

Travail Personnel Encadré

Comment résoudre un problème avec un programme ?

SÉVIGNÉ Alfred

LOPEZ Aliaume

DESMARAIS Yann¹

Lycée Georges Pompidou, 1S1

4 mars 2012

1. Merci à Mme Sirlin et Mme Pivette nos professeurs encadrants

Introduction du Sujet

De nos jours l'utilisation de l'informatique est omniprésente... Dans de nombreux domaines scientifiques elle est utilisée comme un outil pratique qui offre une multitude d'opportunités. Les ordinateurs sont de plus en plus performants et offrent une puissance de calcul impressionnante qui peut maintenant aider dans de nombreux domaines. Nous avons donc inscrit la problématique du TPE dans cette logique de la résolution d'un problème scientifique, et plus particulièrement biologique, grâce à un programme de simulation. Au cours de ce rapport on suivra chronologiquement l'évolution du développement de ce programme : D'abord la première partie mettra en évidence les objectifs et les buts du logiciel. Elle sera consacrée à la vision globale des fonctionnalités du programme que l'on voulait implémenter, elle constitue en quelque sorte son cahier des charges. La deuxième partie est une description du développement du programme en lui-même, elle expliquera également les expériences et recherches auxquelles nous avons procédé pour obtenir les informations nécessaires à celui-ci. Enfin la troisième partie est en premier lieu la présentation du programme final mais également une explication des limites de la simulation et des problèmes qu'il peut éventuellement soulever.

Le travail sur ce programme fut une bonne expérience, tant sur le plan virtuel que les données recherchées pour le créer. Quelques problèmes dans la création de ce programme, mais c'était l'effet recherché d'après la problématique. Le gros de nos ambitions ont pu aboutir et chacun y a mis du sien.

SÉVIGNÉ ALFRED

Ce TPE a été une véritable expérience pour nous et nous a ouvert les portes de la programmation qui restait jusqu'à présent un monde assez hermétique à nos yeux.

DESMARAIS YANN

Ce TPE m'a donné l'occasion de travailler avec des véritables contraintes, et un véritable but. Il m'a aussi permis de travailler avec des non-programmeurs ce qui m'a ouvert l'esprit quand à la réalisation d'un projet en équipe.

LOPEZ ALIAUME

Table des matières

I Mise en place du projet	4
1 Études préliminaires	5
1.1 Objectifs	5
1.2 Recherches	5
1.3 Problèmes	6
2 Cahier des charges de la simulation	7
II Réalisation du projet final	8
3 Le programme général	9
3.1 Présentation des outils	9
3.2 Programmer le contexte	9
3.3 Programme et boucle principale	10
3.4 Le rectangle, la base de l'affichage	11
3.5 Exemple avec le menu	11
3.5.1 Les boutons	12
3.5.2 Le menu	12
4 Études sur les cellules	13
4.1 Choix des cellules	13
4.2 Expérience personnelle	13
4.2.1 Protocole	14
4.2.2 Interprétation et résultats	14
4.2.3 Recherches complémentaires	14
5 Réalisation de la simulation	16
5.1 Avant d'implémenter	16
5.2 Cellule	16
5.2.1 Division	16
5.2.2 Couleur	17
5.2.3 UV	17
5.3 Gérant	18
5.3.1 Définition	18
5.3.2 Stockage des cellules	18
5.3.3 Manipuler les cellules	19
III Rétrospective	20
6 Le programme face aux attentes	21
6.1 Rappel rapide du programme final	21
6.2 Les différences avec notre vision du projet initial	21
7 Limites et approfondissements du programme	22

A Introduction à la programmation	23
A.1 Variables	23
A.2 Contrôles	23
A.3 Fonctions	24
B Complexité algorithmique	26
B.1 Définition	26
C Archiver des données	27
C.1 Les Types de base	27
C.2 Pointeurs	27
C.3 Tableaux	28
C.4 Ennumération	29
C.5 Structures	30
C.6 Listes	30
D Programmation Orientée Objet	32
D.1 Historique	32
D.2 Héritage	32
E Levures	34
E.1 Les levures	34
E.2 L'expérience souche Ade2 et UV	34
F Sitographie et Bibliographie	36

Première partie

Mise en place du projet

Chapitre 1

Études préliminaires

1.1 Objectifs

Le but exact que nous nous sommes fixés pour le programme était l'étude du temps de vie et de division des cellules. Nous avons décidé pour plus de simplicité de nous en tenir à un seul type de cellule. Cela amena à des recherches qui nous orientèrent vers le choix des cellules de levures, qui sont des organismes eucaryotes dont le temps de division correspondait à notre échelle de temps. En effet les levures mettent deux heures pour se diviser une fois quand une cellule humaine en met vingt. Nos objectifs étaient qu'à l'aide du programme, à n'importe quel moment d'une culture de levure, on puisse savoir combien de cellules il y avait dans la colonie. Cela impliquait que le logiciel comprenne un compteur de temps et que l'on puisse stopper la simulation à tout moment pour voir l'état des cellules. A l'origine nous avions prévu deux environnements distincts dans l'interface ce qui permettait d'avoir une culture témoin et un autre test. Cette idée était motivée par le projet de soumettre aux levures des contraintes et de voir comment celle-ci y réagissait. Nous voulions en effet voir quelles étaient les différences entre une culture normale et une, par exemple, exposée à des agents mutagènes tels que les UV. Pour la représentation des cellules nous avions pensé à quelque chose de simple qui permettait une représentation sur un grand nombre de clones à l'échelle d'une colonie entière. Nous pensions le menu comme une fenêtre classique avec une barre d'outil en haut ou apparaîtraient les différentes fonctionnalités, les deux environnements de simulation et une barre dans le bas de la fenêtre qui afficherait les données de la simulation (nombres de cellules, temps de division etc).

1.2 Recherches

Pour réaliser cette vision première que nous avions du programme nous avons procédé il fallait réunir plusieurs conditions :

- Collecter les informations nécessaires à une simulation réaliste
- Faire nous-même les expériences au cas où l'information n'est pas trouvée
- Trouver des outils informatiques performants et qui répondent à nos attentes
- Trouver les fonctions impossibles à implémenter et en proposer d'autres en remplacement

Pour collecter les informations nécessaires nous avons pu utiliser internet (voir bibliographie), quelques livres notamment le livre scolaire de SVT de première et de seconde (voir bibliographie), en se renseignant auprès de notre professeur de SVT ou de nos parents (les parents étaient pour chacun de nous soit biologistes soit informaticiens) pour certains détails.

Pour certaines informations concernant surtout les UV et leurs effets nous avons eu l'occasion de faire une expérience pour tester les colonies de levures exposées à ces rayonnements. Cette expérience est plus détaillée dans la partie deux dans le chapitre consacré aux UV.

Pour le choix des outils numériques nous nous sommes orientés vers un langage qui possède de nombreux avantages dont la simplicité et la rapidité. Malgré tout il avait des problèmes de portabilité. Cela nous empêcha donc de fournir le programme avec ce document, il sera présent lors de la présentation orale et sera pleinement testable...

Ensuite pour les fonctions qui n'ont pas abouti, nous avons procédé par tâtonnement, en essayant de voir laquelle serait la plus cohérente avec l'objectif du programme et celles qui le rendraient plus pratique et facile d'accès. Nous avons par exemple ajouté le fait de pouvoir placer une grille de comptage dans chaque

environnement pour mieux dénombrer les cellules, ou alors le fait de pouvoir zoomer pour voir le processus de division de plus près.

1.3 Problèmes

Enfin il y a la question des problèmes pratiques qui interviennent dès le début du projet dans la mise en place de celui-ci. Ces problèmes étaient d'abord dans la simulation que nous voulions faire : être le plus réaliste possible en restant dans la mesure du réalisable. Trouver certaines données a été assez difficile. Ensuite il y avait également la nécessité que le programme que nous voulions ne soit pas trop complexe à coder en définitive. En général ce ne fut pas une difficulté majeure, nous avons réussi à poser un projet réalisable et les rares fois où ce ne fut pas le cas nous nous sommes arrangés pour trouver des alternatives à ces fonctionnalités trop complexes. Enfin il y avait les problèmes de choix personnels de chacun, nous avons dû nous mettre d'accord pour les orientations que nous souhaitions pour le logiciel. Au cours du développement et de la création du projet ce ne fut pas non plus très problématique du fait de la bonne entente du groupe et d'une communication assez régulière permettant la constatation des évolutions du programme quasiment en direct. En effet nous avions correspondances par mail importante qui se liait à l'utilisation d'une forge collaborative pour le code du programme : Github (voir sitographie).

Chapitre 2

Cahier des charges de la simulation

Avant de se lancer tête baissée dans une expérience, dans des recherches, ou dans un code, il faut impérativement avoir une idée de ce que l'on veut avoir. Cela comprend le design, les fonctionnalités, les représentations.

Même si cette première vision est large et peut-être très éloignée du programme que nous obtiendrons au final, il faut se fixer un objectif à atteindre.

Voici un aperçu du programme :

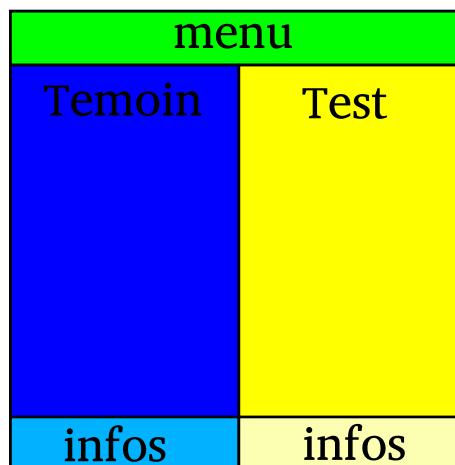


FIGURE 2.1 – Une première maquette

Nous avons :

Un menu qui contrôle les différentes actions

Deux boîtes de pétri qui représentent le témoin et le test

Une barre d'information qui affiche les informations relatives aux simulations

C'est fini pour le design, maintenant il faut définir ce qu'on attend de la simulation :

- Des cellules qui se divisent
- Des UV qui les tuent / font muter
- Ajouter des cellules
- Tuer des cellules
- Compter les cellules

À partir de ces idées nous pouvons avancer vers la réalisation du début de programme qui n'intègre pas encore les valeurs des expériences et recherches. C'est en quelque sorte le patron du programme qui ne reste qu'une maquette permettant d'attaquer la réalisation des bases du programme.

Deuxième partie

Réalisation du projet final

Chapitre 3

Le programme général

3.1 Présentation des outils

Pour réaliser notre programme, il faut des outils logiciels. Voici une liste exhaustive des outils dont nous avons besoin, et le nom ainsi que la description de ceux que nous avons choisi :

Un ordinateur : celui d'Aliaume LOPEZ

Un langage de programmation : nous avons choisi le Vala. Ce langage est en réalité un métalangage, un traducteur qui transforme un véritable langage de programmation : le C. Le C a des avantages, comme la rapidité, et des inconvénients, comme l'insécurité. Le Vala est une surcouche qui permet de simplifier et de clarifier le code, en utilisant des paradigmes de programmation que le C n'a pas à travers une bibliothèque nommée GObject, écrite en C.

Une bibliothèque pour afficher des pixels à l'écran : nous avons choisi la SDL, qui est une bibliothèque pour le C qui permet d'afficher des pixels à l'écran. Elle est plutôt basique, et par la même très simple. On peut donc faire tout ce que l'on veut ... si l'on y met le temps de réfléchir pixel par pixels.

Ces choix ont été faits pour des raisons de performances, mais aussi parce que le C est un langage qui est éprouvé, et dont les vices sont tous connus. Idem pour la SDL. Tous deux sont très bien documentés. Mais, aussi parce que nous avions déjà utilisé ces outils auparavant.

3.2 Programmer le contexte

À partir du cahier des charges de la première partie, nous savons déjà comment mettre en place tout ce qui n'est pas la simulation en elle-même. Car au départ, il n'y a pas encore d'expériences, ni de résultats, on ne peut donc que réaliser ce dont on est sûr d'avoir besoin. Malheureusement, dans la plupart des cas, c'est la chose de plus complexe à réaliser. Par exemple, il faut créer un menu, c'est évident. Or la SDL ne propose rien de tel. Nous allons donc créer notre propre menu, de manière à pouvoir le modifier facilement au cours du développement et le réutiliser dans des programmes ultérieurs. Le menu est en fait un « module ». L'idéal dans la programmation de nos jours, est non pas d'avoir le code avec le moins de lignes possibles, mais le code le plus lisible et le plus modulaire. Chaque module doit être indépendant des autres, être facilement modifiable et extensible. Voici la liste des modules qu'il faut absolument créer :

- Le menu
- La boîte de pétri
- La cellule
- L'affichage¹

Dans chaque module, il y aura différents éléments :

- Des fonctions
- Des variables globales
- Des Classes ou des Structures

Une classe ou une structure est un patron, une définition formelle d'un nouveau type. Après les types classiques comme les nombres ou les lettres, on a la possibilité de créer des combinaisons de type (on peut aussi combiner des structures). Tout ceci est défini clairement dans l'annexe C.5 pour les structures, et D pour les classes.

1. Qui simplifie l'utilisation de la SDL, par exemple une fonction « dessineCellule » qui exécute automatiquement une liste d'instruction SDL

Il faut impérativement que chaque module soit bien configurable, adaptable, réutilisable et modifiable facilement, car nous n'avons pour l'instant encore aucune idée des valeurs exactes de la simulation, il faut donc prévoir tous les cas impérativement.

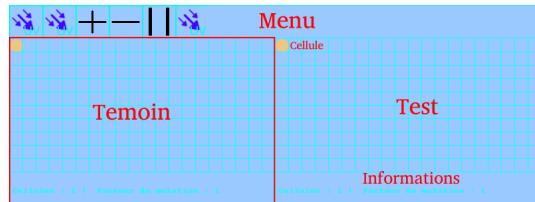


FIGURE 3.1 – Première vue du programme

3.3 Programme et boucle principale

La simulation ne doit s'arrêter que quand on la quitte. On sait donc déjà qu'il y aura une boucle presque infinie au centre du programme, afin d'exécuter chaque action qui fait partie d'un cycle de simulation. On peut donc imaginer une boucle principale de cette manière :

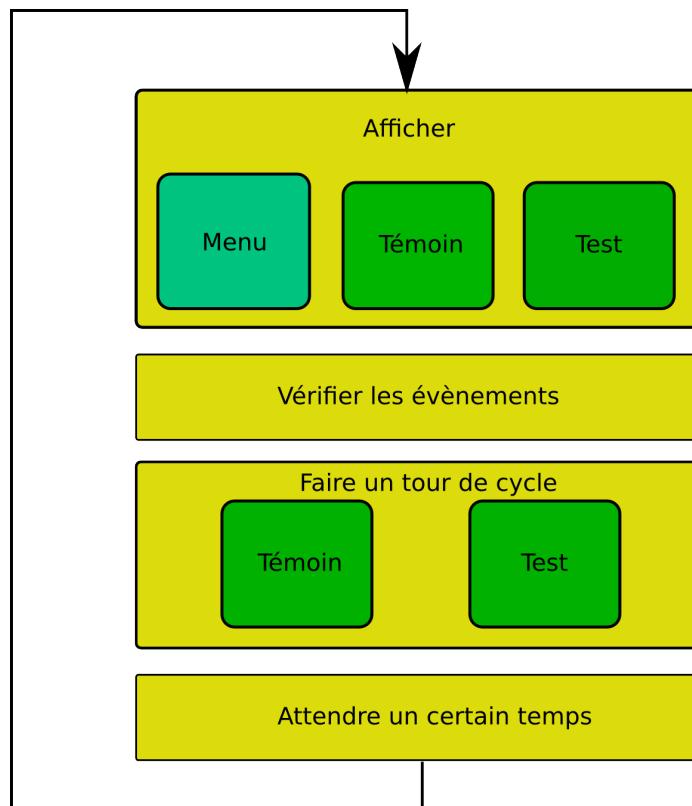


FIGURE 3.2 – Une première idée de boucle

Afficher : Dessiner le menu, la simulation témoin, la simulation test, et tout ce qu'ils contiennent sur l'écran

Vérifier les événements : récupérer le premier évènement produit et le traiter

Faire un tour de cycle : exécuter² un tour de cycle pour la simulation témoin et la simulation test

Attendre : Pour que la simulation soit plus constante, sans cela elle ira aussi vite que possible, et donc sera irrégulière en fonction de la densité de calcul

Recommencer : jusqu'à ce que l'on ai demandé de quitter (géré dans les évènements)

2. C'est un terme volontairement vague, à ce stade, on ne sait pas encore exactement ce que la simulation va faire

On peut imaginer une vision moins naïve de cette boucle en divisant les différentes actions en différents processus distincts. Cette méthode est très utile car les processus s'exécutent en parallèle ou presque³. Ceci permet d'avoir une vitesse d'affichage constante, par exemple 30 images par secondes, une vitesse d'exécution des simulations différente et une boucle qui gère les événements avec une vitesse encore différente.

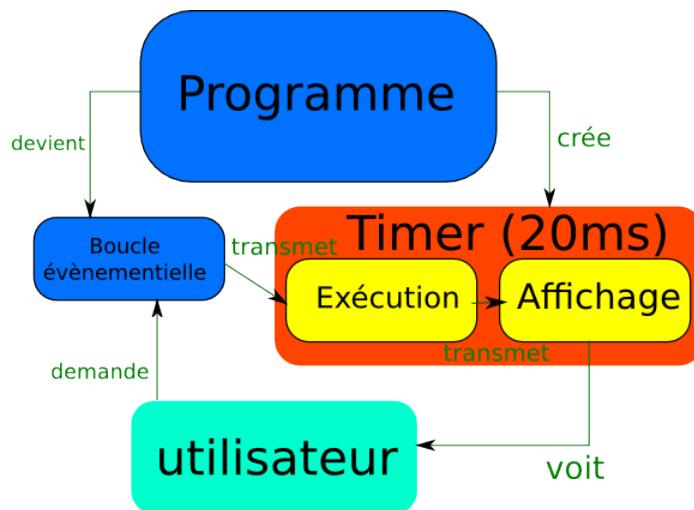


FIGURE 3.3 – Fonctionnement du programme

Bien sûr il faut que les processus puissent communiquer entre eux. Et cela pose problème, étant donné qu'ils s'exécutent en parallèle, et qu'on ne sait pas lequel va être lancé en premier. On peut leur faire partager des variables, mais il faut bien vérifier qu'elle soit à jour, et éviter les modifications simultanées par deux processus d'une même variable! C'est la seule difficulté rencontrée par la séparation en plusieurs processus, la solution a été d'utiliser des verrous : avant de faire une action on demande un verrou sur cette variable et on est sur que personne d'autre que nous ne peut la modifier, ensuite on libère le verrou, et c'est un autre processus qui prend possession de la variable. Cette méthode peut entraîner des ralentissements, des décalages, mais nous ne partageons que 2 variables, donc normalement, il n'y a pas de raisons que ces ralentissements soient visibles.

3.4 Le rectangle, la base de l'affichage

Nous savons que la majorité des éléments de la simulation seront affichés à l'écran. Or pour afficher quelque chose sur une surface, il faut un certain nombre d'informations :

- Position (x, y) sur l'écran
- Taille (w, h)

Nous dessineras uniquement des rectangles, pour des raisons de simplicité, et parce que nous n'avons pas besoin d'autres formes géométriques dans notre simulation⁴.

Avec ces variables, on peut ajouter plusieurs méthodes qui seront utiles :

- déplacer(x, y) ; → redéfinit la position (x, y)
- move (x, y) ; → effectue une translation par le vecteur (x, y)

Nous avons donc notre première classe : `Rectangle`, qui sera la base pour afficher des objets à l'écran.

3.5 Exemple avec le menu

L'exemple du menu étant simple et parlant, nous montrerons comment nous avons pensé ce module. Dans un menu, il y a des boutons. Donc il faut un objet `Menu`⁵, et un objet `Bouton`. Un Bouton est un objet affiché à l'écran, il faut donc le faire hériter de `Rectangle`, un objet `Menu` a une position sur l'écran, il faut donc aussi qu'il hérite de `Rectangle`.

De cette manière on a maintenant des classes comme ceci :

3. C'est le système d'exploitation qui se charge de leur répartir du temps de calcul, ils ne sont donc pas vraiment en parallèle, sauf avec les ordinateurs récents qui ont plusieurs unités de calcul (processeurs)

4. En effet, même si on définit les cellules comme des cercles, pour afficher une image, il faut le rectangle de la taille de l'image, même si l'image elle-même est un rond sur fond transparent

5. De manière à être utilisable dans d'autres programmes, ou pouvoir faire plusieurs menus

3.5.1 Les boutons

Nous allons créer la classe `Bouton` qui hérite de `Rectangle`, mais qui apporte son lot de nouveautés, à savoir tout ce qui est relatif à un bouton :

- une image pour le bouton normal
- une image pour le bouton activé
- une variable pour savoir si le bouton est activé
- une variable de type énumération⁶ pour savoir l'action qu'il entraîne

`Bouton` lui-même n'a pas vraiment de fonctions, tout le code se trouvera dans le gestionnaire des boutons, `Menu`. Mais avant de parler du menu, il faut définir les actions, or nous ne savons pas encore lequelles il y aura, nous allons donc créer une énumération, qui sera traitée dans la boucle des événements. Ajouter une action demandera d'ajouter un item à l'énumération, et un traitement de cet item dans la boucle événementielle.

3.5.2 Le menu

Le menu gère les boutons, son code n'est pas aussi complexe qu'on pourrait se l'imaginer, il contient un tableau de boutons, une fonction `affiche` qui affiche le menu et une fonction `recupBouton` qui retourne le bouton qui se situe sous le curseur⁷.

La seule difficulté réside dans le fait de dessiner correctement les boutons, et au bon endroit. Pour cela il faut savoir que l'écran de la SDL se comporte comme un repère orthonormé, à ceci près qu'il a pour origine le coin haut gauche de la fenêtre, et que son axe des Y est inversé par rapport aux graphiques habituels :

Après ça, il faut aussi savoir que le menu a une position, donc quand on place un `Bouton`, il faut le placer en fonction de la position du `Menu`, et en fonction de la taille d'un `Bouton`. La méthode suivante est utilisée : quand on crée un bouton, on lui donne deux images, une `Action`, et la position du bouton vaut celle de la simulation plus N fois la largeur d'un bouton en x , en fonction du numéro du bouton.

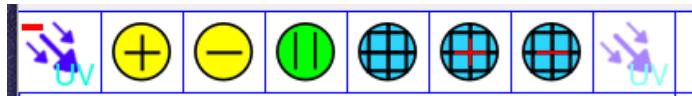


FIGURE 3.4 – Capture d'écran du menu

C'est bon, nous avons un menu fonctionnel, reste à connecter tout cela avec la boucle événementielle et c'est terminé. Nous ne parlerons pas de la boucle événementielle, car elle est en réalité simple. Globalement elle demande si le curseur a cliqué, si oui où. Après elle regarde sur quelle partie du programme le curseur est positionné, et un système de conditions décide de ce qu'il faut faire. Ce code n'est pas intéressant, simplement parce que c'est une suite de conditions imbriquées sans grande complexité.

Conclusion

Nous avons maintenant à travers quelques exemples précis décris toute la mise en place des éléments périphériques à la simulation. Éléments indispensables, mais qui ne concernent pas directement le projet final. Nous allons donc pouvoir passer aux recherches sur la simulation elle-même, qui vont dégager des affirmations et prédictifs pour coder la simulation ensuite.

6. Voir annexe C.4 sur les énumérations, qui parle du menu en exemple

7. En effet, on doit gérer nous même ce sur quoi clique le curseur

Chapitre 4

Études sur les cellules

4.1 Choix des cellules

Les cellules sont la clef de voute de notre simulation, étant donné qu'elles sont les éléments principaux de celle-ci. La première difficulté à été de trouver quel type de cellule nous allions étudier. Pour cela nous avons effectué certaines recherches, pour trouver des cellules simples à comprendre, à manipuler et à représenter. Cela impliquait différents facteurs :

La taille des cellules : les bactéries sont environ 10 fois plus petites que les cellules eucaryotes

La durée de division : élément essentiel pour le programme, et qui varie beaucoup selon les espèces : de 20 minutes à 24h, respectivement pour les bactéries et les cellules humaines

Le métabolisme : les éléments nutritionnels, le fait de survivre en aérobiose ou non, et autre facteurs environnementaux

Le déplacement des cellules : il existe tout types de cellules qui se déplacent ou non, et de différentes manières

L'organisation : cellules isolées, colonies, ou organismes multi-cellulaires

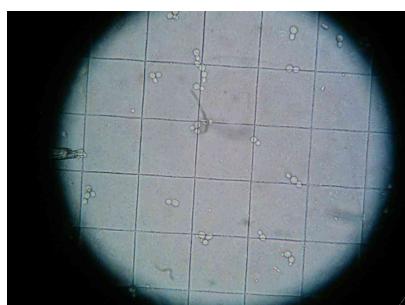


FIGURE 4.1 – Cellules de Malassez

Initialement nous n'étions pas fixés sur le choix d'un type de cellule en particulier, il en découle que le programme, qui avait commencé à être développé en parallèle, peut être configuré pour certains des facteurs.

Ainsi nous avons choisi les levures, qui sont des organismes unicellulaires qui forment des colonies, qui on fait l'objet de nombreuses études précédentes. Ce qui nous a permis d'accéder facilement à des informations les concernant. De plus les levures sont facilement mises en culture ce qui nous a permis de faire de nous en procurer facilement et de faire nos propres expériences.

4.2 Expérience personnelle

Nous avons fait cette expérience pour avoir une idée de l'effet des UV sur les colonies de levures. Notre but était de soumettre les colonies à différents temps d'exposition¹. Nous avions pour projet l'observation au microscope des colonies afin de dénombrer les clones mutés et le taux de mortalité en fonction de la durée d'exposition. Nous voulions procéder à l'aide de lame de comptage telles que les « cellules de Malassez ».

1. Mais à puissance constante

Pour cela nous avons eu besoin de l'aide de notre professeur référent en Sciences de la Vie et de la Terre. Ainsi à l'aide de Madame Sirlin, nous avons réalisé cette expérience, selon le protocole définit ci-après.

4.2.1 Protocole

Matériel

- Six boites de pétri
- Des marqueurs
- De la levure
- Deux lampes à alcool
- Une pipette râteau
- De la verrerie : bêcher, tube à essai ...

Étapes

Tout d'abord il faut créer la gélose dans laquelle nous avons ajouté du glucose, afin de constituer un environnement riche pour le développement des levures. La gélose est une substance composée d'agar-agar, sur laquelle nous allons ensemencer les souches de levure.

Nous avons parallèlement produit une solution de levure de boulanger à faible concentration.

Nous avons répandu la gélose dans les boites de pétri et attendu qu'elle se solidifie. Puis, à l'aide d'une d'une pipette râteau, dans un environnement stérilisé par les lampes à alcool, nous avons répandu la solution de levure et d'eau chaude sur le milieu.

Nous avons crée 6 cultures :

- Deux témoins qui ne seraient pas exposés aux UV
- Deux test qui seraient exposés aux UV durant une nuit
- Deux autres test qui seraient exposés aux UV durant 24h



FIGURE 4.2 – Les six boites de pétri

4.2.2 Interprétation et résultats

Inopportunément, nous n'avons pas eu accès à des microscopes pour analyser les résultats. De plus nous avions utilisé de la levure de boulanger, alors que pour observer des mutations visibles, il aurait fallu des levures ADE2. Malgré tout nous avons bien constaté à l'œil nu la grande disparité du nombre de colonies entre les différents test et témoins. Et donc, n'avons pu que constater la mortalité des UV, mais nous n'avons pas pu calculer des taux précis. Ce genre d'expérience est courante et nous avons pu trouver des résultats probants aussi bien que dans les livres de seconde et première², et des expériences similaires sur internet. Ce qui nous a permis d'enchaîner facilement sur la réalisation du programme.

4.2.3 Recherches complémentaires

Grâce à plusieurs sources nous avons obtenus les informations nécessaires au programme que nous n'avions pas pu récupérer avec l'expérience. Nos sources principales sont : le livre de première de SVT chapitre sur la génétique, activité sur l'exposition de levures aux UV et un site internet d'un professeur de SVT qui propose

2. F

le protocole et le résultat de cette même expérience³. Dans chacun des cas nous avons pu obtenir le tableau récapitulatif des morts et mutation des levures en fonction du temps. C'est ces données que nous avons pu implémenter dans ce programme pour rendre la simulation plus ou moins réaliste lors de l'utilisation de la « fonction UV ». De plus le protocole trouvé sur internet a pu nous donner une idée de comment nous allions procéder lors de notre propre expérience. Enfin ces données ont vraiment pu compléter celles déduites de notre expérience qui, après coup, n'a pu nous fournir les informations voulues.

C'est ainsi que nous avons trouvé que les levures, comme les cellules humaines, avaient un vieillissement cellulaires qui les prédéterminaient à une mort certaine environ après leur 50ème division⁴

Conclusion

À partir des informations recherchées et des interprétations que nous en avons faites, nous avons pu nous lancer dans la véritable réalisation du projet de simulation.

3. Cf F
4. Cf F

Chapitre 5

Réalisation de la simulation

5.1 Avant d'implémenter

Avant de réaliser la simulation il faut savoir que la mise en place des cellules dans la simulation relève de deux choses :

- La mise en place d'une classe **Gerant** de simulation
- La mise en place d'une classe **Cellule**

La **Cellule** se contente d'exister, de mourrir, de muter, d'avoir une couleur, une position. Le **Gerant** lui, est l'environnement de la cellule, il contient toutes les cellules d'une simulation, et c'est lui qui les traite. Pour afficher les cellules, on demande au gérant de le faire. Pour ajouter une cellule à une position, on demande au gérant, quand une cellule veut se diviser, elle demande au gérant, parce que lui sait si elle peut se diviser, ou s'il n'y a plus d'espace libre autour d'elle.

Le code complexe se trouvera donc dans la classe **Gerant**. Elle est avec le système de boucle, le centre du code, autour duquel le reste s'articule.

5.2 Cellule

Avant de représenter une cellule il faut se demander ce que l'on va représenter. Nous avons choisi dans un premier temps de nous simplifier la tâche en la considérant comme un objet simple qui a les membres suivants :

- un nombre de cycles
- un nombre de mutations
- un nombre

Bien entendu, la classe **Cellule** hérite de **Rectangle**, vu qu'elle sera dessinée à l'écran.

Le code complexe des cellules se trouvera en réalité dans le gérant, simplement car il a les informations sur les autres cellules voisines, mais nous en parlerons tout de même dans cette section.

5.2.1 Division

La fonction la plus importante des cellules est la division. C'est aussi la fonction qui pose le plus de problèmes purement algorithmiques. Résumons :

- La division doit se produire une fois toutes les N minutes en fonction des cellules
- La division crée une cellule mère et une cellule fille¹.
- La direction de la division n'est pas déterminée

Donc il faudra créer une nouvelle cellule toutes les N minutes, qui aura les mêmes caractéristiques que sa mère, sauf l'âge, et à une position contiguë à la cellule mère indéterminée.

Pour pouvoir créer une cellule quelque part, il faut vérifier que l'espace est vide. On a donc appliqué la méthode suivante :

- Si la case N est vide alors on divise sinon continuer
- Si une autre case N' est vide alors on divise sinon continuer
- etc ...

¹. Contrairement à ce que l'on pensait encore récemment, les deux cellules filles ne sont pas identiques, et une cellule mère « vieillit » au fur et à mesure des divisions

Pour plus de réalisme, nous avons testé différentes configurations :

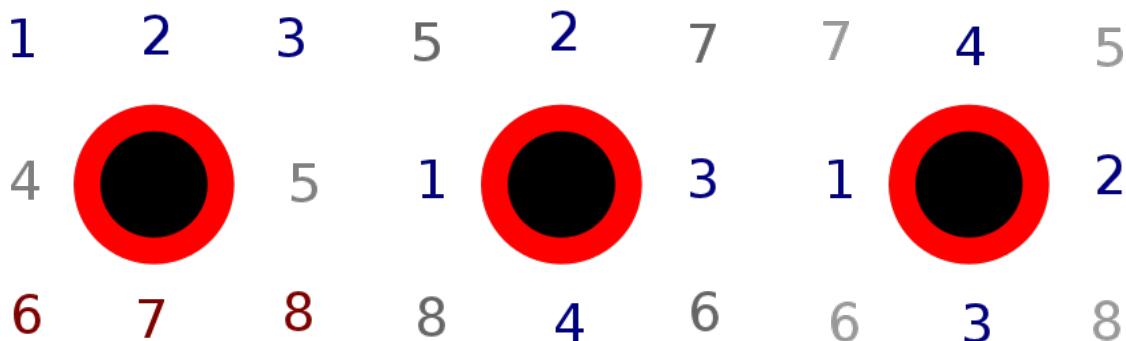


FIGURE 5.1 – Évolution de l'ordre de vérification des cases

Une fois déterminé l'ordre de division, il faut prendre en compte un autre paramètre : lorsqu'il n'y a aucun espace possible, les cellules se « poussent » pour créer assez de place. Faute de temps, nous n'avons pas pu mettre en place une gestion de cette poussée dans les 6 directions, mais uniquement dans les 4 principales. Le code déplace toutes les cellules d'une case, et ensuite permet à la cellule de se diviser. Cette option crée un véritable tapis de cellules qui mélange tous les âges, du fait des déplacements.

5.2.2 Couleur

Une deuxième chose, qui cette fois n'est pas gérée par le **Gérant** mais par l'**Afficheur**, et qui concerne directement les cellules : les codes de couleur. Nous avons choisi d'utiliser des codes de couleur pour signifier les différences entre les cellules selon le modèle suivant :

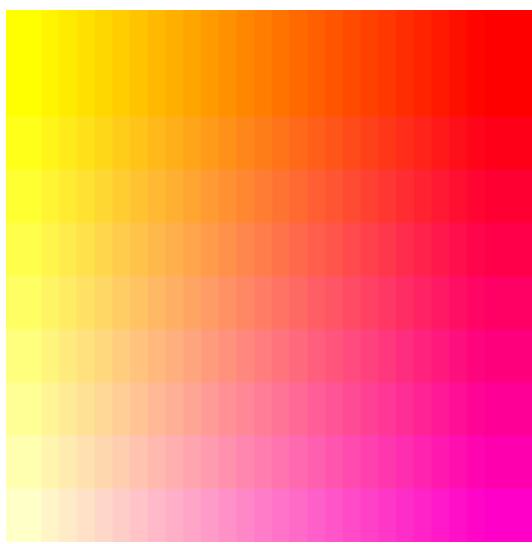


FIGURE 5.2 – Dégradé des couleurs des cellules

La lecture de ce graphhe est la suivante : On part du coin haut gauche pour la cellule normale. Plus on se déplace vers la droite, plus la cellule est vieille. Plus on descend, plus la cellule a subit des mutations.

Malheureusement, il n'est parfois pas très lisible de combiner deux dégradés de couleur sur une seule cellule. Mais c'est le moyen qui est le plus lisible de tous, car il est indépendant de la taille des cellules.

5.2.3 UV

Nous avons souvent parlé des UV, mais jamais nous n'avons implémenté de fonctions qui déterminaient les mutations et la mort des cellules quand les UV étaient activés. Grâce aux résultats des expériences et recherches,

nous avons déterminé les taux de mutation en fonction du temps. Indépendamment nous avons déterminé le taux de mortalité aux UV. On peut les retrouver dans l'annexe E.

Ces taux sont des statistiques. Dans le programme, nous avons utilisé des nombres aléatoires avec des probabilités de sortie qui correspondaient aux statistiques. Nous avons toutefois modifié un peu la courbe des mutations, pour la rendre plus facilement modélisable en une équation simple. Pour le taux de mortalité idem.

Globalement les chiffres tirés correspondent, et la simulation fait ce qu'on attend d'elle. Mais vient le problème du fait que les UV sont très agressifs, et une heure d'exposition tue déjà un peu plus de la moitié des cellules de levures, alors qu'elles mettent environ deux heures à se diviser. Il en résulte un certain décalage, entre la rapidité de réaction aux UV et la vitesse de division, qui peut rendre étrange l'utilisation de la simulation.

5.3 Gérant

5.3.1 Définition

Nous devons définir les actions du **Gerant** :

- Ajouter une cellule
- Tuer une cellule
- Diviser une cellule
- Dessiner les cellules
- Faire exécuter un cycle à toutes les cellules

5.3.2 Stockage des cellules

Devant ceci il nous faut un moyen de conserver les cellules efficacement. Pour cela il faut connaître les structures de données décrites dans l'annexe C.3 et C.6. Pour choisir entre une liste et un tableau², il faut avant tout connaître nos besoins. Au vu des actions définies plus haut voici les actions par ordre d'occurrence à l'exécution :

1. Parcourir
2. Supprimer
3. Ajouter

Au vu des annexes C.3 et C.6, le tableau semble beaucoup plus adapté, étant donné qu'il ne demande qu'un temps constant pour lire une valeur (voir annexe B).

La conservation n'a rien à voir avec l'affichage, puisque toute **Cellule** a une position sur l'écran, indépendante de sa position dans le tableau.

Néanmoins, nous avons décidé pour plus de simplicité, de gérer aussi l'affichage des cellules sous la forme de grille. Donc, la position d'une cellule dépendra de sa case. Ceci n'est pas obligatoire, mais les points suivants défendent cette idée :

- Gérer les cellules « librement » induit une gestion des collisions, chose complexe et coûteuse en rapidité quand elle n'est pas optimisée
- Gérer les cellules en grille permet de mieux contrôler des divisions, et permet d'effectuer plus facilement une grille de comptage que des cellules libres
- Gérer les cellules en grille permet de ne JAMAIS redimensionner le tableau, donc toutes les opérations sont à complexité constante.

Nous avons donc réalisé la conservation des cellules dans un tableau à deux entrées, lignes et colonnes. En réalité, c'est un tableau de tableaux de cellules³. Ce qui revient à un tableau à double entrée. On accède donc à une cellule en faisant : `tableau[ligne][colone]`.

Les cases seront soit vide, soit avec une cellule. Dans le code, il faudra donc bien penser à toujours vérifier que la case contient une cellule avant de faire une action, ce qui cause malheureusement beaucoup de soucis qui sont difficiles à voir facilement.

Il en résulte que les cellules sont sous la forme de grille, ce qui laisse paraître une certaine rigidité dans la simulation. Néanmoins, quand les cellules sont petites, la grille aussi est petite, et le problème de « cases » se ressent moins.

2. Il existe d'autres structures, mais pour cette simulation, liste et tableaux semblaient les plus pertinentes

3. Un tableau est une suite d'élément, sur une seule « ligne »

5.3.3 Manipuler les cellules

La gestion des cellules découle de l'utilisation d'un tableau. Le **Gerant** doit néanmoins conserver d'autres variables, comme le nombre de cellules par exemple. Pour celui-ci, il va falloir mettre en place un compteur. Le principe est d'incrémenter le compteur de 1 à chaque création, et de le décrémenter de 1 à chaque destruction de cellule. On pourrait imaginer faire cela dans chaque fonction qui crée ou détruit une cellule. Mais cela ne serait sûrement pas judicieux. Au contraire, nous utiliserons les constructeurs et destructeurs⁴ de la Classe **Cellule** pour incrémenter et décrémenter. Il en découle qu'il n'y a que 2 lignes de code à ajouter, aucun risque de bug, et un code plus lisible.

De même, le **Gerant** contient un code que l'on trouve souvent, avec juste quelques variantes : le parcours du tableau entier. En effet, pour afficher, pour exécuter, ou pour tout supprimer, on utilise une même boucle qui parcourt le tableau, avec un code différent à l'intérieur. Au lieu d'utiliser Ctrl-C et Ctrl-V nos amis, nous allons introduire un concept magnifique de la programmation : le passage de fonction en argument. En paramètre d'une fonction on donne des variables, mais on peut aussi donner des fonctions ! De ce fait, nous pouvons créer une fonction **parcourir** (*fn fonction*) qui prend en paramètre la fonction à appliquer, et l'applique sur tous les éléments du tableau. Le code est plus modulaire, et plus facilement modifiable. Si la structure de stockage des cellules changeait, il suffirait de changer le code dans **parcourir** plutôt que de changer toutes les fois où l'on a parcouru le tableau en entier.

Conclusion

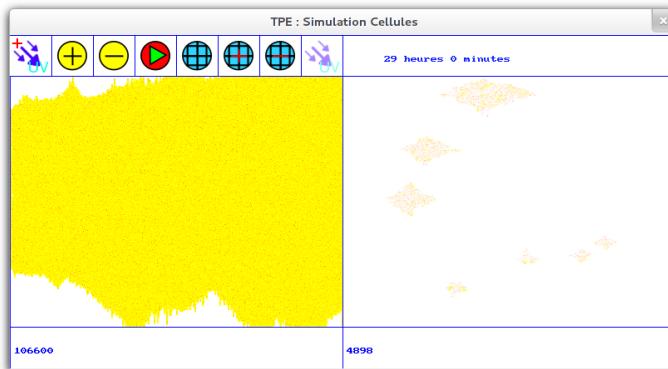


FIGURE 5.3 – Capture d'écran du programme en fonction

Nous avons maintenant un programme fonctionnel, dont nous avons expliqué le développement en prenant l'exemple du **Menu** comme objet périphérique de la simulation, et dont nous avons décrit les recherches de données qui ont permis de coder la simulation elle-même.

Le programme fait ce qu'on veut de lui, mais il reste quand même relativement simpliste, et en cela on est en mesure de se demander s'il répond bien aux attentes de départ.

4. Voir l'annexe D sur la POO

Troisième partie

Rétrospective

Chapitre 6

Le programme face aux attentes

6.1 Rappel rapide du programme final

Le programme se présentait sous la forme d'une fenêtre composé d'une barre d'outils, deux zones de simulations et d'une barre d'information. Les fonctions de bases étaient : Ajouter une cellule, retirer une cellule, mettre en pause, afficher la grille/éditer sa taille et enfin activer les Ultra-violet/les désactiver. Il y avait deux zones distinctes permettant une séparation utile pour faire un témoin et un test. La barre d'information donnait le nombre de cellules qui étaient présentes dans l'ensemble de l'environnement ou dans une zone délimité par la grille. Les Cellules étaient colorées différemment en fonction de leur âge. De plus il existait une fenêtre de paramétrage qui permettait de changer le temps de division, l'échelle de temps, l'échelle de taille et la répartition des cellules.

6.2 Les différences avec notre vision du projet initial

Le projet une fois mené à bien ressemblait en grande partie à ce que nous pensions au début. Les quelques différences qui méritaient d'être notées sont : L'interface et l'aspect graphique la forme très simples de cellules, la gestion des mutations par UV, l'édition des caractéristique des cellules et certaines fonctions que nous n'avions pas prévu dès le début du TPE...

D'abord l'aspect graphique n'était pas exactement celui escompté, nous pensions pouvoir trouver le moyen de faire des boutons plus esthétique dans la barre d'outils. Cela restait un détail mais plus important dans ce registre était la représentation des cellules : Nous planchions au départ pour une représentation plus réaliste avec le noyau des cellules voir certains organites. Nous avons changé d'avis lorsque nous avons constaté que lorsque l'on observait les cellules dans leur globalité de loin ces inscriptions brouillaient la représentation. Ainsi nous avons décidé pour le programme un simple dégradé de couleur pour l'âge des cellules.

Pour les mutations par le problème se trouve plutôt du côté des recherches peu fructueuses que nous avions menés pour trouver les taux. Nous n'avons trouvé que peu d'information et parfois contradictoires. Dans notre expérience l'un des objectif était de trouver ce taux de mutation, hors nous étions trompé de souche de levures : nous avions pris de la levure de boulanger qui ne forme pas de mutation visible à l'œil nu, nous n'avons constaté qu'après coup qu'il n'y avait que certaine souche comme les levures Ade2 qui change de couleur après mutation voir annexe levures.

Ensuite une des modifications qui ont été faites eut, cette fois ci, un effet bénéfique : Nos cellules ne possédaient que certaines caractéristiques fixes mais grâce à une modification nous avons créé un menu pour éditer celle-ci. Maintenant le programme a gagné une forme d'évolutivité car il peut théoriquement simuler différents types de cellules pour ceux que l'utilisateur connaisse les caractéristiques de chacun d'eux. C'est une différence avec notre vision initiale puisque nous pensions que le programme se contenterait de paramètres uniques pour les cellules.

Enfin nous avons eu l'idée d'ajouter certaines fonctions qui rende le programme plus complexe. Il y a le fait de pouvoir choisir les échelles, le zoom, les grilles qui permettent un comptage facilité et les UV qui se contentent de détruire un certains nombres de cellules, ne possédant pas les pourcentages de mutations non-létale. Ces fonctionnalités n'étaient pas prévu dès le départ mais nous avons ressenti le besoin de les ajouter pour améliorer la qualité du programme et les opportunités qu'il ouvrait...

Chapitre 7

Limites et approfondissements du programme

Le programme tel qu'il existe actuellement est fini. Mais fini ne veut pas dire qu'il est terminé.

Un programme n'est jamais terminé, il y a toujours des choses à apporter, des bugs à corriger, même les versions finales ne le sont pas. Chaque nouvelle version apporte son lot de nouveautés, et son lot de frustration du fait de ne pas avoir pu finir tout ce que l'on voulait créer.

Le programme aujourd'hui n'est plus modifié jusqu'à la présentation orale, de manière à ne pas avoir des fonctions qui ne sont pas soigneusement décrites dans ce rapport.

Mais nous avons quand même dressé une liste des limites qu'il a actuellement, et des fonctions que l'on pourrait lui ajouter. Cette liste a été créée au fur et à mesure de la conception par toute l'équipe, chacun apportant sa vision de la simulation. Mais le plus gros des idées, viennent des personnes à qui nous avons fait tester le programme, généralement des amis ou nos parents, qui chacun dans leur domaines, ont critiqué. Ci-dessous une liste exhaustive des fonctions suggérées :

La forme des cellules : les cellules sont uniquement représentées comme des rectangles de couleur. Il eu été intéressant de pouvoir mettre une image plus réaliste en conservant le jeu des couleurs. Il faudrait pouvoir définir ses propres images de cellule, ce qui permettrait de renforcer la généricité du programme.

Les caractéristiques non rendues : le programme ne peut pas montrer de façon visible toutes les caractéristiques des cellules. Par exemple la taille de la membrane, le caryotype ou certains métabolismes. Du fait de la « surcharge » d'information que cela engendrerait. Il aurait fallu pouvoir « zoomer » sur une cellule en particulier pour avoir des informations précises sur celle-ci.

La portabilité : le manque de portabilité du programme rend son installation sur un ordinateur plus difficile que prévue. C'est un problème non négligeable pour la distribution, il nous a par exemple été impossible de vous livrer le programme sous la forme d'un exécutable unique, du fait des librairies utilisées, qui auraient du être installées séparément, ce qui aurait été beaucoup moins simple qu'une vidéo de démonstration.

La gestion du temps : il manque une gestion automatisée du temps qui soit plus simple et intuitive. Actuellement la configuration du programme demande une petite gymnastique mentale qui est désagréable, surtout quand on sait qu'il est possible de se l'éviter. Une autre fonctionnalité de gestion du temps serait de pouvoir modifier la vitesse de la simulation durant l'exécution du programme, pour ralentir durant l'exposition aux UV, et accélérer durant des phases de division par exemple.

Les divisions : les divisions « poussantes » n'existent que pour les 4 côtés, et non pas les diagonales, il en résulte que les colonies ont une forme de losange.

La 3D : en réalité les cellules se divisent dans l'espace tri-dimensionnel. Les levures par exemple forment des colonies sous forme de « bulles », qui ne dépassent pas une certaine hauteur du fait de la gravité, mais qui peut quand même être conséquente. Plutôt que de faire de la 3D, on aurait pu imaginer une légende de densité avec des couleurs, mais elle pose le problème des autres légendes qui utilisent déjà des couleurs.

L'environnement : modifier les UV c'est bien, mais ce n'est pas vraiment suffisant, dans le principe, le logiciel devrait gérer la nutritivité de l'environnement, la température, la capacité d'ajouter des substances comme les antibiotiques. En résumé, pouvoir agir beaucoup plus sur l'environnement de la simulation.

Random : souvent en biologie quand il faut ensemencer des cultures, on utilise des billes de verre et on secoue, ce qui a pour effet de disperser la solution de manière aléatoire et donc statistiquement quasi-uniforme. Il faudrait utiliser une fonction de ce type pour l'ensemencement de départ de la simulation.

Annexe A

Introduction à la programmation

Dans ce TPE dédié à la réalisation d'une simulation, nous allons devoir poser quelques bases de la programmation rapidement. Nous parlerons uniquement de la programmation en Vala, le langage choisi, de manière à se concentrer sur le fond, plutôt que sur la forme, car d'autres langages utilisent des concepts identiques mais avec des syntaxes parfois singulièrement différentes.

Nous tenterons de mettre en relation la programmation et les mathématiques, de manière à bien faire comprendre des concepts qui peuvent paraître abstrait de prime abord.

Vala est un langage dit « haut niveau », c'est à dire qu'il permet de faire simplement des choses complexes, et qu'il permet à certain moment une certaine abstraction du fonctionnement de la machine. Contrairement à des langages dits « bas niveau » qui collent directement au fonctionnement de la machine. Il en découle que ce que nous dirons ici, n'est valable que pour une certaine catégories de langages, et que la syntaxe que nous utiliserons, dans un souci de simplicité, sera uniquement celle du Vala.

Si vous avez déjà quelques notions en algorithmie, vous n'aurez pas besoin de lire le chapitre sur les variables, mais le chapitre sur les fonctions vous sera utile quelque soit votre niveau.

A.1 Variables

Les variables sont le concept de base d'un programme. Une variable est une boîte, sur laquelle on pose une étiquette, et dans laquelle on met une valeur.

```
1 int a = 6;
```

Ce code crée une boîte étiquetée « a » qui est de type « int »¹, et qui a pour valeur « 6 », le point virgule signifie la fin d'une commande. Cette opération est nommée affectation.

En parlant d'opération, il en existe un certain nombre :

- addition avec le symbole « + »
- soustraction avec le symbole « - »
- division avec le symbole « / »
- multiplication avec le symbole « * »
- modulo² avec le symbole « % »

La priorité opératoire suit les mêmes idiomes que les mathématiques. Les calculs peuvent comporter des variables :

```
1 int a = 3;
2 int b = a * 6; // b = a * 6 => 3 * 6 => 18
3 a = b + 3;
```

Après avoir défini une variable, son contenu peut encore changer durant le reste du programme.

A.2 Contrôles

Encore une fois cette section peut être passée par les personnes qui ont déjà une idée sur l'algorithmie. Les structures de contrôle sont une partie d'un langage, voici des contrôles les plus courants en Vala :

1. int est l'abréviation d'integer, soit un nombre entier. La liste des principaux types du vala est visible dans l'annexe C.1
2. Le modulo est le reste d'une division euclidienne

if (condition) code else if (condition) code else code : si une condition est vraie alors on exécute le code entre accolades, sinon si une autre est vraie³ on exécute le code entre accolades, sinon si aucune des précédentes n'a été vérifiée on exécute le code entre accolades

while (condition) code : tant qu'une condition est vérifier, exécuter le code suivant

for (a = départ ; condition ; a += ?) code : effectue la boucle tant que la condition est vraie, mais à chaque fois, la variable a est modifiée, généralement on met a dans la condition de manière à répéter un certain nombre de fois une action.

Exemples :

```

1 int a = 5;
2 while (a < 60) { // tant que a < 60
3     a = a*2; // on le multiplie par 2
4 }
5
6 if (a > 80) {
7     a = 80;
8 } else if ( a > 70) {
9     a = 70;
10 } else {
11     a = 60;
12 }
```

Le but de cette annexe n'est pas d'apprendre à programmer, mais à donner certaines connaissances indispensables à la compréhension du TPE. Du moment que le concept est saisi, vous pouvez passez à la suite.

A.3 Fonctions

Les fonctions en programmation sont en réalité très similaires aux fonctions mathématiques. Si on définit par exemple en mathématique une fonction :

$$f(x) = x^2 + 3x + 5$$

En Vala on la notera :

```

1 // le int devant le f indique le type de la valeur de retour
2 int f (int x) {
3     return x * x + 3 * x + 5; // la fonction renvoie la valeur du calcul suivant
4 }
```

La grande différence est que en Vala il faut préciser le type d'entrée et le type de retour, car il n'y a pas que des entier quand on programme. Sinon tout est assez proche. Pour l'utilisation de notre fonction :

```
1 int a = f(56); // a prend la valeur du retour de la fonction f
```

Mais une fonction en programmation est bien plus qu'une fonction mathématique, parce qu'elle peut faire autre chose que retourner une valeur, et avoir plus d'un paramètre, par exemple :

```

1 int puissance (int x, int pow) {
2     int i;
3     int retour = 1;
4     for (i = 0; i < pow; i++) {
5         retour = retour * x;
6     }
7     return retour;
8 }
```

Ici on utilise une boucle pour effectuer un certain nombre d'action, mais on peut aussi utiliser une fonction dite « récursive » c'est à dire qui s'appelle elle même lors de son exécution :

```

1 int factorielle (int x) {
2     if (x == 1) {
3         return 1;
4     } else {
5         return factorielle (x - 1) * x;
6     }
7 }
```

3. Sous-entendu uniquement si la précédente à échoué

Attention, il faut bien définir au moins un cas ou la fonction ne s'appelle pas, sinon elle est infinie !

Les fonctions peuvent prendre des paramètres de n'importe quel type, elles peuvent avoir une valeur de retour de n'importe quel type, et peuvent faire autant de calculs intermédiaires qu'elles veulent avant de donner une valeur de retour, ou même de ne renvoyer aucune valeur. Par exemple la fonction de la SDL qui permet d'afficher un pixel à l'écran, ne renvoie rien, elle se contente d'afficher un pixel.

Annexe B

Complexité algorithmique

B.1 Definition

Quand les scientifiques se sont posé la question d'énoncer formellement et rigoureusement ce qu'est l'efficacité d'un algorithme ou au contraire sa complexité, ils se sont rendus compte que la comparaison des algorithmes entre eux était nécessaire et que les outils pour le faire à l'époque étaient primitifs. Dans la préhistoire de l'informatique (les années 1950), la mesure publiée, si elle existait, était souvent dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé. Une approche indépendante des facteurs matériels était donc nécessaire pour évaluer l'efficacité des algorithmes. Donald Knuth fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa série *The Art of Computer Programming*.

Wikipédia

La complexité algorithmique est donc une mesure indépendante de tout facteur matériel, ou même logiciel. Il existe plusieurs manières de calculer la complexité algorithmique, nous nous contenterons d'utiliser la plus simpliste, celle qui définit l'ordre de grandeur du nombre d'opération nécessaires à la réalisation du résultat en fonction nombre d'entrée N , dans le pire des cas.

Par exemple le parcours d'un dictionnaire. Il y a différentes manières de coder, la méthode naïve est la suivante : parcourir tous les noms jusqu'à tomber sur le bon. Cette méthode demande au pire pour un dictionnaire de N entrées un nombre N d'opération (dans le pire des cas le nom est à la fin). Il existe aussi la méthode par dichotomie : prendre le nom du milieu. Regarder si le nom cherché est au dessus ou en dessous. prendre encore la moitié, et ainsi de suite. Le principe est de découper le dictionnaire en deux à chaque fois. Dans le pire des cas pour un dictionnaire de N éléments, il faudra couper le dictionnaire $\log_2(N)$ fois (par définition du logarithme de base deux). Le deuxième algorithme est donc beaucoup plus efficace que le premier. C'est là tout le principe de mesurer la complexité algorithmique de différentes méthodes, savoir les-quelles seront les plus efficaces. Imaginons que nous voulions accéder à la valeur N d'un tableau en C : il faut effectuer une addition, puis récupérer la valeur du pointeur (cf C.3), soit une complexité de 1. Mais pour accéder à la même valeur N dans une liste, il faut effectuer N opérations (cf C.6).

C'est donc un bon indicateur afin de déterminer comment aborder un problème en fonction des utilisations que nous faisons des variables.

Annexe C

Archiver des données

En programmation il est très courant de vouloir conserver des données durant l'exécution du programme.

Nous parlerons ici uniquement des données qui sont dans la mémoire vive, c'est à dire la mémoire tampon de l'ordinateur, contrairement à la mémoire morte, qui est souvent un disque dur ou un CD. Pour conserver une information dans la mémoire vive, il faut généralement créer une variable. Nous détaillerons ici tout ce que l'on peut faire avec des variables.

C.1 Les Types de base

Une machine étant une suite de transistors, et la mémoire vive une suite de cases vides et pleines : tout est un nombre binaire. Mais les langages de programmation permettent de faire abstraction de cette complexité technique. Il existe donc en Vala différents types :

- **int** : entier, avec des variantes comme : « **uint** » (entier uniquement positif), ou « **int32** » (grand entier)
- **bool** : booléen, un type qui vaut 0 ou 1, communément **true** et **false**
- **float** : nombre à virgule
- **double** : nombre à virgule à double précision
- **char** : caractère (en réalité c'est un entier positif de maximum 256, qui est traduit en caractères ensuite)
- **string**¹ : tableau de char, donc du texte vu qu'un char est un caractère.

Ces types sont la base du langage, et on les retrouve souvent car c'est la forme la plus simple de conserver une donnée. Mais il existe un autre type, légèrement plus complexe, qui permet de faire beaucoup plus de choses.

C.2 Pointeurs

Pour commencer il faut comprendre comment fonctionne la mémoire de notre ordinateur :

Adresse	Valeur
1600	10
1601	23
1602	505
1603	8

FIGURE C.1 – La mémoire

Ce code demande une case mémoire avec une adresse, et met le nombre 3 dedans.

1. String = chaîne, sous entendu chaîne de caractères

```
1 int a = 3;
```

On peut récupérer l'adresse d'une variable. Par exemple pour afficher la valeur de a, puis son adresse en mémoire :

```
1 printf ("La valeur est %d", a);
2 printf ("L'adresse est %p", &a);
```

Le plus intéressant est qu'une adresse est un nombre, donc on peut la conserver dans une variable aussi ! La syntaxe est la suivante (avec type étant type du langage) :

```
1 Type age = valeur;
2 Type* pointeurSurAge = &age;
```

Ce qui donne en mémoire :

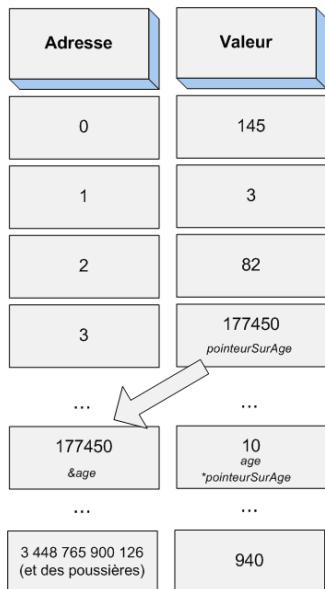


FIGURE C.2 – Un exemple de pointeur

On peut ensuite accéder à la valeur pointée par un pointeur avec la syntaxe suivante :

```
1 *pointeurSurAge; // renvoie la valeur de age
```

Les pointeurs ne sont que des variables qui contiennent des adresses mémoire. Mais pour que l'ordinateur comprenne comment traiter la valeur pointée, le pointeur doit avoir le type de la variable (suivie de l'étoile du pointeur).

Une application directe du pointeur est le tableau, qui va permettre de conserver un ensemble de données.

C.3 Tableaux

Un tableau est une suite de cases mémoire du même type :

```
1 int tableau[4]; // Un tableau de 4 cases avec des int
2 printf ("%d", tableau[1]); // valeur de la case 1
3
4 tableau[10] = 2; // modifie une valeur du tableau
```

En réalité, voilà à quoi ressemble un tableau dans la mémoire de l'ordinateur² :

2. Image tirée de : <http://www.siteduzero.com/tutoriel-3-14015-les-tableaux.html>

Adresse	Valeur
1600	10
1601	23
1602	505
1603	8

FIGURE C.3 – Un tableau dans la mémoire

Lorsqu'un tableau est créé, il prend un espace contigu en mémoire : les cases sont les unes à la suite des autres. L'ordinateur « saura » que c'est un tableau, et quand on demande la 3ème valeur, il prend la 3ème en partant de la première case du tableau.

Donc en réalité, en Vala, un tableau est un pointeur vers la première variable d'une suite de variables contigues. On peut donc considérer que :

```

1 *(tableau); // envoie la valeur de la case d'adresse 'tableau'
2 tableau[0]; // envoie la case 0
3 *(tableau + 3); // envoie la valeur de la cases d'adresse 'tableau' + 3
4 tableau[3]; // envoie la case 3 ( quatrieme car on compte de 0 )

```

Les syntaxes sont différentes, mais mènent au même résultat, et expliquent un peu mieux le fonctionnement du tableau.

Il en découle que le tableau a une taille fixe. Et que pour l'agrandir, il faut trouver un espace suffisant de cases mémoire contigues.

Le langage C n'impose pas à une implémentation de vérifier les accès, en écriture comme en lecture, hors des limites d'un tableau ; il précise explicitement qu'un tel code a un comportement indéfini, donc que n'importe quoi peut se passer. En l'occurrence, ce code peut très bien marcher comme on pourrait l'attendre [...] ou causer un arrêt du programme avec erreur [...] ou encore corrompre une autre partie de la mémoire du processus [...] ce qui peut modifier son comportement ultérieur de manière très difficile à prévoir.

Wikipédia

Il faut donc faire très attention à ne jamais dépasser la taille d'un tableau, sous peine d'avoir des résultats innatendus, ou un plantage complet de la machine qui lance la simulation.

C.4 Ennumération

L'énumération permet de créer un nouveau type qui peut prendre un certain nombre de valeurs dans un ensemble fini, imaginons un type « Volume » qui peut avoir uniquement les valeurs suivantes : FORT, MOYEN, FAIBLE, NULL, INFINI :

```

1 enum Volume {
2     FORT, FAIBLE, MOYEN, NULL, INFINI
3 };
4
5 Volume a = Volume.FORT;

```

Ceci est utilisé pour clarifier le code, il est toujours plus lisible de faire des conditions avec des mots, plutôt qu'avec des nombres. Par exemple, le menu a un certain nombre d'actions, au lieu de leur associer un nombre, on leur associe une valeur de l'énumération ActionMenu.

Mais si ceci permet quasi-uniquement de clarifier le code, il existe un moyen de créer nos propres types plus complexes.

C.5 Structures

Une structure est un type qui permet de combiner des variables :

```

1  typedef struct Personne {
2      int age;
3      string nom;
4  };
5
6  Personne p;
7  p.age = 16;
8  p.nom = "jaques";

```

C'est à partir de ce type de structure que l'on peut définir des choses plus complexes.

C.6 Listes

Les listes sont une des choses plus complexes décrites dans la section précédente, et qui ne sont malheureusement pas gérées directement par le C³. On fait en réalité la chose suivante :

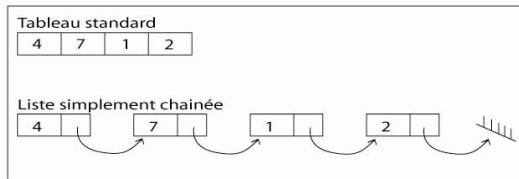


FIGURE C.4 – Une liste chainée et un tableau

```

1  struct element
2  {
3      int valeur;
4      element *nxt; // pointeur vers l'element suivant
5  };

```

On a donc uniquement des structures qui se pointent les unes vers les autres, sans avoir besoin de zones contigues. Les opérations d'ajout et de suppression d'élément sont plus rapides :

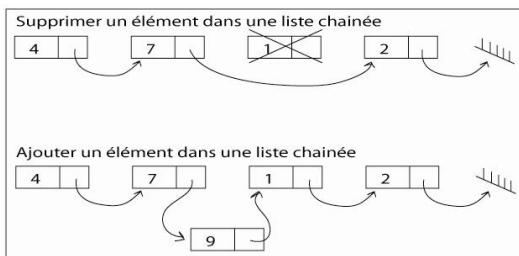


FIGURE C.5 – Ajout et suppression d'un item à la liste

Mais pour récupérer un élément, on est obligé de parcourir la liste. Ce qui fait perdre du temps. Pour récupérer l'élément 3, il faut aller au un, puis au deux, et enfin regarder la valeur du numéro 3 !

Ce qui est bien moins efficace qu'un tableau. Il n'y a donc pas que des avantages, ni que des inconvénients. Les listes ne sont pas toujours simplement chainées, on peut aussi en avoir des doublement chainées, c'est à dire que chaque élément pointe vers le suivant ET le précédent. Il peut aussi y avoir des listes cycliques, c'est à dire que le dernier élément pointe vers le premier.

3. Mais le Vala permet d'en utiliser plus simplement en faisant le travail à notre place

Conclusion

C'est la fin de cette annexe sur la manière de conserver une information en informatique, particulièrement en Vala. Cette n'a pas vocation d'être exhaustive, et ne doit pas être considérée comme un tutoriel, elle est juste une nécessité pour expliquer les choix faits durant la réalisation du programme, et afin de montrer que ces choix étaient réfléchis et non totalement arbitraires, car de ces choix découlent une grande partie de l'apparence de la simulation.

Annexe D

Programmation Orientée Objet

D.1 Historique

La définition de wikipédia est assez claire et concise, nous la compléterons par un exemple :

La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique élaboré par Alan Kay dans les années 1970. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait communiquer avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; la communication entre les objets via leurs relations permet de réaliser les fonctionnalités attendues, de résoudre le ou les problèmes.

Wikipédia

La programmation orientée objet introduit la notion de « classe ». Une classe est un patron, une définition d'un objet. Chaque « classe » définit la structure d'un type d'objet. Cette définition contient des variables « membres », et des fonctions membres dites « méthodes ».

Imaginons une Classe **humain**, qui a des variables *internes* et des fonctions *internes*. On pourra créer des objets à partir de ces patrons. C'est à dire créer un être humain à partir de la classe qui définit **humain** pour conserver l'exemple. Chaque objet à la même structure, mais peut évoluer différemment au cours de l'exécution. On peut créer deux humains avec une variable **couleur des yeux** par exemple. Pour le premier elle peut valoir **bleu** et pour le second **vert**. La différence entre une classe et une structure est qu'une Classe peut contenir des variables ET des fonctions. Par exemple la fonction **direBonjour** de la Classe **humain** utilise la variable **nom** et écrit : bonjour, je m'appelle + nom. Cette même fonction aura des effets différents sur différents objets.

Une classe peut avoir comme variable membre n'importe quel type. Comme une classe est un type, une classe peut en contenir une autre, c'est même toute l'utilité en général.

D.2 Héritage

Une notion très importante qui vient avec la POO est l'héritage. Un exemple vaut tout les discours :

- Une classe **Humain**
- Une classe **Femme** qui hérite d'**Humain**
- Une classe **Homme** qui hérite d'**Humain**

La classe **Humain** définit des variables et fonctions propres aux humains. Ces fonctions sont réutilisées dans la classe **Homme** et la classe **Femme**, qui sont des humains, mais avec des caractéristiques supplémentaires et des réactions différentes. Au lieu de réécrire deux fois toutes les fonctions et attributs d'**Humain** dans les classes **Femme** et **Homme**, on créer une classe qui les regroupe, et on fait hériter **Homme** et **Femme** de celle ci.

En plus de permettre une certaine paresse, cette méthode a le mérite de permettre une chose : **Homme** et **Femme** peuvent être considérés comme des **Humain**. Si on veut créer une fonction qui compare la taille de deux personnes, on a juste à faire ceci :

```
1  bool plus_grand_que (Humain p, Humain s) {  
2      if (p.taille > s.taille) {  
3          return true;  
4      } else {  
5          return false;  
6      }
```

7 }

On peut donner à cette fonction aussi bien deux **Femme**, deux **Homme** ou un **Homme** et un **Femme**. Étant donné qu'ils sont tous « convertibles » en **Humain**.

Il existe d'autres principes dans la POO que nous n'aborderons pas ici simplement parce que c'est une annexe, qui même si elle est nécessaire à la compréhension, n'est pas le centre de ce TPE.

Annexe E

Levures

E.1 Les levures

Les levures sont des champignons unicellulaires aptes à provoquer la fermentation des matières organiques animales ou végétales. Les levures sont employées pour la fabrication du vin, de la bière, des alcools industriels, des pâtes levées et d'antibiotiques. Les levures sont des micro-organismes eucaryotes, ainsi possèdent-elles les caractéristiques structurelles propres à ce type cellulaire et d'autres plus spécifiques aux levures elles-mêmes. Il existe de nombreuses formes de levures qui ont des caractéristiques différentes. Par exemple les « levures de boulanger » (*Saccharomyces cerevisiae*) ont un processus de division complexe et assez différent des autres cellules : Elles forment des bourgeonnements. D'autres types de levures comme par exemple celles de souche Ade2 (souvent utilisées dans des expériences notamment en génétique) se divisent normalement. De plus les levures possèdent deux principaux métabolismes :

- La respiration classique en aérobiose qui transforme glucose et oxygène en énergie, eau et CO₂



- La fermentation alcoolique qui permet de faire de l'énergie sans oxygène en produisant de l'éthanol et du CO₂

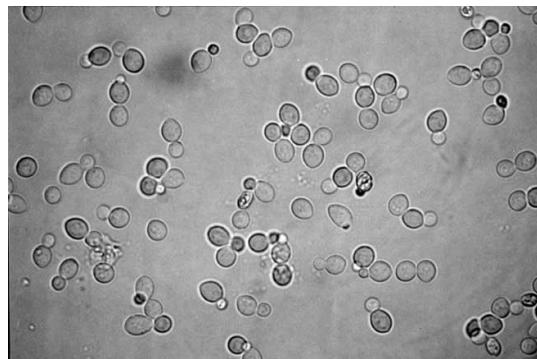
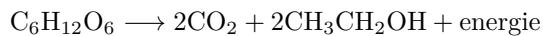


FIGURE E.1 – *Saccharomyces cerevisiae*, levure de boulanger

E.2 L'expérience souche Ade2 et UV

La souche Ade2 des levures est une souche qui possède la caractéristique d'avoir un gène qui lorsqu'il est muté, modifie la couleur de la cellule. Nous nous sommes servis de ce taux de mutation visibles, pour approximer le taux de mutation absolu d'une levure.

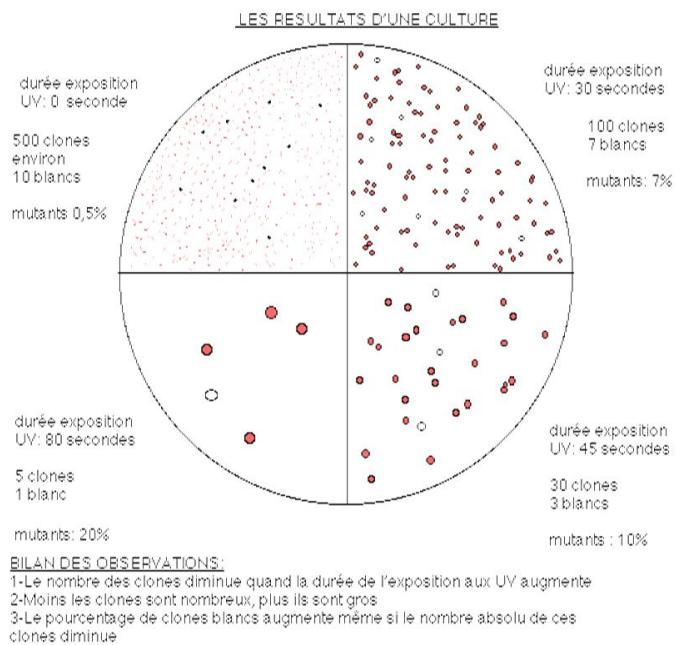


FIGURE E.2 – Résultats de l'expérience

Cette expérience montre deux effets des UV sur les cellules de levures :

Les mutations létales engendrées par les ultra-violets modifient les gènes essentiels au fonctionnement général de la cellule et finissent par la tuer. Ensuite certaines de ces levures qui ont résisté aux rayons ont mutées et cette même mutation engendre une modification de la couleur de ces individus. Grâce à cette expérience on peut déterminer le taux de mutation et de mortalité de ces levures en cas d'exposition au l'UV...

Les observations de cette expérience concernant les mutations sont consignées dans ce tableau :

Durée d'exposition aux UV (secondes)	Nombre de clones par cm ²	Nombre total de clones blancs
0	200	20
30	22	3
45	12	2
80	7	1

Ces informations on été tirées de Wikipédia, article sur les levures/Mutagenèse chez une souche de levure rose ADE 2 et sur le site <http://www.snv.jussieu.fr/vie/informations/2/22ADN/mutagenese/levureADE/levureade.htm> comme spécifié dans la sitographie, annexe F.

Annexe F

Sitographie et Bibliographie

Programmation

<http://fr.wikipedia.org> : définitions formelles de certains points et détails historiques précis qui ont permis d'améliorer les annexes

<http://www.siteduzero.com/> : site qui comprend un tas de tutoriels divers sur tous types de langages de programmation, utilisé pour certaines des illustrations pédagogiques des annexes

<http://www.developpez.com/> : site qui comprend un tas de tutoriels sur des aspects complexes de la programmation et qui nous a orienté dans la rédaction des annexes

Biologie

- <http://www.snv.jussieu.fr/vie/informations/2/22ADN/mutagenese/levureADE/levureade.html>] article sur les levures mutagénèse chez une souche de levure rose ADE2.
- <http://fr.wikipedia.org/wiki/Levure> définitions de wikipédia qui ont orienté nos recherches
- Manuel Scolaire de SVT 1ère et 2nd, Bordas, 2011 compte rendus des activités pratiques sur les levures, en particulier sur la génétique.
- http://biologie.univ-mrs.fr/upload/p235/UE_levurepart2.pdf

Fiches de Synthèse Personnelles

Sévigné Alfred

Lors de la composition des équipes à la première séance, je n'avais pas vraiment choisi. J'ai pris celle où il restait de la place. C'est-à-dire avec Aliaume et Yann. Nous n'avions pas vraiment les mêmes goûts. Nous avons donc chacun exposés nos idées et nous sommes tombés sur plusieurs choix de sujets. Nous avons commencés avec une première problématique après une ou deux séances. Il était question de m'apprendre à programmer et de faire un programme. Il ne me serait jamais venu à l'idée d'apprendre à programmer sans ce TPE. Mais mes nombreuses absences et notre difficulté à trouver des horaires pour se regrouper perturber le travail. De plus je ne comprenais pas exactement ce que m'expliquais Aliaume quant au fonctionnement du logiciel de programmation Python. Nous abandonnâmes donc la première problématique pour une autre conseillé par les professeurs et par nous-mêmes. La deuxième, c'était seulement le programme et dire les problèmes que nous avions rencontrés. Seulement, il restait déjà peu de temps mais le programme avait bien avancé. Ils me donnèrent alors la responsabilité du design dont nous discutions à chaque entrevue ou en parlant par e-mail. Le programme était fini, il fallait s'attaquer à la rédaction du compte-rendu. Ce fut le plus dur car il a fallu accélérer le mouvement. Mais on y est quand même arrivé. En conclusion, je pourrais dire qu'avec la première problématique, j'exprimais peu d'intérêt. Mais à la deuxième en tant que designer, je me sentis un peu plus concerné. Je regrette que le travail fut plutôt laborieux de mon côté car je n'y connaissais rien du tout dans ce thème. Surtout quand Aliaume et Yan discutait du programme, et à certains moments, j'avais du mal à les suivre. Ce fut une bonne expérience qui m'a appris quelques petits trucs sur la programmation. J'ai pu aussi reprendre le cours sur les cellules où je n'avais pas été exemplaire pendant mes recherches et même en savoir plus.

Desmarais Yann

Pour débuter le groupe s'est formé plus ou moins par défaut, mais cela ne posa pas de problèmes majeurs et nous nous entendions relativement bien. Les plus grosses difficultés du TPE furent le choix du sujet et de la problématique... Nous nous sommes décidés pour nous orienter vers le numérique et l'informatique, encouragé par les professeurs encadrants : Le sujet nous plaisait et Aliaume possédait un certains nombres de connaissances en programmation. On décida en premier lieu d'orienter la problématique vers une approche pédagogique de la programmation : Aliaume était le « professeur » et nous étions les « élèves ». On remarqua que ce n'était pas l'angle approprié, de par les rôles trop différents sans lesquels chacun de nous aurait dû intervenir et aussi à cause de nombreuses autres difficultés rencontrées. Ainsi nous avons changé la problématique qui est devenue : « Comment résoudre un problème scientifique à l'aide de la programmation ». Aliaume s'est mis à développer un programme et s'occuper de la facette algorithmique et à s'occuper de la facette algorithmique du projet, Alfred de la simulation et de l'aspect de l'interface et moi au données scientifique nécessaires. Notre problèmes était de chercher à savoir en combien de temps se développe une colonie de levures et à y étudier des mutations dues aux UV. J'ai fourni des informations tirées d'internet et de recherche tiré d'expériences réalisée sur des souches de levures Ade2. Ensuite nous avons-nous même tenté l'expérience ce qui nous a permis de manipuler et de nous servir des laboratoires. Ce fut, en fin de compte, une expérience intéressante qui nous a montré la difficulté du travail de groupe et ouvert les porte du monde de la programmation et de la biologie.

Lopez Aliaume

Au départ je voulais faire mon TPE avec Yann, mais je n'avais encore aucune idée de sujet à traiter. Nous avons par la suite eu l'obligation de former des groupes de trois, et nous avons donc intégré Alfred au groupe, sans qu'aucun de nous n'ai d'idée, même vague de ce que l'on pouvait faire.

Notre plus gros problème lors de ce TPE fût de trouver un sujet qui nous intéressent tous, qui soit à la portée de tous, et en général, d'en proposer un tout court. Durant les premiers mois, j'ai vraiment eu la sensation que le groupe tournait à vide. Durant quelques séances, l'objectif était uniquement de définir une problématique.

Finalement nous nous sommes orientés vers la programmation, sous une de mes impulsions, car je pensais que ma « compétence » dans le domaine apporterais un peu plus de dynamique au TPE. Malheureusement, la problématique était beaucoup trop portée sur l'aspect didactique de la programmation, et par conséquent, le TPE requérait des séances plénières pour avancer sérieusement. Séances que nous n'étions pas en mesure de fournir, car nous avons toujours eu du mal à nous retrouver régulièrement, pour diverses raisons. Le travail manquait de motivation et de dynamique. L'idée que je me faisais du TPE s'est effritée au contact de la réalité, ce que j'imaginais comme une recherche de la connaissance en groupe et une sorte de solidarité face à des problèmes intéressant était juste de la bureaucratie, chacun tentant de justifier son travail, moi y compris, et chacun ne faisant rien car la problématique était très peu sujette au travail en réalité.

Heureusement Yann a su tenir tête à Madame Pivette pour changer la problématique en cours de route, car le TPE n'avait presque pas avancé. À partir de cette modification significative, qui sans changer le sujet, changeait grandement l'angle d'approche de celui-ci, nous avons pu créer une certaine dynamique de groupe, à partir d'une dynamique individuelle. En effet cette nouvelle problématique permettait plus aisément le travail individuel et la mise en commun ultérieure, ce qui a permis des progrès rapides et des séances plénières intéressantes.

Durant ce travail j'ai eu l'occasion de confronter ma façon de penser avec d'autres personnes, qui elles ne programment pas, et qui de manière naïve, ont parfois apporté des solutions très efficaces. De plus ce fut pour moi l'occasion de programmer pour un projet véritable. On m'a fourni un cahier des charges, on vérifiait l'avancée, c'était une ambiance de production, bien qu'un peu détendue tout de même. La seule chose dommage est que l'on ai pas eu plus de temps pour effectuer des recherches et des expériences complémentaires afin d'améliorer la simulation. Il a été vraiment déplaisant de voir notre seule expérience en Biologie ne pas apporter de résultats utilisables.

Au final je trouve que ce TPE a été une assez bonne expérience, mais la confusion et le manque de but au départ a vraiment été pénible.