Le Faucheux Antoine
Mielle Malcolm

Rob3

Projet de C

Tron

Sommaire

1) *Planning et objectifs:*

Méthode travail.

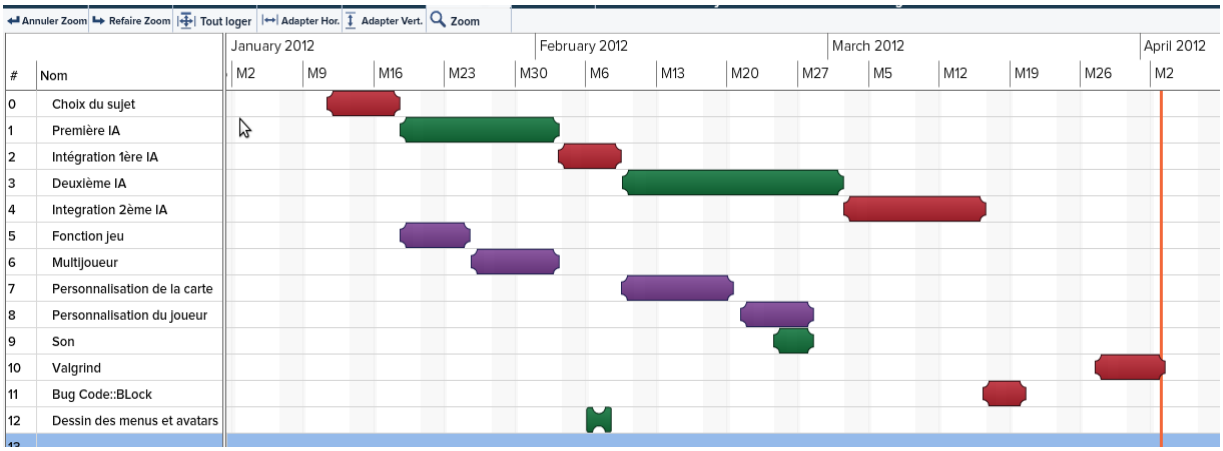
Nous avons choisi de travailler en parallèle même si ce mode de travail était toute fois déconseillé aux gens sans forte expérience de la programmation. Même si nous n'étions pas à proprement parler des « experts » du C, nous avions déjà travaillé ensemble lors du premier semestre à la création d'un jeu (toute fois nettement plus simple que celui ci). Ce projet étant dans la même veine, nous avions une idée précise des objectifs à atteindre et de la manière de les réaliser. Nous avons alors appliqués le même schéma de répartition des taches et de fonctionnement qu'auparavant, cette méthode s'étant avérée payante au premier semestre. Après tout, « on ne change pas une équipe qui gagne ».

De plus, nous nous sommes consciemment donné des objectifs élevés. Partir d'un schéma de travail que nous connaissions et que nous savions efficace nous paraissait alors une obligation dans le but de les atteindre et le travail parallèle, plus rapide, c'est véritablement imposé.

Cependant, travail parallèle ne signifie pas travail séparé. Ainsi, sur le diagramme de Gant (présenté plus loin dans le document) sont représentées les activités dominantes que nous avons fait durant toute la durée du projet. En violet celles d'Antoine Le Faucheux et en vert celles de Malcolm Mielle. Ces couleurs sont représentative de celui qui était le plus impliqué dans l'activité mais aucune d'entre elle n'a vraiment été réalisé seul. Chacun codait la partie dont il était responsable mais la résolution de la majorité des bugs et l'intégration des programmes entre eux passaient par un travail commun et l'avantage de pouvoir avoir un œil extérieur sur son travail. Ce n'est donc pas seul que nous travaillions mais en groupe. Les activités rouges, sont celle où le travail était entièrement fait en commun.

Dans notre apprentissage de la SDL et des MakeFile nous avons entre autre beaucoup été aidé par le site du zéro.

Diagramme de Gant



Ceci est notre diagramme de Gant final, après les modifications que nous lui avons apporté au vu des difficultés que nous rencontrions. Comme on peut le voir, nous avons marché en parallèle surtout au début du projet et avons concentré nos efforts en commun à la fin, en vue de régler tous les problèmes de mémoires inhérents à notre programme.

2) *Critères de succès*

La politique de test que nous avons choisi était différente selon que l'on travaillait sur l'IA ou sur la partie graphique.

Partie IA

La politique de test sur la partie intelligence artificielle est basée sur une série de test unitaire. Nous placions les robots dans une des situations qu'ils étaient susceptibles de rencontrer et nous observions sa réponse tout en connaissant celle qu'ils devaient nous donner (puisque nous savons à quoi ressemble l'environnement, nous pouvons prévoir ses réactions).

Avant tout, il faut savoir que les robots se contentent de renvoyer la prochaine position sur laquelle il vont aller. Ils ne se déplacent pas, et ne touchent en rien à la grille originelle (celle qui constitue le « vrai » jeu) car ces actions sont dirigées dans la partie graphique.

Le but du test unitaire est donc uniquement de vérifier que le choix du robot est un choix logique et en adéquation avec sa manière de réfléchir.

Les quelques situations communes que nous avons testés sont :

- Le robot n'a aucun obstacles et aucun ennemis.
- Le robot est sur une carte vide avec des murs sur le chemin.
- Le robot est entouré de mur et n'a aucune autre solution que de mourir (il renvoie alors un NULL).
- Le robot n'a qu'une seule et unique solution.
- Le robot a deux solutions mais une est meilleure que l'autre (il possède plus d'espace libre après). Il doit donc choisir la meilleure solution.
- Le robot est sur une position où il n'a qu'un seul mur à côté de lui.
- Le robot possède jusqu'à 5 ennemis et a toute les possibilité de mouvement.

Ceci est, bien entendu, un ensemble de tests génériques et d'autres tests unitaires ont été faits pour chacun des deux bots.

Le deuxième critère de fonctionnalité du robot est un test en « live » dans le programme. Si le robot était capable de tenir un match sans mourir de manière trop stupide, sans faire de mouvement impossible et sans faire de segmentation fault, il était alors considéré comme « valide ». Les nombreux problèmes rencontrés lors des test de fonctionnement dans le programme final ont par ailleurs permis de définir de ***nouveaux tests unitaires*** afin de cerner les situations particulière où le robot pouvait planter. C'est pourquoi le test unitaire - Le Robot est dans le coin en bas à droite de la carte - existe pour le BotAlea. Le robot était en effet capable de sortir de la carte par le bas ou la droite, erreur dû à un mauvaise conversion lors de changement de tableau. Un simple test unitaire nous a permis de régler ce problème en quelques minutes.

Ainsi nous pouvons penser avoir testé toutes les possibilités d'action des bots et toutes les bugs ou situation particulière qu'ils pouvaient rencontré grâce à l’application de cette méthode de vérification croisée.

Partie Graphique.

Pour juger la partie graphique, nous nous sommes basés sur l'expérience. Si nous affichions l'image que nous voulions, si les joueurs réagissaient comme on le souhaitait aux demandes, si toutes les fonctionnalités présentent sur la fenêtre étaient « efficaces » et si en effectuant des opérations non conforme à ce qui était attendu et demandé (comme donner les même touches de contrôle à deux joueurs, cliquer n'importe où sur l'écran ou appuyer sur d'autres touches que celles demandées...) le jeu ne plantait pas ou ne présentait pas de bug, alors le programme était considéré comme valide.

Nous avons réfléchie sur un autre critère de validité qui n'est pas a proprement parlé « graphique » : en effet, il s'agit de la saisie sécurisé. C'est une chose tellement simple que de rentrer un nombre la ou le programme demande une lettre que nous avons travaillé sur un moyen d'obliger l'utilisateur a ne pas faire planter le programme. Pour cela, nous avons choisi de restreindre le nombre de touche utilisable par l'utilisateur, ici, lorsque le programme demande une lettre, seules les « touches lettres » sont activées.

3) Les imprévus techniques :

Intégration de l'IA :

Nous avons eu de nombreux problèmes lors de l'intégration de la première IA. Notamment à cause du fait que l'IA elle même comprenait un certain nombre de bug que l'utilisation en temps réel dans le jeu permettait de détecter.

Nous avons adopté une méthode de test unitaire et l'utilisation d'un MakeFile pour tester le bon fonctionnement de notre programme et trouver ses problèmes. De cette manière chaque « groupe de fonction » est isolée des autres dans un fichier C, le programme est plus simple a lire et a débugger.

Ainsi, pour résoudre les problèmes que nous rencontrions, nous avons simplement isoler les cas où le robot faisait planter le jeu et créer une sorte de représentation miniature de la situation, que nous étudions directement dans la console.

Les principaux cas testés sont : le robot a toutes les position autour de lui libre, il en une ou plus de bloqué, il ne peut pas atteindre la position qu'il cherche et il n'a pas de solution car il est encerclé et a perdu.

Par exemple, à un moment le robot plantait en s'approchant d'un mur ou lorsqu'il perdait. Nous avons créer un programme qui plaçait directement le robot contre un mur, dans une petite carte, sans possibilité de mouvement, pour étudier ce qu'il nous renvoyait. Nous avons ainsi trouvé que le bug provenait du fait que dans ces cas, il renvoyait un NULL qui nous avions oublié de gérer par le programme d'affichage en SDL qui créait alors une **segmentation fault**.

De plus, l'intégration des IA a occupé une part significative de notre temps car, en plus de problèmes « de code » détaillés précédemment, nous ajouterons qu'il fallait réfléchir à une IA aux réactions intéressantes. Or, la première IA construite n'était absolument pas efficace dans le type de situation du jeu (où seulement 2 ou 3 joueurs au maximum s'affrontent). Elle avait pour raisonnement de chercher le « centre de gravité » de la carte (les murs représentant des forces) et de s'y rendre par le plus court chemin grâce à un algorithme Astar.. Ainsi, elle restait loin du danger et des adversaires, tout en posant des murs de manière équitable en eux. Bien que cette technique soit intéressante dans l'idée, en pratique, avec un faible nombre de joueurs, elle s'avère mortelle pour l'IA. En effet, le centre de gravité ne se déplaçant pas assez, le robot avait tendance à stagner en un point et à se tuer tout seul. C'est pourquoi il a fallu entièrement repenser la deuxième IA.

La seconde est bien plus subtile : elle prend le meilleur chemin autour d'elle lorsqu'elle ne vous voit pas, c'est à dire le chemin entouré par le moins de case. Mais dès que vous rentrez dans son champ de vision, elle entre en action. Elle calcule, alors que vous êtes toujours dans son champ de vision, la meilleur position que peut occuper chacun de ses adversaires au prochain tour, à savoir la position à partir de laquelle chacun d'eux peut atteindre le plus de cases en premier. Ensuite, elle déplace virtuellement ses adversaire à cette meilleure position et calcule alors la meilleure position pour elle. Ainsi, elle a trouvé la position qui statistiquement lui donnera ensuite le plus d'opportunité et réitère le raisonnement tant que vous êtes dans son champ de vision.

Cette IA, est donc bien plus réactive et même si son intégration a finalement demandé deux fois plus de temps que pour la première, le jeu en valait la chandelle car elle est un meilleur adversaire et offre un type de jeu supplémentaire.

Code::Block :

Séance du 20 mars 2012 :

Le plus gros imprévu que nous avons eu à gérer a été le plantage de la compilation sur Code::Block deux semaines avant de rendre le projet. Alors que nous pouvions lancer le programme, jouer et modifier les personnages (en sommes un programme déjà bien avancé), nous ne pouvions du jour au lendemain, plus le compiler. Le compilateur nous affichait un nombre élevé d'erreur de type : **error : stray '\351' in program** . Nous n'avons strictement aucune idée d'où se problème pourrait venir vu que l'ordinateur était éteint entre le moment où il marchait et le moment où il plantait et que nous n'avons copié aucune partie de code à partir d'un outil de traitement de texte ou autre. Donc pas de changement de UT-8 vers autre chose, ni d'apparition de caractères spéciaux et d'espaces insécables entre autre (cela faisait parti des possibles causes de ce bug que nous avons trouvé sur internet).

Pour résoudre ce problème, nous n'avons trouvé d'autre solution que de conserver seulement le code « pur », tout désinstaller (l'IDE et tous les liens tel que SDL, FMOD, TTS) et tout réinstaller en créant un nouveau projet. Cette solution était extrêmement frustrante car nous avions l'impression de retourner en arrière de 4 à 5 semaines. Quand nous avons réussi à relancer le jeu après avoir reformaté tous les liens, nous n'arrivions plus à avoir la musique par exemple. Alors qu'elle marchait depuis 4 semaines, d'un seul coup, elle ne fonctionnait plus.

Alors que nous devions finir le projet lors de cette séance, finaliser le code, intégrer le deuxième robot et faire un de jeu avec plusieurs AI, nous avons passé notre séance à tenter de corriger le problème et à commencer l'écriture du rapport.

La perte de temps dû à ce problème incompréhensible de Code::Block s'élève a près de 8 h de travail. Par ailleurs, le dysfonctionnement de la bibliothèque FMOD reste toujours un mystère pour nous. Tout le code « avant bug » est toujours présent en commentaire (notamment toutes les en-têtes des fonctions).

4) *Les imprévus temporels :*

Valgrind :

Il nous a été demandé d'utiliser Valgrind afin d'analyser plus finement notre programme. Valgrind est un outil permettant de trouver des erreurs pouvant passer inaperçues par inadvertance comme le fait de free une structure par exemple.

Ainsi, au début, nous avions d'énormes fuites de mémoires notamment à cause des fonctions récursives utilisées par les robots et quelques free de Surface SDL qui n'étaient pas bien effectués. Et, alors que nous pensions que le passage sous Valgrind ne serait l'affaire que de quelques heures, nous nous sommes retrouvés à passer plusieurs jours dessus à régler ces fuites de mémoire et à lire des tutoriels afin de comprendre toutes les fonctionnalités de Valgrind. Nous avons ainsi eu à utiliser des exécutables compilés avec l'option de débogage et l'option « --leak-check=full » abondamment avant d'arriver à un résultat.

Il est aussi des fuites que nous n'avons pu régler. En effet, dans la fonction SDL_Init (fonction que nous n'avons pas programmé nous-même mais qui nous sert à initialiser l'affichage graphique) il existe une fuite de mémoire interne, signalée par Valgrind et que nous ne pouvons régler. De plus, le temps nous ayant fait défaut pour cette fois, il est d'autres fuites dont nous n'avons pas eu le temps de trouvé la solution. Toutes les fuites mémoires ne sont donc pas enlevées de notre programme mais, du moins, les plus importantes ont disparues.

Nous avons donc eu de gros décalage par rapport à nos prévisions sur la fin de notre projet. Pour palier à ce manque de temps, nous avons abandonné l'idée (déjà juger improbable au début) de créer un mode de jeu en réseau et nous nous sommes concentré sur le code. Ainsi les robots sont parfaitement « propre » et ne comporte aucune fuite de mémoire. Par contre, la SDL nous a posé beaucoup plus de problème (du fait que les fuites nous semblait venir d'endroit que nous ne pouvions contrôler (comme SDL_Init) ou alors d'objets que l'on ne pouvait pas free et que l'on pensait avoir free à un moment ou un autre du programme.

5) *Conclusion :*

Au cours de ce projet, nous avons pu évaluer notre capacité à travailler en équipe ainsi qu'améliorer la communication dans le but de résoudre les problèmes rencontrés. Nous nous étions fixé des objectifs très élevés par rapport a notre niveau passé de connaissance de programmation dans l'unique but de nous motivé à apprendre d'avantage.

Les nombreux problèmes rencontrés au cours de l'écriture du code, de l'assemblage des différentes partie des codes, de la connaissance des IDE, nous ont forcé à aller nous auto instruire sur des forums anglais, dans des lires de programmation afin de débugger notre programme. De plus l'utilisation d'outil comme Valgrind nous a permis d'acquérir de nouvelles connaissances et compétences, faisant de nous des programmeurs plus « professionnels ».