PROJET SIAM : COMPTE-RENDU

Sommaire

Chapître 1 : Introduction

Dans le but de mettre en œuvre de bonnes pratiques de programmation et un algorithme de code structuré, nous avons réalisé le codage d'un jeu de Siam pour le module CSC2. Il est important de préciser que nous ne sommes pas partis de zéro, la structure globale du logiciel et un certain nombre de fonctions avaient été réalisés au préalable.

Afin de coder nos fonctions nous avons fonctionné avec une nouvelle méthode de programmation, la méthode dite « par contrat ». Cette méthode consiste à avoir la description des fonctions, ce qu'elles nécessitent en paramètres et ce qu'elles garantissent en retour, dans le fichier « .h ». Notre travail a donc été de reprendre ces contrats de fonction et de coder le corps de ces fonctions dans nos fichiers « .c », ou inversement lorsque le corps de la fonction était donné il nous fallait remplir le contrat de la fonction.

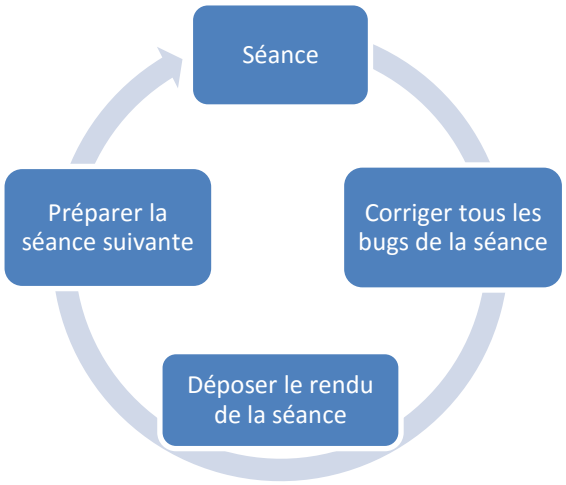
Au cours de ce projet, nous avons fonctionné en binôme. Le travail de codage a été réalisé par étape, de niveau de difficulté croissant. Les premières séances étaient principalement des séances de création de fonctions bas niveau, permettant des vérifications d'intégrité ou des fonctions de test pour vérifier que nos fonctions faisaient ce que l'on attendait. Jusqu'aux dernières séances ou nos codions des fonctions de haut niveau permettant une interface avec l'utilisateur, concrètement ces fonctions permettent à l'utilisateur de jouer.

Nous allons donc voir comment nous avons réalisé ce projet avec notre organisation du travail au sein du binôme, comment nous avons réalisé nos vérifications tout au long du développement du jeu, comment notre fonction de poussée à été réalisé et enfin les différentes alternatives que nous avons rencontré.

Chapître 2 : Organisation générale de notre projet

a. Méthode de travail en binôme

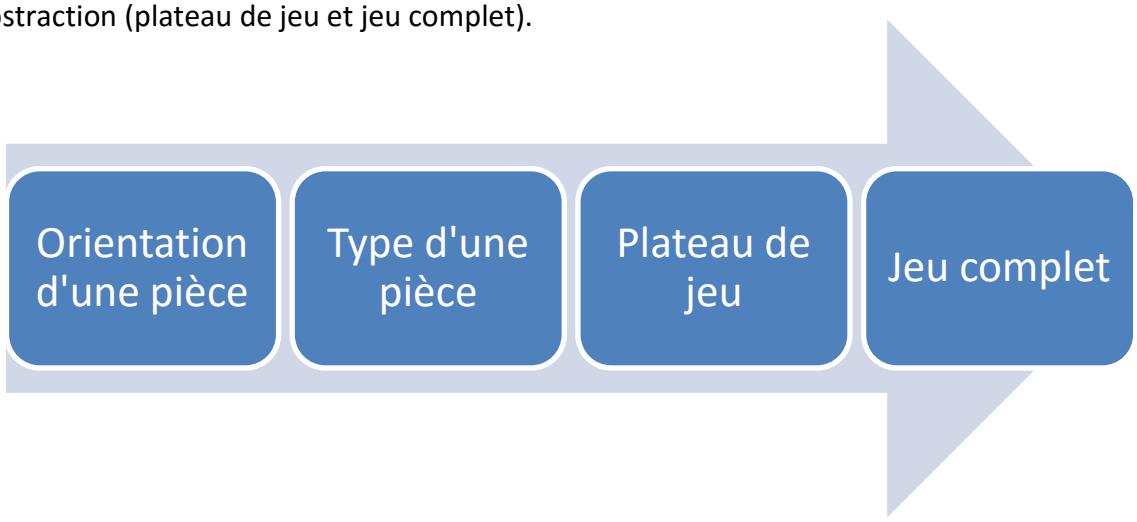
Les binômes de projet étant imposés dès le début, il est alors important de trouver comment s'organiser pour être le plus efficace possible. Comme nous ne nous connaissions pas vraiment avant le début de ce projet (nous venons d'univers différents : Valérian de la prépa intégrée et Samy de DUT GEII, nous avons donc du cerner les points forts de chacun et les mettre en avant. Valérian était déjà familiarisé avec l'environnement de travail Linux depuis plusieurs années, il a donc principalement réalisé les manipulations de gestion du projet dans un premier temps (ouverture des fichiers du projet, utilisation du terminal, exécution du script de vérification, dépôt du rendu…). Peu après, Samy a lui aussi pu mettre en pratique ces manipulations. Samy et Valérian ont tous deux réalisé un travail de réflexion conséquent sur l'algorithme et un codage soutenu, non seulement en classe mais en dehors des heures de cours (notamment pour la poussée). Nous avons donc ainsi travaillé en binôme sur le même code ensemble.
Cependant, trouver du temps libre pour finir le codage de la séance précédente et préparer la suivante est une tâche délicate, surtout quand on est en 3ème année à CPE Lyon. En effet, nous disposons d'une masse de travail très importante et avons peu d'heures de libres à l'école pour nous mettre à jour dans chaque module… De plus, Samy n'avait pas Linux sur son ordinateur personnel pendant les premières séances, il était donc impératif de travailler sur les ordinateurs de l'école. Nous avons réussi à trouver des heures de libres le jeudi après-midi après nos séances de soutien en Maths-Algèbre. Au cours de ces créneaux, nous n'avions pour la plupart du temps pas réussi à finir le travail prévu dans la séance précédente car notre programme rencontrait un bug. Ce bug était dû à une erreur de programmation, la plupart du temps représentée par une assertion lors de l'exécution d'une séquence de tests de notre programme. Pour corriger ces assertions, nous avons attaqué le problème à la source, en simulant le déroulement du programme jusqu'à l'entrée dans la fonction bloquante et apporté des correctifs au programme.



Nous avons également rencontré des erreurs de segmentation. C'est d'ailleurs ces erreurs qui nous ont pris le plus de temps pour les corriger car nous ne maitrisions pas vraiment les logiciels pour les débuguer au début. Par exemple, nous avons passé plus de 6 heures à essayer de résoudre une erreur de segmentation sans succès. Celle-ci était dû à une assertion d'une fonction « « plateau_etre_integre » qui dépendait d'une assertion d'une autre fonction, ce qui rebouclait le programme indéfiniment sans que nous puissions nous en apercevoir. C'est avec l'aide d'un professeur, de la démonstration des lignes de commandes « valgrind » et avec l'utilisation du logiciel « kdgb » que nous avons pu comprendre cette erreur.
Les vacances de Noël ont été une période décisive dans l'avancement de notre projet. Pour la première fois, nous disposions d'assez de temps libre et avons pu nous investir considérablement chez nous pour réaliser les fonctions de poussée (détaillées ci-après dans le rapport). En ce qui concerne le logiciel utilisé, nous avons utlisé « Kate » dans un premier temps puis avons découvert « QtCreator » par la suite et l'avons adopté.

b. Organisation du code, des fonctionnalités et vérification de l'intégrité

Nous avons codé les fonctionnalités du programme en suivant une analyse « top-down », à savoir les hauts niveaux d'abstraction en premier (orientation et type d'une pièce), puis les bas niveaux d'abstraction (plateau de jeu et jeu complet).



Les fonctionnalités de basse abstraction imposent la gestion des tours de jeu, à savoir associer à chaque commande du joueur une action. Cette action est pour la plupart du temps une introduction de pièce ou un déplacement. Pour chaque fonctionnalité, nous avons fait en sorte de respecter le contrat associé à la fonction et avons ainsi vérifié l'intégrité permanente du jeu. Pour se faire, nous avons mis en place des contrôles des paramètres d'entrée et de sortie. Pour les paramètres d'entrées, si ceux-ci dépendaient de l'utilisateur (exemple : saisie de coordonnées),

nous avons utilisé des « if ». Pour les paramètres ne nécessitant pas de saisie du joueur, nous avons utilisé des assertions, évoquées dans la partie a). Par exemple, lors de passage de pointeur en paramètre, nous avons constamment vérifié si celui-ci n'était pas nul. C'est une bonne pratique de programmation à adopter car ce genre d'erreur peut être très difficile à débugger. Nous avons également utilisé des fonctions de vérification d'intégrité sur les entités manipulées dans les assertions (plateau_etre_integre, piece_etre_integre) et dans des ifs (coordonnees_etre_integre, orientation_etre_integre…). Nous avons testés les fonctions au fur et à mesure de l'avancement du projet, manuellement dans un premier temps. Cependant, pour anticiper un maximum de cas, réaliser une séquence de tests s'est avéré bien plus pratique (pour plus d'informations, voir la partie 4 « Tests »).

Chapître 3 : Déplacement des pièces

a. Déplacement classique et introduction

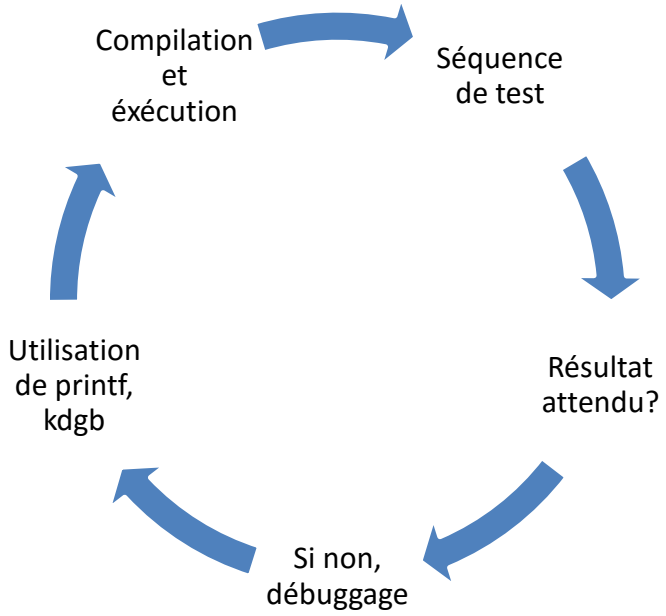
La gestion des coups de jeu se fait depuis l'API : on associe à une action un coup valide ou non et on appelle la fonction associée. Nous avons dans un premier temps eu à gérer les cas de déplacements classiques, autrement le déplacement et d'introduction d'une pièce sur une case vide. Pour étudier la case ciblée, nous avons utilisé la fonction « plateau_obtenir_piece_info ». Dans tous nos cas de déplacement, nous avons opté pour l'utilisation de « switch case ». Nous sommes conscients que cette solution est loin d'être la solution recommandée et la plus succincte mais c'est celle que nous avons jugé la plus souple et d'utilisation et compréhensible.
Dans le cas de l'introduction ou du déplacement d'une nouvelle pièce, nous exécutons au préalable la fonction « plateau_modification_introduire_piece_etre_possible » ou « plateau_modification_deplacer_piece_etre_possible ». Ces fonctions vérifient les conditions d'intégrité du plateau, étudie le type de la case ciblée et renvoie 1 ou 0 selon la possibilité du mouvement. Elles vérifient également d'autres cas à prendre compte, à savoir si les coordonnées sont bien dans le plateau, si le type de la piece à déplacer est bien un animal, si la direction du déplacement et de l'orientation de fin sont bien intègres… Si le mouvement est possible, on utilise la fonction « piece_definir » pour une introduction. Dans le cas d'un déplacement, on recopie la pièce déplacée sur la case suivante puis on remplace la pièce initiale par une case vide sur la case précédente. Lors de notre réflexion à l'algorithme, le problème était de savoir quelle était cette « case suivante », c'est pourquoi nous avons utilisé des « switch case » en créant de nouvelles coordonnées d'une case en fonction de la direction du déplacement.

b. Déplacement par poussée

Jusqu'à présent les tentatives de déplacement ou d'introduction d'une pièce sur une case non-vide étaient considérées comme invalides. Lors de la mise en place de la nouvelle règle de poussée, il a fallu apporter beaucoup de correctifs à notre structure de programme. Dans les fonctions « plateau_modification_introduire_piece_etre_possible » et « plateau_modification_deplacer_piece_etre_possible », nous avons appelé la fonction « poussee_etre_valide » et renvoyé 0 ou 1 en fonction du retour. La fonction « poussee_etre_valide » sollicite 4 nouvelles fonctions de calcul de forces que nous avons créées : « forces_poussee_haut », « forces_poussee_bas », « forces_poussee_gauche », « forces_pousee_droite ». Ces fonctions sont appelées en fonction de la valeur de l'orientation et les calculs de force reposent sur des coefficients de poussée : 0 pour une case vide, -9 pour un rocher, 10 pour un animal de même sens, -10 pour un animal en contre-sens et 0 pour un animal de sens différent. La première étape consiste à calculer un seuil de poussée, correspondant à la première case de la ligne/colonne la plus proche du joueur où il y a une case vide. C'est jusqu'à ce seuil que sera effectuée la poussée. On parcourt ensuite les cases de la ligne/colonne jusqu'à la case «seuil» et on incrémente la somme des forces en fonction des informations de la case (utilisation de la fonction « plateau_obtenir_piece_info ») à l'aide des coefficients précédents. Si la somme des forces devient négative pendant le parcours, c'est qu'une pièce bloquante a été rencontrée, on prendra alors soin de rendre impossible la poussée, quelques soit les pièces qui suivent. On valide alors ou non la poussée si la somme des forces est supérieure à 0. Après avoir vérifié d'autres conditions, comme par exemple un éventuel changement d'orientation pendant la poussée, on peut maintenant réaliser cette poussée. Cette fonction aura des comportements différents en fonction de l'orientation du déplacement, comme dans les fonctions précédentes. On commence alors par recopier sur la case « seuil » la pièce précédente, remplacer la case de la pièce précédente par une case vide et ainsi de suite, jusqu'à la pièce à l'origine de la poussée. On utilise alors les fonctions « piece_definir » et « piece_definir_case_vide ». La poussée par introduction a également été implantée, nous avons pour cela modifié la fonction « plateau_modification_introduire_piece_etre_possible ». Si le type de la case sur laquelle on veut introduire une pièce n'est pas une case vide, on teste si la poussée est possible. Cette poussée est possible ou non en vérifiant l'orientation de la pièce qu'on pose et en calculant la somme des forces de la ligne/colonne. Si l'introduction est possible, on appelle dans « plateau_modification_introduire_piece » la fonction « poussee_realiser » puis « piece_definir » : on aura alors poussé la pièce occupant la case puis introduit la piece du joueur.

Chapître 4 : Tests de notre projet

Pour tester notre code, nous avions pour l'habitude de compiler notre projet, de corriger les éventuelles erreurs puis de l'exécuter et de réaliser des séquences de tests manuels. Par exemple, pour tester les cas de poussée, nous avons repris les 12 cas du cahier des charges et simulé ces cas en ligne de commandes une par une. C'est grâce à ces tests que nous avons pu comprendre où étaient nos erreurs puis comprendre nos erreurs. Lors du débuggage du programme, nous avons utilisé énormément de printf pour afficher des valeurs de variables, de puts pour afficher le déroulement du programme (par exemple, si une fonction rentrait dans une boucle ou non, quelles étaient les valeurs des coordonnées de la fonction de poussée, etc).



Cependant après avoir pris du recul, nos séquences de tests étaient longues et ne couvraient pas vraiment tous les cas de figure possibles. Ce n'est que bien plus tard que nous avons découvert une nouvelles fonctionnalité qui nous a permis de gagner beaucoup de temps et de rigueur : les tests unitaires puis les tests d'intégration. Il suffit de spécifier un plateau rempli initialement, de saisir une suite de commandes et de spécifier la sortie attendue : on peut alors comparer si la sortie obtenue correspond à nos attentes et valider ou non par conséquent le code. Nous avons réalisé plusieurs tests d'intégration, eux-mêmes rattachés à un script de tests central. C'est grâce à ces tests que nous avons réussi à identifier d'autres erreurs que nous n'aurions probablement pas trouvé par une méthode classique, comme par exemple une assertion lorsqu'une ligne entière est remplie de pièce et que, dans un cas valide de poussée, la pièce située au début de la ligne tente de réaliser une poussée.

Chapître 5 : Suppléments

Nous avons également apporté des améliorations au projet de base. Une gestion de fin de partie est requise, puisque le but du jeu est avant tout de gagner. Pour cela, nous avons créé de nouveaux fichiers « victoire_siam.c » et « victoire_siam.h ». Ces fichiers mettent en œuvre une fonction « victoire_siam_victoire » qui renvoie une structure de condition de victoire, constituée de l'activation de la victoire et du joueur associé.
Pour tester si une victoire a eu lieu, on scrutera le nombre de rocher à chaque tour. Si celui-ci est inférieur à 3, cela signifie qu'un rocher a été sorti du plateau et il faudra alors rechercher le joueur gagnant. Pour cela, on récupèrera les informations sur la pièce à l'origine de la poussée avec la fonction « plateau_obtenir_piece_info ». On stockera ces informations dans une structure « piece_poussante ». Ensuite, on récupèrera les informations sur la pièce en bordure ayant poussé le rocher à l'extérieur. Les coordonnées de cette pièce dépendront alors de l'orientation de la pièce à l'origine de la poussée. On mettra alors à joueur une nouvelle structure « piece_gagnte » grâce à la fonction « joueur_obtenir_numero_a_partir_animal ».
Cette fonction est appelée constamment dans l'API. Néanmoins, cette fonctionnalité n'est que partielle puisque une erreur de segmentation est renvoyée lors d'un cas de victoire. Nous pensons que l'erreur peut venir d'un dépassement de tableau lors d'un test sur une condition de cette fonctionnalité.

Chapître 6 : Conclusion

En conclusion, nous avons trouvé ce projet très intéressant, c'est d'ailleurs pourquoi nous l'avons considéré comme un véritable challenge. Nous avons avant tout pris plaisir à coder ce programme car c'est une mise en application très complète de nos connaissances. Nous avons tous les 2 des affinités pour l'informatique (et en voulons d'ailleurs en faire notre spécialité en 4ème année), c'est pourquoi il est important pour nous de travailler dans des gros projets comme celui-ci. C'est d'ailleurs ce genre de projet que nous rencontrerons par la suite dans notre futur métier (si nous poursuivons nos études et notre travail dans ce domaine), nous l'avons donc vécu comme une mise en situation. Ces 6 séances de projet nous ont montré la méthode à adopter pour de tels projets (organisation du l'équipe, analyse top-down, programmation par contrat) et les outils à utiliser (qtcreator, scripts de tests, logiciels de debug…). Il ne s'agit donc pas de tout oublier mais de se servir de ces nouveaux acquis pour tous nos travaux informatiques, aussi bien à CPE que dans notre future vie professionnelle.