Compte-rendu de Projet Final

Algorithme et Programmation : 2ème Semestre

UNIVERSITÉ PARIS-EST MARNE-LA-VALLEE

12 mai 2018

Créé par: GARCIA Ludovic, SAVANE Kévin

Introduction: Ariane et les Minotaures

Ariane et les Minotaures est un projet dont le but est de créer un labyrinthe composé :

- De murs verticaux ou horizontaux.
- De personnages : Ariane est un personnage contrôlé par le joueur, Thésée un personnage « objet » que l'on doit transporter jusqu'à la Porte.
- De Minotaures qui sont les principaux ennemis d'Ariane (et Thésée) qui ont pour rôle de foncer sur le joueur si il ne fait pas attention.
- D'une Porte qui est la sortie du labyrinthe.

Une des personnes de notre groupe, cultivé dans les arts du jeu-vidéo indépendant a remarqué certaines similitudes dans le concept du projet avec le jeu Crypt of the NecroDancer. C'est pourquoi il a pensé que l'idée de modifier certaines textures ou parties du gameplay en cohésion avec ce jeu aurait pu être intéressante. En effet une des améliorations du projet dans les consignes nous demande de redéfinir le jeu non plus comme un jeu en « tour par tour » mais comme un jeu en temps réel et de ce fait les Minotaures peuvent circuler librement vers le joueur tout comme dans le jeu précédemment énoncé.

On peut donc dire dans une première réflexion que ce projet nous inspirait des idées déjà présentes dans d'autres créations et nous permettait petit à petit d'entrevoir la façon dont nous allions opérer.

Dans un second temps, nous nous sommes demandé quelle serait la partie où nous aurions le plus de difficultés en analysant chaque tâche obligatoire du projet qui sont au nombre de trois.

Les points que nous pensions difficiles :

- ❖ Le Solveur

Les points où nous pensions être plus à l'aise :

- ✓ Création du labyrinthe
- ✓ Gestion des déplacements
- ✓ Condition de victoire ou de défaite
- ✓ Analyse syntaxique

Etablir les points difficiles dès le départ nous permettait de mieux comprendre le code que nous allions implémenter dans le programme pour surmonter cette épreuve.

Etape 1 : Représentation et chargement des niveaux

Après avoir vu l'énoncé de la tâche numéro un, nous avions compris qu'il nous était imposé de créer une fonction ayant un rôle d'analyseur syntaxique agissant sur les fichiers du dossier « maps/ ». En effet, ces fichiers sont des fichiers « .txt » comportant du texte censé représenter les murs, les personnages et objets du labyrinthe. Le premier but était donc de créer un convertisseur de texte en structure de données matricielles de dimensions (2*n+1) x (2*n+1).

Comme nous pensions beaucoup au jeu Crypt of the NecroDancer, on a préféré dès le départ créer une fonction (« choixDesign() ») permettant de choisir entre le design d'origine et celui de ce jeu-là. La seule modification que ce choix apportait était simplement le dossier dans lequel on allait charger les images des modèles : soit on chargeait les modèles d'origines, soit les autres.

Ensuite nous avions créé une fonction simple (« choixLabyrinthe() ») demandant à l'utilisateur le type de labyrinthe qu'il voulait. Le numéro saisi par l'utilisateur sera une clé de dictionnaire qui permettra à la fonction d'ouvrir le fichier correspondant à la valeur de la clé entrée par le biais d'une autre fonction. Cette autre fonction (« ouvrirFichier(fichier) ») permet l'ouverture du nom du fichier mis en argument. La particularité de cette fonction est qu'elle permet aussi de séparer les lignes lors de la lecture du fichier en faisant des retours à la ligne à la fin de chacune d'entre elles, car le texte se situait sur une seule ligne uniquement.

Pour résumer, nous possédions alors un programme permettant d'ouvrir le fichier qu'on veut, dans l'un des deux design et qui permettait de rendre plus aisée la lisibilité des fichiers textes pour la matrice plus tard. Il ne nous restait plus qu'à insérer le labyrinthe dans la matrice.

Le groupe s'est donc penché sur la création de la fonction « creerLabyrinthe(fichier) » prenant en paramètre le fichier texte du labyrinthe correspondant. L'idée était ici de créer une première liste vide et d'ajouter, pour chaque ligne du texte, une liste vide à l'intérieur de la première à chaque fois en faisant une boucle « for ». Et ensuite de la même manière on allait inspecter chacun des éléments de la ligne pour les ajouter à la liste de la ligne correspondante en créant une seconde boucle.

Cette fonction composée d'une double boucle « for » nous retournait donc notre matrice de caractères selon le labyrinthe que l'on choisissait. Celle-ci constituait le cœur même du labyrinthe. Tous les éléments nécessaires pour créer le labyrinthe étaient à notre portée et après cela, il nous manquait plus qu'à le concrétiser.

Etape 2 : Réalisation du moteur de jeu

La seconde partie de ce projet consiste à programmer l'interface du jeu. Le travail est similaire à la première tâche dans un sens puisque l'on a également un travail de conversion à faire. En effet, dès le départ comme avant, nous avions des données initiales que nous devions utiliser pour « créer » une structure de données différente. Ici, on va convertir les données stockées dans la matrice en données « graphique ».

Pour cela, il fallait créer une fonction dessinant l'environnement du labyrinthe que l'on appellera « drawLaby(laby, taille_case, design) ». Cette fonction est composée d'une double boucle « for » pour parcourir la matrice intégralement. Pour chacun de ses éléments nous allons vérifier son identité (-, |, A,H,V…) et à chaque fois, faire une action sur une interface carrée « upemtk » dont la taille est modifiable plus bas. Par exemple, on créer une ligne verticale ou horizontale pour les tirets, et on charge l'image correspondante pour les lettres, et cela toujours à l'emplacement des coordonnées « x » et « y » de l'objet en question dans la matrice. Modifier la taille du « tk » modifie la variable taille_case, jouant elle-même un grand rôle pour dessiner le labyrinthe dans la fonction.

L'environnement du labyrinthe étant achevé, il fallait à présent faire fonctionner les déplacements d'Ariane et pour y remédier, nous avions créé une fonction « movePerso(laby,perso,touche,char) ». Le principe est de vérifier si l'utilisateur appuie sur une des touches directionnelles. Si oui, on vérifie dans la même direction si la case voisine est libre et s'il y a un mur et, par la suite, on retourne la nouvelle position du personnage ainsi que le labyrinthe entier. Pour réussir cette fonction, les éléments mis à notre disposition à la fin de la Tâche 1 dans l'énoncé comme les coordonnées des cases voisines et murs voisins nous ont permis la réalisation de cette fonction.

Remarques suite à cela :

- L'argument « perso » est retourné par une fonction « positionPersos(laby) » qui utilise les coordonnées du labyrinthe pour retrouver la position respective de chaque personnage et l'implémenter dans un dictionnaire.
- Nous avons ensuite crée une fonction « arianeEtThesee(laby,perso,taille) » permettant aux deux personnages de se réunir en mettant des conditions vérifiant si Thésée se trouve sur une des cases voisines d'Ariane.

Après avoir créé les personnages et leurs déplacements, il restait les Minotaures, pour cela nous avons créé « mino(laby,perso,char) » permettant aux minotaures horizontaux et verticaux de se déplacer en fonction de la position du héros seul ou avec Thésée. Le plus grand problème de cette fonction était de faire fonctionner le déplacement à longue portée des Minotaures qui ne fonctionnait pas à cause de légers oublis d'incrémentation dans la boucle.

Puis enfin, pour pouvoir gérer les déplacements à chaque tour, on a créé une fonction « moveQui » s'occupant d'appeler les fonctions de déplacement d'Ariane ou d'Ariane et Thésée ainsi que des Minotaures dans le bon ordre. A la fin, on a rajouté « victoire(laby,perso,victory=None) » vérifiant si la partie est gagnée ou perdu en comparant les coordonnées de la Porte, Ariane et Thésée ainsi que les Minotaures (notamment vérifier si ils sont au même emplacement).

Etape 3 : Réalisation du solveur et ajouts

Cette étape semblait être pour nous la bête noire du projet final. On était quasiment sûrs qu'il serait obligatoire d'utiliser des fonctions récursives pour faire fonctionner le solveur du labyrinthe. On a donc commencé à créer une fonction « solveur » tout en se disant que l'on devrait avoir recours à la grande majorité des anciennes fonctions du labyrinthe. Malheureusement, plus tard, on s'est rendu compte que certains types de données du labyrinthe n'étaient pas adaptés au solveur et on rencontrait souvent des erreurs et on se mettait à tout mélanger. On a donc par la suite préféré tout réadapté au solveur mis à part les fonctions de déplacement du personnage.

On a donc pour cela recréer des fonctions dessinant le labyrinthe comme « drawMur », « actualiserPerso » et « gameState » correspondant à peu près respectivement à : « drawLaby » et « positionPersos ». Le but était ici d'adapter les données au solveur.

Ensuite nous avons recrée la fonction « solveur(laby,C,perso,taille_case,config = set()) ».

La première partie du solveur se focalisait sur l'utilisation des fonctions précédemment créée pour initialiser le labyrinthe ainsi que le chargement des images et des coordonnées des personnages dans une liste. Ensuite il fallait établir les conditions d'arrêt. Avec « C », un dictionnaire contenant les positions de chaque entité, on a créé des conditions dans le cas où par exemple les coordonnées d'Arianne et Thésée sont sur la porte ou bien si celles d'un des Minotaures se trouve sur l'un d'eux.

Pour la deuxième partie du programme, il fallait maintenant trouver comment faire en sorte que le solveur vérifie pour Arianne, les cases qui lui sont accessibles et donc pour cela nous avons mis en place trois variables, qui sont pour chacune d'entre elle des listes. Une comportant la liste des cases voisines depuis n'importe quelle coordonnée d'Arianne, une autre comportant les murs voisins et enfin la liste des touches claviers. Le concept était donc ici de parcourir ces listes toutes en même temps. En effet grâce à la commande zip(), nous avons pu réaliser cela. On a donc crée une boucle parcourant ces listes et tant qu'on avait un mur on continuait la boucle, sinon on réalisait les déplacements de tous les personnages en fonction de la situation. Puis on stockait les déplacements dans un tuple nommé Csuivant.

La troisième partie du programme consistait à rendre récursive cette fonction. Pour cela on vérifiait si « Csuivant » se trouvait dans l'ensemble « config » qui est l'ensemble de toutes les coordonnées déjà parcourues. Pour résumer, si les coordonnées d'Arianne ou Arianne et Thésée ne sont pas des cases déjà parcourues, on effectue un appel récursif de la fonction solveur jusqu'à qu'on est une condition d'arrêt et ensuite le solveur se termine.

Remarques :

- Le solveur est fonctionnel sur les labyrinthes 1 et 5 ainsi que la « sandbox », cependant quelques erreurs inconnues persistent sur les autres labyrinthe essentiellement lorsque Arianne et Thésée se rencontrent.
- Après avoir fini le solveur un membre de notre groupe : Kevin Savane, pensa plus judicieux de rendre plus attractif et attirant l'aspect de l'interface du labyrinthe en créant un menu illustré par ses soins.

Ce que nous avons appris :

Le projet d'Ariane et les Minotaures fut un projet intéressant malgré les difficultés rencontrées durant la programmation du jeu. Mais nous nous sommes rendus compte bien évidemment que ces difficultés font pleinement partie de la construction de notre apprentissage de la programmation et nous a permis de gagné beaucoup d'expérience et d'idées nouvelles sur ce qu'on peut faire avec quelques « simples » lignes de code. Créer un jeu rend le programme ludique et nous donne envie d'explorer d'autres possibilités qui ne nous sont pas forcément demandées dans les consignes et c'est cela qui pour nous rend la programmation intéressante.