Rapport Projet Programmation 2 - Partie 1

HURIOT Sacha, YAX Nicolas

February 26, 2019

A Turtle's Holistic Adventures

Les lumières venaient de s'éteindre, les chercheurs rentraient chez eux après une nouvelle journée infructueuse de tests. Seuls les animaux cobayes restaient à l'intérieur du laboratoire, témoins de l'obscurité et du silence froid de leur cellule respective. La tortue, dernière représentante de son espèce dans cette installation souterraine, s'apprêtait à fermer les yeux quand soudain, une lumière forte, un bruit assourdissant, les minces barreaux de sa cage se brisent net. Éblouie par les lumières perçantes, elle peine à discerner la silhouette qui se penche au-dessus de sa tête et y dépose un casque léger. La silhouette disparaît aussi vite qu'elle est apparue mais la tortue ne le remarque même pas, un flot d'informations et de pensées surgit là où quelques instants auparavant il n'y avait que de simples idées. La tortue reprend ses esprits et, pour la première fois, comprend la situation dans laquelle elle se trouve. Tandis que l'alarme du système de sécurité résonne toujours aussi forte, la tortue ne se focalise plus que sur un objectif : sortir d'ici. Elle fait le tri dans ces nouveaux savoirs dont elle ignore l'origine mais est certaine de la validité : la seule sortie est par l'ascenseur dont l'utilisation requiert un badge, suite à l'explosion les androïdes du système de sécurité du laboratoire ne vont pas tarder à être déployés et s'ils la trouve hors de sa cage, s'en est fini. Elle décide de se réfugier dans la cage d'ascenseur qui paraît être sure pour l'instant. Ensuite chemin, elle aperçoit deux prototypes des armes développées dans ce laboratoire, des objets intelligents commandés par la pensée. En se concentrant, et grâce à ses capacités tout juste acquises, elle en prendre le contrôle et réussit là où tous les tests de ces dernières semaines échouaient. Dans l'ascenseur, elle trouve un oiseau également libéré par l'explosion et elle découvre une nouvelle fonction de son casque : elle peut transmettre ses pensées à son nouveau compagnon. Elle donne une des deux armes à l'oiseau et commence à former un plan pour s'échapper : rassembler une armée parmi les cobayes et descendre au quinzième sous-sol, là où se trouve un prototype de téléporteur.

Principes fondamentaux du jeu Ce jeu vise à être un mélange entre un Roguelike et un RTS (Real Time Strategie). Il combine donc une progression dans un dédale de niveaux avec des objets a récupérer afin d'améliorer son expérience de jeu et pour progresser plus facilement. Chaque étage se déroule en 2 temps : une première phase de combat (RTS) qui vise a nettoyer le niveau puis une phase dite de loot (récupération d'équipements par l'activation de mécanismes). Pour inciter le joueur a prendre des risques, un timer est lancé au début de la phase de combat et permet au joueur de récupérer plus de récompenses s'il parvient à vaincre les ennemis rapidement. Ces équipements pourront alors être associés aux membres de son équipe afin de les rendre

plus forts. Si un personnage meurt pendant la phase RTS, il est definitivement mort et le joueur perd aussi l'équipement que son personnage portait. Il est aussi important de comprendre que l'on contrôle la tortue qui contrôle les autres animaux par la pensée. Les clics sont donc les ordres donnés par la tortue aux autres, le joueur joue donc la tortue.

Ce qui marche actuellement La version actuelle est une démo assez élaborée de ce qui se passera pendant la phase RTS (nous pensons que c'est la phase la plus dynamique et donc la plus interessante à présenter en partie 1). Actuellement, les déplacements à la souris ainsi que les attaques automatiques fonctionnent et un certain nombre de personnages sont déjà présents en jeu et peuvent être testés (voir le Readme pour les contrôles). On peut dès à présent actionner certains mécanismes mais pendant la phase de combat (pour le moment c'est plutot prévu pour la phase de loot mais ceci donne un aperçu des potentialités du jeu). Les étages sont déjà générés aléatoirement ainsi que les mécanismes et les récompenses. Toutes les ressources graphiques ont été faites par l'équipe.

Implémentation du moteur RTS - Events Pour implémenter le moteur RTS, j'ai utilisé une architecture classique (voir fichier LePlan.class.violet.html pour le détail). Le principe fondamental et interessant de l'implémentation est sa polyvalence. En effet l'objectif initial et principal de l'implémentation est de pouvoir ajouter n'importe quelle nouveauté facilement. J'ai utilisé un système d'events : ce sont des fonctions Unit $\Rightarrow$ Int qui révèlent toute la compléxité du code. Ces fonctions sont stockées dans des listes dans un objet Clock et seront executées périodiquement. Il y a 2 périodes disponibles : la macro periode (T=0.1s) qui va regrouper les evenements de haut niveau (bouger, attaquer, ...) et la micro periode (T=0.01s) qui va s'occuper des evenements plus bas niveau et ayant besoin d'une plus grande fréquence d'appels (affichage graphique, animations, ...). Ainsi presque toutes les actions vont être des events ce qui explique que les classes n'ont que peu de méthodes pour la plupart car l'idée est de pouvoir implémenter la suite du code en n'ajoutant que des events et sans toucher au reste de la structure du code. C'est pourquoi cette implémentation est polyvalente car la gestion des events est 'Turing Compléte' dans le sens où tout peut être implémenter avec (on pourrait transformer le jeu en Tetris sans avoir à transformer le reste du code).

Implémentation du moteur RTS - Architecture classique Pour ce qui est de la description de l'architecture (voir fichier LePlan.class.violet.html $\rightarrow$ toutes les méthodes ne sont pas dans le graphique car on n'aurait pas la place), j'ai utilisé une classe Personnage pour représenter les personnages en général puis je leur associe un Jeton quand je les place sur le terrain (les personnages peuvent exister sans jeton mais pas l'inverse). Ces personnages utilisent des compétences (qui sont des events c'est à dire a peu près n'importe quoi) comme se déplacer ou bien attaque. Ils peuvent aussi avoir des compétences passives (pas utilisées pour le moment) d'où l'utilisation de l'héritage. Pour ce qui est de la gestion du terrain, j'ai une classe Environnement qui contient toute l'information nécessaire au déroulement de la phase RTS. Elle a une Clock associée qui vise a synchroniser tous les events. Enfin pour ce qui est de l'affichage graphique, nous avons utilisé ScalaFx car Swing paraissait pas approprié pour afficher beaucoup d'informations et faire des phases RTS. Les fonctions utiles sont dans l'objet Graphics2 (car il y avait un Graphics1 utilisant ScalaFx de manière naïve

qui ne marchait pas bien). J'ai choisis d'utiliser un Canvas au final car je voulais que le moteur graphique soit indépendant du reste du code (on ne lui demande que d'afficher l'etat courant du jeu et on ne garde que très peu d'informations graphiques dans le reste du code. Etant habitué de l'orienté objet je n'ai pas eu de difficultés particulières excepté avec scalafx contre qui j'ai du longuement batailler car la documentation sur internet est très pauvre.

Génération aléatoire du plan du niveau ('Schematics.sclala') J'ai dû trouver un moyen de générer aléatoirement un niveau qui reste assez labyrinthique, esthétique et cohérent. Pour cela j'ai choisi de une pièce rectangulaire de 21x15 cellules dont les couloirs (cellules sur lesquelles les personnages peuvent se rendre) ne sont que de largeur 1. Les cellules en bordure sont toutes des couloirs, ainsi qu'un passage au milieu de la pièce et deux rectangles à droite et à gauche de ce passage. L'aléatoire repose sur les liens (de longueur deux) entre le cadre extérieur et les rectangles intérieurs. J'ai rajouté des contraintes de symétrie et de resserrement (pour laisser de la place aux mécanismes plus tard). Il y a 307 200 plans différents qui peuvent tous sortir avec une probabilité égale.

Génération aléatoire des récompenses et matrice des sprites ('Mechanisms.scala') J'utilise un plan de niveau pour y placer les mécanismes sur les obstacles de la pièce (pour la phase de loot). J'ai créé une classe abstraite pour représenter les paquets de sprites associés à chaque objet. Les circuits se décomposent en deux boutons qui peuvent être activés et qui sont chacun reliés par des tuyaux aux coffres-forts ou cellules de prison. Ici aussi le loot est généré aléatoirement (avec une probabilité égale pour l'instant). On obtient, à la fin, un background (une matrice de listes des sprites, rangés par layer croissant) sur lequel les personnages vont pouvoir combattre et avec lequel ils pourront interagir après le combat.

Production de la matrice finale des sprites ('Display.scala') Je rassemble ici le background (les tiles et mécanismes) qui n'est pas recalculé à chaque fois (c'est la valeur 'main_grid') et la liste des personnages présents dans le niveau. Pour le premier rendu, comme on a pas encore implémenté la transition d'un niveau au prochain, on part avec une liste de personnages donnée par 'load_demo_version1()'.

Création des sprites et personnages ('ressources/' et 'bddPersonnages.scala') J'ai créé différentes catégories de sprite, il y a les trois ' background_tile ' qui remplissent le fond du niveau, les 'sprite_character ' pour les personnages qui se déplacent, les ' sprite_mechanism ' pour les circuits de loot, les ' sprite_object ' pour les équipements et les 'sprite_tile ' pour les contenants de récompense (et l'eau pas encore implémentée). Cette nomenclature permet de factoriser le code pour produire la matrice des sprites à afficher. J'ai aussi créer les sept personnages contrôlés par le joueur et l'ennemi (unique pour l'instant) contrôlé par l'ordinateur, avec une répartition des caractéristiques variées (vitesse, portée, puissance et cadence d'attaque).