Rapport de TER

19 mai 2018

Table des matières

I Présentation du projet **3**

1 Introduction **4**
1.1 But du projet 4
1.2 Cahier des charges 4
1.3 Notion d'agent 5
1.3.1 Système multi-agents 5
1.3.2 Representation dans notre projet 5

2 WarBot : Le mode par défaut **6**
2.1 Principe 6

II Réalisation du projet **7**

3 Partie "Moteur" **8**
3.1 État de l'art de l'ancien projet. 8
3.2 Refonte du noyau. 9
3.3 Réalisation 9
3.3.1 Retour aux bases. 9
3.3.2 Integration dans le projet. 12
3.3.3 Améliorations des bases. 12
3.3.4 Fonctionalités finales. 12
3.4 Amélioration possible 12
3.4.1 Gestion des unités. 12
3.4.2 Gestion des scènes. 14
3.4.3 Les différentes maps 14
3.5 Fonctionnalités 14
3.6 Amélioration possible 14
3.7 Phase de conception 15

4 Partie Interface Graphique **16**
4.1 Présentation 16
4.2 Etude de l'ancienne interface 16
4.3 Nouvelle Interface 17

4.3.1 Menu Principal 17
4.3.2 Menu des Paramètres 18
4.3.3 Editeur de Comportement 19
4.3.4 Element dans le jeu 25

5 Partie "Interpreteur" 26
5.1 Présentation et Attente 26
5.2 Etude de l'ancien projet et récupération de ce qui est utile 26
5.3 Conception et implémentation 28

6 Partie "Game Design" 31

III L'avenir du projet 32

7 Amélioration possible 33

Première partie

Présentation du projet

Chapitre 1

Introduction

1.1 But du projet

L'objectif de ce projet est la réalisation d'un jeu basé sur un modèle multi-agent. L'idée générale du projet est dans la continuité de celui de l'année dernière et sur le même thème. L'outil utilisé est Unity 3D, un moteur de jeu employé dans un grand nombre de réalisations de hautes qualités. Notre projet est opérationnel sur Windows et pourrait être porté sur Mac ou encore Android. Ce projet consiste de réaliser un jeu que l'on peut qualifier de programmeur et de permettre, notamment, à de jeunes personnes de se familiariser avec le monde de la programmation. L'utilisateur pourra donc créer un comportement pour des robots appelés "unité" afin de remplir des objectifs du jeu.

Metabot est un projet modeste réalisé à partir du logiciel Unity 3D par un groupe d'étudiants débutants dans l'utilisation de cet outil. Malgré le peu d'expérience dans la création pure de ce genre d'applications, le projet actuel est le fruit d'un travail important et d'une implication entière de toute l'équipe. Il a donc pour unique prétention de communiquer notre amour du jeu vidéo et de la programmation.

1.2 Cahier des charges

Notre but était de modifier le programme existant afin qu'il devienne plus générique. C'est à dire qu'il ne se limite plus au simple jeu Warbot mais qu'il permette l'implémentation de différents jeux orientées agents facilement. Pour cela il fallait donc généraliser la gestion des différentes unités et de leur comportement (action/perception/statistique). De plus, l'ajout d'actions et de moyens de perceptions doit se faire de façon intuitive de même pour les différentes exigeances de jeu (règles/conditions de victoire).

1.3 Notion d'agent

1.3.1 Système multi-agents

1.3.2 Representation dans notre projet

Chapitre 2

WarBot : Le mode par défaut

2.1 Principe

Dans WarBot, deux à quatre équipes se battent sur un terrain pour les ressources afin de survivre et d'éliminer les autres équipes et d'être la dernière ne vie. Des ressources apparaissent sur la carte et peuvent être converties en unités ou en soin.

Deuxième partie

Réalisation du projet

Chapitre 3

Partie "Moteur"

3.1 État de l'art de l'ancien projet.

Pour commencer, attardons nous un moment sur l'ancien projet. L'an dernier, l'objectif premier était de réussir à implémenter le jeu Warbot sous Unity. Le constat que l'on peut faire de ce logiciel est que l'équipe précédente à montrer qu'il est possible de créer un jeu multi-agent en se servant de ce moteur de jeu. En effet, le Warbot Unity d'origine doté de son menu principal, son interface de paramétrage de partie et sa scène de jeu a déjà pas mal d'atout visuellement parlant. Mais il n'est pas exempt de bugs et reste assez limité. Dans cette version, il est donc possible de lancer une partie à deux joueurs uniquement, de régler le nombre de ressources créées par minutes ainsi que la quantité d'unités souhaitées de chacun des cinq types disponibles : WarBase, WarExplorer, WarHeavy, WarEngineer et Warturret. Une seule carte de jeu rectangulaire et sans obstacles est présente.

Durant une partie, le joueur peut effectuer plusieurs actions :

- Activer l'affichage d'un tableau indiquant le nombre de chaque unités détenue par les deux équipes.
- Modifier la vitesse du jeu (accélération / ralentissement / pause).
- Créer une unité pour l'une des deux équipes mais le nombre d'unités est mis à jour pour la mauvaise équipe.
- Supprimer une unité mais le nombre d'unités n'est pas mis à jour.
- Déplacer une unité.
- Activer l'affichage des stats d'une unité (vie,ressources détenues, type).
- Activer l'affichage du suivi visuel des envoies de message entre unités.

- Activer l'affichage du groupe auquel appartient une unité mais l'utilisation de groupe dans les comportements n'est pas fonctionnelle.
- Activer l'affichage de la barre de vie d'une unité.
- Activer l'affichage de la barre de ressource d'une unité.

A voir si a compléter

3.2 Refonte du noyau.

Après s'être pencher en profondeur dans le code nécessaire au bon fonctionnement du jeu, on s'est très vite rendu compte de ses limites en terme de généricité. On a donc pris la décision en accord avec Mr Ferberr et malgré quelques réticences de sa part, de reprendre le "moteur de jeu" from scratch.

3.3 Réalisation

La réalisation du projet a été effectuée en suivant une méthode agile. Mr Ferber avait pour cela le rôle à la fois de chef de projet et de client, le projet s'est donc découpé en plusieurs phases de rush entrecoupés par des réunions régulière avec lui. Il nous est donc paru pluq judicieux de vous expliquer le déroulement de notre travail de manière chronologique. On reviendra plus en détail sur cette méthode et son application à notre situation dans la partie gestion de projet.

3.3.1 Retour aux bases.

Afin de mener à bien la création du moteur de jeu, et dans le but de ne pas géner l'avancer de nos camarades des autres équipes de ce projet, on a choisi de ne pas toucher directement au jeu original mais plutôt de repartir à zéro en créant un nouveau projet sur Unity tout en étant conscient de la difficulté futur que serait son intégration.

Suite à la décision de reprise du moteur de jeu from scratch, nous avons commencé à mettre en place les mécanisme de contrôle d'une unité, M. Ferber nous conseillant de repartir sur la base de la version Java de Warbot. Le premier pas a été de simplement faire bouger une unité. Pour cela, nous avons utilisé le coposant NavMesh que nous avons attaché au GameObject Unit représentant l'unité. Grâce à cela l'unité était capable de parcourir un chemin jusqu'à arriver à sa destination en évitant les différents obstacles avec lesquelles elle pourrait rentrer en collisions,cette dernière étant detecter grâce au composant Collider. M. Ferber nous avait demandé d'étudier la possibilité de créer des zones restrictives, sur lesquelles certaine unité ne seraient pas capables d'aller ou se déplaceraient à une vitesse différente. Cette opération fut simple à mettre en place avec le composant NavMesh, il suffisait d'indiquer à l'unité que si sa destination se

trouvait à un emplacement interdit par exemple elle ne pouvait pas s'y rendre. Mais l'utilisation du NavMesh ne convenait pas au jeu Warbot. En effet, dans le jeu l'utilisateur est libre du choix du comportement de l'unité mais doit aussi faire avec les alléa du terrain, il doit prendre en compte la possibilité que l'unité soit bloquée et doit donc changer de direction. Or, avec l'utilisation du NavMesh cette opération était faite de manière automatique, le NavMesh embarque un algoritme calculant automatiquement un chemin permettant rejoindre le but demandé, son utilisation a donc été rapidement abandonnée. Les zones restrictives ont elles aussi étaient annulées, leur conception sans NavMesh nous ayant parue bien trop complexe et chronophage à ce niveau du projet pour un élément non primordial.

A partir de ce moment, nous avons donc simplement modifié la position du composant Transform (composant determinant la position, rotation et l'échelle de chaque objet dans la scène), du GameObject Unit en fonction du vecteur de mouvement voulu afin de déplacer l'unité.

Une fois ces premiers mouvements rendus possibles, nous nous sommes attarder sur la conception des instructions de l'unité. Celles-ci étant constituées d'une suite de conditions (les percepts) et d'une action terminant le tour de l'unité. La valeur d'un percept était donc calculée à partir d'une fonction booléenne vérifiant les conditions à remplir. Par exemple, le percept de ressource, devant vérifier si une ressource était à proximité. Nous avons commencé par créer la gestion de l'action de la manière suivante : pour créer une action il fallait hériter de la classe Action et surcharger sa méthode do() en créant l'action voulue.

Avant de passer a un exemple concret de la gestion d'une instruction, on va expliquer comment fonctionnait le champs de vision d'une unité et le calcul des objets "vus" par celle-ci à ce moment du projet. Le GameObject Unit a un composant Collider sphérique, une sphère tout autour de lui. Ce Collider a son booléen IsTriggered actif, ce qui signifie que la sphère n'est pas un objet physique auquel on peut se heurter, mais que l'on peut détecter tout GameObject à l'intérieur de celle-ci. Dans cette sphère, on décide d'un angle fixe qui sera l'angle de champs de vision de l'unité.

AJOUTER LE CODE !

Voici comment ce mets à jour la liste des objets "vus" par l'unité :

- On récupére la liste des objets présents dans la shpère.
- Pour tous les objets présents dans la sphère.
- Si celui-ci a son centre de gravité dans l'angle de champs de vision de l'unité, on l'ajoute à la liste des objets vus.

Voici un exemple concret d'exécution d'une instruction "Si l'agent voit une ressource, il la récupère", symbolisée par les classes PerceptRessource et ActionPick :

AJOUTER LE CODE !

Pour cette instruction on a donc accès à son percept et son action, la vérification des percepts se fait de la manière suivante :

- On récupére la liste des percepts nécessaires à l'instruction.
- On récupére la liste des percepts de l'unité.
- On récupére la valeur de chaque percepts demandés.
- Si toutes leurs valeurs sont vrai on exécute l'action demandée.

AJOUTER LE CODE !

La valeur de chaque percept de l'unité est mise à jour à chaque unité de temps, pour PerceptRessource cela se passe comme suit :

- Pour tous les objets présents dans la liste des objets vus de l'unité.
- On cherche s'il en existe un qui a un composant "ItemHandler" (ItemHandler est propre aux Prefab de type Ressource cela nous permet de vérifier que l'objet trouvé est bien une ressource).
- Si c'est le cas on modifie la valeur de la "target" de l'unité (la cible de l'unité pour ce tour de jeu) pour qu'elle soit égale à l'objet trouvé.

AJOUTER LE CODE !

L'action de récupération d'un objet par une unité se déroule comme ceci :

- On récupère le composant Objet de la "target" de l'unité (ce composant est la valeur de l'objet à récupérer).
- On ajoute cet objet dans le composant inventaire de l'unité.
- On détruit le GameObjet "target" (on détruit la représentation physique de l'objet sur la scène).

Notre objectif était de simplifier l'ajout de percepts et d'actions à une unité. La définition d'une classe par action et par percept nous semblait donc assez coûteuse nous avons donc étudié l'une des spécificités du C#. Les type "delegate" propre au C# dont la déclaration est semblable à une signature de méthode. Elle a une valeur de retour et un nombre quelconque de paramètres de type quelconque. Nous l'avons donc utilisé afin d'encapsuler les méthodes correspondant aux actions et aux percepts et ce afin d'éviter la crétion de multiple classes. A la suite de ce changement, la classe Percept s'est dotée d'un attribut dictionnaire[String,Listener], un percept étant définit par son nom et son Listener associé. Le Listener étant un delegate représentant la fonction qui correspond à la fonction de calcul de sa valeur.

delegate
percept agressive percept common actionUnit Cette phase a duré .....
retour aux bases
fonctionnalités finales
creation d'un jeu
amélioration possibles avec maxime

3.3.2 Integration dans le projet.

La phase d'intégration.

Problèmes liés à l'intégration.

Résolution des problêmes.

3.3.3 Améliorations des bases.

3.3.4 Fonctionalités finales.

3.4 Amélioration possible

3.4.1 Gestion des unités.

La gestion des unités de manière générique nous a poussé à réfléchir aux éléments minimum définissant celle-ci. Elles nécessitent donc 6 scripts de base :

- Brain

- Stats
- Percept
- Actions
- ActionNonTerminales
- Messages

Le corps

Le corps de l'unité correspond à son modèle 3D, et son GameObejct Collider associé.

Pour le jeu warbot nous avons réaliser plusieurs modèles 3D.

- Base [Photo] [Descriptif]
- Explorer [Photo] [Descriptif]
- Light [Photo] [Descriptif]
- Heavy [Photo] [Descriptif]

L'esprit

Afin de pouvoir fonctionner, les unités possede 3 parties distinctes :

- La gestion des actions possibles.
- La gestion des perceptions.
- L'identité propre de l'unité.

Perceptions. Concept de percept : qu'est ce qu'un percept ?

Un percept est condition nécessaire, dans l'architecture de subsomption, pour effectuer une action.

Création d'un percept

Gestion du percept de son appel à sa réalisation.

Prenons par exemple le percept "PerceptAlly", pour celui-ci nous ....

Actions.

Non terminales. Une action non terminale est une composante d'une action terminale, par exemple "se tourner vers un ennemi",celle-ci ne termine pas le tour de l'unité,

Création d'une action non terminale.

Gestion de l'action non terminale de son appel à sa réalisation.

Prenons par exemple l'action non terminale "ActionTurnEast", pour cellei-ci nous ....

Terminales. Une action terminale est considérée comme un ordre donné à l'unité à chaque unité de temps, à la différence de l'action non terminale, celle-ci termine le tour de l'unité,

Création d'une action terminale.

Gestion de l'action de son appel à sa réalisation.

Prenons par exemple l'action "Fire", pour cellei-ci nous ....

3.4.2 Gestion des scènes.

Une scène, comme l'unité, détient elle aussi des composants minimums, ceux-ci ont été regroupé dans un GameObject appelé "MetabotNecessary". Ils se résument en 8 scripts :

- MainCamera : la caméra principale.
- TurnManager : s'occupe de la gestion des tours.
- RessourceGenerator : un générateur de ressource.
- MinimapCamera : camera permettant l'affichage de la minimap.
- ItemManager : gère le comportement des objets.
- HUD : canvas d'affichage tête haute.
- UnitManager : gestionnaire d'unité qui comporte les 4 equipes.

3.4.3 Les différentes maps

Nous avons réaliser un ensemble de cinq maps chacune ayant son thème et ses particularités :

- Mountain [Photo] [Descriptif]
- Plain [Photo] [Descriptif]
- Desolate [Photo] [Descriptif]
- Garden [Photo] [Descriptif]
- Simple [Photo] [Descriptif]

3.5 Fonctionnalités

3.6 Amélioration possible

Malgré tous nos efforts, il persiste toujours un soucis d'optimisation .....

3.7 Phase de conception

Notre conception est basée sur notre savoir en matiere de programation orientée agent tout en gardant à l'esprit la notion de généricité qui est au coeur de notre projet. Le but de notre moteur de jeu est de permettre à des développeurs de pouvoir réaliser un mode de jeu orienté agent, de facon simple.

Chapitre 4

Partie Interface Graphique

4.1 Présentation

La partie Interface Graphique comprend principalement l'habillage visuel des éléments avec lequel l'utilisateur va interagir pour jouer au jeu. MetaBot étant un jeu pour "programmeur", le joueur interagit surtout avec le menu principal et l'éditeur de comportement afin de créer des équipes pour pouvoir les faire s'affronter.

L'interface graphique que nous avons créé se décompose en quatre parties : le menu principal, le menu de paramètres, l'éditeur de comportement et des fonctionnalités directement en jeu.

4.2 Etude de l'ancienne interface

L'ancienne interface nous a semblé plutôt sommaire, et les actions y étaient limitées ; pour l'éditeur on pouvait seulement choisir une équipe, une unité et les Contrôles/Conditions/Actions basiques (Il y avait aussi plusieurs bugs présents mais on ne parle ici que de l'interface).

Le menu principal contenait un bouton de nouvelle partie, un bouton vers l'éditeur, un bouton pour quitter la partie et une case à cocher pour arrêter le son.

Notre première décision a été de revoir totalement l'interface, ce que nous pouvions faire grâce aux nouveaux éléments apportés. En effet il fallait rajouter de nouvelles fonctionnalités basiques à cette interface qui en proposait un nombre limité, et rendre l'aspect visuel moins austère.

4.3 Nouvelle Interface

4.3.1 Menu Principal

Le menu principal donne accès aux options majeures du jeu.

Lancer une Partie

Ce bouton permet de lancer une partie du jeu MetaBot. Le lancement de la partie prend en compte le mode de jeu sélectionné, les équipes choisies, le nombre de chaque unités en début de partie, le nombre de ressources maximum (nombre de ressource présentes en jeu au même moment), la carte de jeu.

Bouton Éditeur de Comportement

Ce bouton permet d'accéder à l'éditeur de comportement, que nous détaillerons plus loin.

Bouton Paramètres

Permet d'accéder aux paramètres du jeu, tels que le réglage du volume, le choix de mode de jeu, la langue, etc...

Choisir le nombre d'équipe

Nous avons la possibilité de choisir le nombre d'équipes participant à la partie. Les valeurs sont dans un menu déroulant et vont de 2 à 4.

Choisir les équipes

Dans le cadre de chaque équipe se trouve un menu déroulant avec les noms de toutes les équipes présentes dans les fichiers du jeu. On peut donc en sélectionner une pour qu'elle participe à la prochaine bataille. De plus, à coté du nom de l'équipe, on retrouve aussi son score (ELO).

Bouton "Reload Team"

Ce bouton permet de recharger les équipes. L'intérêt est de permettre à l'utilisateur d'ajouter manuellement au dossier des équipes des nouevaux fichiers d'équipes, et les voir apparaître simplement en cliquant sur ce bouton, sans avoir à relancer le jeu.

Choisir carte de jeu

On peut directement choisir le lieu de la partie en cliquant sur les flèches de part et d'autre de l'aperçu de la carte. Il existe pour le moment cinq cartes : Moutain, Plain, Simple, Desolate et Garden.

Choisir nombre de départ de chaque unité

En dessous de l'aperçu de la carte, il y a les noms des unités existantes. Sous ces noms, le chiffre indique le nombre de ce type d'unité présent au lancement de la partie. On peut incrémenter ou décrémenter ce chiffre à l'aide des boutons "+" et "-" à coté de celui-ci.

Barre de Chargement

Quand on lance une partie, une barre de chargement apparaît et indique l'avancement du chargement de la partie.

4.3.2 Menu des Paramètres

Changer le Volume de la musique

Ce slider indique le niveau sonore de la musique. On peut le modifier en cliquant dessus et en déplaçant la valeur de 0 jusqu'à 100. 0 correspond à un arrêt de la musique et 100 au volume maximal. De plus, le volume sonore dans le menu est le même dans l'éditeur de comportement et dans le jeu lui même.

Choisir nombre de ressource maximum dans le jeu

Dans cette case on peut entrer un chiffre entier qui indiquera le nombre maximum de ressource présentes simultanément en jeu.

Choisir le mode de jeu

Ce menu déroulant permet de choisir le mode de jeu de la prochaine partie. Il y a actuellement deux modes, le mode MetaBot et le mode RessourceRace. Si le mode choisi est RessourceRace alors deux autres paramètres apparaissent :

- le temps imparti
- le nombre de ressources à atteindre pour gagner.

Choisir la langue

Les boutons en forme de drapeau indiquent les langues disponibles pour le jeu. Pour changer de langue il suffit d'appuyer sur le drapeau voulut et de faire Valider les paramètres.

Bouton Retour

Ce bouton permet de retourner à l'écran du menu principal.

Bouton Valider

Ce bouton valide les paramètres définit au dessus et revient au menu principal en ayant appliquer ces paramètres.

Bouton Quitter

Ce bouton ouvre une boite de dialogue demandant à l'utilisateur s'il veut vraiment quitter le jeu. Il peut ainsi choisir de revenir sur le menu principal ou fermer le jeu.

4.3.3 Editeur de Comportement

Pour la refonte de l'éditeur de comportement, ce qui nous a paru le plus optimal fut de repartir d'une interface vierge. Nous avions identifié un nombre trop important de changements à effectuer pour justifier des modifications de l'ancienne interface. Nous avons commencé par schématiser deux interfaces potentielles.

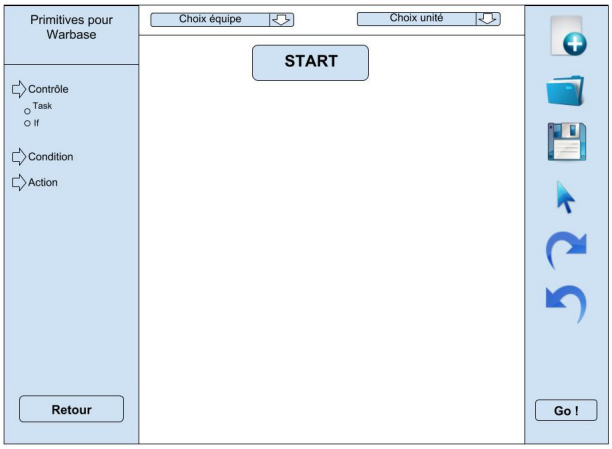

FIGURE 4.1 – Première ébauche

Le première modèle s'appuie sur l'esprit de l'ancienne interface, avec le panneau de contrôles fortement similaire. Cependant, après concertation lors d'une réunion, nous avons écarté ce modèle, car il ne semblait pas optimisé. La barre d'outils paraissait bien trop importante, et la zone d'édition de comportement s'en voyait restreinte.

Quant au second modèle, il nous semblait organisé de manière plus logique, et intuitive. Toutes les options et choix se trouvent à gauche de la fenêtre, alors que la partie droite se consacre à l'édition du comportement.
Nous avons donc décidé de partir sur ce modèle.

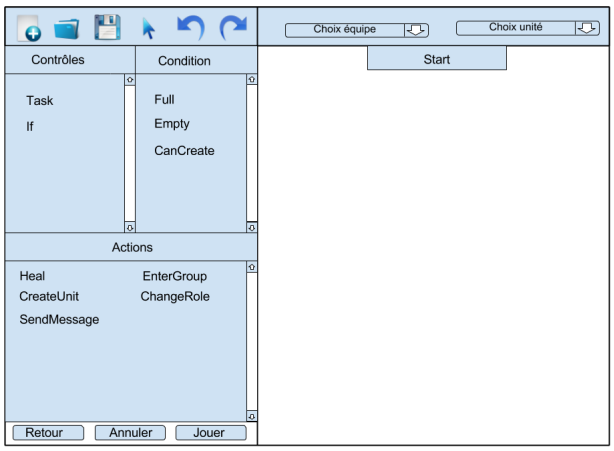

FIGURE 4.2 – Seconde ébauche

ToolBox

A) **Bouton Nouveau Comportement**
Ce bouton permet de vider le comportement de l'unité de l'équipe courante. Si on veut tout effacer sur l'unité où on est on appuie dessus au lieu de tout supprimer à la main.

B) **Bouton Chargement Comportement**
Ce bouton permet de charger le comportement de l'unité actuellement sélectionnée dans le DropDown prévu à cet effet. La première chose à faire est d'effacer le comportement actuellement chargé, s'il existe. A l'aide du path, et du nom de l'équipe, nous allons pouvoir appeler une méthode présente dans l'interpréteur, pour transformer le fichier .xml correspondant à l'équipe courante, en un comportement, qui sera stocké dans un dictionnaire.
Nous allons ensuite récupérer la position de la pièce StartPuzzle, pour pouvoir placer les pièces correctement dans l'éditeur. Nous avons donc notre point de départ.
Il ne reste plus qu'à placer les pièces. Nous allons donc récupérer la liste d'instructions associée à chaque unité, et placer chaque instruction dans leur ordre d'apparition. (Un "if" pour commencer, suivi de sa ou ses conditions, suivies d'une ou plusieurs actions)

Pour finir, on réinitialisera la scrollbar de l'éditeur, pour permettre à l'utilisateur de voir directement le début du comportement.

C) **Bouton Sauvegarde du Comportement**
Ce bouton sauvegarde le comportement de l'unité de l'équipe courante. Si on change d'unité dans la même équipe mais sans sauvegarder, alors le comportement de l'unité précédente sera perdu.
Voyons de plus près deux de ses fonctions primordiales au bon fonctionnement du script :

- **createBehavior()** :
  Nous allons créer un comportement, copiant le comportement présent dans l'éditeur. Pour commencer, nous allons répertorier toutes les pièces "If" présentes. Ensuite, pour chaque pièce If de la liste, nous allons créer l'instruction comprenant les conditions et actions, grâce à l'interpréteur.
  Une fois chaque instruction créée pour chaque "If", nous pouvons appeler la fonction createXML().

- **createXML()** :
  Appelle la fonction behaviorToXml(), présente dans l'interpréteur, qui prend en paramètres le nom de l'équipe courante, le chemin où écrire le fichier .xml, le nom de l'unité, et la liste d'instructions correspondante.

D) **Bouton "Undo"**
Va permettre d'annuler la création ou la suppression d'une pièce. Lors de la création d'une pièce, cette pièce sera ajoutée dans une liste "listPieces", qui nous permettra de garder une trace de l'ordre dans lequel les pièces ont été créées.

- **Undo()** :
  Cette fonction requiert une liste de pièces non vide pour fonctionner, c'est donc ce que l'on va vérifier en premier. Si la liste "listPieces" n'est pas vide, alors on stocke la dernière pièce de cette liste dans une variable "pieceToUndo".
  Maintenant, il faut déterminer quelle action utilisateur nous devons annuler (création ou suppression).
  Si "pieceToUndo" est active, alors nous devons simuler sa suppression, en la passant inactive. Il faudra également l'ajouter dans la liste "recoverList", qui a un comportement similaire à "listPieces", pour reconstituer une annulation.
  Si, en revanche, la pièce est inactive, il faut alors la faire réapparaitre à l'écran, en la passant en active.

E) **Bouton "Redo"**
Permet de restituer une annulation préalablement faite. Lorsqu'on clique

sur le bouton Annuler, on va conserver l'action annulée dans une liste, qui nous servira donc à la restituer lors d'un clic sur le bouton Redo.

F) **Bouton de retour au menu principal**
Ce bouton ramène l'utilisateur au menu principal. Avant de cliquer dessus il faut penser à bien sauvegarder le comportement en cours pour ne pas le perdre.

Sélection des équipes / unités

A) **Choix de l'équipe**
Ce menu déroulant permet de choisir l'équipe sur laquelle on veut travailler.

B) **Choix de l'unité**
Ce menu déroulant permet de choisir l'unité de l'équipe sur laquelle on va opérer les changements de son comportement.

C) **Bouton Création d'équipe**
Juste à droite des équipes se trouve un bouton en forme de croix, il permet de créer un nouvelle équipe. Si on appuie une boite de dialogue s'ouvre et nous demande le nom de la nouvelle équipe. Après avoir validé le nom, la boite de dialogue se ferme et la nouvelle équipe est présente dans le menu déroulant des équipes.

D) **Bouton Suppression d'équipe**
Ce bouton permet de supprimer définitivement l'équipe courante. Une boite de dialogue demandera confirmation.

Informations sur l'unité

A) **Affichage des statistiques de l'unité courante**
Quand on choisit une unité, sur sa droite apparaît ses valeurs dans un cadre, cela correspond à ses statistiques.

B) **Affichage du modèle 3D de l'unité courante**
Dans le même cadre apparaît aussi le modèle 3D de l'unité courante. C'est l'apparence qu'aura l'unité en jeu.

Liste des pièces

A) **Boutons de choix de la catégorie de la pièce**
Sur la gauche, de manière verticale, se trouve cinq noms de catégories qui sont les Contrôles, les Conditions, les Actions, les Messages et les Actions non terminales. En cliquant sur l'une de ces catégories, on affiche la liste des éléments de cette catégorie dans la zone directement à droite.

B) **Zone de sélection de la pièce suivant la catégorie**
C'est dans cette zone qu'apparaît la liste des éléments des catégories de pièces de l'éditeur. La génération des modèles des pièces se fait de manière dynamique. Chaque type de pièce a un prefab associé.
Un script contenant une méthode pour chaque type de pièces disponibles (Conditions, Actions, etc...), permet ce dynamisme. Voyons de plus près la gestion d'une des catégories de pièces (leur comportement est semblable).

- **UpdateCondition()** : Nous commençons par récupérer une structure UnitPerceptAction, contenant le nom d'une unité, ainsi que toutes les conditions, actions, et messages inhérents à l'unité.
  A partir de là, nous pouvons créer, à l'aide du prefab correspondant, une pièce, avec le label contenu dans notre strucutre. Nous allons parcourir la structure pour créer toutes les pièces disponibles pour l'unité, et modifier leur placement, en leur ajoutant un vecteur, pour qu'elles ne se superposent pas.

```
new Vector2(0, -button.GetComponent<RectTransform>().rect.↩
    height)+ deltaVect;
```

  Dans cette fonction, nous nous occupons uniquement de la liste Conditions présente dans notre structure. Il y a une fonction pour chaque liste de la structure.

C) **createPuzzle.cs** : Lors de l'appel à ce script, nous récupérons le prefab et le label associé au type de pièce concerné. Par exemple, pour une pièce "Conditions", nous récupérerons le prefab des pièces "¡Conditions"¿, ainsi qu'un label nommé "PERCEPT". Le traitement de ce label se fait dans le script ConditionEditorScript.cs.
Nous plaçons ensuite la pièce créée

- **create()** : Lors d'un clic sur le modèle de pièce que l'on veut ajouter au comportement courant, cette fonction est appelée. Une position par défaut est définie dans l'éditeur, qui déterminera où la pièce sera placée lors de sa création.
  A chaque pièce créée, nous l'ajoutons dans la liste listPieces, utilisée pour la fonction Annuler.

Les pièces sont présentes sous forme de cases avec leur nom à l'intérieur (certaines possèdent même des menus déroulant pour choisir la valeur voulue une fois dans la zone d'édition). La couleur des pièces dépend de leur catégorie. Les pièces peuvent être sélectionnées et déplacées dans la zone d'édition du comportement grâce au glissé/déposé (Drag & Drop).

Editeur

A) **Zone éditeur de comportement de l'unité et de l'équipe courante**
Dans cette zone arrivent les pièces venant de la zone de sélection. Elles

se placent à l'aide du curseur de la souris sur une grille invisible. Si une pièce est valide, alors elle est colorée, sinon, elle est grise. Les pièces "IF" possèdent un cadenas dans le coin supérieur droit, il permet de déplacer, en plus de la pièce IF, tout les éléments valides qui lui sont rattachés (hors IF). Les pièces de contrôles sont considérées comme valides si elles sont placées en dessous d'autres pièces du même type ou en dessous de la pièce "Start".
Les autres pièces doivent se rattacher à des pièces de contrôle, en haut à droite pour les pièces Condition et en bas à droite pour les autres. Les pièces hors Contrôle sont ainsi valides lorsqu'elles sont au bon endroit, sur leur ligne, adjacente au contrôle, ou adjacente à une autre pièce du même type, valide.

Voyons les scripts permettant cette gestion.

- **ManageDrapAndDrop.cs** :
  Ce script s'occupe de la gestion du déplacement des pièces, ainsi que de leur placement, sur une grille aimantée.
  - **OnMouseDrag()** :
    Une fois qu'un clic a été fait sur une pièce, cette fonction s'occupe de mettre à jour les coordonnées de la pièce, de telle sorte qu'elles soient égales à celles du pointeur de la souris. Pour valider les nouvelles coordonnées de la pièce, il faut appeler la fonction OnMouseUp().
  - **UpgradeGridPosition()** :
  - **OnMouseUp()** :
    Cette fonction appelle la fonction UpdateGridPosition(), puis attribue à la pièce courante sa nouvelle position. Une fois cela fait, il faut vérifier que la nouvelle position de la pièce est toujours un emplacement valide, d'un point de vue comportement. La fonction appelle pour ça le script StartPuzzleScript.cs, que nous verrons plus bas.
- **StartPuzzleScript.cs** :
  Ce script va nous permettre de vérifier si une pièce "If" est bien placé en dessous de la pièce Start.
  - **UpdateAllValidPuzzles()** :
    Cette fonction, appelée dans ManageDragAndDrop.OnMouseUp() va nous servir à vérifier si la position de la pièce "If" est correcte. Pour commencer, nous passons la variable booléenne "isValid" de toutes les pièces à false, puis nous allons vérifier leur placement.
    Pour se faire, nous allons appeler la fonction UpdatePuzzle() du script IfPuzzleScript.cs.
    Ensuite, si la pièce Start a bel et bien une pièce "If" juste en dessous d'elle, alors on passe la valeur isValid du "If" à true.

- **IfPuzzleScript.cs** :
  Ce script s'occupe de vérifier le placement de toutes les pièces actuellement sur l'éditeur. Si leur positionnement n'est pas correct, leur couleur sera grise. Sinon, une pièce Contrôle sera verte, une pièce Condition sera bleue, une pièce Action Non Terminale sera orange, une pièce Action sera Rouge, et une pièce Message sera Jaune.
  - **UpdateCondPuzzle()** :
    Cette fonction parcours toutes les pièces présentes sur l'éditeur. Si la pièce observée est une pièce de type Condition, et qu'elle est placée à droite d'une pièce "If", au niveau de sa première ligne, alors on l'attribue à une variable.

```
if (currentGridPos + new Vector2(1,0) == puzzleGridPos ↩
    && typePuzzle == PuzzleScript.Type.CONDITION)
```

    Cela nous permettra, dans UpdatePuzzle, de savoir que nous avons une pièce Condition placée à notre droite, sur la bonne ligne. Le comportement des fonctions UpdateIfPuzzle() et UpdateActPuzzle se comportent similairement. Le seul changement est la place dans l'éditeur. Une pièce "If" devra être située directement en dessous de notre "If" courant, et une pièce Action / Action non terminale / Message devra se trouver directement à droite de notre pièce "If", sur sa deuxième ligne.
  - **UpdatePuzzle()** :
    Cette fonction récupère les pièces adjacentes à notre pièce actuelle, à l'aide des scripts UpdateIfPuzzle(), UpdateCondPuzzle, UpdateActPuzzle(). Il met ensuite à jour les valeurs des pièces adjacentes, à savoir leur variable isValid, ainsi que leur variable NextPuzzle, qui leur permet de savoir quelle pièce leur est adjacente.

4.3.4 Element dans le jeu

Réglage du volume du son

En bas à droite de la fenêtre de jeu se trouve une icône de son et un slider. Le slider permet, comme dans le menu des paramètres, de modifier le volume du son. L'icône sert de bouton ; si on le presse, le son passe à 0 et l'icône devient barrée. Si l'on appuie de nouveau, il redevient classique, et le son est restitué à sa valeur précédente.

Chapitre 5

Partie "Interpreteur"

repasser sur le texte pour ajouter exemples et illustrations !

5.1 Présentation et Attente

La partie "Interpréteur" est la partie la moins visible du projet MetaBot, mais il s'agit de la partie du projet servant de clé de voute du jeu. Comme dit plus tôt, la particularité de MetaBot est que le joueur, qui pourra être considéré comme le programmeur, va préparer en amont une cohésion d'équipe à travers le comportement et va pouvoir lancer un match contre une autre équipe, afin d'évaluer quel comportement sera le meilleur. L'interpreteur permet de faire la liaison entre l'éditeur du comportement ou l'utilisateur va développer son comportement, en utilisant un ensemble de d'instructions que nous avons prédéfinies et la partie moteur, ou le fonctionnement des unités est inscrit, ainsi que les différents modes de jeux.

Le langage et l'ensemble des instructions nécessaires est alimenté par l'équipe Game Design qui nous a donné des exemples de messages ou de spécificités du langage qui pourrait être nécessaire. On pouvait ensuite tous en discuter en pesant le pour et le contre, afin de définir si la fonctionnalité allait être mise en place , et de quelle façon.

5.2 Etude de l'ancien projet et récupération de ce qui est utile

Au départ, il a été nécessaire de remettre en place un outil permettant de récupérer un comportement, qui était uniquement graphique, dans l'éditeur afin de pouvoir le renvoyer à la partie Moteur, pour que l'ensemble des unités puissent l'exécuter. Nous avons pris connaissance de ce que l'ancien groupe avait mis en place et avons trié ce qui nous semblait correspondre à notre version du projet. Il

était obligatoire d'écrire le comportement récupéré de l'éditeur dans un fichier, afin de le récupérer , pouvoir le modifier , le déplacer , et le conserver pour plusieurs parties. La solution mise en place par l'ancien groupe pour le stockage, qui était d'utiliser un fichier XML correspondait parfaitement à notre besoin, car la syntaxe et l'organisation en nœuds de ce genre de fichiers, permettait une récupération simple et claire des instructions. Nous avons ainsi pu récupérer une bonne partie de leur système d'écriture et de lecture de leur projet, tout en adaptant l'autre partie à nos besoins.

La partie organisant les instructions à été complètement supprimée, et nous avons repensé un système plus générique et plus compréhensible pour les groupes qui vont récupérer ce projet plus tard.

5.3 Conception et implémentation

La construction des Instructions des unités dans la partie Interpréteur a grandement été facilitée et la liaison des instructions avec leur exécution a été décalée dans le moteur. Ainsi on ne manipule que des chaînes de caractères lors de l'accès dans le fichier, et le moteur effectue la liaison avec une structure de "Delegate" comme expliqué plus haut, rajoutant beaucoup de généricité au projet. D'un point de vue de la sécurité, celle-ci a été décalée dans l'éditeur du comportement, afin de ne pas traiter des comportements erronés , mais prévenir a l'avance et même empêcher les erreurs d'apparaitre.

Le but de l'interpréteur était de traiter des fichiers de ce type :

```
<behavior>
  <teamName>Default Team</teamName>
  <unit name="Explorer">
    <instruction>
    </instruction>
  </unit>
  <unit name="Base">
  </unit>
  <unit name="Light">
  </unit>
  <unit name="Heavy">
  </unit>
</behavior>
```

Ainsi nous avons 1 fichier par équipe écrite par l'utilisateur, chacun détaillant le comportement de chaque unité sous la même forme :

```
<behavior>
  <teamName>Default Team</teamName>
  <unit name="Explorer">
 <instruction>
      <parameters>
        <PERCEPT_ENEMY />
      </parameters>
      <message>
        <ACTN_MESSAGE_HELP>
          <Light />
        </ACTN_MESSAGE_HELP>
      </message>
      <actions>
        <ACTION_MOVE />
      </actions>
    </instruction>
    <instruction>
      <parameters>
        <PERCEPT_BLOCKED />
      </parameters>
      <actions>
        <ACTION_MOVE_UNTIL_UNBLOCKED />
```

```
      </actions>
    </instruction>
  </unit>
</behavior>
```

Le comportement d'une unité est une suite d'instructions, dont l'ordre est très important. La hiérarchie va déterminer la priorité qu'aura l'action sur le tick. Ainsi l'action la plus prioritaire dont les conditions sont acceptées sera l'action effectuée sur ce tick. Chaque instruction est toujours organisée de la même façon : - Une liste de conditions à remplir pour que l'instruction soit considéré comme acceptée - Une liste d'actions qui ne terminent pas l'action, appelées Actions non terminales, mais qui peuvent être les messages partagés entre les unités par exemple. Ces actions non terminales seront exécutés avant : - L'action terminale, qui va être exécutée à la fin du tick pour l'ensemble des unités et va être l'action qui va être "physiquement" fait par l'unité, par exemple : Avancer, Tirer, Récupérer ou donner une ressource,Se soigner , ...

Comme dit précédemment, l'interpréteur relie les 2 parties majeures du projet. Ainsi, une fonction permet de transformer un fichier ".wbt" (format des équipes WarBot) vers un comportement d'équipe utilisable par l'équipe moteur, et une fonction permettant la transformation d'un comportement récupéré de l'éditeur graphique vers un fichier XML, pour sa sauvegarde.

Un nouveau travail d'interprétation a vite été ressenti dans le projet. En effet, le jeu étant destiné à aider les jeunes à appréhender la programmation, une traduction française était nécessaire. Toujours dans l'idée de rester générique, un traducteur a été mis en place et attaché à l'ensemble des textes qui seront affichés à l'utilisateur.

//pas clair a changer La traduction fonctionne ainsi : Un premier script s'occupe de vérifier la langue qui doit être utilisée dans le fichier de configuration du jeu et va charger en mémoire un fichier de traductions :

Le fichier va être parcouru, et un ensemble de couples clé,valeurs vont être récupérés , la clé correspondant à la valeur initiale d'un bouton/ texte affiché, la valeur étant la traduction qu'il faudra affiché à la place de la clé.

Ce script est ainsi rattaché au "GameManager" , un objet Unity qui n'est pas détruit lorsque l'on change de scène. Ainsi les traductions sont conservées lorsqu'on passe de l'éditeur au menu principal , du menu principal aux cartes de jeu, ...

Un second script est lui attaché à tous les Objets du jeu qui vont devoir être traduits. Celui-ci va , à la création de l'objet auquel il est rattaché, chercher la langue actuellement affichée, récupérer comme clé le texte original de l'objet auquel le script est affilié, et va stocker la traduction, et l'afficher à l utilisateur.

En Runtime, si la langue est changée dans les paramètres du jeu, les script de traduction des objets vont mettre à jour leur langue, et récupérer la nouvelle traduction dans le "GameManager".

Chapitre 6

Partie "Game Design"

Troisième partie

L'avenir du projet

Chapitre 7

Amélioration possible