Rapport de TER

20 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 . . . 4
1.3.1 Système multi-agents . . . 4
1.3.2 Representation dans notre projet . . . 4

2 WarBot : Le mode par défaut **5**
2.1 Principe . . . 5

II Réalisation du projet **6**

3 Partie "Moteur" **7**
3.1 État de l'art de l'ancien projet. . . . 7
3.2 Refonte du noyau. . . . 8
3.2.1 Retour aux bases. . . . 10
3.2.2 Intégration dans le projet. . . . 13
3.2.3 Améliorations des fonctionnalités. . . . 14
3.2.4 Fonctionalités finales. . . . 18
3.3 Amélioration possible . . . 18
3.3.1 Gestion des unités. . . . 18
3.3.2 Gestion des scènes. . . . 19
3.3.3 Les différentes maps . . . 19
3.4 Fonctionnalités . . . 20
3.5 Amélioration possible . . . 20
3.6 Phase de conception . . . 20

4 Partie Interface Graphique **21**
4.1 Présentation . . . 21
4.2 Etude de l'ancienne interface . . . 21
4.3 Nouvelle Interface . . . 21
4.3.1 Menu Principal . . . 21
4.3.2 Menu des Paramètres . . . 22
4.3.3 Editeur de Comportement . . . 23
4.3.4 Élément dans le jeu . . . 29

III L'avenir du projet **30**

5 Amélioration possible **31**

6 Bugs **32**

IV Annexe **33**

7 Scripts Remarquables **34**
7.1 ObjectPool 35
7.2 MovableCharacter 37

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 exigences 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.

L'ancien projet est une adaptation du jeu Warbot crée en java en utilisant la librairie MadKit, permettant la conception et la simulation de système multi-agents. En utilisant comme base ce projet, et en utilisant la hiérarchie de classe proposé dans le code java dans un moteur de jeu relativement bien assisté comme Unity, de nombreux problèmes de conceptions peuvent apparaître.

Unity, pour rappel, est un moteur de jeu développé par Unity Technologies. Ce logiciel à la particularité d'etre "orienter assets". Les scripts associé à chaque objet (appellée GameObject) dérive de la classe "MonoBehaviour", ce qui permet d'avoir accées à un ensemble de méthodes et d'attribut nécessaire à la création de comportement et d'interaction.

Dans cette section, nous allons développer les problèmes que nous avons rencontrer dans la réalisation d'un moteur de jeu générique sur la base de l'ancien projet et de l'explication de la nécessiter de recréer un moteur à partir de zéro.

Nous avons donc remarqué des problèmes du faite de ce choix de conception. Tout abords les scripts en eux-même ne sont pas adaptés au développement d'un programme sur Unity et surtout ne sont pas générique. En effet, les unités sont codés en dur dans le code, ne permettant pas l'ajout de nouvelles unités de façon simple. Ainsi pour rajouter de nouvelles unités, il faut créer de nouvelles classes correspondant a de nouvelles unités. Cette hiérarchie des classes est un parti-pris que l'équpe de l'année passée à choisi en se basant sur le code java de WarBot. Cependant, la problématique de notre sujet de TER ne nous permet pas d'avoir ce genre de conception dans l'idée de rendre l'ajout des unités plus simple. Ainsi, il fallait reprendre la conception des unités et voir les composants que l'on peut garder de façon unique et modulable pour toute les unités.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using System;
5  using WarBotEngine.Projectiles;
6  using System.Security.Cryptography;
7
8  namespace WarBotEngine.WarBots {
9
10     public class AttackController : MonoBehaviour {
11 [...]
12         public bool Reloaded()
13         {
14             return (this.reloading + this.reload_time < Time.time);
15         }
16
17         public bool Shoot()
18         {
19             if (!this.Reloaded())
20                 return false;
21             this.Fire();
22             return true;
23         }
24 [...]
25         public void Fire() {
26             SoundManager.Actual.PlayFire(this.gameObject);
27             Instantiate(this.projectile , this.warprojectile_emitter.transform.
       position , this.warprojectile_emitter.transform.rotation , this.transform);
```

```
28              Instantiate(this.muzzle_flash , this.warprojectile_emitter.transform.
        position , this.warprojectile_emitter.transform.rotation , this.transform);
29              this.reloading = Time.time;
30          }
31      }
32  }
```

Listing 3.1 – Code du script AttackController.cs de l'ancien projet

La gestion des actions et des perceptions aussi est problématique. Le code de ces actions sont codés en dur dans des scripts correspondant a des unités d'un certain type, empêchant alors de rendre générique le fait de rajouter des actions sans devoir modifier tout les codes nécessitant le nom des actions tel que l'interpréteur. Le code ci-dessous montre le script "AttackController.cs" de l'ancien projet. Dans ce code, l'actions Shoot depend des fonction Reloaded et Fire. Reloaded peut être considérer comme une perception. Le fait que les actions et les perceptions ne sont pas clairement définie entant que tels posent des problèmes de compréhension du code et de faite le rajout de façon simple une nouvelle action et une nouvelle perception. De plus, cela pose le problème de la gestion des actions et des perceptions de l'interpréteur, car pour chaque action créer dans le script, il faut indiquer à l'interpreteur quelle fonction permet d'activer l'action du comportement en cours.

Ainsi dans le code du listing 3.5, on peut voir que pour activer la fonction Shoot() du script AttackController, il faut créer une fonction du même nom, puis renseigner toutes les unités qui peuvent effectuer l'action. Le problème est qu'il faut, pour chaque action, créer une fonction dans le fichier Unit.cs, et ainsi le fichier Unit.cs est surchargé, atteignant les 2000 lignes de codes !

```
1  using ...;
2  namespace WarBotEngine.Editeur
3  {
4      [...]
5      public class Unit
6      {
7          /// <summary>
8          /// The unit shoot a projectile if is reloading
9          /// </summary>
10         /// <returns>Return true if action success and false otherwise</returns>
11         [PrimitiveType(PRIMITVE_TYPE.ACTION)]
12         [UnitAllowed(WarBots.BotType.WarHeavy)]
13         [UnitAllowed(WarBots.BotType.WarTurret)]
14         [PrimitiveDescription("Fait tirer l'unité (termine l'action si réussi)")]
15         public bool Shoot()
16         {
17             return this.agent.GetComponent<WarBots.AttackController>().Shoot();
18         }
19     }
20 }
```

Listing 3.2 – Extrait du code du script Unit.cs

La suite de l'examen du code de l'ancien projet et avec l'accord de notre encadrant en exposant les problèmes qu'engendre la reprise de l'ancien moteur, nous avons décider de recréer un nouveau moteur de jeu pour repartir sur des bases plus générique.

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.

3.2.1 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.

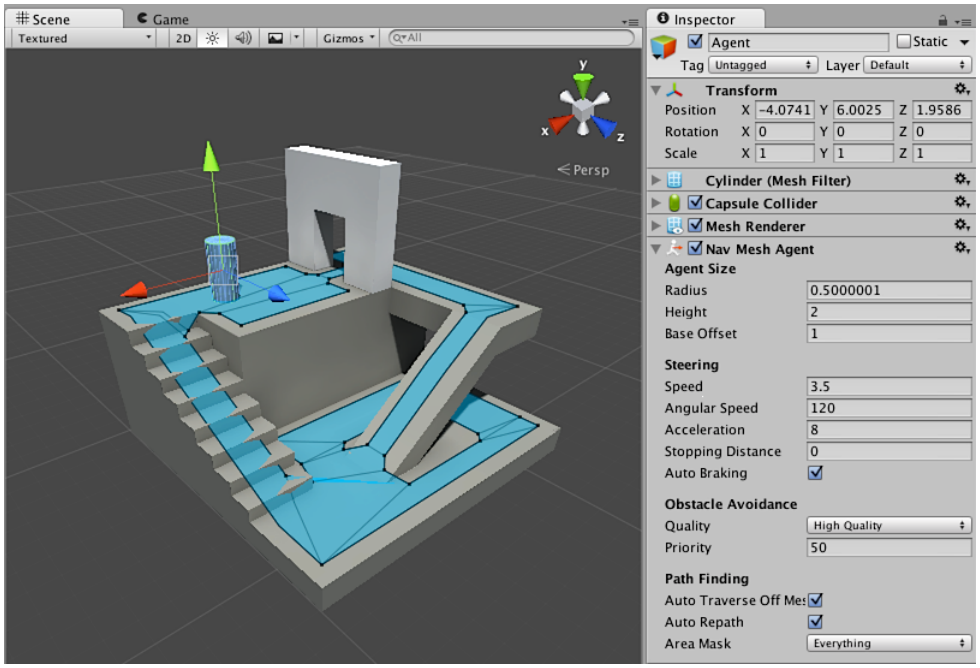

FIGURE 3.1 – Exemple d'application du composant *NavMesh* dans Unity. L'agent se déplace à l'intérieur de la zone bleue.

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.

```
1       public override void Do()
2       {
3           Objet obj = _target.GetComponent<Objet>();
4           Inventory unitInventory = GetComponent<UnitManager>().GetComponent<Inventory
        >();
5           unitInventory.add(obj);
6           obj.getPicked();
7       }
```

Listing 3.3 – Extrait du code du script ActionPick.cs première version.

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" en utilisant sa méthode getPicked(), qui applique la fonction Destroy(GameObject obj) de Unity (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 est un delegate représentant la fonction de calcul de valeur du percept. Le même mécanisme est mis en place pour les actions et est toujours d'actualité aujourd'hui.

```
1   _percepts["PERCEPT_FOOD"] = delegate ()
2           {
3               GetComponent<Stats>().SetTarget(null);
4               foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
5               {
6                   if (gO && gO.tag == "Item")
7                   {
8                       GetComponent<Stats>().SetTarget(gO);
9                       return true;
10                  }
```

```
11              }
12              return false;
13          };
```

Listing 3.4 – Extrait du code du script PerceptUnit.cs

```
1   _actions["ACTION_PICK"] = delegate () {
2               GameObject target = GetComponent<Stats>().GetTarget();
3               if (target != null)
4               {
5                   Objet obj = target.GetComponent<ItemHeldler>()._heldObjet;
6                   Inventory unitInventory = GetComponent<Inventory>();
7                   if (!unitInventory.isFull())
8                   {
9                       unitInventory.add(obj);
10                      Destroy(target);
11                  }
12              }
13          };
```

Listing 3.5 – Extrait du code du script ActionUnit.cs

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

3.2.2 Intégration dans le projet.

Une fois qu'on a estimé que le moteur etait assez générique

La phase d'intégration.

Maxime

Problèmes liés à l'intégration.

Maxime

Résolution des problèmes.

Maxime

3.2.3 Améliorations des fonctionnalités.

Optimisation du temps de chargement

L'un des gros problèmes que l'on a rencontré est le fait que le temps de chargement lors d'une partie était beaucoup trop élevé et cela nuisait au bon déroulement de la partie, atteignant parfois deux minutes de temps de chargement lorsque l'on voulait générer 200 unités. Pour cela nous avons étudié plusieurs pistes pour résoudre ce problème.

1. La gestion des actions des unités. La première piste de recherche que l'on a étudié est d'essayer d'optimiser la gestion des actions des unités. Lorsque l'on a conçu les unités, elles faisaient leurs actions dans une fonction utilisée par Unity ("Update") qui est appelée à chaque calcul d'image. Ainsi, pour chaque unité présente sur la scène, Unity calcule l'action que celle-ci doit faire. Cependant, certaine action nécessite plus de temps en fonction de sa complexité, ce qui pouvait engendrer que des unités avaient des tours de retard par rapport à d'autres unités.
Pour palier à ce problème, nous avons donc délégué la gestion des tours des unités à un script "TurnManagerScript.cs" (Listing 3.6).

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TurnManagerScript : MonoBehaviour
6  {
7      public float _timeTick;
8      public float _ticksPerSeconds;
9
10     // Update is called once per frame
11     void FixedUpdate ()
12     {
13         _timeTick += 0.02F * Time.timeScale;
14         _ticksPerSeconds = (1.0f / 0.02F) * Time.timeScale;
15         if (_timeTick >= 0.04f) // Est-ce que 0.4 secondes ce sont écoulées ?
16         {
17             _timeTick -= 0.04f;
18
19             foreach (GameObject unit in GameObject.FindGameObjectsWithTag("Unit"))
20             {
21                 if (unit.GetComponent<Brain>()) // Comporte un "Brain"
22                 {
23                     unit.GetComponent<Brain>().UnitTurn(); // Fait l'action de l'unité
24                 }
25             }
26         }
27     }
28 }
```

Listing 3.6 – Code du script TurnManagerScript.cs

Ce script permet donc de contrôler le fait que chaque unité ne commence pas un nouveau tour si d'autres unités n'ont pas fini ou fait le leurs. Cependant, cette optimisation n'a pas permis de réduire

le temps de chargement d'une partie, restant toujours à prés de deux minutes pour la génération de 200 unités.

2. *ObjectPool*, les "bassins" d'objets. Pour essayer de mieux localiser le problème, nous avons effectué des tests sur le temps de chargement en fonction du nombre d'unités. Les résultats sont consignés dans le tableau ci-dessous. On peut donc voir que le nombre d'unité à générer influe directement sur le temps de chargement.

Nombre d'unités à générer	Temps de chargements (moyenne de 3 tests)
0 unités	0.49 secondes
1 unités	0.70 secondes
5 unités	3.13 secondes
20 unités	11.50 secondes
50 unités	27.84 secondes
100 unités	50.23 secondes
200 unités	162.20 secondes
500 unités	276.71 secondes

Pour empêcher la génération des unités pendant le temps de chargement, nous avons décidé d'utiliser une méthode dite "du bassins d'objets" (ou *ObjectPooling*). Cette méthode consiste à créer un ensemble d'objets dans un objet appelé *Pool*, puis de les rendre inactifs pour que les scripts contenus dans ces objets ne soient pas exécutés.

```
1 public struct Pool
2 {
3     public string tag; // Label du Pool
4     public int number; // Nombre d'objets
5     public GameObject prefab; //Objet de référence
6 }
```

Listing 3.7 – Code de la structure *Pool* de ObjectPool.cs

Cette méthode permet de ne pas créer une unité directement, mais de juste prendre un objet Unit dans le bassin correspondant. Les bassins sont représentés par une structure définit au listing 3.7. Elle comporte un *tag* représentant le nom du bassin, un *number* correspondant au nombre d'unités à créer dans ce bassin, et enfin un *prefab* qui est l'objet qui servira de référence pour la copie des objets. Le script ObjectPoolScript.cs (Listing 7.1) permet la gestion de plusieurs *Pools*, et dans notre cas, nous utilisons 3 *Pools* : *Light*, *Heavy* et *Explorer*.

Cependant, le temps de chargement est sensiblement le même en utilisant ce principe. Le problème vient donc de la conception même des objets des unités.
Pour trouver d'où viens précisément le problème, on a donc décidé de créer 200 unités et de désactiver certain composant. En utilisant ce procédé, on a pu découvrir et comprendre l'origine du problème.

3. Simplification des collisions. La gestion des collisions entre unités et leur environnement ce faisait avec un *MeshCollider*. Le *MeshCollider* est un composant de type *Collider* qui permet la gestion des collisions notamment en envoyant des messages aux différents acteurs de la collision. Le *MeshCollider* à la particularité d'épouser plus ou moins la forme du maillage de l'objet que l'on met en arguments.

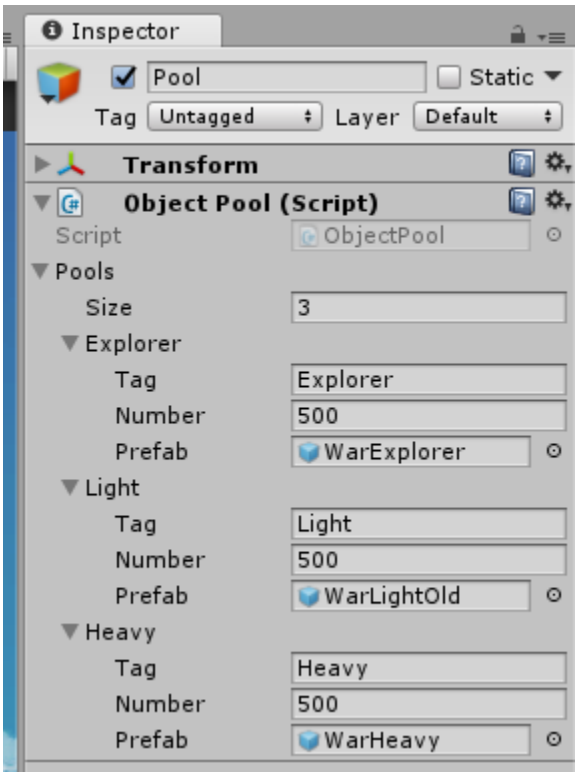

FIGURE 3.2 – Visualisation dans l'*Inspector* d'Unity des *Pools* du script ObjectPool.cs

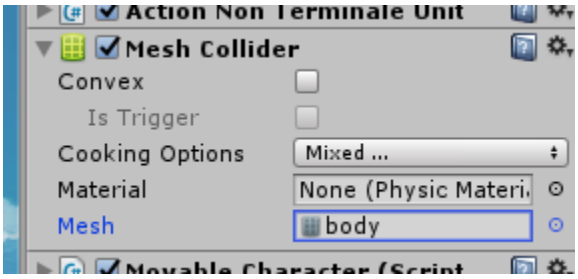

FIGURE 3.3 – Un composant *MeshCollider*

Néanmoins, le *MeshCollider* est un composant gourmand en ressource pour calculer les potentiels collisions avec son environnement et, surtout, les maillages que nous utilisons comportent un nombre relativement conséquent de points et de faces. Nous avons résolu ce problème en supprimant les composants *MeshCollider* des unités en les remplaçant par des *BoxColliders*. Les *BoxColliders* sont des composants de type *Collider*, qui permet la gestion des collision. Ce composant est définit par un pavé ce qui réduit considérablement les calculs pour connaître les éventuelles collisions avec celui-ci.

Effectivement, le temps de chargement est considérablement réduis passant de plus de 4 minutes pour 500 unités à une poignée de secondes.

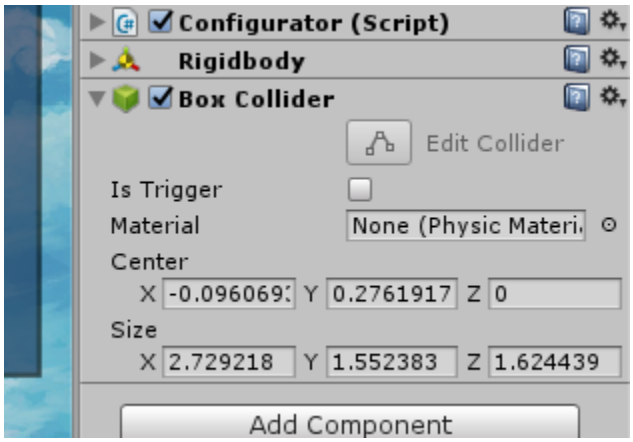

FIGURE 3.4 – Un composant *BoxCollider*

Amélioration des collisions.

Après avoir découvert et réglé le problème de chargement des parties. Nous avons remarqué de nombreux bugs par rapport aux collisions des unités. En effet, certaines unités ne reconnaissait pas l'environnement et traversait le décor. Pour résoudre ce problème, nous avons donc définit certaines règles de collisions.

1. Une unité rentre en collision si un objet est devant lui.
2. Une unité rentre en collision si l'angle du point de collision par rapport a son axe frontal est inférieur à 90°.

```
1  public class MovableCharacter : MonoBehaviour
2  {
3  [...]
4      void OnCollisionStay(Collision other)
5      {
6          collisionObject = null;
7          if (other.gameObject.tag != "Ground")
8          {
9              foreach (ContactPoint contact in other.contacts)
10             {
11                 float a = Utility.getAngle(gameObject.transform.position, contact.
       point);
12                 float b = GetComponent<Stats>().GetHeading();
13                 float A = Mathf.Abs(a - b);
14                 float B = Mathf.Abs( 360+ Mathf.Min(a,b) - Mathf.Max(a, b) ) ;
15                 if (Mathf.Min(A, B) < 90f)
16                 {
17                     collisionObject = other.transform.gameObject;
18                     break;
19                 }
20             }
21         }
22     }
23 [...]
24 }
```

Listing 3.8 – Extrait du code de MovableCharacter.cs

Dans l'extrait du code ci-dessus, on peut voir que l'objet qui est en collision avec l'objet est mis à jour seulement si l'angle est inférieur à 90°. Pour éviter de rentrer en collision avec le sol, l'objet ne doit pas comporter le *tag* "Ground".

3.2.4 Fonctionalités finales.

Mehdi

3.3 Amélioration possible

Mehdi et Maxime

3.3.1 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 ....

3.3.2 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.

3.3.3 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]

3.4 Fonctionnalités

3.5 Amélioration possible

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

3.6 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.

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.

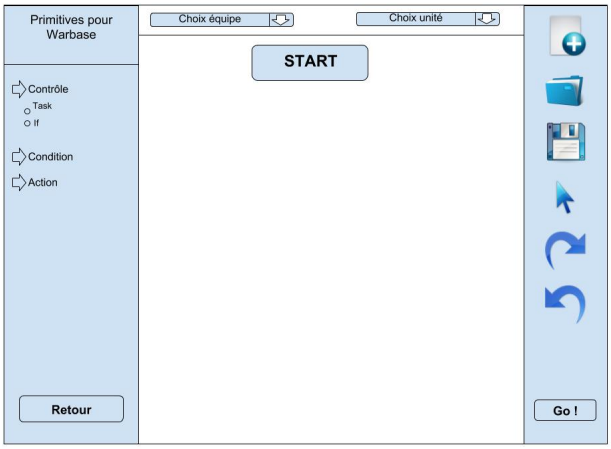

FIGURE 4.1 – Première ébauche

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.

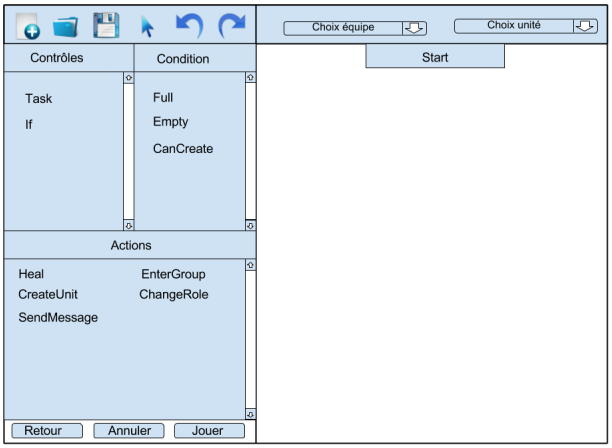

FIGURE 4.2 – Seconde ébauche

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 Drop-Down 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.

```
1 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.

```
1  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 Élément 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.

Troisième partie

L'avenir du projet

Chapitre 5

Amélioration possible

Chapitre 6

Bugs

Quatrième partie

Annexe

Chapitre 7

Scripts Remarquables

7.1 ObjectPool

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class ObjectPool : MonoBehaviour
6  {
7      static public Dictionary<string, Queue<GameObject>> dictPools = new Dictionary<
       string, Queue<GameObject>>();
8      public Pool[] pools;
9
10     static bool created = false;
11
12     void Awake()
13     {
14         if (!created)
15         {
16             DontDestroyOnLoad(this.gameObject);
17             created = true;
18         }
19         else
20         {
21             Destroy(this.gameObject);
22             return;
23         }
24
25
26         foreach(Pool pool in pools)
27         {
28             pool.prefab.SetActive(false);
29             Queue<GameObject> queue = new Queue<GameObject>();
30             for (int i = 0; i < pool.number; i ++)
31             {
32                 GameObject instance = Instantiate(pool.prefab);
33                 instance.SetActive(false);
34                 instance.transform.parent = transform;
35                 queue.Enqueue(instance);
36             }
37             dictPools.Add(pool.tag, queue);
38         }
39
40     }
41
42
43     static public GameObject Pick(string tag, Vector3 position, Quaternion rotation)
44     {
45         GameObject obj = dictPools[tag].Dequeue();
46         if (obj)
47         {
48             obj.transform.position = position;
49             obj.transform.rotation = rotation;
50             obj.SetActive(true);
51
52         }
53         return obj;
54     }
55
56     static public GameObject Pick(string tag, Vector3 position)
57     {
58         return Pick(tag, position, Quaternion.identity);
```

```
59      }
60
61      static public GameObject Pick(string tag)
62      {
63          return Pick(tag, Vector3.zero, Quaternion.identity);
64      }
65
66
67
68
69
70  }
71
72
73  [System.Serializable()]
74  public struct Pool
75  {
76      public string tag;
77      public int number;
78      public GameObject prefab;
79  }
```

ObjectPool.cs

7.2 MovableCharacter

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MovableCharacter : MonoBehaviour
6  {
7      public float speed;
8      public Vector3 vectMov;
9      public bool _isblocked;
10     public float _offsetGround;
11     public float _offsetObstacle;
12     public float _edgeDistance;
13     public bool _obstacleEncounter;
14
15
16     public GameObject collisionObject;
17
18     private Vector3 nextposition;
19
20     public void Start()
21     {
22         _isblocked = isBlocked();
23     }
24
25     public void Move()
26     {
27         _isblocked = isBlocked();
28         if (!isBlocked())
29         {
30             vectMov = Utility.vectorFromAngle(GetComponent<Stats>().GetHeading());
31             nextposition = transform.position + vectMov.normalized * speed * 0.02f;
32
33             transform.position = nextposition;
34
35         }
36         else
37         {
38             transform.position *= 1;
39         }
40
41     }
42
43     public bool isBlocked()
44     {
45         return collisionObject != null;
46     }
47
48     void OnCollisionStay(Collision other)
49     {
50         collisionObject = null;
51         if (other.gameObject.tag != "Ground")
52         {
53             foreach (ContactPoint contact in other.contacts)
54             {
55                 float a = Utility.getAngle(gameObject.transform.position, contact.
       point);
56                 float b = GetComponent<Stats>().GetHeading();
57                 float A = Mathf.Abs(a - b);
58                 float B = Mathf.Abs( 360+ Mathf.Min(a,b) - Mathf.Max(a, b) ) ;
```

```
59                  if (Mathf.Min(A, B) < 90f)
60                  {
61                      collisionObject = other.transform.gameObject;
62                      break;
63                  }
64
65              }
66              /*
67              for (int i = 0; i < hits.Length; i++)
68              {
69                  RaycastHit hit = hits[i];
70                  if (hit.transform.gameObject != gameObject)
71                  {
72                      collisionObject = hit.transform.gameObject;
73                  }
74              }*/
75          }
76      }
77
78      void OnCollisionExit(Collision other)
79      {
80          collisionObject = null;
81      }
82
83  }
```

MovableCharacter.cs