

RAPPORT DE TER

METABOT
UN JEU MULTI-AGENT GÉNÉRIQUE

Equipe Interface Graphique

FUMEY David

DOREY Benoit

Equipe Moteur

BENLAMINE Mehdi

TEIXEIRA-RICCI Maxime

Supervisé par M. FERBER Jacques

Université Montpellier - Master 1 Informatique

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 Représentation dans le projet	6
II Réalisation du projet	8
2 Partie Moteur	9
2.1 État de l'art de l'ancien projet.	9
2.2 Refonte du noyau.	10
2.2.1 Retour aux bases.	12
2.2.2 Intégration dans le projet.	23
2.2.3 Améliorations des fonctionnalités.	24
2.2.4 Fonctionnalités finales.	33
2.3 Modélisation 3D, sons et animation.	33
2.3.1 Les unités Warbot.	33
2.3.2 Les différentes cartes.	34
2.3.3 Animations.	36
2.3.4 Sons.	38
2.4 Amélioration possible	40
3 Partie Interface Graphique	41
3.1 Présentation	41
3.2 Etude de l'ancienne interface	41
3.2.1 Menu Principal	41
3.2.2 Editeur de comportement	42
3.2.3 La sélection des équipes	44

3.2.4	Améliorations envisagées	44
3.3	Nouvelle Interface	45
3.3.1	Menu Principal	45
3.3.2	Menu des Paramètres	50
3.3.3	Editeur de Comportement	53
3.3.4	Élément dans le jeu	66
III	L'avenir du projet	67
4	Amélioration possible	68
5	Bugs	69
IV	Annexe	70
6	Scripts Remarquables	71
6.1	ObjectPool	72
6.2	MovableCharacter	74
6.3	Brain	76
6.4	PerceptUnit	79
6.5	LoadFile	82

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 fonctionne 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 "unités" 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

Dans cette section, nous allons développer la notion d'agent dans un système multi-agent.

1.3.1 Système multi-agents

Un agent dans un système multi-agents est une entité autonome définie par un comportement créé par son auteur. Les agents peuvent aussi interagir avec leur environnement grâce à des perceptions et qui lui permet de lui donner une représentation de ce dernier.

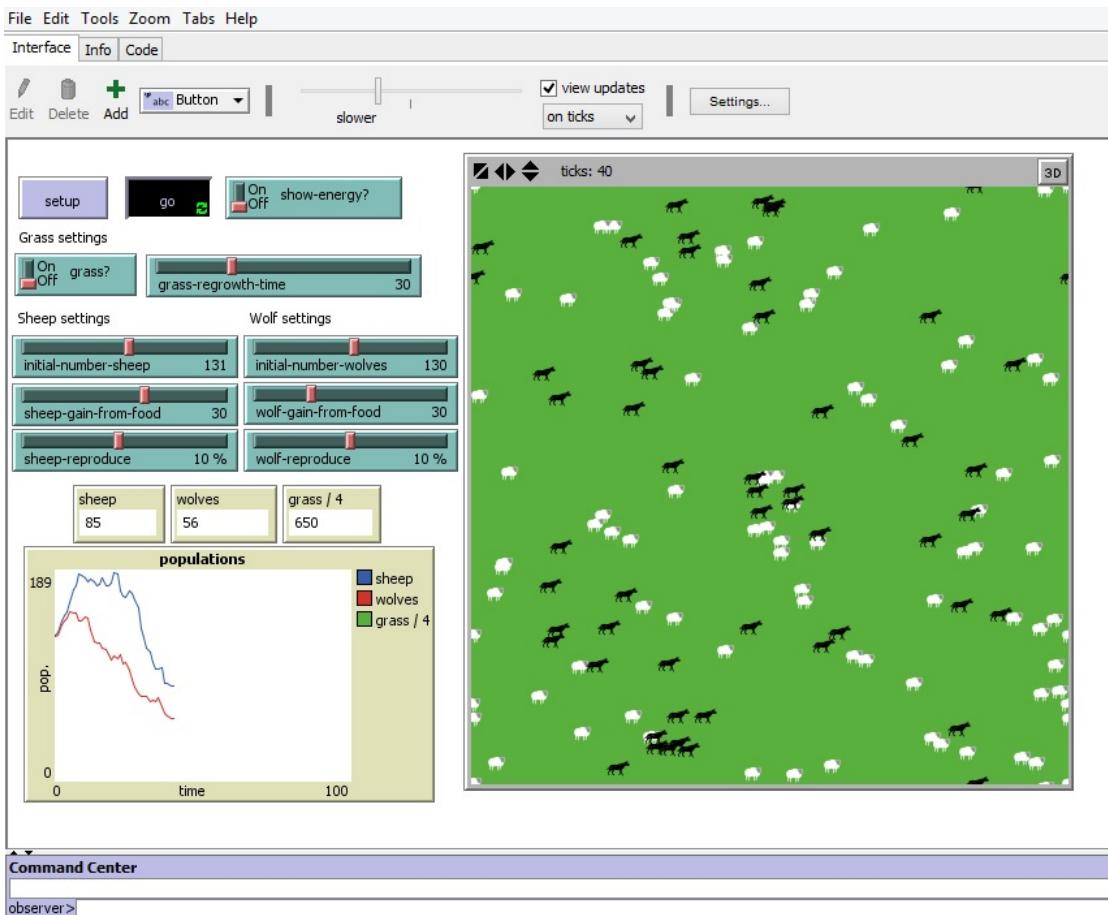


FIGURE 1.1 – NetLogo : Un logiciel de simulation multi-agents.

L'individualisme au service du collectif. L'intérêt des agents est l'idée qu'une action individualiste et égoïste permet un comportement collectif d'un ensemble d'agents pour réaliser des actions qu'un individu seul ne pourrait pas accomplir. Pour permettre de réaliser des objectifs de groupe, les individus peuvent communiquer entre eux afin de permettre de déléguer ou de s'unir dans des tâches plus ardues.

Représentation et simulation Ce genre de système permet, en créant des comportements d'agents simple, la simulation de société plus complexe. Ainsi, on peut étudier les comportement globales des entités selon ce que l'on veut observer.

Les systèmes multi agents dans de multiples domaines. Les systèmes multi-agents sont utilisées dans de nombreux domaines, notamment dans les jeux vidéos et le cinéma. Dans ces domaines, les systèmes multi-agents sont utilisées pour représenter des mouvement de foules avec des comportement très simples, ou, dans le cas du jeu vidéo, permettre de créer des comportements aux personnages non joueurs pour les rendre plus intéressant pour le joueurs.

1.3.2 Représentation dans le projet

WarBot : Le mode par défaut

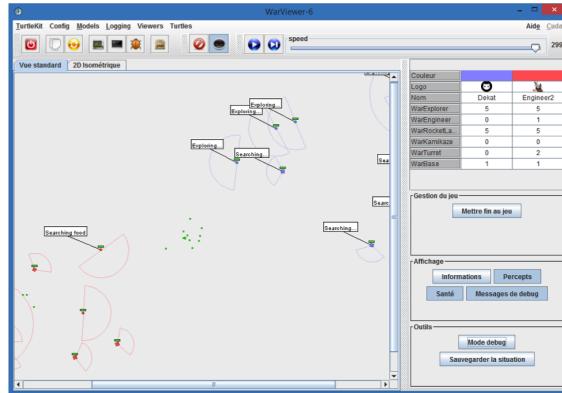


FIGURE 1.2 – WarBot, la version Java.

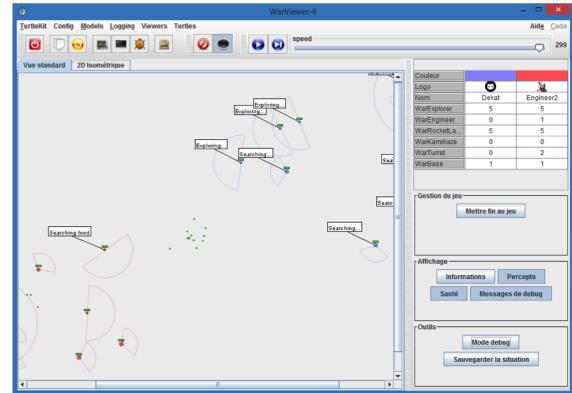


FIGURE 1.3 – WarBot, la version Unity.

Warbot Java en quelques mots

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

Une équipe est composé d'un ensemble de sept unités : la WarBase (base), le WarExplorer (explorateur), le WarEngineer (ingénieur), le WarLight (char léger), le WarHeavy (char lourd), le RocketLauncher (lance-missile) et la Turret (tourelle).

Le joueur doit programmé le comportement de son équipe, en particulier une architecture de subsomption ou une machine à état finie peuvent être développées dans ce but.

Différences avec Warbot Unity

Sur notre version actuelle de Warbot Unity, jusqu'à quatre équipes peuvent combattre ensemble pour leur survie. Quatre unités sont disponibles : la Base, l'Explorer, le Light et le Heavy, pour ce qui est de la définition du comportement, seule l'architecture de subsomption est disponible.

Deuxième partie

Réalisation du projet

Chapitre 2

Partie Moteur

2.1 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 visuels, 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 statistiques d'une unité (vie, ressources détenues, type, ...).
- Activer l'affichage du suivi visuel des envoies de messages 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.

2.2 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 proposée 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'être "orienter assets". Les scripts associés à chaque objet (appelé GameObject) dérive de la classe "MonoBehaviour", ce qui permet d'avoir accès à un ensemble de méthodes et d'attributs nécessaire à la création de comportements et d'interactions.

Dans cette section, nous allons développer les problèmes que nous avons rencontré 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 nouveau moteur à partir de zéro.

Nous avons donc remarqué des problèmes du fait de ce choix de conception. Tout abord, 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 codées en dur dans le code, ne permettant pas l'ajout de nouvelles unités de façon simple et rapide. Ainsi pour rajouter de nouvelles unités, il faut créer de nouvelles classes correspondant à ces nouvelles unités. Cette hiérarchie des classes est un parti-pris que l'équipe 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 permettait 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     {
27         SoundManager.Actual.PlayFire(this.gameObject);
28         Instantiate(this.projectile, this.warprojectile_emitter.transform.
position, this.warprojectile_emitter.transform.rotation, this.transform);
29         Instantiate(this.muzzle_flash, this.warprojectile_emitter.transform.
position, this.warprojectile_emitter.transform.rotation, this.transform);
30         this.reloading = Time.time;
31     }
32 }
```

Listing 2.1 – 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 à 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 fonctions Reloaded et Fire. Reloaded peut être considérer comme une perception. Le fait que les actions et les perceptions ne sont pas clairement définies entant que tels posent des problèmes de compréhension du code et de faire 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'interpréteur quelle fonction permet d'activer l'action du comportement en cours.

Ainsi dans le code du listing 2.10, 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(PRIMITIVE_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 2.2 – 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 décidé 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. M. 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 régulières avec lui. On exposera 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.

2.2.1 Retour aux bases.

Afin de mener à bien la création du moteur de jeu, et dans le but de ne pas gêner l'avancer de nos camarades des autres équipes de ce projet, on a choisi de ne pas toucher directement au projet 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 de zéro, nous avons commencé à mettre en place les mécanismes 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 composant NavMesh que nous avons attaché au GameObject Unit représentant l'unité.

Le composant *NavMesh* est un composant d'Unity permettant de déplacer des agents sur des zones (appelé *Navigation Mesh*). Ces zones peuvent avoir différentes propriétés (comme le coût de

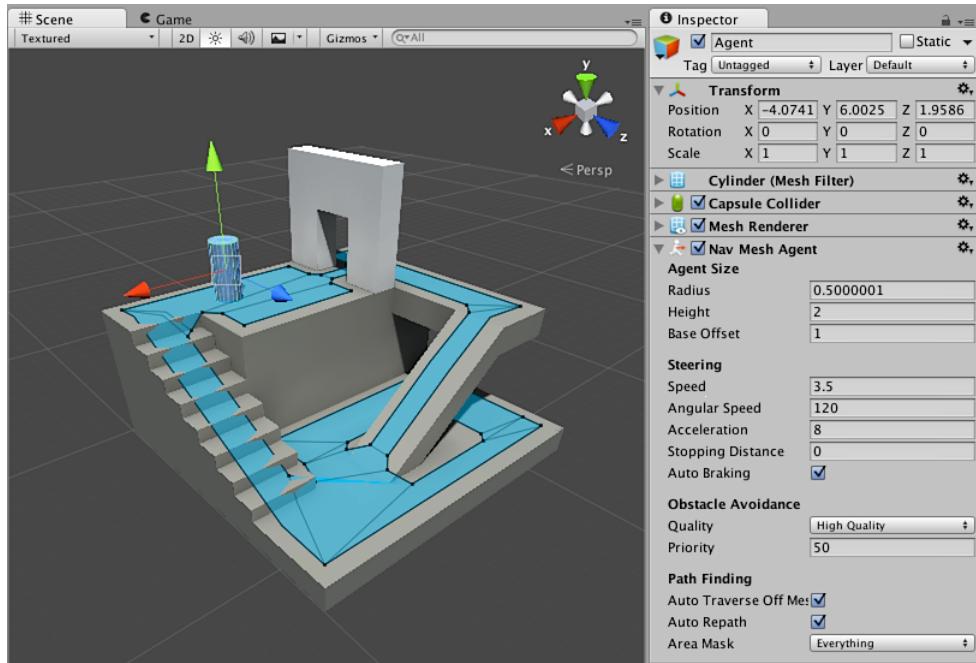


FIGURE 2.1 – Exemple d’application du composant *NavMesh* dans Unity. L’agent se déplace à l’intérieur de la zone bleue.

déplacement) et un algorithme de recherche de plus court chemin est utilisé lorsqu’on demande l’ordre à une unité de se déplacer dans le NavMesh.

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 entrer en collisions, ces dernières étant détectées 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, car l’unité peut se déplacer uniquement dans les zones qui lui sont autorisées.

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, et doit prendre en compte la possibilité que l’unité soit bloquée. De ce fait, l’utilisation du composant NavMesh n’est pas judicieux. En effet, ce composant permet de calculer le plus court chemin vers une destination. Cependant, si la destination n’est pas atteignable, le composant NavMesh calcule le point le plus proche dans les zones autorisées de l’unité. Or, cette opération était faite de manière automatique, 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 déterminant 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, on a enfin pu créer notre premier comportement qui consistait à dire à l'unité de bouger si elle n'était pas bloquée.

```

1     void Update()
2     {
3         PerceptStructure[] listePercepts = GetComponent<UnitManager>().GetComponent<
4             PerceptManager>()._percepts;
5         if (!listePercepts[0]._percept._value) // Percept bloquée? faux
6             _actions[0].Do(); // Move
7     }

```

Listing 2.3 – Extrait du code du script Brain.cs première version.

Mais on peut constater que cela nous obligé à écrire l'architecture de subsomption correspondant au comportement en brut, ce qui nuisait à notre besoin de généricité. On s'est donc attardé sur la conception des instructions de l'unité.

Celles-ci furent constituées d'une suite de conditions (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 à un exemple concret de la gestion d'une instruction, nous allons expliquer comment fonctionnait le champ de vision d'une unité et le calcul des objets "vus" par celle-ci. Le GameObject Unit a un composant Collider sphérique, une sphère tout autour de lui. Ce Collider détecte 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.

```

1     void OnTriggerStay(Collider other)
2     {
3
4
5         float h = GetComponent<UnitManager>()._stats._heading * Mathf.Deg2Rad;
6         Vector3 A = new Vector3(Mathf.Cos(h), 0, Mathf.Sin(h)).normalized * _distance;
7         Vector3 B = (other.transform.position - transform.position).normalized *
8             _distance;
9         float angle = Vector3.Angle(A, B);

```

```

10         if (angle <= _angle && !_listOfCollision.Contains(other.gameObject))
11     {
12
13         _listOfCollision.Add(other.gameObject);
14     }
15     else if (angle > _angle && _listOfCollision.Contains(other.gameObject))
16     {
17         _listOfCollision.Remove(other.gameObject);
18     }
19 }
```

Listing 2.4 – Extrait du code du script Sight.cs première version.

Pour la mise à jour de la liste des objets vus par l'unité, on procède comme suit :

- On récupère la liste des objets présents dans la sphère.
- Pour tous les objets présents dans la sphère, si celui-ci a sa position dans l'angle de champ de vision de l'unité, on l'ajoute à la liste des objets vus.

```

1 public class Instruction : MonoBehaviour {
2
3     [SerializeField]
4     PerceptStructure[] _listePerceptsVoulus;
5     [SerializeField]
6     Action _action;
7
8     public bool verify()
9     {
10         PerceptStructure[] listePerceptsUtilisables = GetComponent<UnitManager>().
11         GetComponent<PerceptManager>()._percepts;
12         bool verifie = true;
13         foreach(PerceptStructure p in _listePerceptsVoulus)
14         {
15             foreach(PerceptStructure p2 in listePerceptsUtilisables)
16             {
17                 if(p._name.Equals(p2._name))
18                 {
19                     verifie = p2._percept._value;
20                 }
21             }
22             if (!verifie)
23             {
24                 break;
25             }
26         }
27         if (verifie)
28     }
```

```

29             _action.Do();
30             return true;
31         }
32
33
34     return false;
35 }
36 }
```

Listing 2.5 – Classe Instruction du fichier Instruction.cs première version.

Cet exemple montre l'exécution d'une instruction demandant à un agent de récupérer une ressource s'il l'a perçoit, symbolisée par les classes PerceptRessource et ActionPick : Pour cette instruction on a donc accès à son percept et son action, la vérification des percepts se fait de la manière suivante :

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

```

1   override public void update()
2   {
3       bool res = false;
4       foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
5       {
6           if (gO && gO.GetComponent<ItemHandler>())
7           {
8               GetComponent<Stats>().SetTarget(gO);
9               res = true;
10              break;
11          }
12      }
13      _value = res;
14  }
```

Listing 2.6 – Extrait du code du script PerceptRessource.cs première version.

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 2.7 – 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).

On vient de voir la récupération d'une ressource par une unité, arrêtons-nous un instant sur le concept et la représentation d'une ressource dans le jeu. Sur Unity, un Prefab est une sauvegarde d'un GameObject avec ses composants et ses propriétés afin que celui-ci puisse être utilisé autant de fois qu'on le souhaite sans avoir à le reconfigurer.

Du point de vue de sa modélisation, une ressource est représentée par une sphère contenant une autre plus petite en son centre. Elle détient également un système à particules permettant de générer ces dernières, tournant autour du centre de la sphère la rendant plus dynamique.

Pour sa représentation informatique, une ressource est un Prefab détenant un script ItemHandler, uniquement composé d'un attribut de type Objet.

Le script Objet a donc plusieurs attributs ce qui permet la création d'objet de façon générique utilisable dans le gameplay :

- Son nom : string _name.
- Sa valeur : int _value.
- Sa taille : int _size.

- Son coût : int _cost.
- Sa représentation physique : GameObject _gameObject.
- Son script, contenant une unique fonction use(GameObject unit), qui permet de définir la manière dont celui-ci est utilisé par une unité : ItemScript _itemScript.

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 types "delegate" propre au C# ont une déclaration semblable à une signature de méthode. Elle a une valeur de retour et un nombre quelconque de paramètres de type quelconque. Nous les avons donc utilisées afin d'encapsuler les méthodes correspondant aux actions et aux percepts et ce afin d'éviter la création de multiples classes.

```

1  public abstract class Percept : MonoBehaviour
2  {
3      public delegate bool Listener();
4      public Dictionary<string, Listener> _percepts = new Dictionary<string, Listener>()
5      ;
6  }

```

Listing 2.8 – Extrait du code du script Percept.cs

A la suite de ce changement, la classe Percept s'est dotée d'un dictionnaire composée d'une string comme clef et d'un *Listener* comme valeur. 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.

L'intérêt d'avoir mis en place un dictionnaire comportant des strings en clefs permet notamment d'obtenir la liste des actions possible de l'unité.

```

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 2.9 – Extrait du code du script PerceptUnit.cs

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

Listing 2.10 – Extrait du code du script ActionUnit.cs

Hiérarchisation des actions et des percepts. Pour permettre d'éviter la surcharge des classes *Actions* et *Percepts* à destination de plusieurs type d'unités mais comportant plusieurs éléments en commun, nous avons mis en place une hiérarchisation de ces classes.

```
1  public abstract class Action : MonoBehaviour
2  {
3      /*
4      * Cette classe abstraite permet de créer de nouvelle action.
5      * Pour cela, créer une classe qui dérive de celle-ci.
6      */
7      public delegate void Act();
8      public Dictionary<string, Act> _actions = new Dictionary<string, Act>();
9      public abstract void InitAction(); ///
10 }
```

Listing 2.11 – Code du script Action.cs

```
1  public class ActionCommon : Action
2  {
3
4      void Start(){
5          InitAction();
6      }
7
8      public override void InitAction()
9      {
```

```

10         _actions["ACTION_IDLE"] = delegate () { [...] };
11         _actions["ACTION_GIVE_RESSOURCE"] = delegate () { [...] };
12         _actions["ACTION_HEAL"] = delegate () { [...] };
13     }
14 }
```

Listing 2.12 – Code du script ActionCommon.cs dérivant de la classe Action.cs

```

1  public class ActionUnit : ActionCommon
2  {
3      void Start() {
4          InitAction();
5      }
6
7      public override void InitAction() {
8          base.InitAction(); // IMPORTANT : Permet de récupérer les percepts de la
9          // classe mère
10         _actions["ACTION_MOVE"] = delegate () { [...] };
11         _actions["ACTION_RANDOM_MOVE"] = delegate () { [...] };
12         _actions["ACTION_TURN_AROUND_MOVE"] = delegate () { [...] };
13         _actions["ACTION_PICK"] = delegate () { [...] };
14         _actions["ACTION_BACK_TO_BASE"] = delegate () { [...] };
15     }
16 }
```

Listing 2.13 – Code du script ActionUnit.cs dérivant de la classe ActionCommon.cs

Les listings 2.11, 2.12 et 2.13 font references à des listes d'actions dans différentes scripts. Lorsqu'on attache le script *ActionUnit* à une unité, les actions possibles pour cette dernière sont toutes les actions qui sont dans la hiérarchie.

Ainsi, les actions "ACTION_IDLE", "ACTION_GIVE_RESSOURCE" et "ACTION_HEAL" sont disponibles ayant pour script *ActionUnit* et pour toutes les classes dérivant de celle-ci.

Ci-dessous, voici la première représentation du cerveau de l'unité. Le "Brain" a pour but de gérer le comportement de l'unité, c'est-à-dire de vérifier les instructions de celle-ci et de lui donner l'ordre de faire l'action correspondante ce tour. On peut donc voir le mécanisme présenté plus haut, la confirmation de la présence d'une action dans l'instruction ainsi que la vérification des percepts nécessaires à son exécution.

```

1  public class Brain : MonoBehaviour
2  {
3      public ActionStructure[] _actions;
4      public InstructionScriptable[] _instructions;
```

```

5
6     void Update()
7     {
8         foreach(InstructionScriptable instr in _instructions)
9         {
10            Action actionPossible = this.actionPossible(instr._stringAction);
11            if (actionPossible != null && instr.verify(GetComponent<UnitManager>().
12 GetComponent<PerceptManager>()._percepts))
13            {
14                actionPossible.Do();
15                break;
16            }
17        }
18    }
19
20    public Action actionPossible(string stringAct)
21    {
22        Action presence = null;
23        foreach (ActionStructure actStruc in _actions)
24        {
25            if (actStruc._name.Equals(stringAct))
26            {
27                presence = actStruc._action;
28            }
29        }
30        return presence;
31    }
32}

```

Listing 2.14 – Première version de la classe Brain.cs

Afin de mener à bien le développement du comportement des unités, on a aussi dû créer une carte plus légère et plus petite que celle de l'ancienne génération nous permettant de faciliter nos tests.

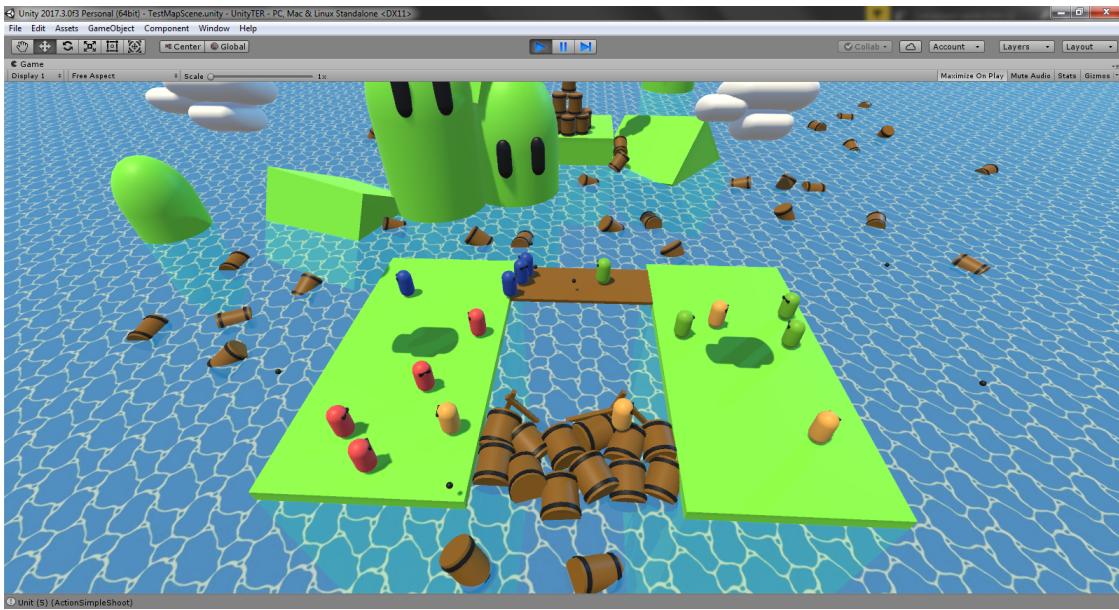


FIGURE 2.2 – Aperçu de la première carte de test.

Sur cette scène a été introduit le déplacement de la caméra principale (MainCamera) calculé en fonction de la position moyenne des unités. Cela permet de donner un effet dynamique à la scène. Ce mécanisme est présent dans le script "FollowCamera.cs" attaché à la caméra principale et est toujours d'actualité pour chacune de nos MainCamera de scène de jeu.

```

1 void FixedUpdate ()
2 {
3     Vector3 res = Vector3.zero;
4     GameObject[] units = GameObject.FindGameObjectsWithTag("Unit");
5
6     foreach (GameObject go in units)
7     {
8         res += go.transform.position;
9     }
10    res *= 1f / (units.Length + 1);
11    currentTarget += (res - currentTarget) * Time.deltaTime * _speed;
12    transform.LookAt(currentTarget);
13 }
```

Listing 2.15 – Extrait du code du script FollowCamera.cs

Cette phase de démarrage a duré environ un mois, durant lequel nous avons pu rendre compte chaque semaine à notre encadrant. Durant celle-ci, on a donc agrémenté notre moteur de jeu en suivant les

directives de M. Ferber, en nous basant sur notre expérience d'utilisation de Unity pour valider les différents ajouts demandés.

2.2.2 Intégration dans le projet.

Lorsque le moteur nous a paru complet, nous l'avons intégré au projet. Cette phase a été source de problèmes du fait que l'ancien projet était encore présent et que de nombreux conflits persistés.

Le lien avec l'interpréteur

Le plus grand défi de l'intégration était de rendre possible la liaison avec l'interpréteur en place, créé par l'équipe Interpréteur du projet. Nous avons donc travaillé en étroite collaboration avec cette équipe afin de comprendre le fonctionnement de l'interpréteur pour que nous puissions faire les modifications nécessaires.

Suite à nos échanges, il a été convenu de modifier la classe Instruction qui servira de pont entre l'interpréteur et le moteur.

```
1  public class Instruction {
2      public string[] _listeStringPerceptsVoulus;
3      public MessageStruct[] _stringActionsNonTerminales;
4      public string _stringAction;
5
6      [...]
7      public Instruction(string[] ins, MessageStruct[] actionsNonTerminales, string act)
8      {
9          _stringAction = act;
10         _listeStringPerceptsVoulus = ins;
11         _stringActionsNonTerminales = actionsNonTerminales;
12     }
13 }
14 [...]
15 }
```

Listing 2.16 – Extrait du code du script Instruction.cs

Cette classe est composée de l'attribut *_listeStringPerceptsVoulus* correspondant à la liste des labels des percept de l'instruction, de l'attribut *_stringActionsNonTerminales* comportant la liste des labels des actions non terminales et d'une string *_stringAction* qui est le label de l'action à faire. Les actions non terminales ne sont, à ce stade, pas encore implémentées, nous décidons tout de même de les laisser pour les implémenter ultérieurement.

Lors de la lecture d'un comportement créé par l'éditeur de comportement intégré dans le jeu, l'interpréteur nous renvoie une instance de cette classe Instruction.

```

1 Instruction (
2     Percepts = [PERCEPT_ENEMY, PERCEPT_LIFE_MAX] ,
3     ActionNT = [] ,
4     Action = ACTION_FIRE )

```

Listing 2.17 – Exemple d'une instruction d'un comportement que peut renvoyer l'interpréteur.

L'exemple montré au listing 2.17 correspond au comportement d'une unité et lui demandera de tirer si l'unité peut percevoir un ennemi, et que sa vie est au maximum. Les actions non-terminales n'étant pas prises en charge à ce moment du développement, la liste est vide. Un comportement n'est donc rien d'autres qu'une liste d'instructions.

Pour garder l'idée d'une structure de subsomption, la liste des instructions est ordonnée de telle sorte que la première instruction dans cette liste est la première à être examinée et son action est exécutée si ces percepts sont activés.

Le lien avec l'éditeur de comportement.

Pour permettre au joueur de faire un comportement avec des éléments autorisés pour l'unité en question, il nous a fallu transmettre à l'équipe s'occupant de l'éditeur graphique les actions et percepts autorisés pour chaque type d'unités.

Les clefs des dictionnaires. Ainsi chaque unité doit contenir une classe dérivant d'*Action* et de *Percept*. Ainsi dans l'objet *GameManager*, on renseigne les unités dans la section "List Unit Game Object". Cette section permet de capturer les différentes actions et percepts des différents unités et de les consigner dans un fichier.

Ce fichier est repris par l'éditeur de comportement pour créer les pièces de puzzles en fonction des unités.

2.2.3 Améliorations des fonctionnalités.

Ajout de fonctionnalités des instructions.

Un des enjeux du moteur était de pouvoir rendre les instructions plus intéressantes pour le joueur. Il a donc fallu améliorer les instructions qui étaient composées d'une simple liste de percepts, d'actions non terminales et d'une action en rajoutant des fonctionnalités.

```

1 Behavior [LIGHT] = {
2     1. Instruction (   Percepts = [PERCEPT_ENEMY, PERCEPT_LIFE_MAX] ,
3                     ActionNT = [] ,

```

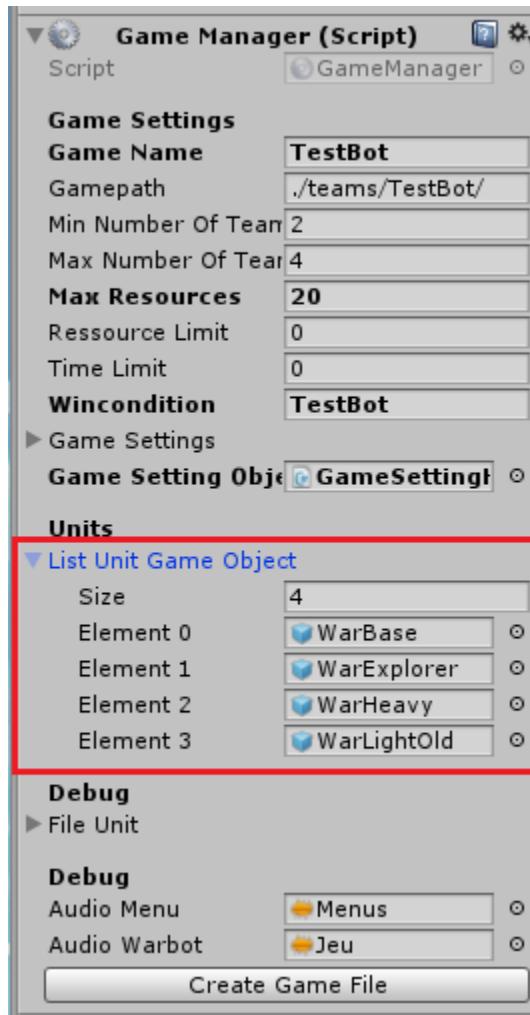


FIGURE 2.3 – Visualisation dans l'*Inspector* d’Unity des unités présente dans le GameManager.cs

```

4      Action = ACTION_FIRE ) ,
5  2. Instruction (   Percepts = [PERCEPT_ENEMY, NOT_PERCEPT_LIFE_MAX] ,
6      ActionNT = [MESSAGE_HELP(@ HEAVY)] ,
7      Action = none) ,
8  3. Instruction (   Percepts = [ NOT_PERCEP_LIFE_MAX] ,
9      ActionNT = [] ,
10     Action = ACTION_HEAL ) ,
11 4. Instruction (   Percepts = [] ,
12      ActionNT = [] ,
13      Action = ACTION_MOVE ) }
```

Listing 2.18 – Exemple d’un comportement que peut renvoyer l’interpréteur.

Le comportement défini en listing 2.18 comporte plusieurs instructions résumant les possibilités du moteur.

1. Négation. La plus grande amélioration que l'équipe de Game Design nous avait demandé d'apporter au moteur sur le plan des instructions était de mettre en place la négation des percepts.

Pour rappel, un percept est une fonction anonyme qui renvoie un booléen. La composition de plusieurs percepts donne l'illusion d'un tableau de bord, permettant à l'unité de se faire une idée de son environnement. Cependant, les instructions fonctionnent actuellement en utilisant une liste conjonctive de percepts. En d'autres termes, tous les percepts d'une instruction doivent être activés pour pouvoir exécuter l'action de l'instruction.

Il nous fallait donc trouver un moyen pour rendre possible la négation des percepts. Ainsi, en travaillant de concert avec l'équipe Interpréteur, nous avons convenu d'un préfixe *NOT_* que l'on concatène au label des percepts, comme vu aux instructions 2. et 3. du listing 2.18.

```
1 PERCEPT_LIFE_MAX; //Activé lorsque "TRUE"  
2 NOT_PERCEPT_LIFE_MAX; //Activé lorsque "FALSE"
```

Listing 2.19 – Un percept et son inverse.

En utilisant cette manière, on permet à l'éditeur de comportement de transcrire le choix du joueur (négation ou pas) via un fichier XML que l'interpréteur peut traduire en *Instruction* pour notre moteur.

```
1 public class Brain : MonoBehaviour  
2 {  
3     [...]  
4     bool Verify(Instruction instruction)  
5     {  
6         bool flag = true;  
7         foreach (string percept in instruction._listeStringPerceptsVoulus)  
8         {  
9             if (!( _componentPercepts._percepts.ContainsKey(percept.Replace("NOT_", ""))  
10                && (percept.Contains("NOT_") ^ _componentPercepts._percepts[percept.  
11                    Replace("NOT_", "")]()) )) { flag = false; }  
12         }  
13     }  
14     [...]  
15 }
```

Listing 2.20 – Extrait du code du script Brain.cs

Pour rendre le moteur compatible avec la négation des percepts, on a du modifier la fonction *Verify* (listing 2.20) du script Brain.cs. Cette fonction vérifie si une instruction donné en paramètre est activé. On regarde ainsi si tout les percept sont activés.

Ainsi, une instruction est activé si :

- Il existe le percept dans la liste des percepts de l'unité. (En enlevant le préfixe *NOT_*).
- Le percept ne contient pas le préfixe *NOT_* et le percept renvoie "TRUE".
- Le percept contient le préfixe *NOT_* et le percept renvoie "FALSE".

2. Actions non-terminales. Pour permettre une plus grande liberté dans la conception des comportements des unités, il a était rapidement nécessaire de rajouter des actions dites *non-terminales*. Ces actions sont effectuées au même titre que les actions terminales à la différence près que celle-ci n'empêche pas d'exécuter une autre instruction située hiérarchiquement plus bas dans l'architecture de subsomption si l'instruction ne comporte pas d'action terminant le tour de l'unité.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public abstract class ActionNonTerminal : MonoBehaviour
6  {
7      public delegate void ActNonTerm();
8      public Dictionary<string, ActNonTerm> _actionsNT = new Dictionary<string,
ActNonTerm>();
9      public abstract void InitActionNonTerminal();
10     public string _messageDestinataire;
11     public Message _tmpMessage;
12 }
```

Listing 2.21 – Code du script ActionNonTerminal.cs

Comme pour les actions terminales et les percepts, les actions non-terminales sont constituées d'un dictionnaire composé de chaîne de caractères comme clef et un delegate *ActNonTerm* correspondant à la méthode anonyme de l'action non-terminal.

```

1      _actionsNT [ "ACTN_HEADING_RANDOM" ] = delegate ()
2      {
3          GetComponent<Stats>().SetHeading (Random.Range(0f,360f));
4      };
```

Listing 2.22 – Exemple d'une action non-terminale

Dans le listing 2.22, on peut voir l'exemple d'une action non-terminale. Fonctionnant comme une action classique, l'action non-terminale "*ACTN.HEADING.RANDOM*" permet à une unité de faire une rotation aléatoire.

3. Mouvement tactique. Comme définie dans le listing 2.18, l'instruction 2. permet de faire une action non-terminale même sans d'action terminale. Ainsi on a la possibilité de réaliser des mouvements plus tactiques sans terminer le tour.

Lorsque les percepts de l'instruction 2 sont activés, cela déclenche l'action non terminale en ligne 6 qui envoie un message d'aide aux unités de type *Heavy*. Mais comme il n'y a pas d'action terminant le tour, on continue à regarder les autres instructions tant qu'il n'y a pas d'action terminale à effectuer.

4. Instruction par défaut. Lorsqu'une instruction ne comporte pas de percepts, cette instruction est, par définition toujours valide. Cela permet, entre autre, de créer des cas par défaut que l'on peut mettre à la fin d'un comportement. L'action de cette instruction est alors toujours exécutée si aucune autre action ne peut l'être.

5. Cas exceptionnel. Dans un comportement, il peut arriver que les conditions de toutes les instructions soient fausse, empêchant ainsi l'exécution d'une action. Dans ce cas précis, c'est le moteur qui choisi l'action à effectuer. Actuellement, l'unité exécute l'action *ACTION_IDLE*.

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. Cela nuisait au bon déroulement de la partie, atteignant parfois deux minutes de temps de chargement lorsque l'on voulait générer 200 unités. Pour résoudre ce problème, nous avons étudié plusieurs pistes.

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 propre à Unity ("Update") qui est appelée à chaque calcul d'image. Ainsi, pour chaque unité présente sur la scène, Unity calcule l'action qu'elle doit faire. Cependant, certaines actions nécessitent plus de temps en fonction de leur 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 2.23).

```
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 2.23 – Code du script TurnManagerScript.cs

Ce script permet 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 près 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 unité	0.49 secondes
1 unité	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 2.24 – 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 définie au listing 2.24. 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 6.1) permet la gestion de plusieurs *Pools*, et dans notre cas, nous utilisons 3 *Pools* : *Light*, *Heavy* et *Explorer*.

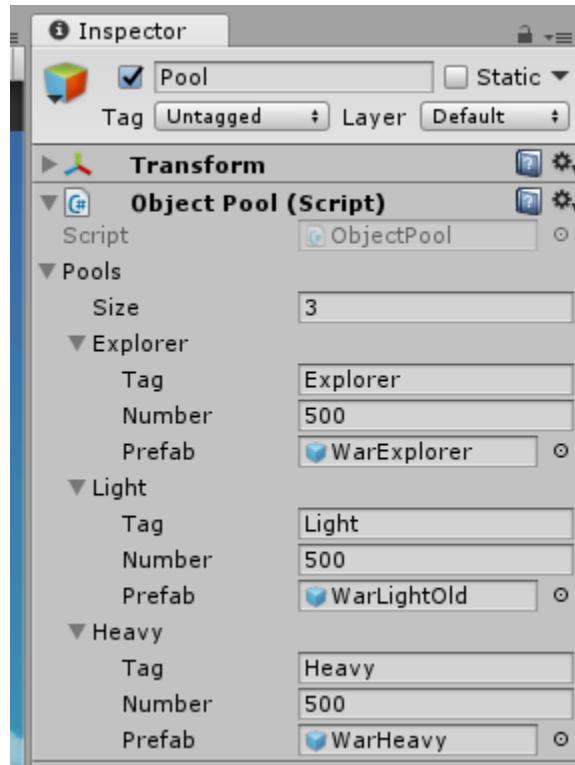


FIGURE 2.4 – Visualisation dans l'*Inspector* d'Unity des *Pools* du script ObjectPool.cs

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 certains composants. 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 l'environnement se 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 argument.

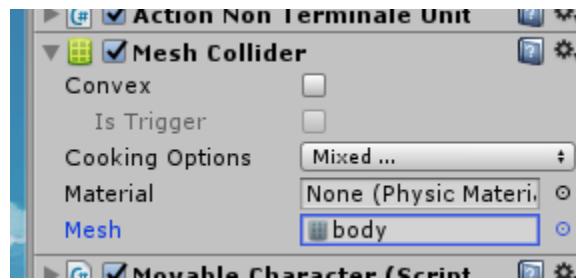


FIGURE 2.5 – Un composant *MeshCollider*

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 collisions. Ce composant est défini par un pavé ce qui réduit considérablement les calculs pour connaître les éventuelles collisions avec celui-ci.

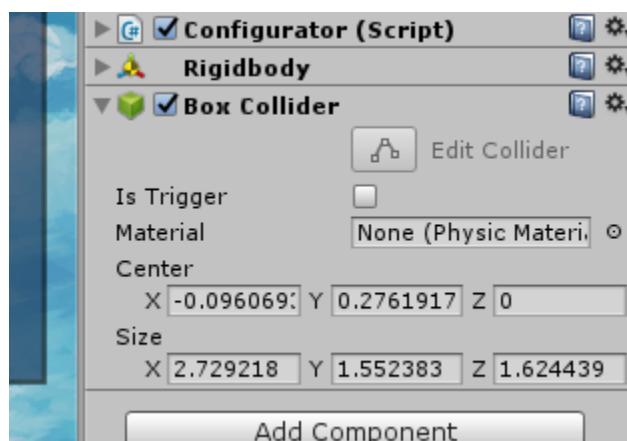


FIGURE 2.6 – Un composant *BoxCollider*

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

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 reconnaissaient pas l'environnement et traversaient le décor. Pour résoudre ce problème, nous avons donc défini 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 à un autre objet 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.
12 point);
13                 float b = GetComponent<Stats>().GetHeading();
14                 float A = Mathf.Abs(a - b);
15                 float B = Mathf.Abs(360 + Mathf.Min(a, b) - Mathf.Max(a, b));
16                 if (Mathf.Min(A, B) < 90f)
17                 {
18                     collisionObject = other.transform.gameObject;
19                     break;
20                 }
21             }
22         }
23     [...]
24 }
```

Listing 2.25 – 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".

2.2.4 Fonctionnalités finales.

Création d'un jeu.

Création des unités. La création des unités nécessite 6 scripts au minimum :

- Brain
- Stats
- Percept
- Actions
- ActionNonTerminales
- Messages

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 équipes.

2.3 Modélisation 3D, sons et animation.

2.3.1 Les unités Warbot.

Afin de pouvoir tester les fonctionnalités que nous avons mis en place, nous avons créé des unités. On a ainsi du concevoir des modèles 3D pour les unités en utilisant le logiciel libre de droit *Blender*.

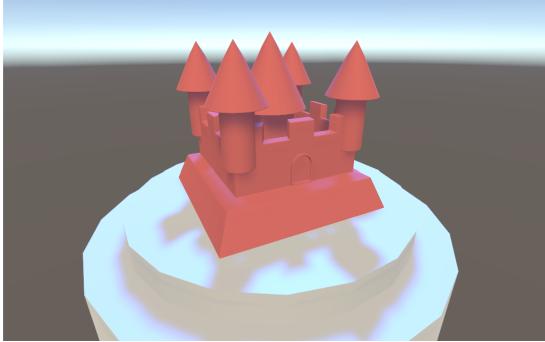


FIGURE 2.7 – L’unité Base de WarBot.

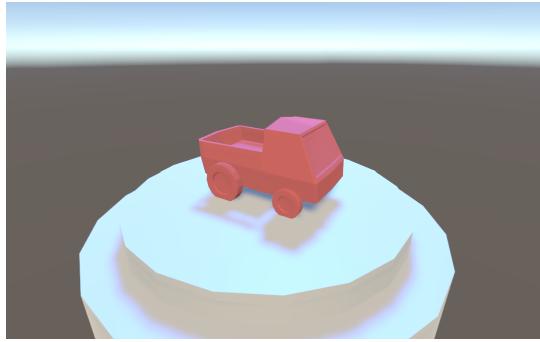


FIGURE 2.8 – L’unité Explorer de WarBot.

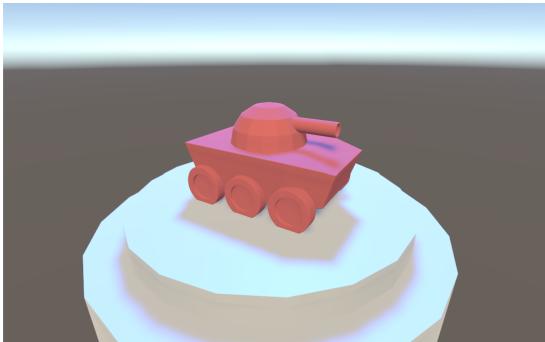


FIGURE 2.9 – L’unité Light de WarBot.

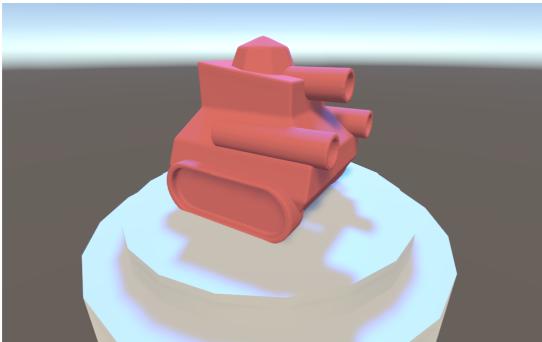


FIGURE 2.10 – L’unité Heavy de WarBot.

2.3.2 Les différentes cartes.

Pour tester les unités, nous avons conçu 5 différentes cartes.

- *Simple* : Carte basique ne comportant pas d’obstacle. C’est une île rectangulaire entourée d’eau.
- *Mountain* : Une île plus élaboré ayant des bords plus irréguliers comportant une chaîne de montagne au sud-est de la carte.
- *Desolate* : Un désert avec des obstacles au centre et des limites de cartes plus anguleux. C’est une carte utilisé pour tester les collisions avec les objets.
- *Plain* : Une carte de taille plus petite entouré de barrière surveillé par des collines aux grands yeux noires.
- *Garden* : Une île flottante aux dessus des nuages. La zone de jeu est entourée par de grandes pierres et un parterre de fleurs aux centre.

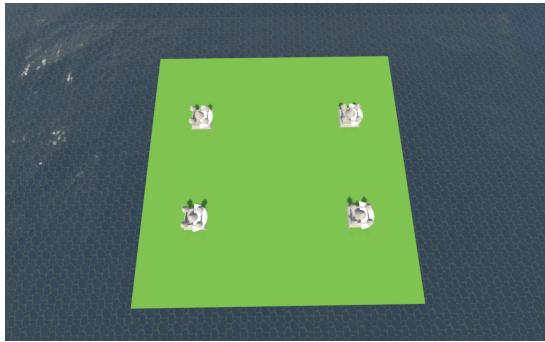


FIGURE 2.11 – L’unité Base de WarBot.



FIGURE 2.12 – L’unité Explorer de WarBot.



FIGURE 2.13 – L’unité Light de WarBot.

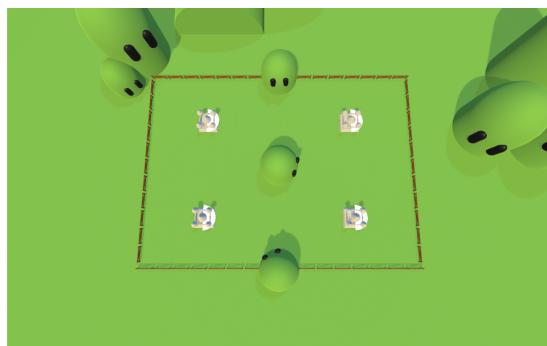


FIGURE 2.14 – L’unité Heavy de WarBot.



FIGURE 2.15 – L'unité Light de WarBot.

2.3.3 Animations.

Différentes animations ont été réalisées à l'aide de systèmes à particules afin d'améliorer le rendu visuel durant les combats entre unités.

Les projectiles. Au moment de la destruction d'un projectile, qui intervient lorsque celui-ci touche une unité ou après un laps de temps suivant son lancement, celui-ci provoque une explosion de particules. Les Prefab "HeavyBullet" et "LightBullet" possède pour ce faire un script "BulletScript.cs". Celui-ci instancie le GameObject d'animation correspondant au projectile au moment souhaité. Ce GameObject peut bien sûr être modifier facilement.

Les unités. Les autres animations concernent les unités qui détiennent toutes quatre animations différentes :

- L'unité se met à fumer légèrement lorsque ses points de vie sont entre 75% et 100%.
- Une fumée plus épaisse se déclenche lorsque ses points de vie sont entre 50% et 75%.
- L'unité est en feu lorsque ses points de vie sont entre 25% et 50%.
- Pour finir elle explose en un amas de particule lorsque ses points de vies atteignent 0. Les GameObject concernant ces différentes animations sont présents au sein des statistiques de chaque unité et peuvent être remplacés simplement.

```

1 [ Header( " Effects" ) ]
2 public GameObject _smallSmoke;
3 public GameObject _largeSmoke;
4 public GameObject _fire;
5 public GameObject _explosion;
6
7 private GameObject _currentEffect;
8 void Start()
9 {
10     [...]
11     _smallSmoke.SetActive( true );
12     _largeSmoke.SetActive( true );
13     _fire.SetActive( true );
14     _explosion.SetActive( true );
15 }
16 [...]
17 void SetSmoke()
18 {
19     if ( _currentEffect )
20     {
21
22     }
23     if ( _health > _maxHealth * 0.75 )
24     {
25         Destroy( _currentEffect );
26     }
27     else if ( _health > _maxHealth * 0.50 )
28     {
29         if ( _currentEffect )
30         {
31             Destroy( _currentEffect );
32         }
33         _currentEffect = Instantiate( _smallSmoke, transform.position, Quaternion.
identity );
34     }
35     else if ( _health > _maxHealth * 0.25 )
36     {
37         if ( _currentEffect )
38         {
39             Destroy( _currentEffect );
40         }
41         _currentEffect = Instantiate( _largeSmoke, transform.position, Quaternion.
identity );
42     }
43     else if ( _health > 0 )
44     {
45         if ( _currentEffect )
46         {
47             Destroy( _currentEffect );

```

```

48         }
49         _currentEffect = Instantiate(_fire, transform.position, Quaternion.
50         identity);
50     }
51 }
```

Listing 2.26 – Extrait du code de Stats.cs

2.3.4 Sons.

Les différents sons et bruitages présents dans le jeu ont été récupérés à partir de banques de sons libres de droit sur internet.

Différents fonds musicaux et effets sonores sont audibles tout au long de l'utilisation du jeu.

Fonctionnement. Pour utiliser le son dans le moteur de jeu Unity, il faut passer par deux composants :

- Audiosource : un AudioSource est attaché à un GameObject pour lire les sons dans un environnement 3D, on peut lire un seul clip audio en utilisant Lecture, Pause et Stop. On peut également ajuster son volume pendant la lecture en utilisant la propriété de volume (expliqué plus en détail dans la partie Interface Graphique). Plusieurs sons peuvent être lus sur une source audio en utilisant PlayOneShot.
- AudioListener : un AudioListener agit comme un périphérique de type microphone. Il reçoit une entrée de n'importe quelle AudioSource dans la scène et joue des sons à travers les haut-parleurs de l'ordinateur. Pour la plupart des applications, il est plus logique d'attacher l'AudioListener à la caméra principale. Il ne peut y avoir qu'un seul AudioListener par scène.

Les fond sonores. Deux ambiances musicales sont proposées sur l'ensemble du jeu.

La première est activée dès le démarrage du jeu. Elle est jouée en boucle sur l'écran principal ainsi que sur l'éditeur sans interruption au passage de scène en scène.

La seconde est la musique en cours de jeu, elle démarre à chaque nouveau lancement de partie et est elle aussi jouée en boucle.

Pour ce faire, le composant AudioSource utilisé est attaché au GameManager dont le script "GameManager.cs" a en attribut les deux AudioClip correspondant aux fonds sonores. Un AudioClip contient les données audio utilisées par un AudioSource.

```

1   public AudioClip audioMenu;
2   public AudioClip audioWarbot;
```

Listing 2.27 – Extrait du code de GameManager.cs

Ci-dessous un exemple de changement de piste audio par appuie sur le bouton "Quit" sur l'écran de jeu. La piste audio correspondant aux menu est jouée.

```
1     public void QuitToMenu()
2     {
3         SceneManager.LoadScene(0);
4         audioSource.clip = gameManager.GetComponent<GameManager>().audioMenu;
5         audioSource.Play();
6     }
```

Listing 2.28 – Extrait du code de HUDmanager.cs

Les projectiles. Les deux projectiles, celui lancé par les chars légers et par les chars lourd produisent des sons lors de leur création et de leur destruction. Ils ont chacun leur propre duo de sons et utilise l' AudioSource du GameManager pour les jouer. Comme se sont des sons ponctuels, on utilise la fonction PlayOneShot de l' AudioSource afin de les jouer.

```
1     public AudioClip _shotSongStart;
2     public AudioClip _shotSongFinish;
3     AudioSource audioSource;
4     void Start()
5     {
6         audioSource = GameObject.Find("GameManager").GetComponent<AudioSource>();
7         if (_shotSongStart != null) audioSource.PlayOneShot(_shotSongStart);
8     }
```

Listing 2.29 – Extrait du code de BulletScript.cs

Les unités. Chaque unité a un son correspondant à sa destruction, ceux-ci sont donc jouer à la mort de l'unité. Ces sons sont gérés de manière identique à celle utilisée pour les projectiles. Leurs lecture se fait dans le script "Stats.cs", en même temps que la mise à jour des points de vies.

```
1  if (_health <= 0)
2  {
3      if (_currentEffect)
4      {
5          Destroy(_currentEffect);
6      }
7      _currentEffect = Instantiate(_explosion, transform.position, Quaternion.
identity);
8      if (_dieSong != null) audioSource.PlayOneShot(_dieSong);
9      Destroy(_currentEffect, 1.5f);
10     Destroy(gameObject);
11 }
```

Listing 2.30 – Extrait du code de Stats.cs

2.4 Amélioration possible

L'objectif de ce projet était de refaire rendre générique la version Warbot Unity réalisée l'an dernier. Le but étant de faciliter la création sur le moteur de jeu Unity, de jeux de stratégie basés sur le paradigme de la programmation orientée agent afin de permettre un véritable apprentissage de ses différentes notions. En ce sens, notre objectif a été rempli, mais les améliorations possibles pour les années à venir sont multiples :

- perfectionner la gestion des collisions des unités.
- ajouter de nouvelles unités au mode Warbot (Tourelles, Ingénieur et RocketLauncher).
- créer de nouveaux percepts, de nouvelles actions terminales et non terminales.
- ajouter la possibilité de créer et supprimer des unités directement en jeu.
-

Chapitre 3

Partie Interface Graphique

3.1 Présentation

La partie Interface Graphique fait le lien entre l'utilisateur, et toutes les fonctionnalités existantes dans le moteur. Elle 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.

3.2 Etude de l'ancienne interface

L'ancienne interface, sur laquelle nous avons du baser notre travail, se décomposait en trois parties.

- **Le menu principal**
- **L'éditeur de comportement**
- **La sélection d'équipes**

3.2.1 Menu Principal

Le menu principal, à première vue, est assez classique. Quatre options s'offrent à nous :

- Lancer une nouvelle partie
- Accéder à l'éditeur de comportement
- Quitter le jeu
- Activer / Désactiver le son

L'aspect purement graphique est cohérent, puisque le fond d'écran représente une scène de jeu. Nous pouvons donc apercevoir deux modèles 3D d'unités présentes en jeu, ainsi qu'une partie du décor sur mequel se déroulera le combat.



FIGURE 3.1 – Menu Principal, Ancienne Interface

3.2.2 Editeur de comportement

L'éditeur de comportement se décompose en deux écrans distincts :

1. Le menu Pause

Ce menu apparaît par défaut lors d'une tentative d'accès à l'éditeur de comportement si aucune équipe n'existe. Il permet donc de créer une nouvelle équipe, en entrant son nom.

De plus, il permet de revenir au menu principal, de lancer une nouvelle partie, de quitter le jeu, ou bien tout simplement de fermer ce menu pour accéder à l'éditeur de comportement.

Ce menu est toutefois "caché". Pour y accéder, il faut appuyer sur la touche "ECHAP" du clavier pour le voir s'afficher.

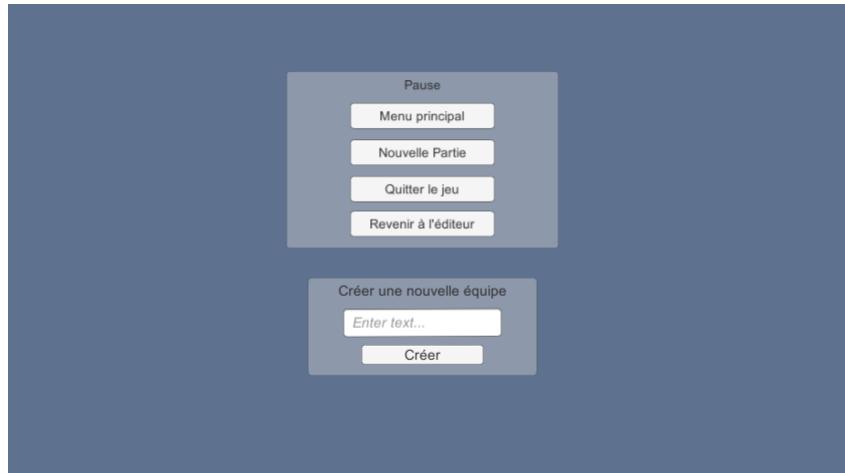


FIGURE 3.2 – Menu Pause, Ancienne Interface

2. L'éditeur en lui même

Nous avons maintenant devant nous un éditeur sobre. En haut à gauche se trouve le nom de l'équipe courante, ainsi que l'unité sur laquelle on va travailler.

Juste en dessous, une liste de primitives, utiles à la création de comportement.

Et enfin, à droite, la zone d'édition.

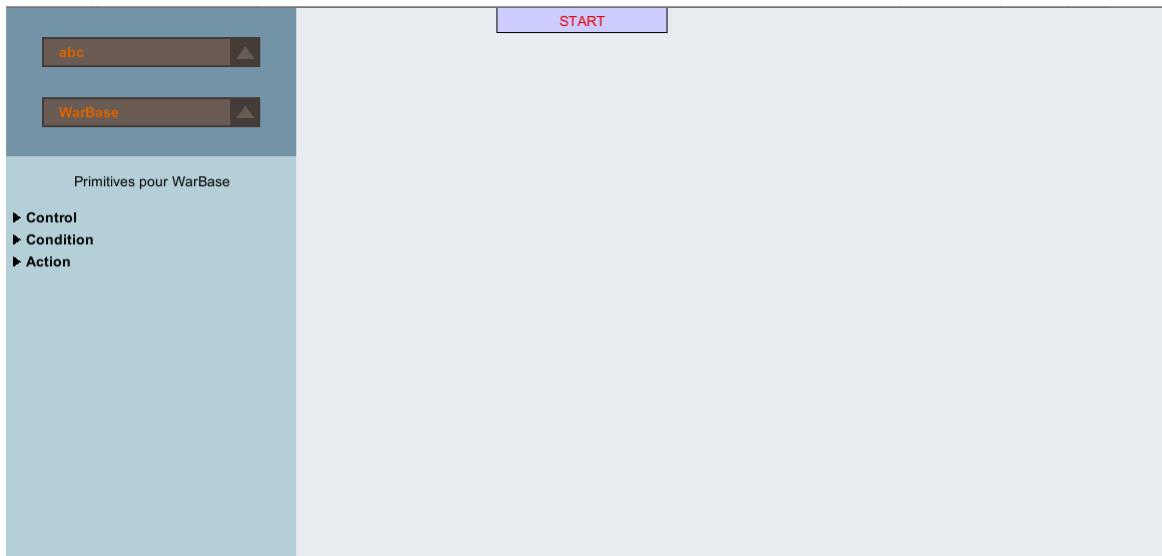


FIGURE 3.3 – Editeur de Comportement, Ancienne Interface

3.2.3 La sélection des équipes

Une fois le bouton "Nouvelle partie" cliqué, nous arrivons sur un nouvel écran, permettant de choisir les équipes qui vont participer, ainsi que le nombre de chacune de leurs unités qui commenceront directement en jeu.

Cet écran permet également de revenir au menu principal, ou bien tout simplement de lancer la partie. Cette étape supplémentaire demandé à l'utilisateur ne nous paraissait pas spécialement utile. L'écran ne propose rien d'autre que la gestion des paramètres de partie. L'espace utilisé est moindre pour justifier l'affichage d'un nouvel écran.

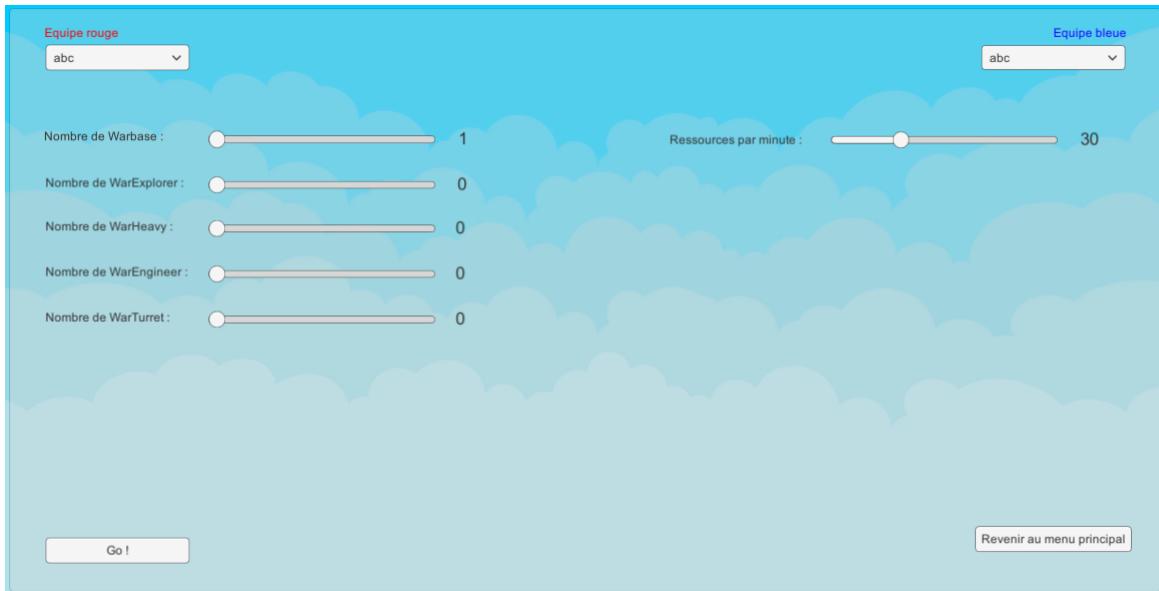


FIGURE 3.4 – Sélection des équipes, Ancienne Interface

3.2.4 Améliorations envisagées

Les fonctionnalités primordiales au bon fonctionnement du jeu sont disponibles :

- La sélection de l'équipe et de toutes les unités
- La liste des primitives concernant l'unité sélectionnée
- La description et le placement des primitives
- La création et l'édition de l'architecture de subsomption via l'aire de jeu

Ceci dit, nous souhaitions revoir quelques points.

Premièrement, nous voulions que tout soit le plus intuitif possible pour l'utilisateur. Il fallait que toutes les fonctionnalités proposées soient accessibles facilement. Par exemple, la création d'équipe n'est pas

évidente à trouver. Il faut passer par un menu "caché" (appuyer sur la touche "ESC" depuis l'éditeur de comportement) pour pouvoir y accéder. Nous voulions éviter ce genre de problèmes. Des petits soucis d'optimisations étaient également présents, tel que la difficulté d'aimanter deux blocs. Si on ne posait pas le bloc à un endroit précis, il disparaissait. Il fallait alors le sélectionner à nouveau dans la liste de primitives, et retenter l'opération. A force, cela peut vite devenir redondant.

Bien que l'éditeur soit fonctionnel, il nous paraissait assez austère, pas suffisamment "User Friendly". L'utilisateur passe la majeure partie de son temps sur cet écran. Il fallait donc qu'il soit le plus agréable possible à l'œil, pour ne pas lasser l'utilisateur, et lui donner envie de passer du temps dessus.

Enfin, l'absence totale de paramètres nous a semblé dommage. On ne peut qu'activer ou désactiver le son, mais pas régler le volume. L'utilisateur ne sait pas comment gérer les équipes qui vont participer, ainsi que leur nombre d'unités, avant d'avoir cliqué sur Nouvelle Partie. Ce comportement ne paraît pas intuitif. L'ajout d'une gestion des paramètres nous semblait donc indispensable.

Nous avons donc décidé de revoir l'organisation de l'interface depuis le début. Adapter cette interface n'aurait pas été justifié, car trop de changements étaient à prévoir, tant sur le point technique que graphique.

Nous souhaitions également proposer une cohésion graphique entre chaque écran, et le jeu lui-même.

3.3 Nouvelle Interface

Nous devions repartir d'une toute nouvelle interface, tout en s'inspirant de l'ancienne. Comme dit précédemment, nous voulions éviter au maximum les écrans intermédiaires qui n'étaient pas indispensables. Nous avons donc décidé de mettre en place deux écrans : le menu principal, et l'éditeur de comportement. Dans le menu principal se trouve une fenêtre "pop-up" dédiée à la configuration des paramètres. D'autres fenêtres "pop-up" peuvent apparaître, mais elles n'ont que pour but d'effectuer une vérification de l'action émise par l'utilisateur. Par exemple, confirmer la suppression d'une équipe, confirmer la fermeture du jeu, etc...

3.3.1 Menu Principal

Le menu principal donne accès aux options majeures du jeu. Ce menu doit contenir toutes les fonctionnalités dont l'utilisateur peut avoir besoin pour gérer le lancement d'une partie, sans avoir à accéder à l'éditeur de comportement. Cela comporte donc les paramètres de la partie, le choix des équipes, du nombre d'unités, de la carte, etc...

Passons en revue tous les éléments présents sur cet écran.



FIGURE 3.5 – Menu Principal, Nouvelle Interface

Lancer une Partie

Le bouton "Jouer" 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), et la carte de jeu. Un script rattaché à ce bouton va donc se charger d'aller récupérer toutes ces informations stockées dans les différents "GameObject" de notre scène.

```

1  public void StartGame()
2  {
3
4
5      nbPlayers = int.Parse(_numberplayerDropDown.GetComponent<Dropdown>().
captionText.text);
6      XMLWarbotInterpreter interpreter = new XMLWarbotInterpreter();
7
8      GameObject gameManager = GameObject.Find("GameManager");
9      string gamePath = Application.streamingAssetsPath + "/teams/" + gameManager.
GetComponent<GameManager>()._gameName + "/";
10
11     gameManager.GetComponent<TeamManager>()._teams = new List<Team>();
12
13     for (int i = 0; i < nbPlayers; i++)
14     {
15         Team team = new Team();
16         team._color = _DropDownList[i]._teamColor;
17         team._name = _DropDownList[i]._teamName // .Replace("_", " ");

```

```

18         team._unitsBehaviour = interpreter.xmlToBehavior(gamePath + team._name,
19         gamePath);
20         GameManager.GetComponent<TeamManager>()._teams.Add(team);
21     }
22
23     GameManager setting = GameManager.GetComponent<GameManager>();
24     setting.SetSetting();
25
26     // StartCoroutine(AsynchronousLoad(setting._gameSettings._indexSceneMap));
27 }
28 }
```

Listing 3.1 – Extrait du code de PlayButton.cs

Bouton Éditeur de Comportement

Le bouton ”Editer” permet d'accéder à l'éditeur de comportement, qui permettra à l'utilisateur, entre autres, de créer et/ou modifier le comportement de ses équipes.

Bouton Paramètres

L'icône en forme d'engrenage représente le bouton paramètres. Il va permettre d'afficher la fenêtre ”pop-up” qui contient tous les paramètres du jeu.

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. Sa valeur courante sera récupérée par le script StartGame() (cf Listing 4.1).

En fonction du nombre d'équipes choisies, le nombre de cadres d'équipe changera.

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. Pour récupérer toutes les équipes existantes, nous allons chercher la liste comprenant le nom de toutes les équipes.

```

1 [...]
2 string team = GameObject.Find("GameManager").GetComponent<TeamManager>()._teams[
3     _idPlayer]._name;
4     for (int i = 0; i < _teamDropDown.options.Count; i++)
5     {
6         if (_teamDropDown.options[i].text.Equals(team))
7         {
```

```

7             _teamDropDown.value = i ;
8         }
9     }
10    [ ... ]

```

Listing 3.2 – Extrait du code de TeamMenuHUD.cs

De plus, à côté du nom de l'équipe, on retrouve aussi son score (ELO). Pour son affichage, nous récupérons le score associé à l'équipe courante. La couleur du score dépendra de sa valeur. (rouge = score faible, bleu = score élevé)

```

1  public void Change()
2  {
3      _teamName = _teamDropDown.captionText.text;
4      _powerScoreText.text = TeamsPerformance.GetTeamElo(_teamName).ToString();
5      _powerScoreText.color = ColorElo(TeamsPerformance.GetTeamElo(_teamName));
6  }

```

Listing 3.3 – Extrait du code de TeamMenuHUD.cs

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 nouveaux fichiers d'équipes, et de les voir apparaître en cliquant simplement sur ce bouton, sans avoir à relancer le jeu. Pour permettre cela, nous devons aller chercher le dossier contenant les fichiers des équipes, le parcourir, et mettre à jour notre liste d'équipes.

```

1  teams = new List<string>();
2      string[] fileEntries = Directory.GetFiles(gamePath);
3      foreach (string s in fileEntries)
4      {
5          string team = stringDifference(s, gamePath);
6          if (team.Contains(".wbt") && !team.Contains(".meta"))
7          {
8              teams.Add(team);
9          }
10     }

```

Listing 3.4 – Extrait du code de GameSettingsScript.cs

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.

Chaque changement de carte mettra à jour le nom affiché, ainsi que le visuel de l'aperçu de la carte. Pour cela, il suffit de récupérer le nom de la carte sélectionnée, ainsi que son visuel associé, puis de les afficher.



FIGURE 3.6 – Aperçu de la sélection de carte

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. A chaque clic sur le bouton "+", on incrémente la variable "number" associée à l'unité, et inversement avec "-", dans la limite du possible (c'est-à-dire tout en vérifiant que la valeur maximale du nombre d'unités possible ne soit pas dépassé).

```

1 _number = Mathf.Min(_maxNumber, _number + 1);
2 _unitCounter.text = _number.ToString("00");

```

Listing 3.5 – Extrait du code de UnitSettingMenu.cs

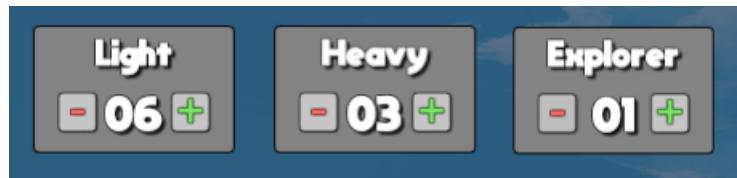


FIGURE 3.7 – Choix du nombre d'unités

Barre de Chargement

Quand on lance une partie, une barre de chargement apparaît et indique l'avancement du chargement de la partie. Ceci est une fonction qui utilise "AsyncOperation". La variable ao va contrôler

l'avancée du chargement de la scène jeu. Nous allons récupérer cette valeur, et la passer à notre slider. La scène se chargera qu'une fois la progression de l'AsyncOperation terminée. De cette manière, l'utilisateur peut voir où en est le chargement de sa partie.

```
1 Ienumerator AsynchronousLoad(int scene)
2 {
3     loadingScreenBar.SetActive(true);
4     yield return null;
5     AsyncOperation ao = SceneManager.LoadSceneAsync(scene);
6     ao.allowSceneActivation = false;
7     while (!ao.isDone)
8     {
9         sliderLoad.value = ao.progress;
10        if (ao.progress == 0.9f)
11        {
12            sliderLoad.value = 1f;
13            ao.allowSceneActivation = true;
14        }
15        yield return null;
16    }
17 }
```

Listing 3.6 – Extrait du code de UnitSettingMenu.cs

3.3.2 Menu des Paramètres

Ce menu regroupe les paramètres de jeu.

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. La valeur de notre slider affectera directement la valeur de notre AudioSource.

Choisir le nombre de ressource maximales dans le jeu

Dans cette case, on peut entrer un chiffre entier qui indiquera le nombre maximum de ressources 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. Le mode MetaBot est le mode classique.

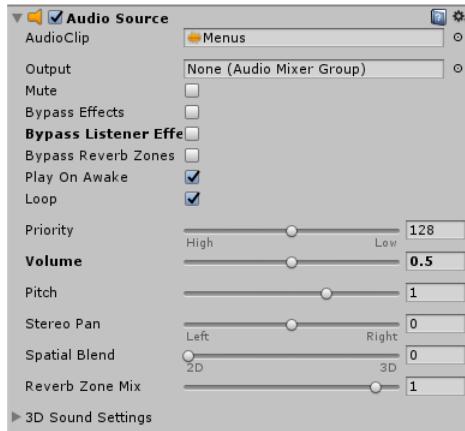


FIGURE 3.8 – Aperçu de l’ AudioSource, dans l’ Inspector



FIGURE 3.9 – Choix du nombre maximal de ressources

Le mode RessourceRace est une course aux ressources. La première équipe à atteindre le nombre de ressources fixé par le joueur gagne.

Si le mode choisi est RessourceRace alors deux autres paramètres apparaissent :

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



FIGURE 3.10 – Paramètres RessourceRace

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 voulu et de cliquer sur ”Valider”. La traduction de chaque texte présent dans l’interface se fait grâce à des fichiers .yml, et .txt. L’appel au traducteur permet ce changement. Nous ne traiterons pas en détail son fonctionnement, car il concerne la partie Interpréteur.

```
1  public void ChangementLangue( string newLangage )
```

```

2     {
3         GameObject.Find("GameManager").GetComponent<LangageLoader>().changeLanguage(
4             newLangage);
5             GameObject.Find("GameManager").GetComponent<Traducteur>().langue = newLangage;
6
7         string[] lines = System.IO.File.ReadAllLines(Application.streamingAssetsPath +
8             "/properties.yml");
9         int cpt = 0;
10        foreach (string line in lines)
11        {
12            if (line.Contains("Language"))
13            {
14                string[] tmp = line.Split('=');
15                tmp[1] = newLangage;
16                lines[cpt] = tmp[0] + "=" + tmp[1];
17                break;
18            }
19            cpt++;
20        }
21        System.IO.File.WriteAllLines(Application.streamingAssetsPath + "/properties.yml",
22             lines);
23    }

```

Listing 3.7 – Extrait du code de ChangeLanguage.cs

Bouton Retour

Ce bouton permet de retourner à l'écran du menu principal. Contrairement au bouton "Appliquer", ce bouton n'applique aucun changement effectué par l'utilisateur.

Bouton Valider

Ce bouton valide les paramètres définis précédemment et revient au menu principal en ayant appliqué ces paramètres.

```

1 public void ApplySettings()
2 {
3     setButtonActive();
4     manageVolume();
5     changeLanguage.ChangementLangue(language);
6     changeGameMode();
7     if (numberResources() == 0)
8     {
9         numberResources();
10    }
11    else
12    {

```

```

13         return ;
14     }
15     window.SetActive(false);
16 }

```

Listing 3.8 – Extrait du code de SettingsButton.cs

On peut apercevoir dans cette méthode, qu'elle appelle une méthode spécifique pour chaque paramètre modifiable. Elle s'occupe donc d'appliquer chacune des modifications effectuées, via l'appel de la fonction adéquate.

Bouton Quitter

Ce bouton ouvre une boîte 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.

3.3.3 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 faire un croquis de deux interfaces potentielles.

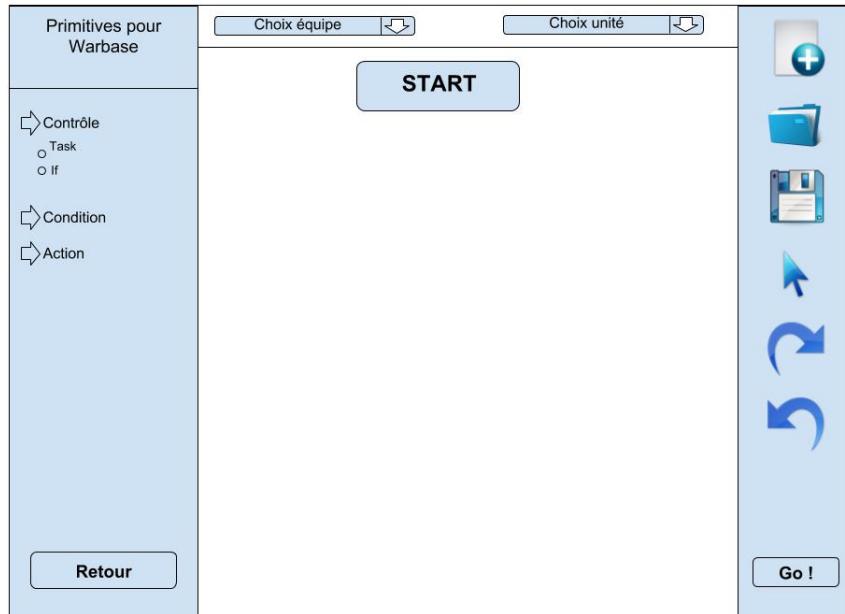


FIGURE 3.11 – Première ébauche

Le premier 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.

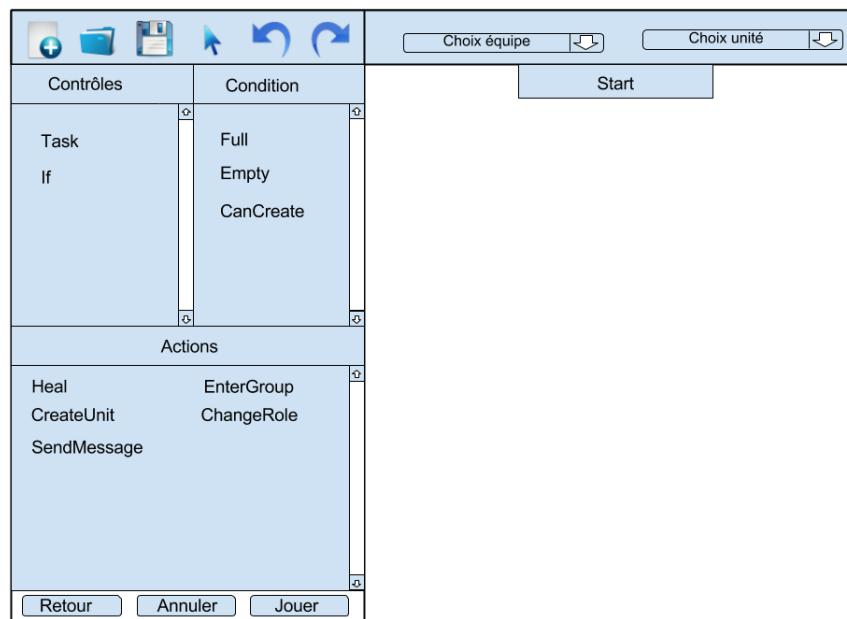


FIGURE 3.12 – Seconde ébauche

Au fur et à mesure de nos avancées, de nouveaux éléments ont pris place dans l'interface. A force d'ajouter des informations et fonctionnalités, nous avons dû légèrement retravailler l'aspect global de l'écran, pour gagner en cohésion, et en ergonomie. En voici un aperçu :



FIGURE 3.13 – Interface Finale

ToolBox

La ToolBox est le panneau en haut à gauche de l'interface, regroupant des actions classiques, telles que la sauvegarde de fichier, la création d'un nouveau comportement, etc...



FIGURE 3.14 – ToolBox, Editeur de comportement

A) Bouton Nouveau Comportement

Ce bouton permet de commencer la création d'un nouveau comportement pour l'unité de l'équipe courante. Cela permet de repartir à zéro, lorsque notre comportement courant ne nous satisfait pas, et nous évite de devoir retirer chaque pièce manuellement.

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.

Voir Annexes pour le script

C) Bouton Sauvegarde du Comportement

Ce bouton sauvegarde le comportement de l'unité de l'équipe courante. Si l'on change d'unité dans la même équipe 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 :

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.

```
1 public void createBehavior()
2 {
3     GameObject startPuzzle = GameObject.FindGameObjectWithTag("StartPuzzle");
4     GameObject ifpuzzle = startPuzzle.GetComponent<StartPuzzleScript>();
5     ifpuzzle;
6     while (ifpuzzle != null && ifpuzzle.activeSelf == true)
7     {
8         // Crée l'instruction (conditions + une/des actions) correspondante au
9         If courant
10        listBehavior.Add(ifpuzzle.GetComponent<IfPuzzleScript>());
11        createInstruction();
12        ifpuzzle = ifpuzzle.GetComponent<IfPuzzleScript>().puzzleIfObject;
13    }
14    createXML();
15    Debug.Log("Saving file done !");
16    listBehavior.RemoveRange(0, listBehavior.Count);
17 }
```

Listing 3.9 – Extrait du code de SaveFile.cs

Une fois chaque instruction créée pour chaque "If", nous pouvons appeler la fonction createXML().

D) 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.

```

1  public void createXML()
2  {
3      //récupération de l'unité actuellement traitée et de l'équipe
4      string teamName = team.captionText.text;
5      string unitName = unit.captionText.text;
6      string path = Application.dataPath + "/StreamingAssets/" + Constants.
    teamsDirectory + Constants.gameModeWarBot;
7      if(teamName != "") {
8          XMLWarbotInterpreter interpreter = new XMLWarbotInterpreter();
9          interpreter.behaviorToXml(teamName, path, unitName, listBehavior);
10     }
11 }
12 }
```

Listing 3.10 – Extrait du code de SaveFile.cs

E) Bouton "Undo"

Ce bouton 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.

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 réapparaître à l'écran, en la passant en active.

```

1  public void Undo()
2  {
3
4      if( cptObjects > 0 ) {
5
6          pieceToUndo = (GameObject)listPieces[cptObjects - 1];
7          // Si la pièce à "undo" est une pièce IF, on oublie pas de mettre à jour
    le compteur de pièces IF
8          if (pieceToUndo.tag == "IfPuzzle") {
9              cptIfPuzzle = numberIfPuzzle() - 1;
10         }
11         // Si la pièce à "undo" est active, alors on la masque, on l'ajoute à
    la liste redo et on met à jour le compteur d'objet
```

```

12         if (pieceToUndo.activeSelf == true) {
13             recoverList.Add(pieceToUndo);
14             pieceToUndo.SetActive(false);
15             cptObjects--;
16         }
17         // Si la pièce à "undo" est inactive, alors on l'active et on change son
18         // ordre dans la liste des pièces
19         else if (pieceToUndo.activeSelf == false) {
20             pieceToUndo.SetActive(true);
21             listPieces.Remove(pieceToUndo);
22             listPieces.Insert(0, pieceToUndo);
23         }
24         cptUndo = recoverList.Count;
25     }
26 }
```

Listing 3.11 – Extrait du code de createPuzzle.cs

F) Bouton "Redo"

Ce bouton 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.

```

1 // Restitue la dernière pièce de la liste recoverList
2 public void Redo()
3 {
4     if (recoverList.Count > 0) {
5         GameObject pieceToRedo = (GameObject)recoverList[cptUndo - 1];
6         if (pieceToRedo.tag == "IfPuzzle") {
7             cptIfPuzzle = numberIfPuzzle() + 1;
8         }
9         pieceToRedo.SetActive(true);
10        recoverList.RemoveAt(cptUndo - 1);
11        cptObjects++;
12        cptUndo = recoverList.Count;
13    }
14 }
```

Listing 3.12 – Extrait du code de createPuzzle.cs

G) 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

Ce cadre nous permet de sélectionner l'équipe et l'unité sur lesquels nous souhaitons travailler. Chaque équipe est composée du même type d'unités. Deux boutons sont également disponibles, permettant de créer ou supprimer l'équipe couramment sélectionnée.



FIGURE 3.15 – Sélection des équipes / unités

A) Choix de l'équipe

Ce menu déroulant permet de choisir l'équipe sur laquelle on veut travailler. Un clic sur le menu déroule la liste, qui comprend le nom de toutes les équipes. Cette liste est tenue à jour, dès que l'on ajoute une équipe, supprime une équipe, ou bien que l'on ajoute un fichier équipe à la main, et que l'on appuie sur le bouton "Reload team" du menu principal.

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. Lorsqu'une unité est sélectionnée, son comportement, s'il existe, sera dynamiquement chargé dans l'éditeur, grâce à la fonction `createBehaviorFromXML()` du script `LoadFile` (détailé en annexe).

Il faut aussi charger les statistiques et le modèle de l'unité sélectionnée, toujours dynamiquement. Pour ça, nous allons faire appel à la méthode `ReadStats()` du script `StatsLoader.cs`, qui nous permettra à la fois d'afficher les statistiques de l'unité sélectionnée, mais aussi son modèle 3D. Voici un exemple qui concerne l'unité "Base" :

```
1  [...]
2 // Lecture des statistiques de la base
3     else if (unitName == "Base" && line.Contains("Base"))
4     {
5         readStatsFile(unitName, reader, statBase);
6         imageUnit.sprite = baseSprite.sprite;
7         imageUnit.color = new Color(imageUnit.color.r, imageUnit.color.g,
8             imageUnit.color.b, 255);
9     }
10 [...]
```

Listing 3.13 – Extrait du code de `StatsLoader.cs`

C) Bouton Crédit d'équipe

Juste à droite des équipes se trouve un bouton en forme de "+", il permet de créer une nouvelle équipe. Si l'on clique dessus, une boîte de dialogue s'ouvre et nous demande le nom de la nouvelle

équipe. Pour qu'une équipe soit créée, il faut vérifier plusieurs choses. D'abord, il faut s'assurer que son nom ne contient que des caractères contenus dans a-zA-Z0-9. Nous empêchons également la création d'équipe dont le nom commence par le caractère "Espace". Ensuite, si le nom est valide, il faut que l'on vérifie si cette équipe n'existe pas déjà dans le dossier des équipes.

Enfin, lorsque ces deux conditions sont respectées, alors il faut mettre à jour la liste des équipes, ainsi que le menu déroulant de sélection d'équipes.

```
1 // Vérifie le nom d'équipe entré par l'utilisateur , pour empêcher l'utilisation de
2     // caractères spéciaux
3     // Caractères acceptés : a-zA-Z0-9
4     public void validateName()
5     {
6         string teamName = mainInputField.text;
7         string path = Application.streamingAssetsPath + "/teams/TestBot/";
8
9         List<int> listInt = new List<int>();
10        for (int i = 0; i < teamName.Length; i++)
11        {
12            listInt.Add(Convert.ToInt32(teamName[i]));
13        }
14
15        for (int i = 0; i < listInt.Count; i++)
16        {
17            int result = listInt[i];
18            if ((result > 90 && result < 97) || (result < 65 && result > 57) || (
19                result < 48 && result > 32) || result > 122)
20            {
21                errorText.SetActive(true);
22                Text error = errorText.GetComponentInChildren<Text>();
23                error.text = "Nom invalide ! (a-zA-Z0-9)";
24                return;
25            }
26            if (listInt[0] == 32) //&& result == 32
27            {
28                errorText.SetActive(true);
29                Text errorSpace = errorText.GetComponentInChildren<Text>();
30                errorSpace.text = "Nom invalide ! (a-zA-Z0-9)";
31            }
32
33        if (listInt.Count == 0)
34        {
35            errorText.SetActive(true);
36            Text errorSpace = errorText.GetComponentInChildren<Text>();
37            errorSpace.text = "Nom invalide ! (a-zA-Z0-9)";
38            return;
39        }

```

```

40
41     foreach (string file in Directory.GetFiles(path))
42     {
43         string res = file.Replace(path, "");
44         if (res == teamName + ".wbt")
45         {
46             errorText.SetActive(true);
47             Text error = errorText.GetComponentInChildren<Text>();
48             error.text = "L'équipe existe déjà !";
49             return;
50         }
51     }
52
53     if (!System.IO.File.Exists(Application.streamingAssetsPath + "/ELO/" +
54     teamName + ".elo"))
55     {
56         File.Create(Application.streamingAssetsPath + "/ELO/" + teamName + ".elo").Dispose();
57         File.WriteAllLines(Application.streamingAssetsPath + "/ELO/" +
58     teamName + ".elo", new string[] { 1000 + "" });
59
60         dropOption.Add(teamName);
61         teamDropDown.AddOptions(dropOption);
62         XMLWarbotInterpreter interpreter = new XMLWarbotInterpreter();
63         interpreter.generateEmptyFile(teamName, path);
64         dropOption.Clear();
65         errorText.SetActive(false);
66         window.SetActive(false);
}

```

Listing 3.14 – Extrait du code de CreateTeam.cs

D) Bouton Suppression d'équipe

Ce bouton permet de supprimer définitivement l'équipe courante. Une boîte de dialogue demandera confirmation. Pour supprimer une équipe, il nous suffit de supprimer les fichiers qui lui sont associés, dans le dossier contenant toutes les équipes. Il ne faut pas oublier de mettre à jour le menu déroulant de sélection des équipes, pour qu'elle cesse d'y apparaître.

Informations sur l'unité

Ce cadre a un but purement informatif. Diverses informations concernant notre unité sont affichées. Ces informations sont chargées en temps réel. Si l'utilisateur sélectionne une nouvelle unité, sa représentation 3D ainsi que ses statistiques seront immédiatement affichées dans ce cadre.



FIGURE 3.16 – 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. Cela va permettre à l’utilisateur d’adapter ses stratégies en fonction des statistiques de chaque unité. Ces statistiques ne changent pas d’une équipe à l’autre, elles sont inhérentes à l’unité.

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 structure. 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 concernée. 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.

```
1  GameObject puzzleClone = (GameObject)Instantiate(puzzle, GameObject.Find("MaskEditeur").transform);
2  if (puzzleClone.GetComponent<PuzzleScript>())
3  {
4      puzzleClone.GetComponent<PuzzleScript>()._value = _label;
5 }
```

Les pièces sont présentes sous forme de cases avec leur nom à l'intérieur (certaines possèdent 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

C'est dans cette zone de l'éditeur que l'utilisateur pourra créer et modifier les comportements des unités de ses équipes. Nous voulions cette zone la plus grande possible, et la plus épurée possible, pour rendre le processus de création vraiment optimal.

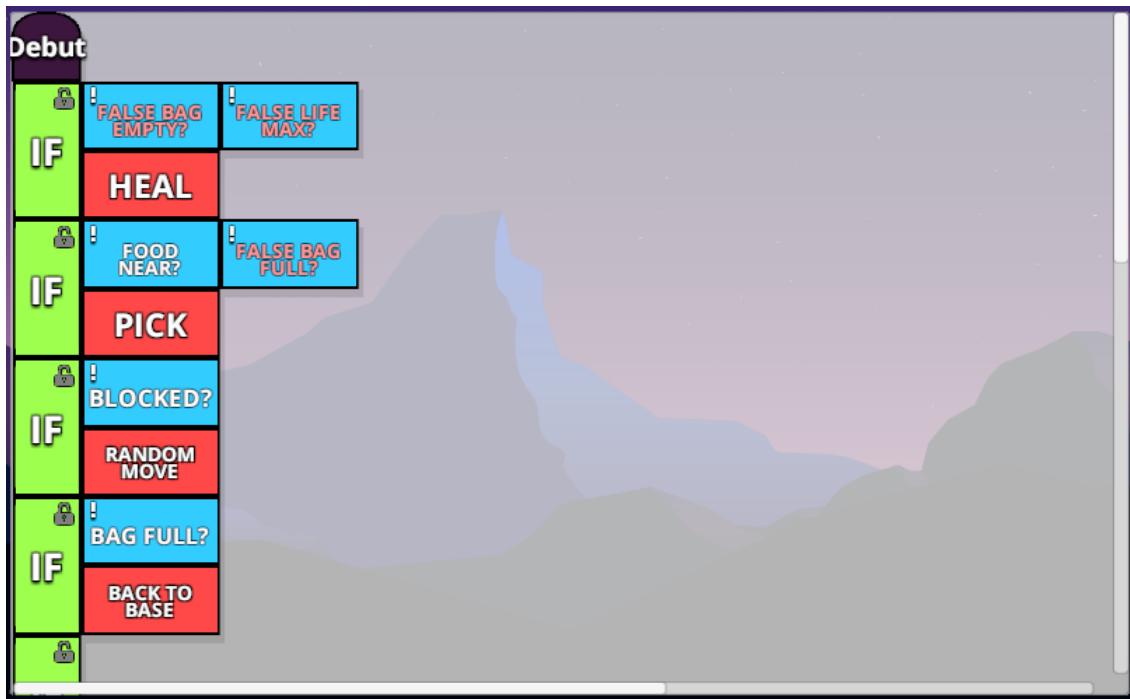


FIGURE 3.17 – Zone d'édition

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, adjacentes au contrôle, ou adjacentes à une autre pièce du même type, valide.

Voyons les scripts permettant cette gestion.

- **ManageDragAndDrop.cs :**

Ce script s'occupe de la gestion du déplacement des pièces, ainsi que de leur placement, sur une grille aimantée.

- **OnMouseDown() :**

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()** :

Lorsque le clic de la souris est relâché (OnMouseUp), il faut mettre à jour la position de la pièce dans l'éditeur.

```
1 posGridX = Mathf.Max((int)GameObject.Find("Editeur").GetComponent<
RectTransform>().rect.x /
2   (int)widthPuzzle, (int)Mathf.Round.GetComponent<RectTransform>().
localPosition.x / widthPuzzle));
3
4 posGridY = Mathf.Max((int)GameObject.Find("Editeur").GetComponent<
RectTransform>().rect.y /
5   (int)heightPuzzle, (int)Mathf.Round.GetComponent<RectTransform>().
localPosition.y / heightPuzzle));
```

- **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 placée 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 est similaire. 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.

3.3.4 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, elle redévient classique, et le son est restitué à sa valeur précédente.

Troisième partie

L'avenir du projet

Chapitre 4

Amélioration possible

Chapitre 5

Bugs

Quatrième partie

Annexe

Chapitre 6

Scripts Remarquables

6.1 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<
8          string , Queue<GameObject>>();
9      public Pool[] pools;
10
11     static bool created = false;
12
13     void Awake()
14     {
15         if (!created)
16         {
17             DontDestroyOnLoad(this.gameObject);
18             created = true;
19         }
20         else
21         {
22             Destroy(this.gameObject);
23             return;
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     static public GameObject Pick(string tag , Vector3 position , Quaternion rotation)
43     {
44         GameObject obj = dictPools[tag].Dequeue();
45         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

6.2 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 }

```

MovableCharacter.cs

6.3 Brain

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  [RequireComponent(typeof(Rigidbody))]
7  [RequireComponent(typeof(SphereCollider))]
8  [RequireComponent(typeof(Stats))]
9  [RequireComponent(typeof(Sight))]
10 [RequireComponent(typeof(Inventory))]
11 public class Brain : MonoBehaviour
12 {
13
14     public int indexInstructionUsed;
15     public List<Instruction> _instructions = new List<Instruction>();
16     public List<InstructionEditor> _instructionsEditor = new List<InstructionEditor>()
17     ;
18     public Percept _componentPercepts;
19     public Action _componentActions;
20     public ActionNonTerminal _componentActionsNonTerminales;
21     private string _currentAction;
22     public MessageManager _messageManager;
23     public int nbInstruction;
24
25
26     void Start()
27     {
28         //GameObject.Find("Canvas").GetComponent<HUDManager>().CreateHUD(gameObject);
29         GameObject.Find("CanvasHUD").GetComponent<HUDManager>().CreateHUD(gameObject);
30         foreach (Instruction I in _instructions)
31         {
32             InstructionEditor IE = new InstructionEditor();
33             IE._listeStringPerceptsVoulus = I._listeStringPerceptsVoulus;
34             IE._stringAction = I._stringAction;
35             IE._stringActionsNonTerminales = I._stringActionsNonTerminales;
36             _instructionsEditor.Add(IE);
37         }
38     }
39
40     public void LoadBehaviour()
41     {
42         if (GetComponent<Stats>()._teamIndex < GameObject.Find("GameManager") .
43             GetComponent<TeamManager>()._teams.Count)
44         {
45             _instructions = GameObject.Find("GameManager").GetComponent<TeamManager>()
46             .getUnitsBehaviours(GetComponent<Stats>()._teamIndex, GetComponent<Stats>().
47             _unitType);
```

```

44         }
45
46     _componentPercepts = GetComponent<Percept>();
47     _componentActions = GetComponent<Action>();
48     _componentActionsNonTerminales = GetComponent<ActionNonTerminal>();
49     // _messageManager = new MessageManager(this.gameObject);
50     _messageManager = GetComponent<MessageManager>();
51 }
52
53     public void UnitTurn()
54 {
55     _messageManager.UpdateMessage();
56
57     nbInstruction = _instructions.Count;
58     if (_instructions != null && _componentActions != null && _componentActions.
59     _actions.Count != 0)
60     {
61         string _action = NextAction();
62         if (_componentActions._actions.ContainsKey(_action))
63         {
64             _componentActions._actions[_action]();
65         }
66         else
67         {
68             _componentActions._actions["ACTION_IDLE"]();
69         }
70     }
71
72     public string NextAction()
73 {
74     indexInstructionUsed = 0;
75     foreach (Instruction instruction in _instructions)
76     {
77
78         if (Verify(instruction))
79         {
80
81             foreach (MessageStruct act in instruction._stringActionsNonTerminales)
82             {
83                 _componentActionsNonTerminales._messageDestinataire = act.
84                 _destinataire;
85                 if (_componentActionsNonTerminales._actionsNT.ContainsKey(act.
86                 _intitule))
87                 {
88
89                     _componentActionsNonTerminales._actionsNT[act._intitule]();
90                 }
91             }
92         }
93     }
94 }
```

```

91             if (instruction._stringAction != "") {
92                 {
93                     return instruction._stringAction;
94                 }
95             }
96             indexInstructionUsed++;
97         }
98         return "ACTION_IDLE";
99     }
100
101    bool Verify(Instruction instruction)
102    {
103        bool flag = true;
104        foreach (string percept in instruction._listeStringPerceptsVoulus)
105        {
106            if (!(_componentPercepts._percepts.ContainsKey(percept.Replace("NOT_", "")))
107                && (percept.Contains("NOT_") ^ _componentPercepts._percepts[percept.Replace("NOT_",
108                "")]())))) { flag = false; }
109
110        // bool flag2 = false;
111        // foreach (string percept in instruction._listeStringPerceptsOu)
112        // {
113        //     if (!(_percepts._percepts.ContainsKey(percept) && _percepts._percepts[
114        percept]())) { flag2 = true; }
115        // }
116        // return (flag && flag2);
117
118        return flag;
119    }
120
121 }
122
123 [System.Serializable]
124 public struct InstructionEditor
125 {
126     public string[] _listeStringPerceptsVoulus;
127     public MessageStruct[] _stringActionsNonTerminales;
128     public string _stringAction;
129 }
```

Brain.cs

6.4 PerceptUnit

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PerceptUnit : PerceptCommon
6  {
7
8      void Start()
9      {
10         InitPercept();
11     }
12
13
14     public override void InitPercept()
15     {
16         base.InitPercept();
17         _percepts["PERCEPT_BLOCKED"] = delegate () { return GetComponent<MovableCharacter>().isBlocked(); };
18
19         _percepts["PERCEPTE_BASE_NEARALLY"] = delegate ()
20         {
21             GetComponent<Stats>().SetTarget(null);
22             foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
23             {
24                 if (gO && gO.GetComponent<Stats>() != null && gO.GetComponent<Stats>()._unitType == "Base" &&
25                     Vector3.Distance(transform.position, gO.transform.position) < 8f && gO
26                     .GetComponent<Stats>()._teamIndex == GetComponent<Stats>()._teamIndex)
27                 {
28                     GetComponent<Stats>().SetTarget(gO);
29                     return true;
30                 }
31             }
32             return false;
33         };
34         /* PERCEPTE */
35         _percepts["PERCEPTE_CAN_GIVE"] = delegate ()
36         {
37             GetComponent<Stats>().SetTarget(null);
38             foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
39             {
40                 if (gO && gO.GetComponent<Stats>() != null &&
41                     Vector3.Distance(transform.position, gO.transform.position) < 8f && gO
42                     .GetComponent<Stats>()._teamIndex == GetComponent<Stats>()._teamIndex)
43                 {
44                     GetComponent<Stats>().SetTarget(gO);
45                     return true;
46                 }
47             }
48         };
49     }
50 }
```

```

44         }
45     }
46     return false;
47 };
48 _percepts["PERCEP_CONTRACT_ELIMINATION"] = delegate () { return GetComponent<
Stats>().HaveContrat() && GetComponent<Stats>().GetContract().type == "Elimination"
; };
49 _percepts["PERCEP_CONTRACT_ELIMINATION_TARGET_NEAR"] = delegate () {
50     Brain brain = GetComponent<Brain>();
51     Sight sight = brain.GetComponent<Sight>();
52     List<GameObject> _listOfUnitColl = new List<GameObject>();
53     if (!GetComponent<Stats>().HaveContrat() || (GetComponent<Stats>().
HaveContrat() && GetComponent<Stats>().GetContract().type != "Elimination")) {
54         return false; }
55     EliminationContract contract = (EliminationContract)GetComponent<Stats>().
GetContract();
56
57     foreach (GameObject gO in sight._listOfCollision)
58     {
59         if (gO && contract._target == gO)
60         {
61             GetComponent<Stats>().SetTarget(gO);
62             GetComponent<Stats>().SetHeading(getAngle(gO));
63             return true;
64         }
65     }
66     return false;
67 };
68 _percepts["PERCEP_FOOD_NEAR"] = delegate ()
69 {
70     return (_percepts["PERCEP_FOOD"]()) && (Vector3.Distance(GetComponent<
Stats>().GetTarget().transform.position, transform.position) < 4f);
71 };
72
73 _percepts["PERCEP_CONTRACT"] = delegate () { return GetComponent<Stats>().
HaveContrat(); };
74 _percepts["PERCEP_FOOD"] = delegate ()
75 {
76     GetComponent<Stats>().SetTarget(null);
77     foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
78     {
79         if (gO && gO.tag == "Item")
80         {
81             GetComponent<Stats>().SetTarget(gO);
82             return true;
83         }
84     }
85     return false;
86 };

```

```
87     }
88
89 }
90 }
```

PerceptUnit.cs

6.5 LoadFile

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using System.Text.RegularExpressions;
4  using UnityEngine;
5  using System.IO;
6  using UnityEngine.UI;
7
8  public class LoadFile : MonoBehaviour
9  {
10     public Dropdown team;
11     public Dropdown unit;
12     public GameObject ifPuzzle;
13     public GameObject condPuzzle;
14     public GameObject actionPuzzle;
15     public GameObject messagePuzzle;
16     public GameObject antPuzzle;
17     public Vector3 positionInitial;
18     public Scrollbar scrollEdit;
19     public Scrollbar scrollPieces;
20     GameObject startPuzzle;
21     NewFile clear;
22
23     // Lit un fichier .xml, et crée le comportement correspondant
24     public void createBehaviorFromXML()
25     {
26         clear = new NewFile();
27         clear.clearEditor();
28         string teamName = team.captionText.text;
29         string unitName = unit.captionText.text;
30         string path = Application.dataPath + "/StreamingAssets/" + Constants.
31 teamsDirectory + Constants.gameModeWarBot;
32         Transform editeurTransform = GameObject.Find("Editeur").transform;
33         XMLWarbotInterpreter interpreter = new XMLWarbotInterpreter();
34         Dictionary<string, List<Instruction>> behavior = interpreter.xmlToBehavior(
35             teamName, path);
36         Vector2 delta = new Vector2(0, -1);
37         if (behavior.ContainsKey(unitName) && behavior[unitName].Count != 0)
38         {
39             GameObject.Find("StartPuzzle").GetComponent<ManageDragAndDrop>().
40             UpdateGridPosition();
41             GameObject.Find("StartPuzzle").GetComponent<StartPuzzleScript>().
42             UpdateAllValidPuzzles();
43             float widthPuzzle = GameObject.Find("StartPuzzle").GetComponent<
44             ManageDragAndDrop>().sizePuzzleX;
45             float heightPuzzle = GameObject.Find("StartPuzzle").GetComponent<
46             ManageDragAndDrop>().sizePuzzleY;
```

```

41         float newX = GameObject.Find("StartPuzzle").GetComponent<ManageDragAndDrop>().posGridX * widthPuzzle;
42         float newY = GameObject.Find("StartPuzzle").GetComponent<ManageDragAndDrop>().posGridY * heightPuzzle;
43         Vector3 newPos = new Vector3(newX, newY, 10);
44         GameObject.Find("StartPuzzle").GetComponent<RectTransform>().localPosition =
45             newPos;
46         GameObject currentIf = GameObject.Find("StartPuzzle");
47         GameObject currentPercept = GameObject.Find("StartPuzzle");
48         GameObject currentAction = GameObject.Find("StartPuzzle");
49
50         foreach (Instruction I in behavior[unitName])
51     {
52             GameObject _ifPuzzle = Instantiate(ifPuzzle, editeurTransform);
53             _ifPuzzle.GetComponent<ManageDragAndDrop>().setGridPosition(currentIf.
54             GetComponent<ManageDragAndDrop>().getGridPosition() + delta);
55             _ifPuzzle.GetComponent<ManageDragAndDrop>().UpdateGridPosition();
56             GameObject.Find("StartPuzzle").GetComponent<StartPuzzleScript>().
57             UpdateAllValidPuzzles();
58             widthPuzzle = _ifPuzzle.GetComponent<ManageDragAndDrop>().sizePuzzleX;
59             heightPuzzle = _ifPuzzle.GetComponent<ManageDragAndDrop>().sizePuzzleY
60             ;
61             newX = _ifPuzzle.GetComponent<ManageDragAndDrop>().posGridX *
62             widthPuzzle;
63             newY = _ifPuzzle.GetComponent<ManageDragAndDrop>().posGridY *
64             heightPuzzle;
65             newPos = new Vector3(newX, newY, 10);
66
67             _ifPuzzle.GetComponent<RectTransform>().localPosition = newPos;
68             currentIf = _ifPuzzle;
69             delta = new Vector2(1, 0);
70             currentPercept = currentIf;
71
72             foreach (string s in I._listeStringPerceptsVoulus)
73         {
74                 GameObject _condPuzzle = Instantiate(condPuzzle, editeurTransform)
75                 ;
76                 _condPuzzle.GetComponent<ManageDragAndDrop>().setGridPosition(
77                 currentPercept.GetComponent<ManageDragAndDrop>().getGridPosition() + delta);
78                 currentPercept = _condPuzzle;
79                 _condPuzzle.GetComponent<PuzzleScript>()._value = s;
80                 if (s.Contains("NOT_"))
81                 {
82                     _condPuzzle.GetComponent<PuzzleScript>().NegationBoutton();
83                 }
84                 delta = new Vector2(2, 0);
85             }
86             delta = new Vector2(1, -1);
87             currentAction = currentIf;
88             if (I._stringActionsNonTerminales.Length != 0)

```

```

81
82         {
83             foreach (MessageStruct s in I._stringActionsNonTerminales)
84             {
85                 GameObject _messPuzzle = null;
86                 if (s._intitule.Contains("ACTN")) { _messPuzzle = Instantiate(
87                     antPuzzle, editeurTransform); }
88                 if (s._intitule.Contains("ACTN_MESSAGE_")) { _messPuzzle =
89                     Instantiate(messagePuzzle, editeurTransform); }
90                     //print(s._intitule);
91                     if (_messPuzzle != null)
92                     {
93                         _messPuzzle.GetComponent<ManageDragAndDrop>().
94                         setGridPosition(currentAction.GetComponent<ManageDragAndDrop>().getGridPosition() +
95                         delta);
96                         currentAction = _messPuzzle;
97                         _messPuzzle.GetComponent<PuzzleScript>()._value = s.
98                         _intitule;
99                         //print("A Dest : " + s._destinataire);
100                        if (_messPuzzle.GetComponent<PuzzleScript>().
101                            messageDropDown)
102                            {
103                                _messPuzzle.GetComponent<PuzzleScript>().
104                                DropDownUpdate();
105                                for (int i = 0; i < _messPuzzle.GetComponent<
106                                PuzzleScript>().messageDropDown.options.Count; i++)
107                                {
108                                    // print("B " + _messPuzzle.GetComponent<
109                                    PuzzleScript>().messageDropDown.options[i].text + "><" + s._destinataire);
110                                    if (_messPuzzle.GetComponent<PuzzleScript>().
111                                        messageDropDown.options[i].text == s._destinataire)
112                                        {
113                                            //print("B Dest : " + s._destinataire);
114                                            _messPuzzle.GetComponent<PuzzleScript>().
115                                            messageDropDown.value = i;
116                                            _messPuzzle.GetComponent<PuzzleScript>().
117                                            messageDropDown.Select();

```

```
118         if (I._stringAction != "")  
119         {  
120             GameObject _actPuzzle = Instantiate(actionPuzzle, editeurTransform  
);  
121             _actPuzzle.GetComponent<ManageDragAndDrop>().setGridPosition(  
currentAction.GetComponent<ManageDragAndDrop>().getGridPosition() + delta);  
122             _actPuzzle.GetComponent<PuzzleScript>()._value = I._stringAction;  
123         }  
124         delta = new Vector2(0, -2);  
125     }  
126     }  
127     GameObject.Find("StartPuzzle").GetComponent<StartPuzzleScript>().  
UpdateAllValidPuzzles();  
128     ResetScrollBarEditorPosition();  
129 }  
130 }
```

LoadFile.cs