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

Première partie

Présentation du projet

Chapitre 1

Introduction

1.1 But du projet

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

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

1.2 Cahier des charges

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

1.3 Notion d'agent

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 définit par un comportement créer 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.

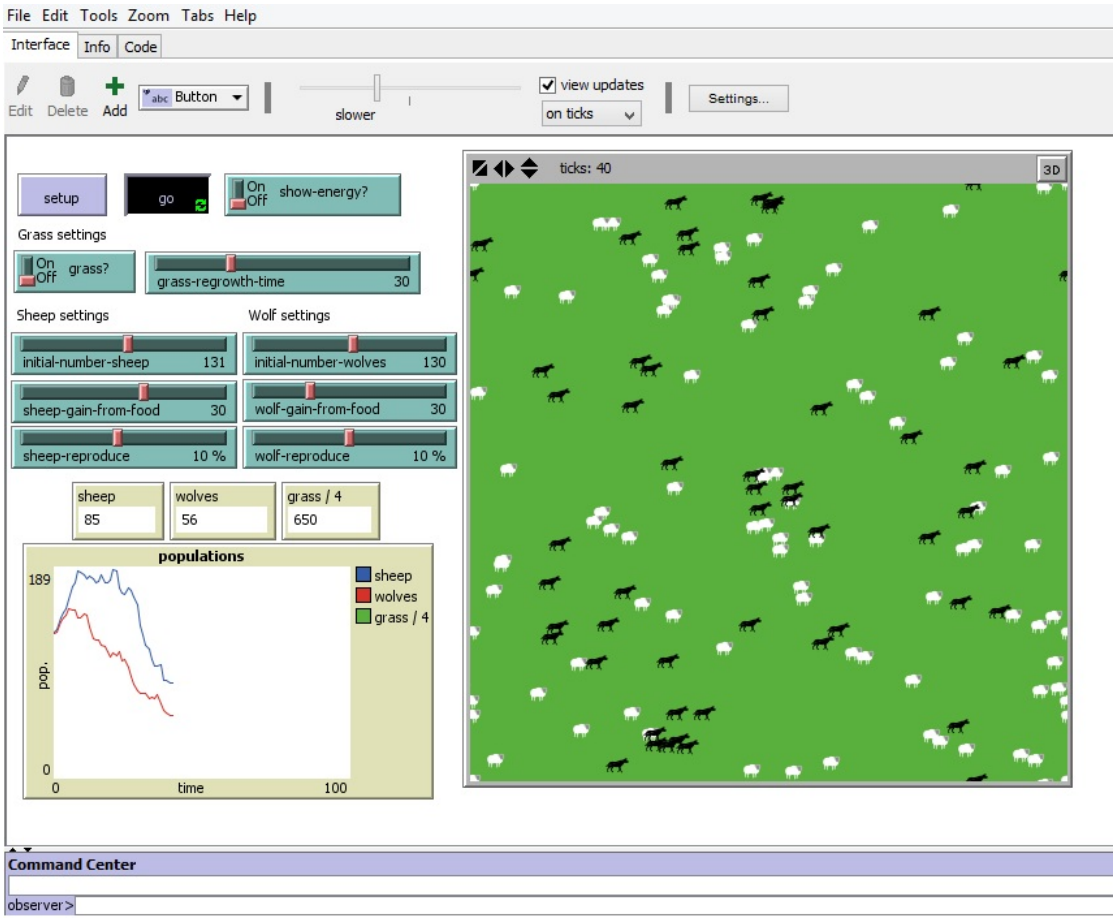

FIGURE 1.1 – NetLogo : Un logiciel de simulation multi-agents.

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 pourrais pas accomplir. Pour permettre de réaliser des objectifs

de groupe, les individues 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 système-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.

Chapitre 2

WarBot : Le mode par défaut

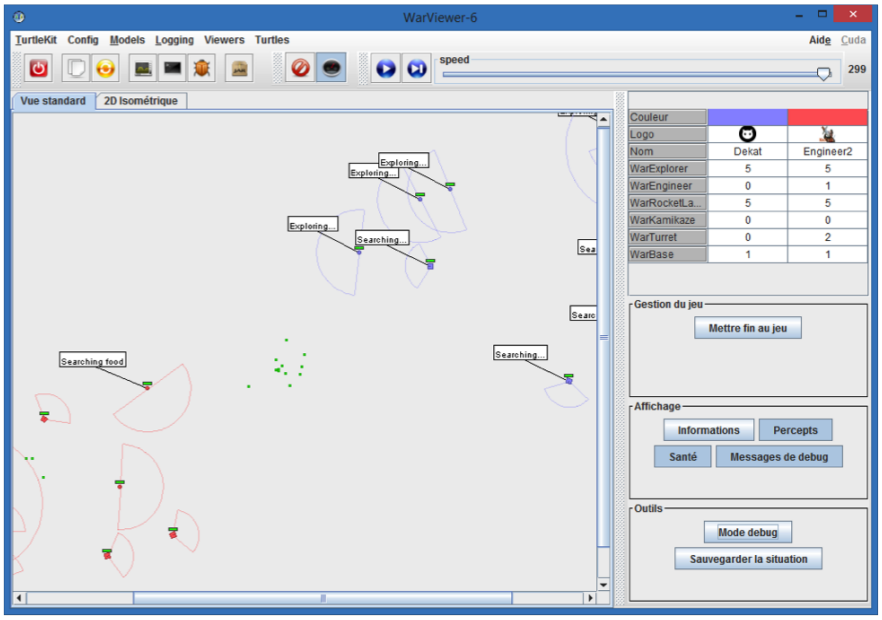

FIGURE 2.1 – WarBot, la version Java.

2.1 Principe

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

Deuxième partie

Réalisation du projet

Chapitre 3

Partie "Moteur"

3.1 État de l'art de l'ancien projet.

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

Durant une partie, le joueur peut effectuer plusieurs actions :

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

- Activer l'affichage du suivi visuel des envoies de message entre unités.
- Activer l'affichage du groupe auquel appartient une unité mais l'utilisation de groupe dans les comportements n'est pas fonctionnelle.
- Activer l'affichage de la barre de vie d'une unité.
- Activer l'affichage de la barre de ressource d'une unité.

A voir si a compléter

3.2 Refonte du noyau.

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

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

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

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

```
1 using System.Collections;
```

```
2  using System.Collections.Generic;
3  using UnityEngine;
4  using System;
5  using WarBotEngine.Projectiles;
6  using System.Security.Cryptography;
7
8  namespace WarBotEngine.WarBots {
9
10     public class AttackController : MonoBehaviour {
11 [...]
12         public bool Reloaded()
13         {
14             return (this.reloading + this.reload_time < Time.time);
15         }
16
17         public bool Shoot()
18         {
19             if (!this.Reloaded())
20                 return false;
21             this.Fire();
22             return true;
23         }
24 [...]
25         public void Fire() {
26             SoundManager.Actual.PlayFire(this.gameObject);
27             Instantiate(this.projectile , this.warprojectile_emitter.transform.
       position , this.warprojectile_emitter.transform.rotation , this.transform);
28             Instantiate(this.muzzle_flash , this.warprojectile_emitter.transform.
       position , this.warprojectile_emitter.transform.rotation , this.transform);
29             this.reloading = Time.time;
30         }
31     }
32 }
```

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

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

problème de la gestion des actions et des perceptions de l'interpréteur, car pour chaque action créer dans le script, il faut indiquer à l'interpreteur quelle fonction permet d'activer l'action du comportement en cours.

Ainsi dans le code du listing 3.9, on peut voir que pour activer la fonction Shoot() du script AttackController, il faut créer une fonction du même nom, puis renseigner toutes les unités qui peuvent effectuer l'action. Le problème est qu'il faut, pour chaque action, créer une fonction dans le fichier Unit.cs, et ainsi le fichier Unit.cs est surchargé, atteignant les 2000 lignes de codes !

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

Listing 3.2 – Extrait du code du script Unit.cs

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

La réalisation du projet a été effectuée en suivant une méthode agile. 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 régulière avec lui. Il nous est donc paru plus judicieux de vous expliquer le déroulement de notre travail de manière chronologique. On reviendra plus en détail sur cette méthode et son application à notre situation dans la partie gestion de projet.

3.2.1 Retour aux bases.

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

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

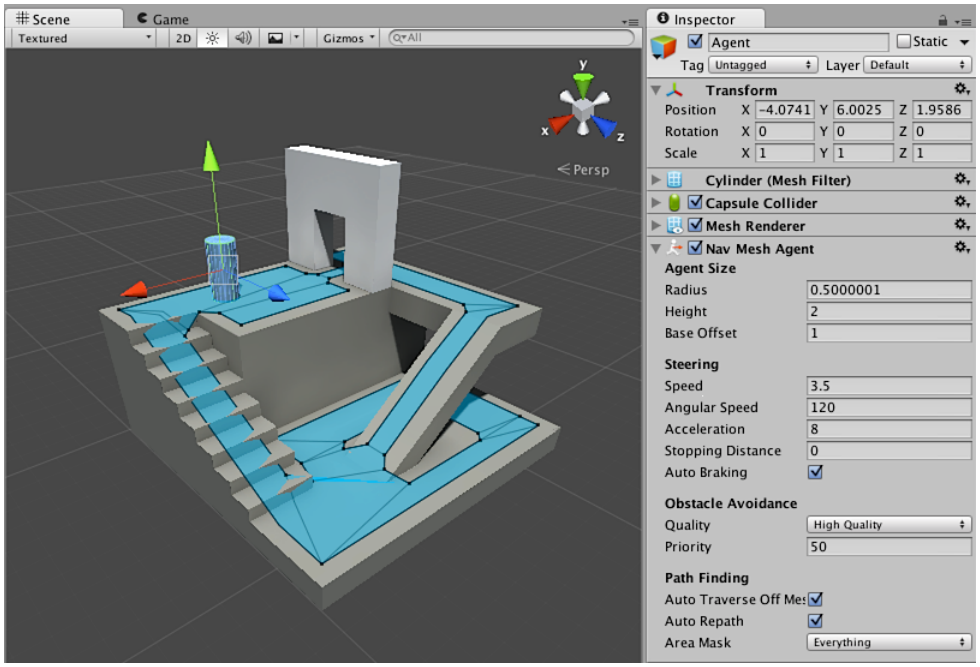

FIGURE 3.1 – Exemple d'application du composant *NavMesh* dans Unity. L'agent se déplace à l'intérieur de la zone bleue.

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

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

Une fois ces premiers mouvements rendus possibles, 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<PerceptManager>()._percepts;
4          if (!listePercepts[0]._percept._value) //Percept bloquée? faux
5              _actions[0].Do(); // Move
6      }
```

Listing 3.3 – 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 (les percepts) et d'une action terminant le tour de l'unité. La valeur d'un percept était donc calculée à partir d'une fonction booléenne vérifiant les conditions à remplir. Par exemple, le percept de ressource, devant vérifier si une ressource était à proximité. Nous avons commencé par créer la gestion de l'action de la manière suivante : pour créer une action il fallait hériter de la classe Action et surcharger sa méthode do() en créant l'action voulue.

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

de celle-ci. Dans cette sphère, on décide d'un angle fixe qui sera l'angle de champs de vision de l'unité.

```
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 *
        _distance;
8
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 3.4 – Extrait du code du script Sight.cs première version.

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

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

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

```
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>()
        .GetComponent<PerceptManager>()._percepts;
11          bool verifie = true;
12          foreach(PerceptStructure p in _listePerceptsVoulus)
13          {
14              foreach(PerceptStructure p2 in listePerceptsUtilisables)
15              {
16                  if(p._name.Equals(p2._name))
17                  {
18                      verifie = p2._percept._value;
19                  }
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 3.5 – Classe Instruction du fichier Instruction.cs première version.

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

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

```
1       override public void update()
2       {
3           bool res = false;
```

```
4           foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
5           {
6               if (gO && gO.GetComponent<ItemHeldler>())
7               {
8                   GetComponent<Stats>().SetTarget(gO);
9                   res = true;
10                  break;
11              }
12          }
13          _value = res;
14      }
```

Listing 3.6 – 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 3.7 – 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 viens 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.

INSERER IMAGE RESSOURCE!!!!!!!!!!!!!!!!!!!!!!!!!!!

Du point de vu de sa modélisation, une ressource est représentée par une sphère en contenant une autre plus petite en son centre. Elle détient égalument un système a particules permettant de générer ces dernières tournent autour de du centre de la shère 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, correspondant à la vision de l'objet ressource sans ses caractéristiques physiques. Cela nous permet de détruire la représentation de celle-ci sur la scène tout en conservant ses propriétés.

Le script Objet représentant a donc plusieurs attributs cela pour rendre générique la création d'un Objet 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 type "delegate" propre au C# dont la déclaration est semblable à une signature de méthode. Elle a une valeur de retour et un nombre quelconque de paramètres de type quelconque. Nous l'avons donc utilisé afin d'encapsuler les méthodes correspondant aux actions et aux percepts et ce afin d'éviter la crétion de multiple classes. A la suite de ce changement, la classe Percept s'est dotée d'un attribut dictionnaire[String,Listener], un percept étant définit par son nom et son Listener associé. Le Listener est un delegate représentant la fonction de calcul de valeur du percept. Le même mécanisme est mis en place pour les actions et est toujours d'actualité aujourd'hui.

```
1   _percepts["PERCEPT_FOOD"] = delegate ()
2           {
3               GetComponent<Stats>().SetTarget(null);
4               foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
5               {
6                   if (gO && gO.tag == "Item")
7                   {
8                       GetComponent<Stats>().SetTarget(gO);
9                       return true;
10                  }
11              }
12              return false;
13          };
```

Listing 3.8 – Extrait du code du script PerceptUnit.cs

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

Listing 3.9 – Extrait du code du script ActionUnit.cs

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 possible ce tour-ci s'il en existe une possible. 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
       >().GetComponent<PerceptManager>()._percepts))
12             {
13                 actionPossible.Do();
14                 break;
15             }
16         }
17     }
18
19     public Action actionPossible(string stringAct)
20     {
21         Action presence = null;
22         foreach (ActionStructure actStruc in _actions)
23         {
24             if (actStruc._name.Equals(stringAct))
25             {
26                 presence = actStruc._action;
27             }
28         }
29         return presence;
30     }
31
32 }
```

Listing 3.10 – 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 d'effectuer

nos tests.

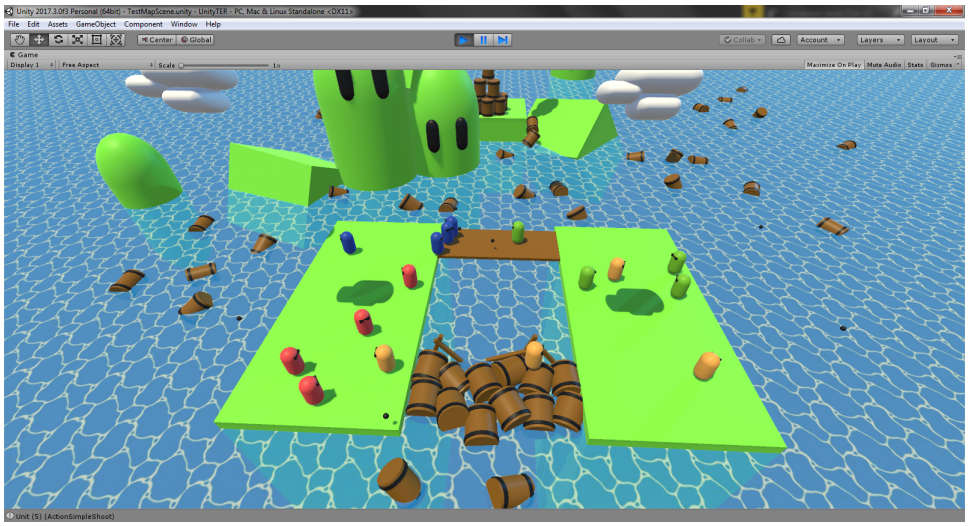

FIGURE 3.2 – Aperçu de la premier carte de test.

Sur cette scène a été intruduit le déplacement de la camera principale (MainCamera) calculé en fonction de la position moyenne des unités. Cela permet de donner un effet dynamique à la scène.

Cette phase de démarrage a duré environ un mois, durant lequel nous avons pu rendre compte chaque semaine à notre encadrant/client. Durant celle-ci on a donc aggrémenté notre moteur de jeu au gré de M. Ferber, en nous basant sur notre expérience d'utilisation de Unity pour valider les différents ajouts demandés.

3.2.2 Intégration dans le projet.

Lorsque le moteur nous paraissait complet, nous l'avons intégré au projet. Cette phase à était source de problème du fait que l'ancien projet etait encore present et que de nombreux conflit persister.

La phase d'intégration.

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

Suite à nos échanges, il a été convenue 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 3.11 – Extrait du code du script Instruction.cs

Cette classe est composé de l'attribut *_listeStringPerceptsVoulus* correspondant à la liste des labels des perceptions 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 implémenté, nous décidons tout de même de les laisser pour les implémenter ultérieurement.

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

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

Listing 3.12 – Exemple d'une instruction d'un comportement que peut renvoyer l'interpréteur.

L'exemple montrer au listing 3.12 correspond au comportement d'une unité et lui demandera de tirer si l'unité peut percevoir un ennemie, et que sa vie est au maximum. Les actions non-terminales n'étant pas pris en charge à ce moment du développement, la liste est donc vide. Un comportement n'est donc rien d'autres qu'une liste d'instruction.

Pour garder l'idée d'une structure de subsomption, la liste des instructions est ordonnée de tel sorte que la premiere instruction dans cette liste est la première à être examiner et exécute si ces perceptions sont activé.

Résolution des problèmes.

Maxime

3.2.3 Améliorations des fonctionnalités.

Actions non-terminales

Ajout de fonctionnalités des instructions.

Un des enjeux du moteur était de pouvoir rendre les instructions plus interessantes pour le joueurs. Il a donc fallu améliorer les instructions qui etait une simple liste de perceptions, d'actions non terminales et d'une action en rajoutant des fonctionnalité.

```
1  Behavior [LIGHT] = {
2  1. Instruction (  Percepts = [PERCEPT_ENEMY, PERCEPT_LIFE_MAX],
3          ActionNT = [],
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_PERCEPT_LIFE_MAX],
9          ActionNT = [],
10         Action = ACTION_HEAL ),
11 4. Instruction (  Percepts = [],
12         ActionNT = [],
13         Action = ACTION_MOVE ) }
```

Listing 3.13 – Exemple d'un comportement que peut renvoyer l'interpréteur.

Le comportement défini en listing 3.13 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 instruction était de mettre en place la négation des percepts.

Pour rappel, un percept (ou perception) est une fonction anonyme qui renvoie un booléen. La composition de plusieurs perception 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 percept. En d'autres termes, tout les percepts d'une instruction doit être activé pour pouvoir exécuter l'action de l'instruction.

Il nous fallait donc trouver un moyen pour rendre possible la négation des perceptions. Ainsi, et en travaillant de concert avec l'équipe Interpréteur, nous avons convenu d'un préfixe *NOT_* que l'on concatène au label des perceptions, comme vu à l'instruction 2. et 3. du listing 3.13.

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

Listing 3.14 – Une perception 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.
       Replace("NOT_", "")]())))  { flag = false; }
11         }
12         return flag;
13     }
14 [...]
15 }
```

Listing 3.15 – 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 3.15) 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. Mouvement tactique. Comme définie dans le listing 3.13, l'instruction 2. permet de faire une action non-terminale même pas d'action terminale. Cela permet ainsi de pouvoir réaliser des mouvement plus tactique et de ne pas terminer le tour.

Lorsque les percepts de l'instruction 2 sont activé, cela déclenche l'action non terminale en ligne 6 qui envoie un message d'aide aux unité 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.

3. Instruction par défaut. Lorsque 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 exécuter si aucune autre action ne peut l'etre.

4. Cas exceptionnel. Dans un comportement, il peut arriver que toutes les instructions ne soit pas vérifier. Dans ce cas précis, c'est le moteur lui même qui prends en charge l'action de l'unité. Actuellement, l'unité éxécute l'action $ACTION\_IDLE$ dans ce cas la.

Optimisation du temps de chargement

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

1. La gestion des actions des unités. La première piste de recherche que l'on a étudié est d'essayer d'optimiser la gestion des actions des unités. Lorsque l'on a conçu les unités, elles faisaient leurs actions dans une fonction utilisée par Unity ("Update") qui est appelée à chaque calcul d'image. Ainsi, pour chaque unité présente sur la scène, Unity calcule l'action que celle-ci

doit faire. Cependant, certaine action nécessite plus de temps en fonction de sa complexité, ce qui pouvait engendrer que des unités avaient des tours de retard par rapport à d'autres unités. Pour palier à ce problème, nous avons donc délégué la gestion des tours des unités à un script "TurnManagerScript.cs" (Listing 3.16).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TurnManagerScript : MonoBehaviour
{
    public float _timeTick;
    public float _ticksPerSeconds;

    // Update is called once per frame
    void FixedUpdate ()
    {
        _timeTick += 0.02F * Time.timeScale;
        _ticksPerSeconds = (1.0f / 0.02F) * Time.timeScale;
        if (_timeTick >= 0.04f) // Est-ce que 0.4 secondes ce sont écoulées ?
        {
            _timeTick -= 0.04f;

            foreach (GameObject unit in GameObject.FindGameObjectsWithTag("Unit")
    )
            {
                if (unit.GetComponent<Brain>()) // Comporte un "Brain"
                {
                    unit.GetComponent<Brain>().UnitTurn(); // Fait l'action de l'
    unité
                }
            }
        }
    }
}
```

Listing 3.16 – Code du script TurnManagerScript.cs

Ce script permet donc de contrôler le fait que chaque unité ne commence pas un nouveau tour si d'autres unités n'ont pas fini ou fait le leurs. Cependant, cette optimisation n'a pas permis de réduire le temps de chargement d'une partie, restant toujours à prés de deux minutes pour la génération de 200 unités.

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

Nombre d'unités à générer	Temps de chargements (moyenne de 3 tests)
0 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 3.17 – Code de la structure *Pool* de ObjectPool.cs

Cette méthode permet de ne pas créer une unité directement, mais de juste prendre un objet Unit dans le bassin correspondant. Les bassins sont représentés par une structure définit au listing 3.17. Elle comporte un *tag* représentant le nom du bassin, un *number* correspondant au nombre d'unités à créer dans ce bassin, et enfin un *prefab* qui est l'objet qui servira de référence pour la copie des objets. Le script ObjectPoolScript.cs (Listing 7.1) permet la gestion de plusieurs *Pools*, et dans notre cas, nous utilisons 3 *Pools* : *Light*, *Heavy* et *Explorer*.

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

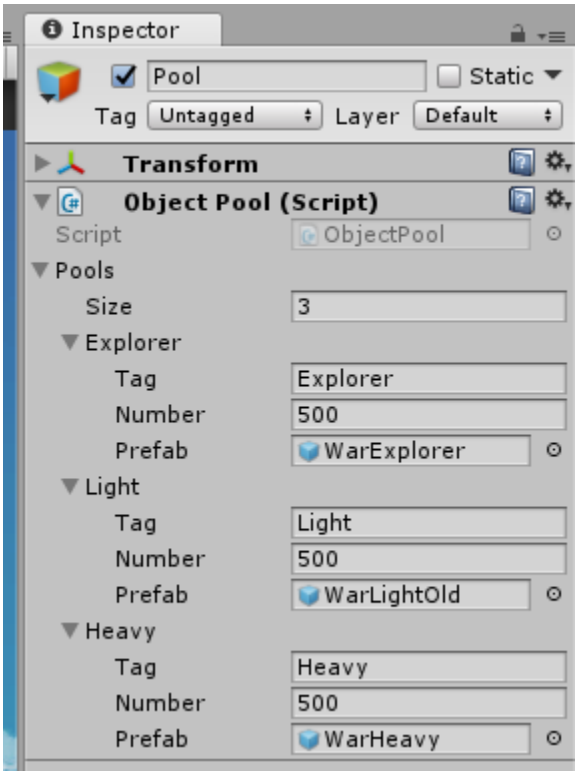

FIGURE 3.3 – Visualisation dans l'*Inspector* d'Unity des *Pools* du script ObjectPool.cs

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

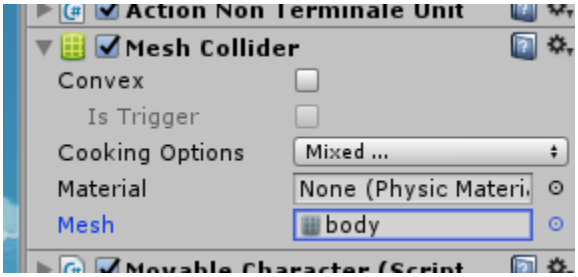

FIGURE 3.4 – Un composant *MeshCollider*

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

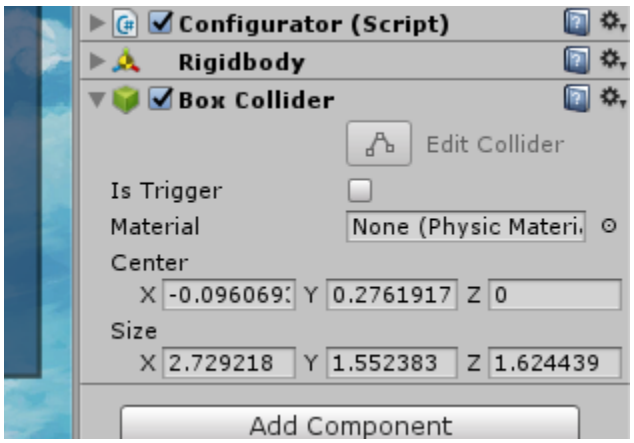

FIGURE 3.5 – Un composant *BoxCollider*

Effectivement, le temps de chargement est considérablement réduis 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 reconnaissait pas l'environnement et traversait le décor. Pour résoudre ce problème, nous avons donc définit certaines règles de collisions.

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

```
1 public class MovableCharacter : MonoBehaviour
2 {
3 [...]
```

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

Listing 3.18 – Extrait du code de MovableCharacter.cs

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

3.2.4 Fonctionalités finales.

creation d'un jeu
Mehdi

3.3 Amélioration possible

Mehdi et Maxime

3.3.1 Gestion des unités.

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

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

Le corps

Le corps de l'unité correspond à son modèle 3D, et son GameObejct Collider associé.
Pour le jeu warbot nous avons réaliser plusieurs modèles 3D.

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

L'esprit

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

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

3.3.2 Gestion des scènes.

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

- MainCamera : la caméra principale.
- TurnManager : s'occupe de la gestion des tours.
- RessourceGenerator : un générateur de ressource.
- MinimapCamera : camera permettant l'affichage de la minimap.

- ItemManager : gère le comportement des objets.
- HUD : canvas d'affichage tête haute.
- UnitManager : gestionnaire d'unité qui comporte les 4 equipes.

3.3.3 Les différentes maps

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

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

3.4 Fonctionnalités

3.5 Amélioration possible

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

Chapitre 4

Partie Interface Graphique

4.1 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.

4.2 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 séléction d'équipes**

4.2.1 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.

FIGURE 4.1 – Menu Principal, Ancienne Interface

4.2.2 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.

2. **L'éditeur en lui même**
   Nous avons maintenant devant nous un éditeur sobre. En haut à gauche se trouvent le nom de l'équipe courante, ainsi que l'untié 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.

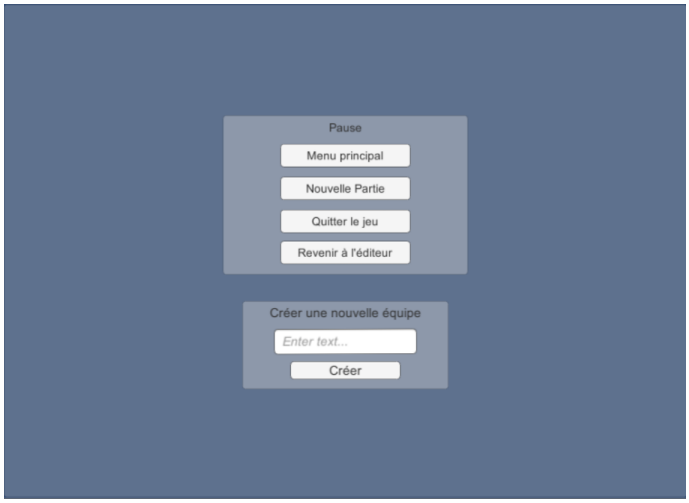

FIGURE 4.2 – Menu Principal, Ancienne Interface

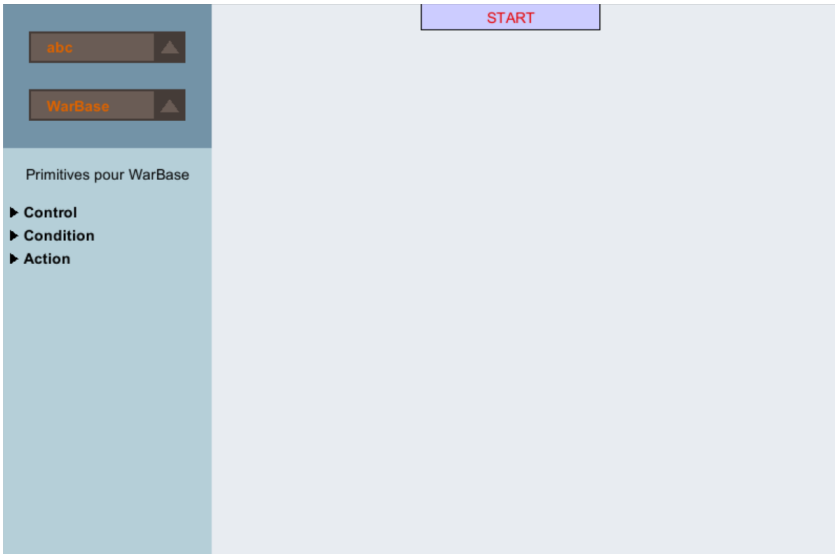

FIGURE 4.3 – Editeur de Comportement, Ancienne Interface

4.2.3 La séléction 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.

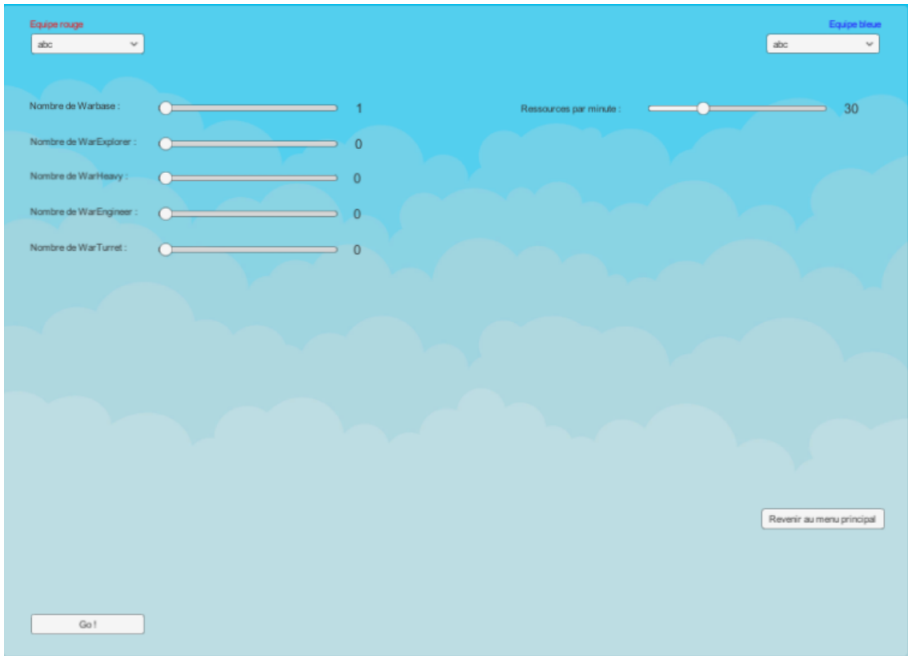

FIGURE 4.4 – Sélection des équipes, Ancienne Interface

4.2.4 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 facilement. Si on ne posait pas le bloc à un endroit précis, elle disparaissait. Il fallait alors la 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 majeur partie de son temps sur cet écran. Il fallait donc qu'il soit le plus agréable à l'œil possible, 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 parait pas intuitif. L'ajout d'une gestion des paramètres nous semblait donc indispensable.

Nous avons donc décidés 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 proposer une cohésion graphique entre chaque écran, et le jeu lui-même.

4.3 Nouvelle Interface

Nous devions repartir d'une toute nouvelle interface, tout en s'insiprant 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 pour but que 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...

4.3.1 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.

Lancer une Partie

Ce bouton permet de lancer une partie du jeu MetaBot. Le lancement de la partie prend en compte le mode de jeu sélectionné, les équipes choisies, le nombre de chaque unités en début de partie, le nombre de ressources maximum (nombre de ressource présentes en jeu au même moment), la carte de jeu. 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
2  public void StartGame()
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, gamePath);
19             gameManager.GetComponent<TeamManager>()._teams.Add(team);
20         }
21
22         GameManager setting = gameManager.GetComponent<GameManager>();
23         setting.SetSetting();
24
25         //
26         StartCoroutine(AsynchronousLoad(setting._gameSettings._indexSceneMap));
27     }
28 }
```

Listing 4.1 – Extrait du code de PlayButton.cs

Bouton Éditeur de Comportement

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

Bouton Paramètres

Ce bouton va 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 choisi, 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[
       _idPlayer]._name;
3              for (int i = 0; i < _teamDropDown.options.Count; i ++)
4              {
5                  if (_teamDropDown.options[i].text.Equals(team))
6                  {
7                      _teamDropDown.value = i;
8                  }
9              }
10 [...]
```

Listing 4.2 – Extrait du code de TeamMenuHUD.cs

De plus, à coté 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 4.3 – 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 4.4 – 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.

Choisir nombre de départ de chaque unité

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

Barre de Chargement

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

4.3.2 Menu des Paramètres

Changer le Volume de la musique

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

Choisir nombre de ressource maximum dans le jeu

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

Choisir le mode de jeu

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

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

Choisir la langue

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

Bouton Retour

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

Bouton Valider

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

Bouton Quitter

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

4.3.3 Editeur de Comportement

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

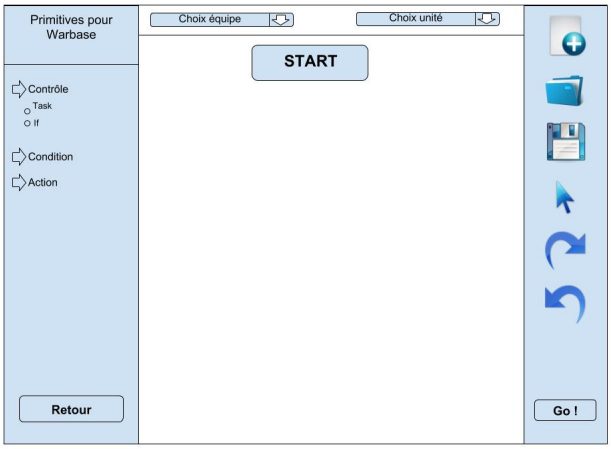

FIGURE 4.5 – Première ébauche

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

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

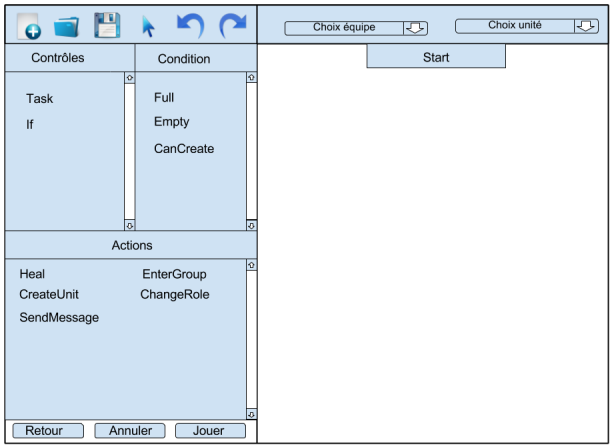

FIGURE 4.6 – Seconde ébauche

ToolBox

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

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

C) **Bouton Sauvegarde du Comportement**

Ce bouton sauvegarde le comportement de l'unité de l'équipe courante. Si on change d'unité dans la même équipe mais sans sauvegarder, alors le comportement de l'unité précédente sera perdu.

Voyons de plus près deux de ses fonctions primordiales au bon fonctionnement du script :

- **createBehavior()** :

  Nous allons créer un comportement, copiant le comportement présent dans l'éditeur. Pour commencer, nous allons répertorier toutes les pièces "If" présentes. Ensuite, pour chaque pièce If de la liste, nous allons créer l'instruction comprenant les conditions et actions, grâce à l'interpréteur.

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

- **createXML()** :

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

D) **Bouton "Undo"**

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

- **Undo()** :

  Cette fonction requiert une liste de pièces non vide pour fonctionner, c'est donc ce que l'on va vérifier en premier. Si la liste "listPieces" n'est pas vide, alors on stocke la dernière pièce de cette liste dans une variable "pieceToUndo".

  Maintenant, il faut déterminer quelle action utilisateur nous devons annuler (création ou suppression).

  Si "pieceToUndo" est active, alors nous devons simuler sa suppression, en la passant inactive. Il faudra également l'ajouter dans la liste "recoverList", qui a un comportement similaire à "listPieces", pour reconstituer une annulation.

  Si, en revanche, la pièce est inactive, il faut alors la faire réapparaitre à l'écran, en la passant en active.

E) **Bouton "Redo"**

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

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

Sélection des équipes / unités

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

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

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

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

Informations sur l'unité

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

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

Liste des pièces

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

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

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

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

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

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

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

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

Editeur

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

Voyons les scripts permettant cette gestion.

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

vérifier leur placement.
Pour se faire, nous allons appeler la fonction UpdatePuzzle() du script IfPuzzleScript.cs.
Ensuite, si la pièce Start a bel et bien une pièce "If" juste en dessous d'elle, alors on passe la valeur isValid du "If" à true.

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

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

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

4.3.4 Élément dans le jeu

Réglage du volume du son

En bas à droite de la fenêtre de jeu se trouve une icône de son et un slider. Le slider permet, comme dans le menu des paramètres, de modifier le volume du son. L'icône sert de bouton ; si

on le presse, le son passe à 0 et l'icône devient barrée. Si l'on appuie de nouveau, il redevient classique, et le son est restitué à sa valeur précédente.

Troisième partie

L'avenir du projet

Chapitre 5

Amélioration possible

Chapitre 6

Bugs

Quatrième partie

Annexe

Chapitre 7

Scripts Remarquables

7.1 ObjectPool

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ObjectPool : MonoBehaviour
{
    static public Dictionary<string, Queue<GameObject>> dictPools = new
    Dictionary<string, Queue<GameObject>>();
    public Pool[] pools;

    static bool created = false;

    void Awake()
    {
        if (!created)
        {
            DontDestroyOnLoad(this.gameObject);
            created = true;
        }
        else
        {
            Destroy(this.gameObject);
            return;
        }


        foreach(Pool pool in pools)
        {
            pool.prefab.SetActive(false);
            Queue<GameObject> queue = new Queue<GameObject>();
            for (int i = 0; i < pool.number; i ++)
            {
                GameObject instance = Instantiate(pool.prefab);
                instance.SetActive(false);
                instance.transform.parent = transform;
                queue.Enqueue(instance);
            }
            dictPools.Add(pool.tag, queue);
        }

    }


    static public GameObject Pick(string tag, Vector3 position, Quaternion
    rotation)
```

```
44      {
45          GameObject obj = dictPools[tag].Dequeue();
46          if (obj)
47          {
48              obj.transform.position = position;
49              obj.transform.rotation = rotation;
50              obj.SetActive(true);
51
52          }
53          return obj;
54      }
55
56      static public GameObject Pick(string tag, Vector3 position)
57      {
58          return Pick(tag, position, Quaternion.identity);
59      }
60
61      static public GameObject Pick(string tag)
62      {
63          return Pick(tag, Vector3.zero, Quaternion.identity);
64      }
65
66
67
68
69
70  }
71
72
73  [System.Serializable()]
74  public struct Pool
75  {
76      public string tag;
77      public int number;
78      public GameObject prefab;
79  }
```

ObjectPool.cs

7.2 MovableCharacter

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovableCharacter : MonoBehaviour
{
    public float speed;
    public Vector3 vectMov;
    public bool _isblocked;
    public float _offsetGround;
    public float _offsetObstacle;
    public float _edgeDistance;
    public bool _obstacleEncounter;


    public GameObject collisionObject;

    private Vector3 nextposition;

    public void Start()
    {
        _isblocked = isBlocked();
    }

    public void Move()
    {
        _isblocked = isBlocked();
        if (!isBlocked())
        {
            vectMov = Utility.vectorFromAngle(GetComponent<Stats>().GetHeading())
    ;
            nextposition = transform.position + vectMov.normalized * speed * 0.02
    f;

            transform.position = nextposition;

        }
        else
        {
            transform.position *= 1;
        }

    }

    public bool isBlocked()
```

```
44      {
45          return collisionObject != null;
46      }
47
48      void OnCollisionStay(Collision other)
49      {
50          collisionObject = null;
51          if (other.gameObject.tag != "Ground")
52          {
53              foreach (ContactPoint contact in other.contacts)
54              {
55                  float a = Utility.getAngle(gameObject.transform.position, contact
        .point);
56                  float b = GetComponent<Stats>().GetHeading();
57                  float A = Mathf.Abs(a - b);
58                  float B = Mathf.Abs( 360+ Mathf.Min(a,b) - Mathf.Max(a, b) ) ;
59                  if (Mathf.Min(A, B) < 90f)
60                  {
61                      collisionObject = other.transform.gameObject;
62                      break;
63                  }
64
65              }
66              /*
67              for (int i = 0; i < hits.Length; i++)
68              {
69                  RaycastHit hit = hits[i];
70                  if (hit.transform.gameObject != gameObject)
71                  {
72                      collisionObject = hit.transform.gameObject;
73                  }
74              }*/
75          }
76      }
77
78      void OnCollisionExit(Collision other)
79      {
80          collisionObject = null;
81      }
82
83  }
```

MovableCharacter.cs

7.3 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     public Percept _componentPercepts;
18     public Action _componentActions;
19     public ActionNonTerminal _componentActionsNonTerminales;
20     private string _currentAction;
21     public MessageManager _messageManager;
22     public int nbInstruction;
23
24
25     void Start()
26     {
27         //GameObject.Find("Canvas").GetComponent<HUDManager>().CreateHUD(
       gameObject);
28         GameObject.Find("CanvasHUD").GetComponent<HUDManager>().CreateHUD(
       gameObject);
29         foreach (Instruction I in _instructions)
30         {
31             InstructionEditor IE = new InstructionEditor();
32             IE._listeStringPerceptsVoulus = I._listeStringPerceptsVoulus;
33             IE._stringAction = I._stringAction;
34             IE._stringActionsNonTerminales = I._stringActionsNonTerminales;
35             _instructionsEditor.Add(IE);
36         }
37     }
38
39     public void LoadBehaviour()
40     {
41         if (GetComponent<Stats>()._teamIndex < GameObject.Find("GameManager").
       GetComponent<TeamManager>()._teams.Count)
```

```
42          {
43              _instructions = GameObject.Find("GameManager").GetComponent<
        TeamManager>().getUnitsBevahiours(GetComponent<Stats>()._teamIndex,
        GetComponent<Stats>()._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._actions.Count != 0)
59              {
60                  string _action = NextAction();
61                  if (_componentActions._actions.ContainsKey(_action))
62                  {
63                      _componentActions._actions[_action]();
64                  }
65                  else
66                  {
67                      _componentActions._actions["ACTION_IDLE"]();
68                  }
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.
        _destinataire;
```

```
84                      if (_componentActionsNonTerminales._actionsNT.ContainsKey(act
    ._intitule))
85                      {
86
87                          _componentActionsNonTerminales._actionsNT[act._intitule
    ]();
88                      }
89
90                  }
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_"
    , "")) && (percept.Contains("NOT_") ^ _componentPercepts._percepts[percept.
    Replace("NOT_", "")]()))) { flag = false; }
107         }
108
109         // bool flag2 = false;
110         // foreach (string percept in instruction._listeStringPerceptsOu)
111         // {
112         //     if (!(_percepts._percepts.ContainsKey(percept) && _percepts.
    _percepts[percept]())) { flag2 = true; }
113         // }
114         // return (flag && flag2);
115
116         return flag;
117     }
118
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

7.4 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["PERCEPT_BASE_NEAR_ALLY"] = 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.GetComponent<Stats>()._teamIndex == GetComponent<Stats>()._teamIndex)
26                 {
27                     GetComponent<Stats>().SetTarget(gO);
28                     return true;
29                 }
30             }
31             return false;
32         };
33         /** PERCEPT **/
34         _percepts["PERCEPT_CAN_GIVE"] = delegate ()
35         {
36             GetComponent<Stats>().SetTarget(null);
37             foreach (GameObject gO in GetComponent<Sight>()._listOfCollision)
38             {
39                 if (gO && gO.GetComponent<Stats>() != null &&
40                 Vector3.Distance(transform.position, gO.transform.position) < 8f
       && gO.GetComponent<Stats>()._teamIndex == GetComponent<Stats>()._teamIndex)
41                 {
```

```
42                      GetComponent<Stats>().SetTarget(gO);
43                      return true;
44                  }
45              }
46              return false;
47          };
48          _percepts["PERCEPT_CONTRACT_ELIMINATION"] = delegate () { return
        GetComponent<Stats>().HaveContrat() && GetComponent<Stats>().GetContract().
        type == "Elimination"; };
49          _percepts["PERCEPT_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"))
        { return false; }
54              EliminationContract contract = (EliminationContract)GetComponent<
        Stats>().GetContract();
55
56              foreach (GameObject gO in sight._listOfCollision)
57              {
58                  if (gO && contract._target == gO)
59                  {
60                      GetComponent<Stats>().SetTarget(gO);
61                      GetComponent<Stats>().SetHeading(getAngle(gO));
62                      return true;
63                  }
64              }
65              return false;
66          };
67
68          _percepts["PERCEPT_FOOD_NEAR"] = delegate ()
69          {
70              return (_percepts["PERCEPT_FOOD"]()) && (Vector3.Distance(
        GetComponent<Stats>().GetTarget().transform.position, transform.position) < 4
        f);
71          };
72
73          _percepts["PERCEPT_CONTRACT"] = delegate () { return GetComponent<Stats
        >().HaveContrat(); };
74          _percepts["PERCEPT_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