

UNIVERSITÉ DE MONTEPLLIER

RAPPORT DE PROJET TER

---

## Warbot : Extension de la version Unity

---



SAVIO Kurt  
GALLEAN Benjamin  
DEAU Guillaume  
CHAMOULAUD Romain

Tuteur : M. J FERBER  
Responsable : M. M  
LAFOURCADE

# Table des matières

<b>I Présentation du projet</b>	<b>3</b>
<b>1 Introduction</b>	<b>4</b>
1.1 But du projet . . . . .	4
1.1.1 Cahier des charges . . . . .	4
1.2 Notion d'agent . . . . .	5
1.2.1 Système multi-agents . . . . .	5
1.2.2 Langage et comportement d'agents . . . . .	5
<b>2 WarBot : Le mode par défaut</b>	<b>6</b>
2.1 Principe . . . . .	6
<b>II Réalisation du projet</b>	<b>7</b>
<b>3 Partie "Interpréteur"</b>	<b>8</b>
3.1 Présentation et Attente . . . . .	8
3.2 Etude de l'ancien projet et conception préalable . . . . .	8
3.3 Conception et fonctionnement Interpréteur . . . . .	12
3.4 Internationalisation du projet . . . . .	15
3.4.1 Mise en place . . . . .	15
3.5 Caméras Spécifiques . . . . .	20
3.5.1 Caméra première personne . . . . .	20
3.5.2 Caméra "Libre" . . . . .	22
3.6 Fin du jeu et Statistiques . . . . .	24
3.7 Score des équipes . . . . .	26
3.7.1 Match-up . . . . .	26
3.7.2 Système de points ELO . . . . .	27
3.7.3 Affichage . . . . .	29
3.8 Nouvelle condition de victoire . . . . .	31
3.9 Début de la création d'une nouvelle carte . . . . .	32
<b>4 Partie "Game Design"</b>	<b>35</b>
4.1 Expérience utilisateur . . . . .	35
4.1.1 Ancienne version de Warbot . . . . .	35
4.1.2 Création d'équipe sous la version actuelle . . . . .	37
4.2 Test logiciel . . . . .	39

4.2.1	Procédé . . . . .	39
4.3	Test logiciel . . . . .	40
4.3.1	Procédé . . . . .	40
4.3.2	Organisation et communication . . . . .	40
4.4	Game Design . . . . .	41
4.4.1	Amélioration demandés . . . . .	41
4.4.2	Game Document . . . . .	45
4.4.3	Design des cartes de jeu . . . . .	49
4.4.4	Equilibrage . . . . .	51
<b>III</b>	<b>L'avenir du projet</b>	<b>55</b>
<b>5</b>	<b>Amélioration possible</b>	<b>56</b>
5.1	Interpréter . . . . .	56
5.2	Game Design . . . . .	57
<b>6</b>	<b>Conclusion</b>	<b>58</b>
<b>IV</b>	<b>Annexe</b>	<b>59</b>
6.1	Code important . . . . .	60
6.2	Documents . . . . .	60

# 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.

#### 1.1.1 Cahier des charges

Le but de notre groupe était donc l'ajout de fonctionnalités à la version Unity du jeu WarBot, tout en corrigeant ces problèmes, pour la rendre utilisable et aussi stable que possible. Après inspection de l'ancien projet et la nécessité de recommencer sur de nouvelles bases, nous nous sommes d'abord concentrés sur la mise en place d'un jeu générique et stable en collaboration avec le groupe MetaBot.

Le travail sur le projet a donc été réparti en 4 sous-groupes de 2 personnes comme ci : Un groupe sur le moteur du jeu lui-même Un groupe sur l'interface graphique et des interactions avec l'utilisateur Un groupe travaillant sur le langage pour le comportement des unités et l'interprétation de ce langage Un groupe de "Game Design"

Ce rapport portera sur le sous-groupe Game Design et le sous-groupe Interpréteur, même si les groupes ont rapidement disparus afin que l'on puisse s'aider comme un groupe de 8 et que chacun puisse découvrir des parties différentes du moteur Unity 3D.

## 1.2 Notion d'agent

### 1.2.1 Système multi-agents

Un agent est une unité simple et autonome qui suit les règles données par un comportement écrit par un utilisateur ou programmeur. Le but d'un Système multi-agent est la mise en collaboration de l'ensemble des agents dans un objectif commun, ou simplement pour observer leur évolution dans un environnement défini. Il existe 2 types d'agents : Les agents cognitifs , qui disposent d'une représentation explicite de leur environnements et des agents qui les entourent. Ils disposent d'un but et peuvent s'organiser en société, tout en effectuant des collaborations si nécessaire. Les agents réactifs répondent de manière automatique à des stimuli externe / changements dans son environnement. Cet agent n'a pas de but précis et n'est pas capable de s'organiser en société. Même si les agents réactifs ne sont pas organisés et capable de s'adapter à leur environnement, un ensemble d'agents réactifs sera capable d'effectuer des actions évoluées. C'est le cas d'une société d'insectes comme les fourmis par exemple.

### 1.2.2 Langage et comportement d'agents

Le langage et les échanges entre agents seront donc 2 points clés pour avoir une bonne coordination au sein d'une société. Il est bien sûr possible d'implémenter une équipe n'utilisant aucune communication, mais les performances de l'équipe dans la réalisation de leur but seront très faible et dans le cas d'une confrontation avec une équipe organisée, leur victoire serait plus liée au hasard que de meilleures performances.

Un exemple d'agent nécessitant de la communication pour utiliser ses compétences correctement est le cas du War RocketLauncher dans WarBot , qui a une portée de tir supérieure à son champ de vision. Il va devoir s'organiser avec d'autres unités pour réussir son but.

Il existe différentes structures pour représenter le comportement d'agents : Architecture du subsomption, machine à états, ... mais nous nous concentrerons sur l'Architecture de subsomption lors de ce projet, puisqu'elle est plus simple et intuitive pour de jeunes programmeurs, qui pourraient être le public cible d'un jeu comme Warbot.

Une architecture de subsomption se décompose comme un ensemble de cas que pourraient traiter l'agent, organisés de manière hiérarchique, l'ordre est donc très important.

## Chapitre 2

# WarBot : Le mode par défaut

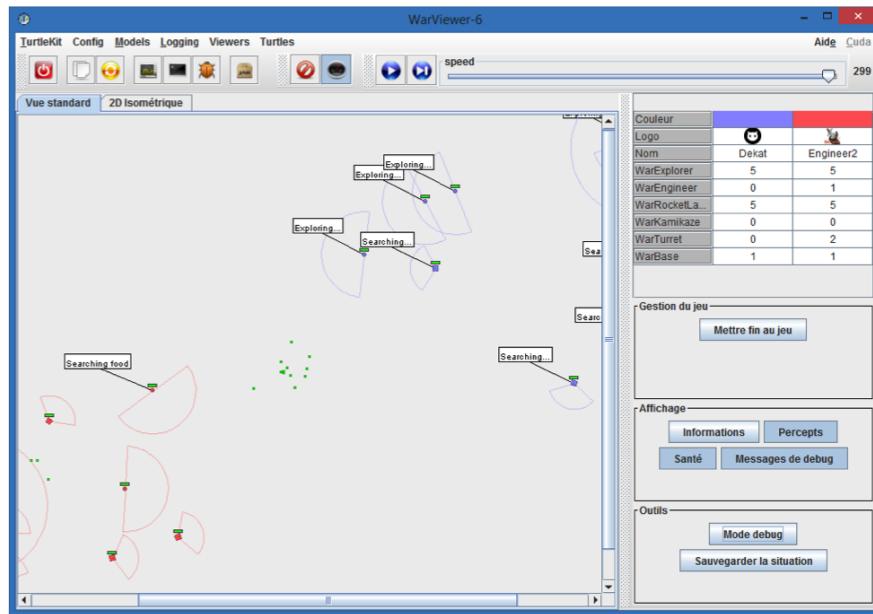


FIGURE 2.1 – WarBot, la version Java.r

### 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 afin d'être la dernière en vie. Des ressources apparaissent sur la carte et peuvent être converti en unité ou en soin. Faire le Détails des unités !

# **Deuxième partie**

# **Réalisation du projet**

## Chapitre 3

# Partie "Interpréteur"

### 3.1 Présentation et Attente

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

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

### 3.2 Etude de l'ancien projet et conception préalable

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

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

La partie concernant le traitement et l'exécution des instructions par les unités a dû être repensée totalement , car même si elle fonctionnait pour leur vision du jeu, était inutilisable dans l'objectif d'être générique. L'ensemble de la gestion de l'exécution des instructions, des droits d'accès à une primitive en fonction du type de l'unité actuelle, et la gestion des groupes étaient réalisés au sein d'un même script cumulant 2136 lignes.

Il a donc été décidé de séparer tout cela.

Le comportement d'une unité est une suite d'instructions, dont l'ordre est très important. La hiérarchie va déterminer la priorité qu'aura l'action sur le tick. Ainsi l'action la plus prioritaire dont les conditions sont acceptées sera l'action effectuée sur ce tick. Chaque instruction est toujours organisée de la même façon : - Une liste de conditions à remplir pour que l'instruction soit considéré comme acceptée - Une liste d'actions qui ne terminent pas le tour - L'action terminale

Ce sont ces variables qui seront utilisées lors du déroulement du jeu pour exécuter les primitives correspondantes.

La gestion des différentes parties à était déplacée dans la conception du moteur, pour favoriser la générativité du jeu.

Les actions ont été représentées à part, et leur fonctionnement est simple. On utilise comme base une fonction anonyme, et les différentes actions sont implémentées dans un Dictionnaire, mettant en relation un "String" correspondant au nom de l'action qu'il faut exécuter, et une fonction anonyme qui sera effectivement exécutée en jeu. Ensuite, un ensemble d'actions communes à toutes les unités seront écrites dans un script "ActionCommon" et celles-ci seront bien partagées par toutes les unités.

```
public abstract class Action : MonoBehaviour
{
    public delegate void Act();
    public Dictionary<string, Act> _actions = new Dictionary<string, Act>();
    public abstract void InitAction();
}
```

Les différences entre unités vont se faire en fonction de leur type. A partir de cet ensemble d'action commune on va spécifier les actions spécifiques ,des unités qui se déplacent, de la base qui est une unité assez spéciale puisqu'elle a la possibilité de créer de nouvelles unités, un groupe d'unités qui peuvent tirer , à travers des sous classes de cette classe commune.

```
public class ActionCommon : Action
{
    void Start(){
        InitAction();
    }

    public override void InitAction() {
        _actions["ACTION_IDLE"] = delegate () { };
    ...
}
```

Comme dit , les actions sont ajoutées via une fonction initialisant le tableau de fonctions anonymes. Il est important que , pour chaque sous groupe d'actions, on appelle les fonctions d'initialisations de tableaux des classes mères afin que l'on récupère les actions partagées par tout le monde.

```
public override void InitAction() {
    base.InitAction();
    _actions["ACTION_MOVE"]
    ...
}
```

Avant d'exécuter leurs actions , les unités doivent vérifier si les conditions que l'utilisateur a défini nécessaire sont bien validées. Pour cela, une structure très similaires de celles des actions a été conservée, avec le même principe de fonctions anonymes. Une spécificité dans la gestion des percepts (et ce sera aussi le cas pour la gestion des messages entre unités) est que l'orientation de l'unité actuelle sera modifiée selon ce qu'elle perçoit. Par exemple, si elle voit une unité agressive , elle va s'orienter vers elle, comme ça l'utilisateur ne devra pas gérer la direction de l'unité sauf si celle ci doit être orientée de manière externe (par exemple changer d'orientation si on est bloqué contre un mur).

```
_percepts["PERCEPT_ENEMY"] = delegate ()
{
    foreach (GameObject g0 in sight._listOfCollision){
    ...
        GetComponent<Stats>().SetHeading(getAngle(g0));
    ...
}
};
```

Vers le milieu du projet, il a été nécessaire de mettre en place des actions non terminales au sein du comportement des unités, puisqu'on ne pouvait plus se limiter à des actions telles que tirer ou avancer. WarBot étant une plate-forme multi-agents, il n'était pas concevable de ne pas mettre en place des messages entre les unités, afin de donner la possibilité au joueur d'exploiter pleinement les possibilités de coordination entre agents.

La structure des Action non terminales est la même que les Actions initiales, mais l'équipe moteur a pu installer un système permettant de les exécuter au cours de la partie, sans qu'elles ne mettent fin au tour de l'unité comme les actions "classiques".

La dernière structure importante était donc les messages, qu'il a fallu instaurer de manière assez différente. Ceux-ci contiennent bien sûr un expéditeur et un destinataire,mais aussi un titre qui va permettre de les différencier, pour savoir si un message va plutôt donner la position d'une ressource, une demande d'aide ou une demande d'attaque. Enfin on conservera la direction vers l'expéditeur du message, afin que le destinataire puisse se tourner facilement, à travers une action non terminale "Se tourner vers expéditeur".

```
public abstract class MessageManager : MonoBehaviour {

    public GameObject sender;
    public List<Message> _waitingMessages = new List<Message>();
    public List<Message> _currentMessages = new List<Message>();
    public string[] _messageType;
    ...
}
```

```
    public void UpdateMessage() {
        _currentMessages = _waitingMessages;
        _waitingMessages = new List<Message>();
    }
}
```

La gestion des messages dans les unités est réalisée sous forme d'une boîte au lettre. Les messages qui arrivent à l'unité pendant le tick de jeu sont stockés dans une boîte "Attente", et les messages que peuvent traiter l'unité pendant ce tick sont dans une boîte "Courante". A chaque début de tick, on va charger la boîte courante avec les messages de la boîte "Attente" et on va vider la boîte d'attente. Ainsi les messages ne vont pas traîner dans les boîtes de réception et ralentir le jeu. Cela implique par contre que les messages qui ne sont pas traités sont perdus, ce qui correspond bien à ce que l'on souhaite, puisqu'ainsi si toutes les ressources à un endroit ont été récupérées entre le moment où une unité envoie le message "Ressource" et que le destinataire le traite, le message n'emmènera pas l'unité à un endroit où il n'y aura plus de ressources. De plus, il est très important que les unités ne puissent pas stocker des positions telles que la base adverse, donc pouvoir garder un message contenant la position de la base adverse et l'utiliser alors que plus aucune unité ne se trouve près de la base adverse est contraire à l'idée de la coordination en multi agents.

L'ensemble de ces représentations permet la séparation de l'interpréteur et de l'exécution des primitives en jeu, l'interpréteur sera donc totalement externe et n'aura pas besoin d'être touché lors d'ajout d'actions, conditions, messages... De plus, cette structure utilisant les fonctions anonymes permet d'être totalement générique puisque l'ajout se fait simplement en ajoutant une entrée dans l'initialisation du tableau correspondant.

### 3.3 Conception et fonctionnement Interpréteur

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

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

Ainsi nous avons un fichier par équipe écrite par l'utilisateur, chacun détaillant le comportement des unités sous la même forme :

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

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

Différentes fonctions intermédiaires ont été implémentées afin de rendre le code plus lisible et plus réutilisable.

```
public Dictionary<string, List<Instruction>> xmlToBehavior(string teamName, string ←
  path)
```

Cette fonction permet d'aller lire le comportement enregistré à l'endroit du paramètres "path" , et dans le fichier appelé "teamName.wbt"

Les fichiers d'équipes sont au format XML , mais leur extension est ".wbt" pour coller au projet.

Tout d'abord, la vérification du fichier dans la fonction xmlToBehavior est un peu plus spécifique que uniquement vérifier si le fichier existe, on va aussi aller vérifier qu'on trouve bien un noeud contenant le nom de l'équipe. Cela permet d'être sûr qu'on ait un fichier de comportement, puisque le chargement des comportements est un moment clé dans le lancement du jeu.

```
if (file.Contains(Constants.xmlExtension) && !file.Contains(".meta"))
{
    using (var stream = new FileStream(file, FileMode.Open))
    {
        if (stream.CanRead)
        {
            Load(stream);
            XmlNode team = SelectSingleNode("//" + Constants.nodeTeam);
            if (team.InnerText != null && team.InnerText.Equals(teamName))
                l_fileName = file;
        }
    }
}
```

La fonction va renvoyer un dictionnaire, mettant en relation un nom de type d'unité (Explorateur, WarHeavy , ....) et sa liste d'Instructions , organisé par ordre de priorité dans l'architecture de subsomption.

```
XmlNodeList l_units = GetElementsByTagName(Constants.nodeUnit);
```

Ici on récupère les listes des noeuds par unité, ainsi que tous les noeuds fils que l'on devra lire. Pour chacun des noeuds d'unités, on va faire appel à la fonction clé de l'interpréteur.

```
public override Instruction whichInstruction(string unitName, XmlNode ins)
```

Cette fonction va récupérer le noeud qu'on va lui passer dans la fonction "xmlToBehavior" , et va aller construire le comportement grâce à l'ensemble des fils du noeud "ins". Il est très avantageux de pouvoir utiliser un noeud pour contenir tout le comportement pour une unité précise ici, puisque le parcours de ces fils est simple et formaté en XML. On va ainsi parcourir tous les cas qui nous intéressent et remplir une Instruction qui sera renvoyée à la fonction "xmlToBehavior". Celle-ci n'aura plus qu'à construire une liste de ces instructions au fur et à mesure du parcours des noeuds , et mettre en relation la clé , le nom de l'unité , et la valeur , la liste d'instructions, dans le Dictionnaire résultat.

Vous pourrez retrouver cette fonctionne en annexe, à la figure XX

Nous avons vu ici le traitement du cas ou on souhaitait récupérer des comportements depuis un fichier pour le renvoyer sous une suite d'instructions. Cette fonction est vitale pour le fonctionnement du moteur, mais elle sert aussi du côté de l'éditeur de comportement pour pouvoir charger un fichier dans la zone des pièces de puzzle.

Il est donc bien sûr important de pouvoir faire le chemin inverse, et donc d'écrire un comportement au format XML depuis un ensemble d'instructions récupérées depuis l'éditeur.

Cette partie sera plus facile à réaliser que la précédente L'idée ici est de stocker la structure XML de chaque instruction dans la classe Instruction même , mais pas dans une variable , plutôt dans une fonction qui va nous renvoyer le noeud XML correspond à ladite instruction. Ainsi il faudra juste ajouter ce noeud au reste du fichier de l'équipe puis le sauvegarder.

Voici à quoi ressemble cette fonction renvoyant la structure XML :

```

public XmlNode xmlStructure()
{
    XmlDocument l_doc = new XmlDocument();
    XmlNode l_ifNode = l_doc.CreateElement("instruction");

    XmlNode paramNode = l_doc.CreateElement("parameters");
    foreach (string c in _listeStringPerceptsVoulus)
    {
        XmlElement t = l_doc.CreateElement(c);
        paramNode.AppendChild(t);
    }
    l_ifNode.AppendChild(paramNode);

    if (_stringActionsNonTerminales.Length > 0)
    {
        XmlNode MsgNode = l_doc.CreateElement("message");
        foreach (MessageStruct c2 in _stringActionsNonTerminales)
        {
            XmlNode t2 = l_doc.CreateElement(c2._intitule);
            XmlNode t2d = l_doc.CreateElement(c2._destinataire);
            t2.AppendChild(t2d);
            MsgNode.AppendChild(t2);
        }
        l_ifNode.AppendChild(MsgNode);
    }

    XmlNode actNode = l_doc.CreateElement("actions");
    if (_stringAction != "")
    {
        XmlNode a = l_doc.CreateElement(_stringAction);
        actNode.AppendChild(a);
        l_Node.AppendChild(actNode);
    }
}

return l_ifNode;
}

```

La construction de la structure XML est rapide, il suffit de parcourir les percepts et les actions non terminales et de construire les noeuds que si elles sont présentes, on va ensuite construire un dernier noeud étant l'action finale , celle-ci étant obligatoire.

Nous avons ici fait le tour des possibilités de l'interpréteur, nous avons vu les représentations des primitives des unités, et comment l'interpréteur a répondu aux besoin du projet.

A partir de ce moment du projet, l'équipe Interpréteur est devenue plutôt libre puisque le plus important était d'aider les autres équipes dans l'avancée de leur partie. De nouveaux travaux allaient être donnés à l'équipe, mais dans les moments d'attente, chacun apportera son aide ailleurs dans le jeu.

## 3.4 Internationalisation du projet

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

### 3.4.1 Mise en place

Dans les fichiers du jeu , on retrouve un dossier contenant ces traductions,sous formes de fichier représentant chacun une langue. Une traduction est simplement une ligne du fichier mettant en relation une clé et sa traduction. Le rajout d'une langue est totalement simple et générique : On ajoute un fichier ayant le nom de la langue que l'on souhaite, et qui doit obligatoirement implémenter l'ensemble des clés présentes dans les autres fichiers, ou dans le fichier "exemple" mis à disposition.

Voici à quoi ressemble les fichiers de traduction :

```
# Percepts
PERCEPT_ALLY=PERCEPT_ALLIE_A_PORTEE
FALSE_PERCEPT_ALLY=PERCEPT_AUCUN_ALLIE_A_PORTEE
PERCEPT_IS_NOT_LOADED=PERCEPT_PAS_RECHARGE
FALSE_PERCEPT_IS_NOT_LOADED=PERCEPT_EST_RECHARGE
# Map en jeu
Recommencer=Recommencer
Quitter=Quitter
Health :=Vie :
Bag :=Sac :
Heading :=Orientation :
Contract :=Contrat :
```

On peut voir que le caractère séparateur est le caractère ' = ' , au départ nous utilisions le caractère ' : ' , mais il y a eu des conflits puisque les textes affichés sur l'interface utilisateur contiennent aussi les ' : '. Il est possible que le caractère ' = ' pose problème au bout d'un moment, si jamais des traductions ajoutées ultérieurement le contiennent. Il faudra donc simplement faire une recherche "Trouver et Remplacer" dans les fichiers de traduction, et changer un seul caractère dans le code du Parser des fichiers de traductions.

La traduction dupliquée de chaque texte comme vue dans la partie "Percepts" de l'exemple plus haut, à laquelle on ajoute un "FALSE\_ " , provient du fait que les conditions qui sont traitées dans le jeu peuvent être mises au négatif. Par exemple, si le comportement d'une unité de mon équipe cherche à savoir si cette unité à "un contrat d'élimination en cours" lorsqu'elle reçoit le message demandant de l'aide depuis la base, il est tout à fait possible que l'on cherche à vérifier que l'on a aucun contrat d'élimination pour pouvoir en accepter un nouveau. Au lieu de dupliquer les conditions dans le code , il a été préféré de pouvoir les rendre négatives.

La traduction fonctionne ainsi : Un premier script, le LangageLoader s'occupe de vérifier la langue qui doit être utilisée dans le fichier de configuration du jeu et va charger en mémoire un fichier de traductions. Ce script va aller chercher la langue qui doit être appliquée au projet dans les fichiers de configuration du jeu à son initialisation, et va récupérer les traductions en lisant le fichier correspondant à la langue souhaitée. Deux structures de données ont été mises en place pour faciliter les traductions

```
[System.Serializable]
public struct Traduction
{
    public string cle;
    public string valeur;
}
```

La première est une Traduction , qui va mettre en relation une clé qui sera le texte initial contenu dans les boutons et textes du projet, et une traduction qu'il faudra afficher. Ce sont les mêmes clés valeur qui seront parsés dans le fichiers de traduction.

L'appel “System.Serializable” va permettre à Unity de transformer cette structure de données dans un format que le moteur peut stocker et reconstruire si il y en a besoin. C'est par exemple le cas pour l'affichage des traductions dans l'inspecteur de Unity, ce qui a permis de grandement faciliter le débogage de ces fonctionnalités.

```
[System.Serializable]
public struct Langage
{
    public string langue;
    public List<Traduction> trads;
}
```

Ceci est la structure qui va permettre de conserver une langue, représentée simplement comme un ensemble de traductions.

Il est théoriquement possible de stocker plusieurs langues à la fois dans le “LangageLoader”, mais on va vider la liste des traductions à chaque fois que la langue est modifiée dans les paramètres, pour ne pas stocker trop de choses en runtime.

Ce script est rattaché à un objet des scènes Unity appelé le Game Manager, qui a été mis en place par l'équipe moteur et n'est pas détruit au changement des scènes. Le fait que cet objet soit conservé à travers les scènes provient d'utilisation de l'appel “DontDestroyOnLoad()” . Ainsi, si il y a besoin de partager des ressources entre les scènes tel que des configurations , ont peut utiliser le GameManager.

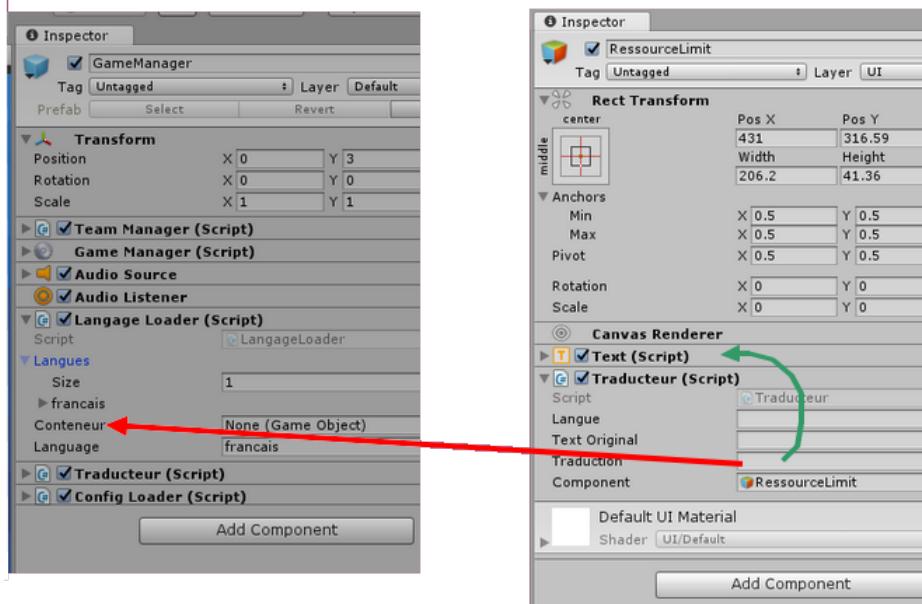


FIGURE 3.1 – Intéraction Traducteur/Objet - GameManager

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

```
public void Traduction()
{
    foreach (Langage l in GameObject.Find("GameManager").GetComponents<LangageLoader>().langues){
        if (l.langue.Equals(langue)){
            foreach (Traduction t in l.trads){
                if (textOriginal.Equals(t.cle)){
                    traduction = t.valeur;
                    return;
                }
            }
        }
    }
    traduction = textOriginal;
}
```

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

```
void Update () {
    if (gameManager.GetComponent<LangageLoader>().language != langue){
        langue = gameManager.GetComponent<LangageLoader>().language;
        Traduction();
    }
}
```

```

        if (component)
            component.gameObject.GetComponent<Text>().text = traduction;
    }
}

```

La traduction est aussi disponible sans rattacher un script "Traducteur" à un Objet de la scène. En effet, il y a beaucoup de situations où on ne peut pas rattacher ce script. Pour cela, une instance du traducteur a été placée dans le GameManager, pour qu'on puisse accéder à une traduction. La fonction `setTextOriginal()` du traducteur se charge donc de changer le texte à traduire puis appliquer la traduction.

Un des endroits qui nous a forcés à pouvoir utiliser le traducteur autrement qu'en étant rattaché à un objet de la scène est le cas des pièces de puzzle. En effet celle-ci ne sont pas dans la scène au lancement du jeu mais sont générées dynamiquement en fonction des interactions de l'utilisateur dans l'éditeur de comportement. On ne pouvait donc pas leur rattacher un traducteur.



FIGURE 3.2 – Traduction en anglais des pièces de puzzle

La langue actuelle est conservée dans un fichier de configuration, pour que le projet conserve la langue au changement de scène et quand on relance le jeu.

```

##### FICHIER DE CONFIGURATION WARBOT #####
Language=francais
Units:
WarBase:

```

FIGURE 3.3 – Fichier configuration

Mais ce fichier est caché, à moins de connaître sa position dans la hiérarchie et de devoir redémarrer le jeu pour changer la langue, il vaudra mieux utiliser les 2 boutons dans les paramètres pour passer en Français ou en Anglais. Si le besoin de nouvelles langues se fait sentir dans le futur, il sera toujours possible de transformer ces boutons en un "DropDown", un menu déroulant, afin de pouvoir sélectionner autant de langues que disponible.



FIGURE 3.4 – Boutons changement de langues

## 3.5 Caméras Spécifiques

Depuis le début de l'existence de WarBot, les caméras ont toujours été des caméras vue de dessus, puisque ce genre de caméras correspond parfaitement à la vue globale que l'on cherche à avoir pour regarder les agents évolués sur le terrain. Nous n'allions pas déroger à la règle et avons conservé cette caméra vue de dessus, mais une spécificité a été implementée par l'équipe moteur qui est une caméra qui va suivre le mouvement moyen des unités afin d'être toujours orienté vers l'action.

Le script va simplement calculer la moyenne des positions de l'ensembles des unités sur le terrain, et va déplacer l'angle de la caméra vers cette position moyenne , grâce à la fonction "LookAt" contenue dans les caméras de Unity.

```
Vector3 res = Vector3.zero;
GameObject[] units = GameObject.FindGameObjectsWithTag("Unit");

foreach (GameObject go in units)
{
    res += go.transform.position;
}
res *= 1f / (units.Length + 1);
currentTarget += (res - currentTarget) * Time.deltaTime * _speed;
transform.LookAt(currentTarget);
```

### 3.5.1 Caméra première personne

Sur une idée de notre encadrant Mr Ferber, il a été décidé de mettre en place une caméra première personne , car cette option permettait à l'utilisateur de se mettre dans la vue complète de l'unité et de pouvoir mieux suivre ce qui peut se passer autour de cette unité, et pour profiter pleinement des possibilités que permet une version Unity du jeu.

Un premier jet de la caméra avait été une caméra totalement dans l'unité, suivant ainsi sa position et sa rotation. La caméra était en soit très bien , mais le fait de suivre la rotation de l'unité était peu agréable car les unités ont tendance à beaucoup bouger et se retourner dans les coins.

D'autres essais ont été réalisés, telle qu'une caméra centrée au dessus de l'unité. On pouvait ainsi voir ce qui se déroulait autour de l'unité , tout en suivant sa position mais pas sa rotation. Idée s'éloignant trop de l'idée de caméra première personne, l'équipe est revenu sur une caméra première personne , tout en cherchant un moyen de ne pas subir les changements de rotation des unités.

La caméra première personne finale est donc située sur le capot de l'unité, mais la rotation de celle ci est déterminée par les mouvements de souris de l'utilisateur. L'utilisateur peut donc observer ce qui se passe autour de lui , tout en suivant l'unité.

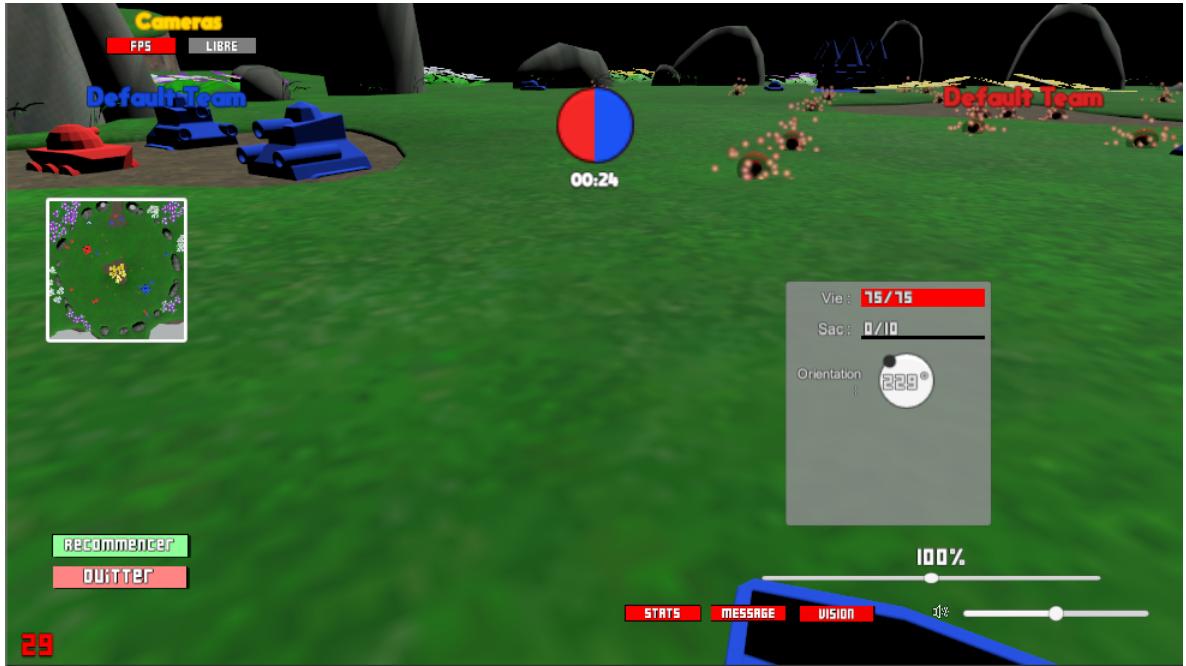


FIGURE 3.5 – Vue “FPS” dans l’unité

On peut aussi remarquer l’apparition d’un petit HUD sur la droite, qui n’est présent que lorsqu’on est en caméra première personne, et qui correspond à l’unité à laquelle on est “rat-taché”. On peut y voir la vie actuelle/max de l’unité, sa quantité de ressource dans le sac, son orientation pour du débug. Il y a aussi sur la gauche une mini carte qui apparaît uniquement en caméra fps, qui permet à l’utilisateur de suivre l’action de manière globale sur la carte de jeu tout en étant en Vue première personne.

Cette mini carte a pu être mis en place avec Unity, puisqu’avec la possibilité de placer une caméra assez haut dans les scènes des cartes du jeu , puis le rendu de cette caméra est redirigé dans une texture. Puis il suffit d’appliquer cette texture à une image , pour que cette image rende en temps réel la carte vue de dessus. On a pu ajouter un cadre à mini carte pour ne pas la confondre avec le sol.

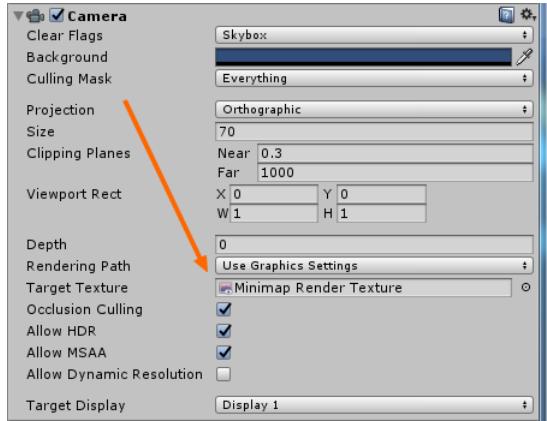


FIGURE 3.6 – Caméra enregistrant dans une texture

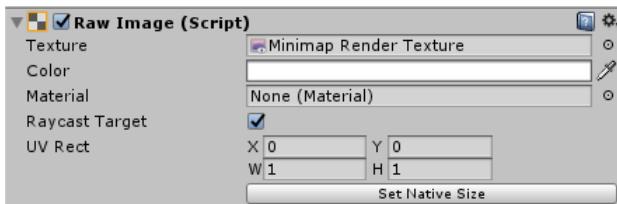


FIGURE 3.7 – Image affichant une texture

### 3.5.2 Caméra “Libre”

Au final, une nouvelle caméra à été ajouté au projet, qui a été inspirée de jeux tel que Age of Empires, Civilisation ou Dota, qui est une caméra qui laisse la liberté à l'utilisateur de se déplacer et de dézoomer ou zoomer à l'utilisateur. Nous avons bien sûr bloqué la caméra sur les côtés de la carte pour éviter que l'utilisateur puisse se déplacer tant qu'il le souhaite d'un côté ou de l'autre. Une limite à aussi été mise en place sur le zoom/dézoom en caméra libre, de telle manière qu'on ne voit pas la fin de la carte sur unity, mais aussi qu'on ne puisse pas traverser les objets en zoomant trop.

Le zoom est très classique, il suffit de scroller vers l'avant avec la souris pour zoomer et dans le sens inverse pour dézoomer. Pour les déplacements de la caméra, tant que la souris est dans l'écran la caméra ne bouge pas , et dès que la souris sort de l'écran la caméra va suivre la direction de la souris que ce soit sur le côté ou en diagonale.

```

if (Input.mousePosition.x <= 2 && Camera.main.transform.position.x > backPosition.x - terrain.bounds.size.x / 4)
    Camera.main.transform.position = new Vector3(Camera.main.transform.position.x - speed, Camera.main.transform.position.y, Camera.main.transform.position.z);
if (Input.mousePosition.y <= 2 && Camera.main.transform.position.z > backPosition.z - terrain.bounds.size.z / 4)
    Camera.main.transform.position = new Vector3(Camera.main.transform.position.x, Camera.main.transform.position.y, Camera.main.transform.position.z - speed);

```

```

if (Input.mousePosition.x >= Screen.width - 2 && Camera.main.transform.position.x <= backPosition.x + terrain.bounds.size.x / 4)
    Camera.main.transform.position = new Vector3(Camera.main.transform.position.x + speed, Camera.main.transform.position.y, Camera.main.transform.position.z);
if (Input.mousePosition.y >= Screen.height - 2 && Camera.main.transform.position.z <= backPosition.z + terrain.bounds.size.z / 4)
    Camera.main.transform.position = new Vector3(Camera.main.transform.position.x, Camera.main.transform.position.y, Camera.main.transform.position.z + speed);

```

Comme vous le voyez , on va se limiter à maximum un quart de la taille du terrain de la scène sur chaque côté, afin de ne pas trop dépasser la scène.

La gestion des deux caméras se fait à travers un système d'états gérés par des boolean, et les déplacements de la caméra s'adaptent à l'état dans lequel on se trouve. Des changements d'états sont possibles mais il sont limités. En effet , passer d'une caméra première personne à une caméra troisième personne sans repasser par la caméra initiale ne va pas fonctionner. On risque de se retrouver à se déplacer tout en étant collés au sol.

L'utilisateur peut choisir sa caméra directement depuis le HUD en jeu, a partir de 2 boutons qui vont suivre les états des caméras.

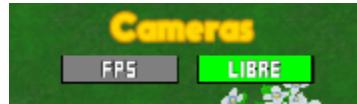


FIGURE 3.8 – Configuration à l'état initial

Initialement, on ne peut passer qu'en caméra libre à partir des boutons, puisque le passage en caméra première personne se fait en cliquant directement sur l'unité voulue. Lorsque le bouton est vert, cela signifie que l'on peut cliquer dessus pour passer sur la caméra correspondante. Si il est rouge cela signifie que l'on peut cliquer dessus pour quitter la caméra actuelle et revenir dans l'état initial. Enfin si il est grisé, cela signifie que cette caméra n'est pas disponible dans ce mode



FIGURE 3.9 – Bloquage de la caméra libre en première personne

## 3.6 Fin du jeu et Statistiques

Avec la quantité de travail pour faire fonctionner le moteur, un écran de fin du jeu n'avait pas encore mis en place, alors qu'il totalement vital ici. Si l'on souhaite que WarBot soit utilisable l'année prochaine pour l'UE Programmation Orientée Agents de Mr. Ferber, il est nécessaire de pouvoir finir le jeu et récupérer le nom du gagnant du match.

La aussi dans l'optique d'une générnicité maximale, il a fallu trouver un moyen de pouvoir ajouter très facilement des conditions et des exécutions de fin. Nous nous sommes ainsi inspirés du fonctionnement des Actions, et utilisons ainsi des fonctions anonymes stockées dans 2 tableaux. Le EndGameManager doit pouvoir vérifier à chaque tick du jeu si la condition de victoire actuelle est validée, et si c'est le cas on va exécuter une séquence d'instructions détaillées plus tard.

```
void Update()
{
    if (_tests[_wincondition]())
        _ends[_wincondition]();
}
```

On va donc initialiser les tableaux à la création du script, ainsi que récupérer la condition de victoire choisie par l'utilisateur.

Il a ensuite fallu montrer à l'utilisateur que le jeu était bien fini, ce qui nous a permis de découvrir l'animation avec Unity. Il s'agira ici que d'animation très simple.

L'animateur dans Unity va récupérer des objets de la scène et on va pouvoir appliquer des transformations à différentes propriétés de ces objets au fur et à mesure du temps.

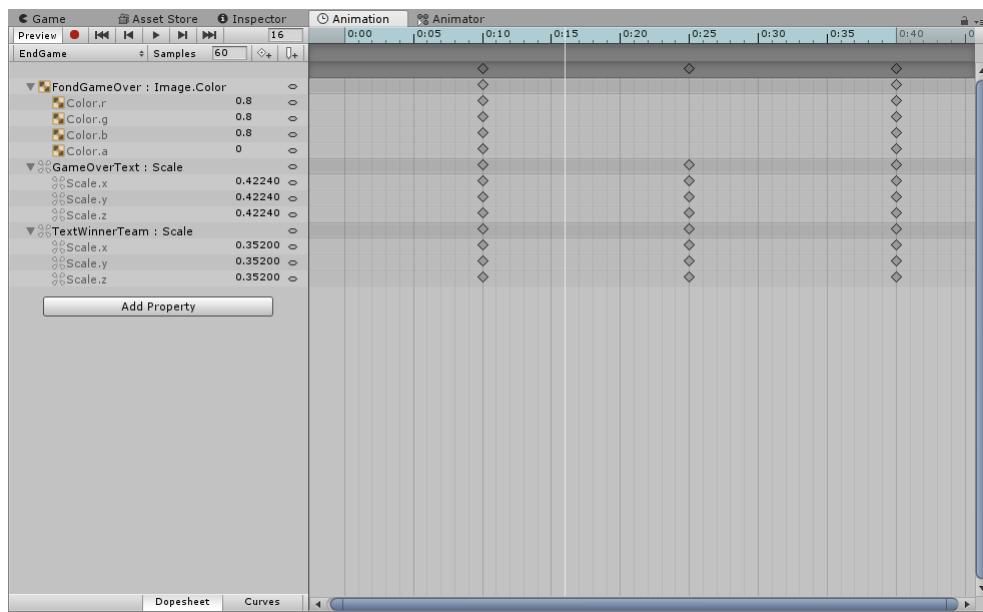


FIGURE 3.10 – Exemple d’animation dans Unity

Sur la photo ci-dessus, nous voyons sur la gauche les différents objets impliqués dans l’animation , et l’écoulement du temps sur le milieu. Il est possible d’ajouter des images clés sur la timeline, et définir les valeurs qu’auront les propriétés des objets à chaque image clé. Par exemple, si à 10 secondes je veux que la taille de mon texte soit à 100% je vais simplement mettre une frame sur 10 sec, et mettre la valeur 1 dans la propriété “Scale” du texte. Unity va donc augmenter linéairement la taille de mon texte entre 0 et 10 secondes.

L’animation ici correspond à l’apparition d’un fond coloré , avec une augmentation progressive de sa transparence sur le temps de l’animation. Par dessus, on va voir 2 textes contenant le mot “Game Over” et aussi le nom de l’équipe gagnante. Ces 2 textes vont grossir de 0 à 1.2x leur taille initiale pour ensuite revenir à 1x la taille initiale, pour ajouter un peu de dynamique à l’animation.

Une fois l’animation terminée, il faut déterminer quand a t-on besoin de jouer cette animation et certains paramètres (doit elle boucler ? , . . . ) Ici , il ne fallait absolument pas que l’animation boucle, l’écran doit rester fixe lorsque le jeu est fini. Il est donc indispensable de simplement jouer l’animation dès qu’on détecte la fin du jeu.

Unity possède un bon outil pour gérer les états des animations, il s’agit d’une machine à état, qui va pouvoir faire passer le jeu d’un état à un autre en fonction de “triggers” , que le développeur peut appeler dans un script.

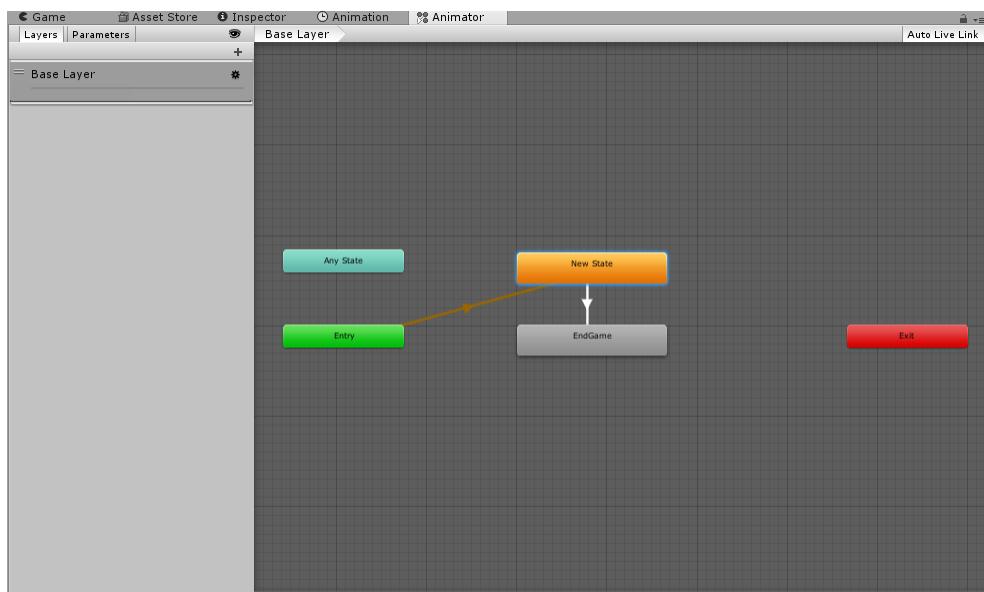


FIGURE 3.11 – Machine à états Unity

Ici, le jeu se trouve dans l’état orange lors de son déroulement. Cet état est juste ici en attente de la fin du jeu et ne fait rien d’autre. La transition entre l’état orange et l’état gris est appelé lorsque le trigger “Gameover” est activé. Arrivé dans l’état gris, Unity va jouer l’animation de

fin du jeu , mais nous avons défini qu'elle ne devait pas boucler , donc les textes vont ainsi rester affichés sur l'écran. La transition vers l'état Exit ne doit pas être faite, sinon l'utilisateur ne pourra pas rester sur l'écran de fin du jeu.

```
anim . SetTrigger ( " GameOver " ) ;
```

Cet appel doit être réalisé après avoir relié l'animation avec le script en utilisant la variable “anim”

Au final, le fond coloré à été retiré de l'animation pour laisser simplement les 2 textes blancs, car ce fond empêche l'utilisateur de voir l'état de la partie à la fin du jeu, et donc voir quelles unités il reste, ou à quel point la partie était serrée / unilatérale. Voici une image de l'écran de fin du jeu.



FIGURE 3.12 – Écran après l'animation

La gestion de la fin du jeu de chaque mode doit intégrer au moins 2 appels obligatoires. Le premier est l'activation du trigger “Gameover” déclenchant l'animation et le second étant les appels aux scripts d'enregistrement du score des équipes.

## 3.7 Score des équipes

### 3.7.1 Match-up

Les match-up permettent d'enregistrer les différents match entre équipes afin de pouvoir afficher les résultats (victoire et défaite) d'une équipe par rapport à une autre. Chaque équipe à un fichier .stat créé lors de la création d'équipe et est modifié à chaque fin de match. le format

de ce fichier est

*TeamName/MatchCount/WinCount*

A la fin d'un match, les fichiers de chaque équipe sont modifiés afin de mettre à jour ces statistiques.

Pour cette mise à jour, pour chaque équipe adverse, 2 cas sont possibles :

- les équipes se sont déjà affrontées avant la recherche, dans la liste des équipes affrontées, l'équipe adverse actuelle et on incrémenté MatchCount. Si l'équipe actuelle est l'équipe gagnante, on incrémentera aussi WinCount

```
int numVal = int.Parse(DetailStat[1]);
numVal++;
DetailStat[1] = numVal.ToString(); //NbMatch
if (Team == Winner)
{
    numVal = int.Parse(DetailStat[2]);
    numVal++;
    DetailStat[2] = numVal.ToString(); //NbVictoire
}
```

- les équipes ne se sont jamais affrontées on ajoute une ligne au fichier tel que TeamName/1/0. cette ligne sera TeamName/1/1 si l'équipe actuelle est l'équipe gagnante

```
i++; //taille du tableau + 1
System.Array.Resize(ref Stats, i);
if (Team == Winner)
    Stats[i - 1] = Teams[0] + "/1/1";
else
    Stats[i - 1] = Teams[0] + "/1/0";
```

Les match-up étaient affichés sur l'écran d'accueil. Cependant, prenant trop de place, ils ont été effacés de l'écran d'accueil et pourraient être affichés sur une fenêtre à part si l'équipe qui va récupérer le projet le souhaite.

### 3.7.2 Système de points ELO

Le système ELO est un système de classement par niveau de capacité relatif d'un joueur pour les jeux à deux joueurs. Pour WarBot, ce système sert à évaluer les capacités d'une équipe complète afin de pouvoir estimer ses performances en combat 1 contre 1.

La formule permettant de calculer le ELO d'un joueur à l'issue d'un match est

$$E_{n+1} = E_n + K * (W - p(D))$$

$p(D)$  est la probabilité de gain par rapport à la différence du ELO des deux joueurs. il peut s'exprimer sous la forme

$$p(D) = \frac{1}{1 + 10^{\frac{-D}{400}}}$$

D est la différence entre le ELO du joueur et le ELO du joueur adverse

$$D = E_{Joueur1} - E_{Joueur2}$$

W est le résultat du match. en cas de victoire W = 1, en cas de défaite W = 0 et en cas d'égalité W = 0.5

K est un facteur dépendant de la volatilité du classement et permet d'amplifier les variations afin que les nouveaux joueurs progressent rapidement vers le réel niveau. Pour WarBot, nous avons utilisé les coefficients suivant

$$K = 80 \iff ELO < 1000$$

$$K = 50 \iff 1000 \leq ELO < 2000$$

$$K = 30 \iff 2000 \leq ELO < 2400$$

$$K = 20 \iff ELO \geq 2400$$

$E_n$  est le ELO du joueur avant le match.

Lors de la création d'une équipe, nous mettons le ELO de cette équipe à 2400. Ce choix réduit cependant les variations dû à K. En effet, K sera à 30 dès lors du premier match d'une équipe mais sera élevé pour une équipe faible afin qu'elle remonte vite en cas de victoire face à une équipe plus haute.

Exemple :

L'équipe 1 avec un ELO de 1800 affronte l'équipe 2 avec un ELO de 2550

Les probabilités de gain des deux équipe sont

$$p_{Equipe1}(D) = 0.01$$

$$p_{Equipe2}(D) = 0.99$$

Les facteurs K des deux équipe sont

$$K_{Equipe1} = 50$$

$$K_{Equipe2} = 20$$

N'ayant pas d'égalité dans WarBot, les possibilités sont les suivantes :

- L'équipe 1 gagne.

$$E_{Equipe1n+1} = 1800 + 50 * (1 - 0.01) = 1800 + 50 * 0.99 = 1800 + 49 = 1849$$

$$E_{Equipe2n+1} = 2550 + 20 * (0 - 0.99) = 2550 + 20 * -0.99 = 2550 - 19 = 2531$$

- L'équipe 2 gagne.

$$E_{Equipe1n+1} = 1800 + 50 * (0 - 0.01) = 1800 + 50 * -0.01 = 1800 - 0 = 1800$$

$$E_{Equipe2n+1} = 2550 + 20 * (1 - 0.99) = 2550 + 20 * 0.99 = 2550 + 0 = 2550$$

Suite à ce calcul, les résultats sont stockés dans des fichiers .ELO afin de pouvoir le re-calculer le ELO à la fin de chaque match et de l'afficher dans un classement.

### 3.7.3 Affichage

Les points ELO des équipes ainsi stockés vont pouvoir être récupérés à chaque fois qu'une équipe va être sélectionnée sur le menu principal , on va afficher son score ELO juste en dessous, en appliquant une teinte, selon si le ELO est supérieur à la valeur initiale (teinte plutôt verte) ou inférieur (teinte plutôt orange).

Voici un exemple



FIGURE 3.13 – Affichage des points

### 3.8 Nouvelle condition de victoire

A la fin du projet , après avoir intégré la gestion de la fin du jeu, Mr Ferber nous a donné l'idée de mettre en place un second mode de jeu pour montrer les possibilités de généricité du projet. Le temps étant assez court, l'idée de mettre en place un mode de jeu entier était impossible , et nous avons mis en place une condition de victoire différente, mais toujours basée sur le mode de jeu WarBot.

Retenant l'idée d'un mode réfléchi par l'équipe Game Design, la possibilité de passer en "Course aux ressources" a été rajoutée dans les paramètres. Si ce mode est sélectionné , l'utilisateur va pouvoir définir un temps limité en secondes avant l'arrêt de la partie, ainsi qu'une limite de ressource à atteindre. Le jeu s'arrête si le temps imparti est atteint, ou si l'une des équipes à atteint le nombre de ressources spécifié dans l'inventaire de leur base. Dans le cas où le temps est atteint, c'est l'équipe ayant le plus de ressources dans sa base qui a gagnée.



FIGURE 3.14 – Écran après l'animation

Le but était de mettre en place un mode rapide mais qui justifierait le fait de pouvoir refaire des équipes adaptées à ce mode, au lieu d'utiliser uniquement les équipes de base qui ont été faites pour WarBot.

Nous avons ainsi pu rajouter une entrée dans les tableaux de fins du jeu, qui au lieu de vérifier le nombre de bases différentes toujours sur le terrain, va vérifier d'un côté dans chacune des bases du jeu si l'une d'entre elles a atteint la limite de ressource que l'on a données, et on va ensuite vérifier sur le Timer de jeu ,qui était déjà présent sur la carte par l'équipe Moteur, si la limite de temps est atteinte. Dans ce cas la on va déclencher la fin du jeu. Il a été nécessaire de traiter un cas d'égalité dans ce mode-ci, puisque si aucune des équipes n'est capable de ramasser des ressources avant la fin du jeu il ne faut pas donner une victoire aléatoirement.

La gestion de l'égalité se fait via un boolean, qui passe à vrai tant que l'on rencontre des bases qui ont le même nombre de ressources que la base considérée comme gagnante. Lors de l'affichage de l'animation , on va afficher le terme "Égalité" à la place d'une équipe gagnante, et surtout il ne faudra surtout pas faire l'appel de changements de stats des équipes.



FIGURE 3.15 – Écran d'égalité

### 3.9 Début de la création d'une nouvelle carte

Souhaitant s'essayer à la création d'une nouvelle carte, et suivant la création du mode de jeu des ressources, nous avons réfléchi à l'ajout d'une carte dans le jeu. Le temps étant trop court, et ayant d'autres priorités, la carte n'a pas été terminée, et ainsi l'équipe qui récupèrera le projet sera libre de pouvoir l'adapter comme bon leur semble.

Même si le but était la découverte de Unity sur la création de scène, nous avons cherché un moyen de rendre cette carte unique et ayant un intérêt. Avec une grande aide de l'équipe Game Design nous avons trouvé une configuration de carte équilibrée et utile.

Les bases ont été écartés aux 4 points cardinaux afin de les éloigner et limiter les combats. L'action est déplacée dans la zone centrale, où les ressources apparaissent et se retrouvent à même distance des 4 bases, les agents vont se déplacer en majorité.

L'organisation des murs autour de chaque base rend plus difficile l'accès aux bases adversaires à cause des collisions qui vont renvoyer les agents dans une autre direction.

Bien sûr une équipe oubliant les ressources va tomber sur la base adverse si elle est située en face de soi, mais il n'est pas sûr que cette technique soit si rentable dans le mode Course aux ressources.

Un thème de carte qui plaît beaucoup et qui n'avait pas encore été exploré dans les 5 cartes du jeu est l'espace. Une idée était de représenter la carte comme un HUB central de vaisseau spatial, ce qui collait bien à la forme qu'on lui donnait.

Il a fallu trouver des textures de murs et de sol en rapport avec le thème, et nous avons pu apprendre le fonctionnement des “Skybox” dans Unity. La skybox d'une scène un ensemble de 6 images correspondant à l'intérieur d'un cube (respectivement haut, bas , gauche, droit, arrière, avant) qui va être rendu par le jeu en après de tout le reste (donc en arrière-plan) , pour ne pas avoir un fond gris peu agréable pour l'utilisateur. La skybox utilisée dans cette carte est tout simplement un fond noir, avec des étoiles pixélisées réparties.

Cette skybox a été générée par l'outil “Spacescape”, qui génère des skybox par couches, en générant des étoiles plus ou moins grosses.

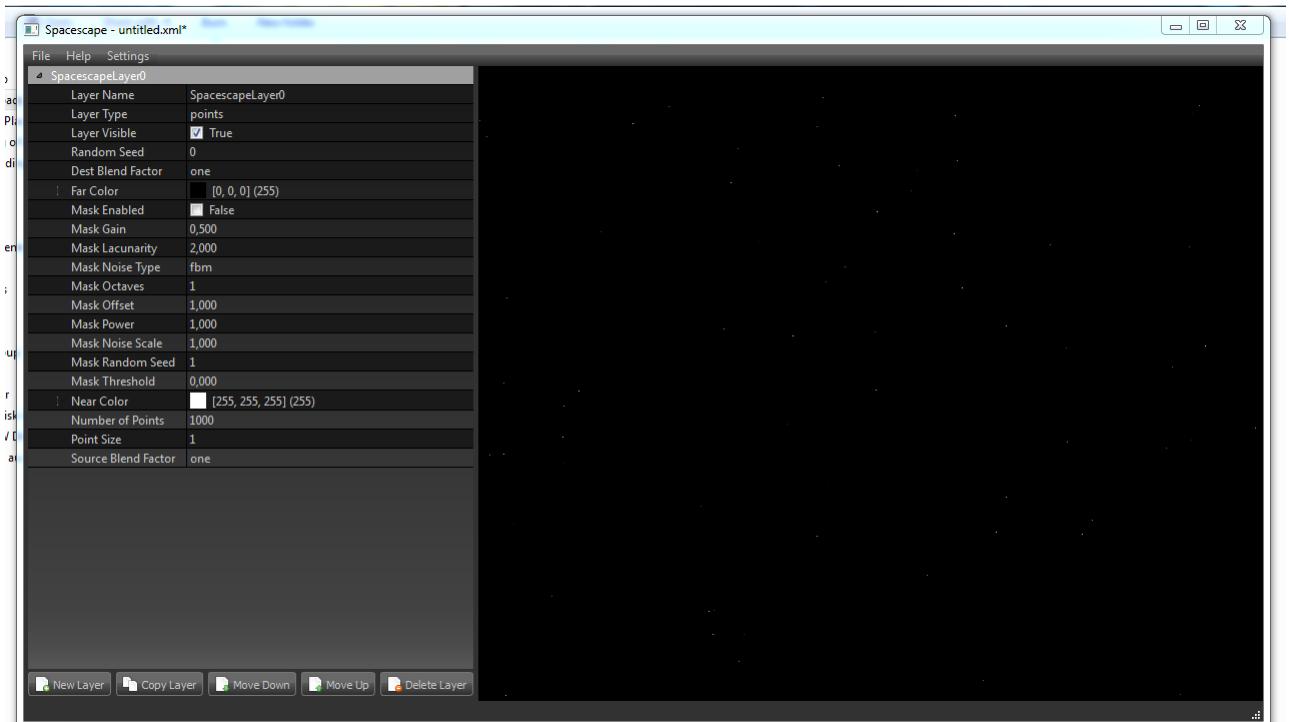


FIGURE 3.16 – Outil Spacescape pour générer une skybox

La carte a d'abord été mise en place sous forme de rectangle, pour bien comprendre le fonctionnement de Unity. Il n'a pas été malheureusement possible de finir et de modifier la forme pour s'adapter au schéma, la carte est donc restée dans les fichiers du projet tel quelle et n'est pas accessible dans le jeu.

Voici une image de l'état de la carte :

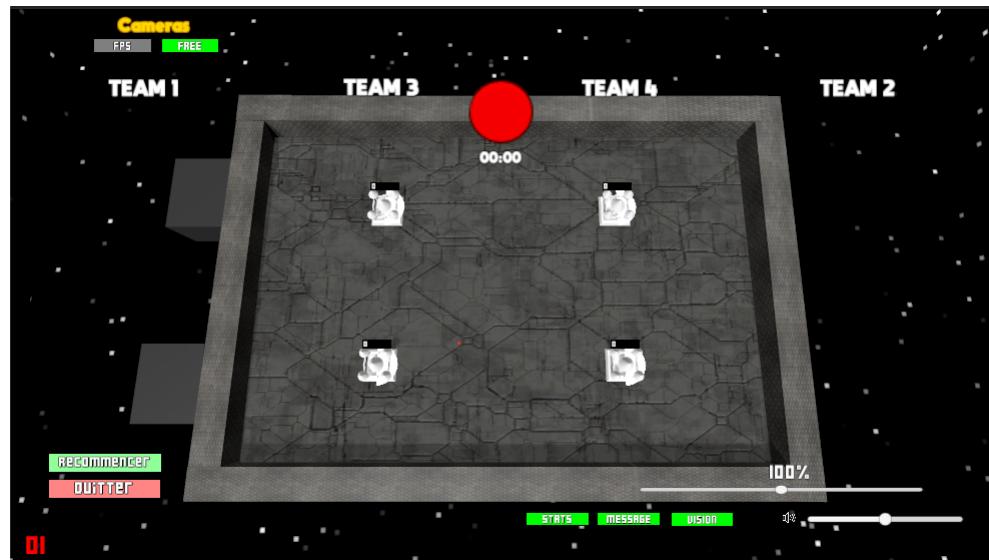


FIGURE 3.17 – Etat actuel de la carte

# Chapitre 4

## Partie "Game Design"

### 4.1 Expérience utilisateur

#### 4.1.1 Ancienne version de Warbot

Les premières semaines de ce projet ont étées l'occasion pour toute l'équipe de se familiariser avec la version de WarBot qui fut créée par les étudiants de l'année précédente. Nous avons donc immédiatement essayé de créer des équipes pour tenter d'apprioyer son éditeur de comportement.

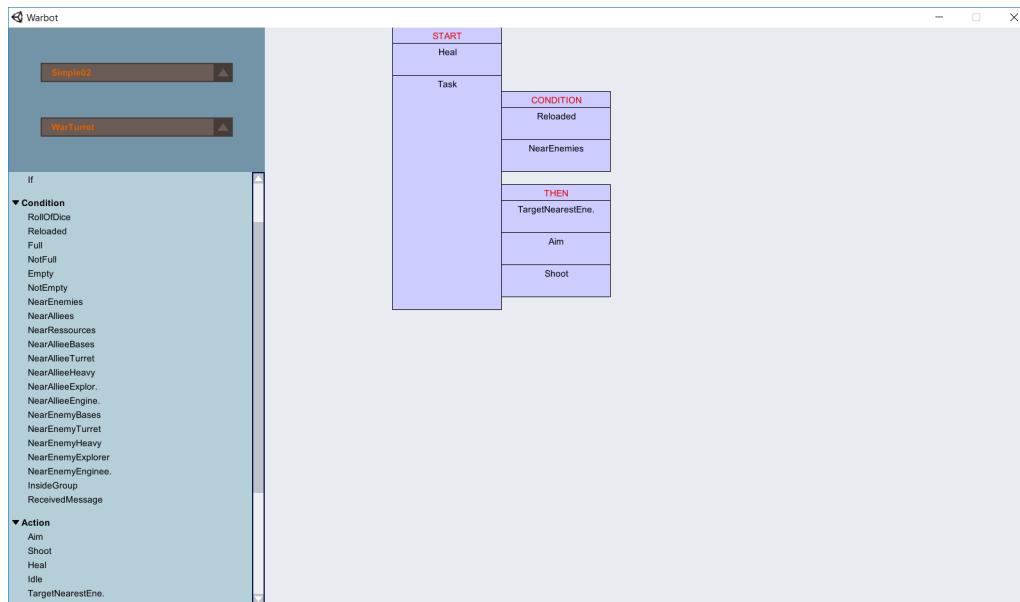


FIGURE 4.1 – Aperçu de l'ancien éditeur

Nous nous sommes très vite rendu compte de plusieurs problèmes d'ergonomie qui rendait la création d'équipe difficile. Tout d'abord, pour placer un bloc, il fallait :

- Cliquer sur le nom d'une primitive dans le menu de gauche
- Maintenir le clic de la souris et déplacer le pointeur de la souris sur un bloc déjà placé
- La primitive est ensuite placée en-dessous du bloc où se trouve le pointeur de la souris

Si cela semble simple sur le papier, en pratique c'est une tout autre histoire car le moindre faux mouvement positionnait le bloc au mauvais endroit ou le faisait disparaître. En effet, si le curseur de la souris ne se trouvait sur aucun bloc au moment où on relâchait le clic, le bloc était tout simplement supprimé.

Dans le cas où l'on avait mal positionné le bloc il fallait alors le déplacer. Cependant, cliquer sur un bloc déjà existant pour le bouger entraînait également le déplacement des blocs situés en-dessous de celui dont nous voulions modifier la position. Il fallait donc, dans un premier temps, sélectionner les blocs en-dessous de la primitive mal placée pour les positionner au-dessus de cette dernière pour pouvoir déplacer la primitive mal placée sans modifier le reste du comportement.

Supprimer un bloc s'il n'est attaché à aucun autre bloc semble être une bonne idée car cette technique permet de ne pas avoir trop de blocs inutiles qui traînent dans l'interface. Seulement, un simple clic sur une primitive supprimait cette dernière ainsi que toutes celles placées en dessous de la primitive supprimée. Ceci s'explique par le fait que lors du clic sur une primitive, l'éditeur estime que nous sommes en train de la déplacer. Or, comme on relâche la souris sans la bouger cette dernière est déplacée à un endroit où il n'y a aucun bloc entraînant ainsi la suppression des primitives.

Ce problème était particulièrement dérangeant lorsque l'on souhaitait utiliser WarBot avec un pavé tactile car une simple pression sur ce dernier est compté comme un clic. Il n'était donc pas rare de supprimer une grosse partie de notre comportement en voulant repositionner son doigt sur le pavé tactile.

A ceci s'ajoute l'absence de certaines fonctionnalités comme la possibilité d'annuler une modification ou de choisir quand sauvegarder le fichier, ce qui implique que chaque erreur était sauvegardée et la seule manière de la corriger était de replacer de nouveaux blocs pour revenir au comportement voulu.

Cette version de WarBot possédait plusieurs types de primitives : les primitives de contrôle if et task, les conditions et les actions. Parmis les actions, certaines terminaient le tour de l'unité (tirer, bouger, se soigner...) alors que d'autres permettaient à l'unité de continuer son tour (Rejoindre un groupe, changer l'orientation de l'unité...). Il était difficile de savoir lesquelles terminaient le tour ou non car il n'y avait aucune différence visuelle entre ces deux types de primitives. Le seul moyen de savoir si une primitive permettait de continuer son tour était donc de passer sa souris sur le nom d'une primitive dans le menu à gauche pour avoir une description de la primitive dans laquelle il était précisé si cette action terminait le tour de l'unité. Dans le cas où l'action n'était pas terminale, aucune précision n'était rajoutée.

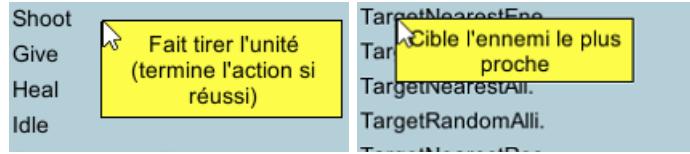


FIGURE 4.2 – Comparaison descriptions

Un dernier problème d’ergonomie important est l’emplacement des boutons dans cet éditeur. En effet, il était possible de revenir au menu principal, de créer une nouvelle partie, de quitter le jeu et de créer une nouvelle équipe depuis l’éditeur. Cependant, les boutons qui permettaient ces actions étaient cachés dans le menu pause de l’éditeur. Le simple fait de pouvoir mettre en pause un écran qui n’en a pas besoin est déjà suffisamment contre-intuitive pour privilégier le fait de placer des boutons directement dans l’interface de l’éditeur plutôt que dans un menu accessible uniquement via la pression de la touche échap du clavier.



FIGURE 4.3 – Fichier configuration

#### 4.1.2 Crédit d’équipe sous la version actuelle

La création d’équipe sous la nouvelle version de WarBot était un mélange de recherche de bugs et d’expérience utilisateur.

Pour créer une équipe nous partions d’une stratégie générale et nous développions les comportements à partir de cette idée. N’ayant pas encore la possibilité d’utiliser des groupes, les stratégies se limitaient souvent au choix entre une équipe défensive ou offensive.

Au vu de notre expérience, il y a une seule chose qui sépare ces deux genres d'équipes : l'ordre dans lequel on veut gérer les actions. Une équipe défensive va préférer mettre ses primitives pour se soigner au plus haut de sa hiérarchie pour que ce soit une action prioritaire, assurant la survie alors qu'une équipe offensive va préférer placer les actions de tirs et de recharge avant les soins pour s'assurer d'effectuer des dégâts plus souvent.

De manière générale, il y a quelques astuces importantes à prendre en compte pour optimiser le comportement d'une équipe. La première est de placer toutes les actions qui ne terminent pas le tour d'une unité comme les envois de message tout en haut de la hiérarchie pour s'assurer que ces actions soit effectués. On fait ensuite suivre les actions qui ont le moins de chances de se produire qui s'avèrent être les actions qui se font alors que l'unité est immobile (tirer, ramasser une ressource...) en les plaçant dans un ordre en particulier suivant la stratégie d'équipe que l'on souhaite appliquer.

La deuxième est de préférer le système de création d'unité à l'aide de la vie car ce procédé permet de créer des unités dès le début de partie alors qu'une équipe qui crée des unités à partir des ressources devra attendre beaucoup plus longtemps avant de voir ses premières unités se former. Les ressources ramenées à la base sert alors à la soigner, ce qui permet de créer de nouvelles unités. L'avantage numérique étant très important dans le mode de jeu WarBot, il vaut mieux s'assurer de posséder beaucoup d'unités dès le départ si nous ne voulons pas nous mettre dans une situation dans laquelle il est presque impossible de reprendre l'avantage. L'inconvénient de cette technique est le fait de ne pas avoir beaucoup de vie mais l'avantage numérique nous semble beaucoup trop important pour arriver à former une équipe efficace qui préfère utiliser les ressources pour créer ses agents.

Dernièrement, il vaut mieux respecter ce genre de comportement pour les déplacements :

```
if(blocked) {
    heading random
    move
}
#Tout mouvement qui demande de se diriger vers quelque chose (base, message...)
if( ){
    move
}
```

De cette manière, le fait de se débloquer est prioritaire à tout déplacement. Ce qui permet de ne pas rester bloqué dans un obstacle.

Lors de la création des équipes, nous avons pu apprécier les améliorations ergonomiques par rapport à l'ancienne version de WarBot. Le code est bien plus facile à lire grâce au code couleur qui différencie bien les différents types de blocs. De plus, le système de suppression de blocs a été changé : il faut maintenant faire un clic droit sur un bloc. Cette méthode permet de placer des blocs sans les rattacher aux blocs du comportement ce qui rend le déplacement de primitives plus simples et agréables.

L'ajout d'un système de grille est également très agréable car les blocs sont naturellement attirés les uns les autres ce qui permet de ne pas avoir besoin d'être précis lorsque que l'on place des blocs.

Il est très facile de comprendre si nos blocs sont pris en compte dans le comportement grâce au système de colorisation des pièces qui sont grisées si elles ne sont pas prises en compte et colorées si la primitive fait parti du comportement.

L'ajout de boutons de sauvegarde, de retour arrière et restauration rende également la partie conception bien plus agréable car il est possible de corriger ses erreurs bien plus simplement qu'avec la version de l'éditeur développé l'année dernière.

La seule chose qui nous dérange encore sur la version actuelle est l'impossibilité d'utiliser la molette de la souris pour agir sur la partie de la fenêtre où l'on construit le comportement d'une unité ce qui demande de déplacer l'ascenseur avec sa souris, chose qui nous éloigne de l'endroit où se situe les primitives du comportement. Comme on utilise cet ascenseur pour pouvoir déplacer les blocs, il est dommage de devoir s'en éloigner avant de pouvoir effectuer le déplacement.

La partie propriétés, bien qu'elle demande à être revu graphiquement fait son travail et permet de se donner une idée de l'intérêt de chaque unité qui permettra aux nouveaux joueurs de rapidement saisir les atouts et les faiblesses de chaque unité. Chose qui leur permettra de concevoir plus rapidement des équipes qui exploitent toutes les unités de manière optimale.

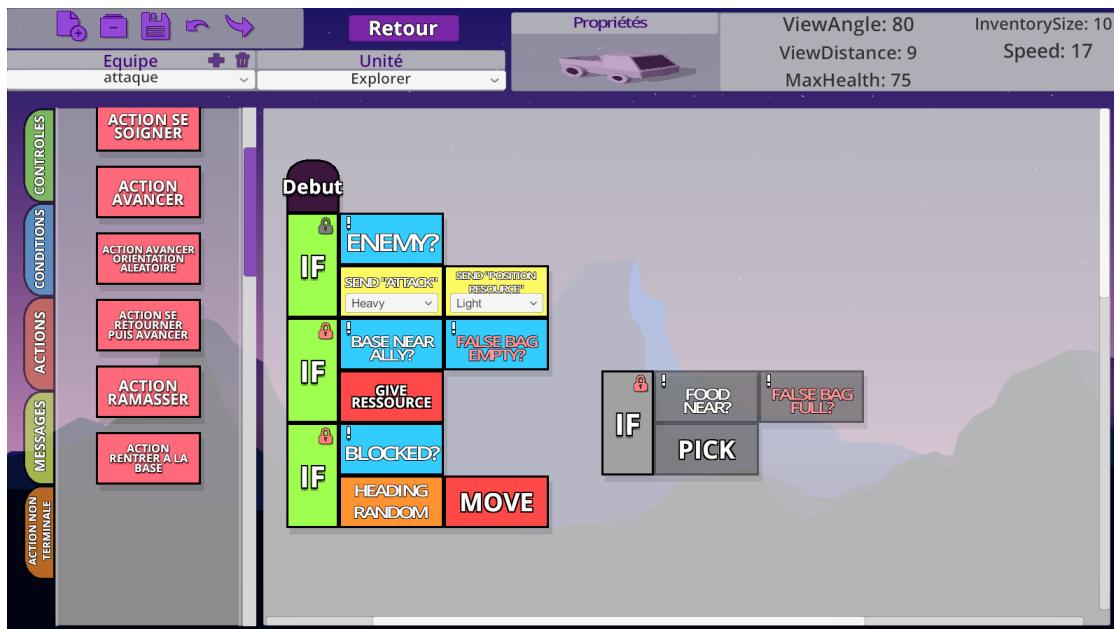


FIGURE 4.4 – Aperçu du nouvel éditeur

## 4.2 Test logiciel

### 4.2.1 Procédé

Malheureusement, les groupes ne sont plus présents dans cette version, ce qui rend le nombre d'option bien plus limité.

La disparition des unités WarEngineer et WarTurret rend également le jeu moins complet. Les

tourelles étant des outils très intéressant pour les stratégie défensive, ces équipes sont bien moins fortes que dans la version faite l'année dernière.

## 4.3 Test logiciel

### 4.3.1 Procédé

Même si nous arrivions à trouver des bugs lors d'une utilisation normale du logiciel, ce n'est pas suffisant pour trouver tous les bugs présents sous WarBot.

En effet, dans le domaine de tests logiciels il existe plusieurs familles de tests. Il est important de préciser que nous avons effectué les tests en boîte noire, c'est-à-dire en ayant très peu, voire pas du tout, de connaissance du code du logiciel. La première est le test des nouvelles fonctionnalités dont le but est de se focaliser sur les nouveautés de la version pour vérifier qu'elles marchent bien. Pour effectuer ces tests, nous avons créé des équipes très simples n'utilisant presque qu'une seule primitive à savoir celle que l'on voulait tester. Nous lancions ensuite une partie et observions le comportement des unités pour déterminer si les ajouts étaient bien opérationnelles.

Une fois les tests des nouvelles fonctionnalités terminés, nous faisions des tests combinatoires dont le but était de vérifier que de mêler plusieurs primitives dans un même comportement ne posait pas de problème. Pour cela, nous complexifions les équipes créées pour tester des primitives à part en rajoutant de nouvelles primitives dans leur comportement.

Une autre famille de test très importante est la famille des tests de non régression. Le but de ces tests est de tester toutes les anciennes fonctionnalités pour vérifier si elles fonctionnent toujours de la même manière que dans la version précédente. Pour cela, nous lancions une partie avec des équipes créées lors des versions précédentes et nous vérifions que les agents n'avaient pas de comportement étranges ou non désirés.

Les derniers tests que nous avons effectués sont les tests aux limites. Ces tests consistaient à pousser WarBot dans ses retranchements en essayant des choses qui étaient susceptibles de ne pas avoir été pensée par les développeurs. Par exemple, nous avons essayé de créer des équipes dont le nom contenaient des caractère spéciaux, de lancer des parties avec énormément d'unités, de constamment changer la vitesse en jeu...

### 4.3.2 Organisation et communication

Une fois qu'un bug était trouvé, nous tentions de le reproduire plusieurs fois afin de comprendre comment le déclencher. Une fois que le déclenchement du bug était compris, nous l'ajoutions à la liste de bugs.

Cette liste étant un Google Document partagé, n'importe qui possédant le lien vers ce document étaient capable de le voir et de l'éditer. Le but était donc de centraliser tout le processus de communication des bugs : n'importe qui pouvait ajouter à la liste un bug qu'il avait découvert, indiquer la correction d'un bug ou la réapparition de certains bugs que l'on croyait corrigés.

Ce document possède plusieurs tableaux : un pour les bugs non traités, un pour ceux qui sont réapparus après une correction, un pour les bugs patchés (non corrigé mais un compromis a été réalisé pour que ça ne pose plus de problème), un pour les bugs à vérifier et un dernier pour les bugs corrigés.

Il nous arrivait également de rajouter des liens vers des vidéos des bugs enregistrées lors de nos sessions de recherche de bugs ou bien des captures d'écrans, rendant ainsi plus explicite les problèmes qu'apportent le bug ou la manière de le réaliser.

Lorsqu'un bug était plus important, comme l'impossibilité de lancer une partie ou de créer un comportement, nous en informions directement l'équipe via notre serveur Discord qui permet d'effectuer des conversations instantanées avec tous les membres du groupe. Ces conversations peuvent être aussi bien textuelles que vocales. L'intérêt de passer directement par Discord était de provoquer une situation d'urgence où toute l'équipe était mise au courant du problème sans qu'ils aient besoin de passer sur le Google Document.

## 4.4 Game Design

### 4.4.1 Amélioration demandés

Deux semaines après le début du projet, il nous a été demandé de rédiger un document expliquant comment nous voulions voir langage évoluer. Nous avons donc commencé à imaginer comment améliorer le langage de l'éditeur de l'ancienne version.

Tout d'abord, nous voulions introduire une nouvelle fonctionnalité : l'objectif. Au vu de la nature d'une Finish-State Machine qui est le concept sur lequel se base le comportement des agents de WarBot, il est très dur de réaliser des stratégies car la moindre réponse à un réflexe nous fait perdre notre trajectoire car l'orientation de l'unité est modifiée. L'objectif est donc un objet contenant simplement des coordonnées qui se conserve au fil des tours, contrairement à la cible. Il a été conçu notamment pour se diriger vers des messages. Le but est de conserver la position de l'émission d'un message pour pouvoir se diriger vers ce dernier à tout moment.

Ainsi, il est possible de réagir à des réflexes sur un tour puis de modifier son orientation en direction de l'objectif pour reprendre le chemin vers ce dernier aux tours suivants. Cette fonctionnalité permet également de tirer vers une position en dehors de notre champ de vision, possibilité imposée par M. Ferber qui veut que les WarHeavy aient une distance de vue très faible pour que les autres unités soient obligées de les guider.

Le gros problème de l'ancien éditeur venait de son ergonomie. Pour l'améliorer, nous avons proposé un système de code couleur et de forme de pièces, réalisé à l'aide d'un logiciel de traitement d'image pour servir de référence à l'équipe interface :

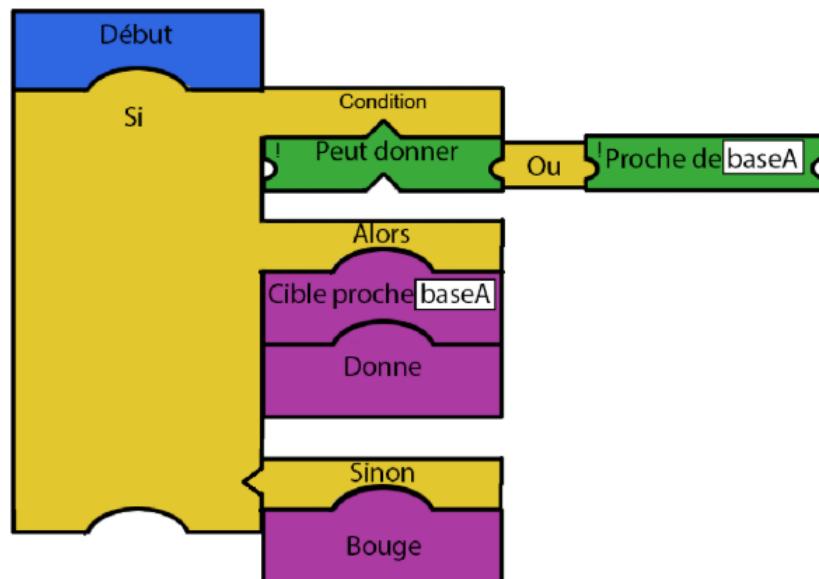


FIGURE 4.5 – Aperçu d'un comportement avec la refonte proposée

Ici, toutes les structures de conditions sont en jaunes, toutes les actions en violettes et toutes les conditions en vertes.

Le système de forme de bloc permettait également de comprendre immédiatement où on pouvait placer des blocs. Les actions peuvent être positionnées sous les blocs possédant un creux ovale alors que les conditions ne peuvent être positionnées qu'en-dessous des blocs possédant des creux triangulaires. Les actions terminales ne possèdent pas de creux en bas du bloc pour exprimer le fait qu'il est impossible de positionner d'autres blocs à leur suite.

Les conditions ont également des "!" en haut à gauche de leur bloc. Un clic sur ce dernier changeait la couleur du bloc exprimant ainsi que nous ne testions plus si la condition est vraie mais si la condition est fausse.



FIGURE 4.6 – Condition normale et sa négation

Au vu de notre expérience utilisateur, nous avons également voulu l'ajout de plusieurs primitives. Les premières étant un nouveau groupe de blocs : les variables. Voici la liste de celle que nous voulions rajouter à WarBot :

- Vie actuelle qui donne la vie restante de l'unité
- Ressource actuelle qui donne le nombre de ressources dans le sac de l'unité.
- Porté de tir qui donne la porté de l'arme de l'unité
- Distance objectif qui donne la distance entre l'unité actuelle et son objectif
- Distance cible qui donne la distance entre l'unité actuelle et sa cible
- Nombre membre qui donne le nombre de membre au sein d'un groupe précisé en paramètre
- Lancé de dé qui retourne un nombre aléatoire entre 1 et la borne maximale passée en paramètre
- Constante qui permet d'écrire n'importe quelle donnée

Ces variables servaient surtout à utiliser les opérateurs de comparaison ( inférieur, supérieur, égale) qui est l'une des conditions que nous voulions ajouter :

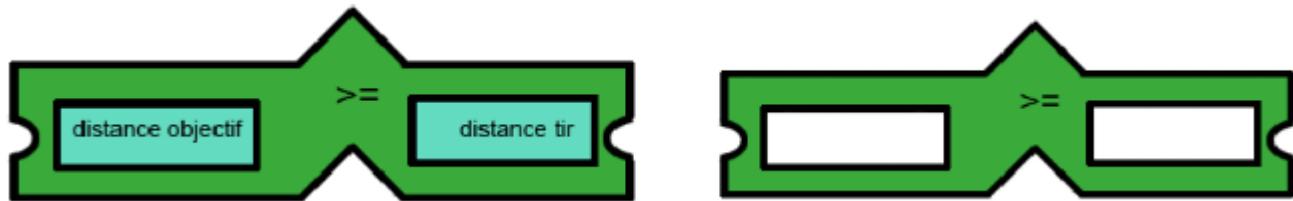


FIGURE 4.7 – Condition de comparaison

Les blocs bleus cyans représentent les variables qui sont de simples rectangles pour exprimer le fait qu'on ne peut les positionner qu'ici.

Les autres conditions que nous voulions ajouter sont :

- A un objectif qui retourne true si l'unité à un objectif
- Niveau de vie qui prend un double entre 0 et 1 comme paramètre et qui renvoie true si

$$vieAtuelleDeL'unite > vieMaximaleDeL'unite * parametre$$

- Niveau de ressource qui prend un double entre 0 et 1 et qui renvoie true si

$$nombreRessourceDeL'unite > tailleInventaireDeL'unite * parametre$$

Pour les derniers ajouts de primitives, il nous fallait deux primitives pour gérer la notion d'objectif :

- Définir objectif qui affecte à cette variable un objet possédant les coordonnées de la cible actuelle
- Supprimer objectif qui permet de libérer l'unité de son objectif.

Dernièrement, nous avons listé quelles primitives n'était pas intéressantes et celles qu'il fallait modifier :

- Task car c'est une primitive qui ne se différencie pas assez de la primitive if pour être conservée
- Not full et not empty car la possibilité d'exprimer la négation d'une condition les rends redondantes avec les négations de full et empty
- SelectUnite ne servait qu'à créer des unités. En effet, il fallait sélectionner le type d'unité que nous voulions créer puis appeler la primitive create unit. Le fait d'appeler simplement create unit et de sélectionner l'unité voulu via un menu déroulant était donc suffisant.

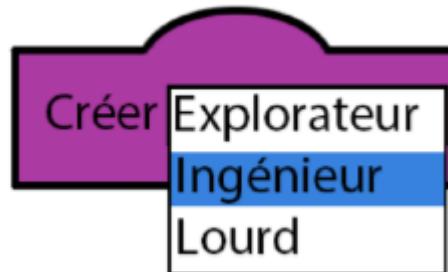


FIGURE 4.8 – Le menu déroulant déroulé

- TargetNearest existait en plusieurs exemplaire (un pour chaque type d'unité, le tout multiplié par deux pour différencier les ennemis des alliés). Nous voulions les remplacer par une variable "Cible" que l'on aurait affecté via une primitive qui aurait eu un menu déroulant pour sélectionner ce qu'on voulait cibler (ennemi, allié, message, ressources...)
- Near avait les mêmes problèmes que TargetNearest. Nous voulions donc utiliser une solution similaire en créant une primitive "proche de" ayant le même système de menu déroulant.

#### 4.4.2 Game Document

L'un des deux buts majeurs de notre équipe lors de ce projet a été de créer de nouveaux modes de jeu, différent de WarBot.

Il existe deux approches créatrices majeures en conception de jeu. L'approche bottom-up où l'on part d'un ou plusieurs éléments de gameplay auquel on tente d'appliquer un contexte est l'une d'entre elles. C'est par exemple le cas du jeu Splatoon de Nintendo qui n'était au départ qu'un prototype qui utilisait le gyroscope de la console Nintendo Wii U afin de repeindre un cube. A partir de cette idée principale d'autres éléments comme le fait de se déplacer dans sa peinture ont été rajouté puis il a été décidé du contexte et du scénario du jeu à la toute fin. L'autre approche est l'approche top-down dans laquelle on commence par déterminer le scénario ou une intention de jeu pour lesquels on crée des mécaniques qui s'inscrivent dans l'expérience voulue. C'est l'approche qui est choisi pour des jeux à licence comme les Batman Arkham où il faut créer des mécaniques qui s'appuient sur la personnalité de Batman et du monde décrit dans ses comics. D'autres jeux comme Journey partent d'une envie de créer une certaine expérience de jeu qui dans le cas présent était de créer un jeu social se basant sur le partage d'émotions. Nous avons choisi l'approche top-down car notre but dans la création de nouveau mode était de créer des modes se différenciant du mode WarBot. Il était donc plus logique de partir d'une intention de jeu précise suffisamment différente de ce qui est demandé dans le mode WarBot pour se détacher de ce dernier.

Nous avons donc commencé à chercher des idées en ayant eu pour consigne la très vague instruction "faire quelque chose de différent de WarBot sans trop s'en éloigner". Parmis ces idées, nous avons décidé de développer celles que nous trouvions les plus pertinentes à savoir un jeu en coopération basé sur la collecte de ressources, un simili RPG où les agents devaient accepter des quêtes de personnages non joueur et un jeu d'énigme.

Au final, seulement l'idée du jeu de coopération a été accepté par le reste de l'équipe mais il a été jugé trop complexe et il nous a donc été demandé de retravailler ce mode. On nous a également demandé de travailler sur la conception d'un mode de jeu de capture de drapeau car implémenter un mode de jeu qui partage un grand nombre de primitives avec WarBot semblait bien plus faisable que la création d'un mode de coopération qui demandait de plus amples réflexions afin de le faire fonctionner sous le moteur de jeu développé cette année.

Après avoir sélectionné les idées à développer, nous avons commencé à penser la conception des modes. En partant de l'idée générale, nous avons définis quel genre d'objectifs nous voulions imposer aux joueurs pour rendre le mode intéressant. Par exemple, pour le mode de capture de drapeau, nous voulions obliger le joueur à créer une équipe capable de défendre une zone. Une fois cet objectif défini, nous réfléchissions aux problèmes éventuels que l'idée allait impliquer et nous ajoutions de nouvelles règles et mécaniques dans le but de diriger le mode de jeu dans la direction voulue. Toujours pour la capture de drapeau, nous avions peur que les parties soient trop passives car dans la version la plus courante de ce mode de jeu, chaque équipe possède un drapeau à défendre et doit en plus récupérer celui de l'adversaire. Nos craintes se basaient sur l'hypothèse que l'attaque serait moins intéressant que la défense, ce qui aurait entraîné de nombreuses parties interminables où aucune des deux équipes n'arrive à récupérer le drapeau adverse. Pour résoudre ce problème, il a été décidé qu'il n'y aurait qu'un seul drapeau n'appartenant à aucune équipe et que l'équipe qui le ramènerait à sa base remporterait le match.

Une fois toutes les mécaniques définies nous passions à la rédaction du game document dans lequel nous expliquons les règles du mode (objectifs, interaction entre agents...), les éléments de jeu dont nous avons besoin, les intentions du mode et en quoi il se sépare de WarBot, une liste des primitives nécessaire pour ce mode ainsi qu'une ébauche de direction artistique qui sert de contexte pour justifier les éléments de gameplay.

La grande difficulté de cette partie était le fait que le joueur n'intervient pas au cours de la partie : son seul but est de créer des comportements pour une équipe qui jouera toute seule. Ce problème devient double parce que nous n'étions pas particulièrement au courant des méthodes utilisées lors de conception d'un jeu ce qui nous a donc demandé à la fois d'apprendre la méthodologie et de l'adapter à un jeu où il n'y a pas de joueur.

En effet le simple fait de ne plus prendre le joueur en compte dans l'équation enlève des principes fondamentaux de la conception de jeu comme la boucle de gameplay ou le game feel. Le principe de la boucle de gameplay est que pour un objectif donné, il existe un challenge et une récompense qui doit être proportionnelle à la difficulté qui lui a été associé. Il existe des boucles de gameplay à plusieurs niveaux : micro, moyen et macro. Voici ce que des boucles de gameplay dans un RPG pourrait être.

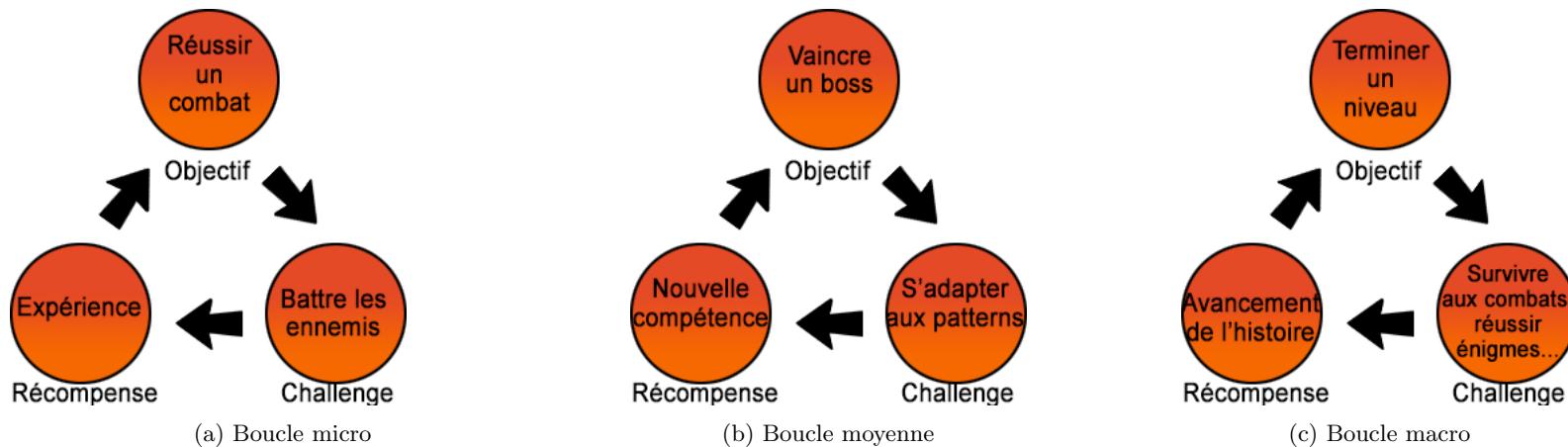


FIGURE 4.9 – Boucle de gameplay d'un RPG

Seulement, avec WarBot, nous devons supprimer les boucles micro et moyenne pour ne conserver qu'une boucle macro :

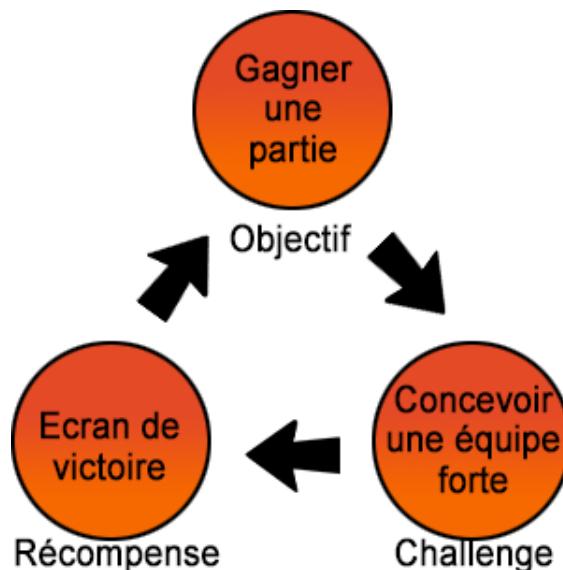


FIGURE 4.10 – Boucle de gameplay de Warbot

Normalement, nous devrions posséder des boucles moyennes elles mêmes composées de boucles micros au sein de la boucle macro pour stimuler l'intérêt du joueur au cours de la partie, chose impossible avec WarBot car même si l'on peut définir des objectifs au niveau moyen (défendre la base, lancer une attaque...) et micro (détruire une unité, créer une nouvelle unité...) il est

impossible de créer du challenge au sein de la partie pour le joueur. Ce challenge est néanmoins présent lors de la conception d'une équipe mais comme il est impossible d'avoir une récompense directement après avoir surmonté le challenge car il faut lancer une partie pour obtenir cette récompense, l'effet de la résolution d'un objectif est amoindri. Le fait que le challenge soit toujours le même pour un objectif différent peut également entraîner l'ennui chez le joueur qui l'amènera à se désintéresser du jeu.

Le game feel, quant à lui, est un concept défini par Steve Swink dans son livre Game Feel : A Game Designer's Guide to Virtual Sensation. Ce concept explique que chaque jeu doit penser le contrôle de son personnage ainsi que les mécaniques de son jeu afin de servir une intention de jeu. Par exemple, un jeu de combat comme Street Fighter 5 va donner peu de contrôle aérien à ses personnages et faire en sorte que les coups des personnages soient suffisamment lent pour permettre aux joueurs de lire et punir ses adversaires. De même, les coups lents permettent de lier plus facilement les coups d'un combo ensemble car le jeu souhaite que les joueurs s'adaptent et réagissent à la phase de jeu actuel plutôt que de demander de bêtement appuyer sur des touches en espérants qu'il se passe quelque chose. Ces décisions amènent à un ressenti de lourdeur et de rigidité du gameplay qui dirigent l'expérience de jeu vers ce que les concepteurs ont prévus. A l'opposé, on retrouve des jeux comme Super Mario 64 où les concepteurs ont voulu donner une grande liberté de mouvement à Mario car le but des game designer est de s'assurer que le cœur du jeu, à savoir les mouvements, était suffisamment intéressant pour porter le jeu tout entier. Pendant les premiers mois de développement, ce jeu ne possédait d'ailleurs aucun niveaux car les concepteurs se sont assurés que les déplacements à eux seuls étaient amusants en laissant Mario sur un sol plat où le seul but était d'attraper un lapin, sans risque d'échec. Mario donne l'impression de flotter dans les airs, ce qui a été décidé afin que le joueur puisse facilement modifier sa direction en l'air pour atterrir sur les plateformes.

Le problème avec WarBot est alors évident : le joueur ne contrôle pas ses unités ce qui implique qu'il est impossible de déterminer un game feel qui aurait pu nous servir pour déterminer quel genre de mécaniques et d'objectifs nous aurions pu implémenter si nous avions un game feel que nous souhaitions implémenter au jeu.

#### 4.4.3 Design des cartes de jeu

Lors de la décision d'ajouter un mode de jeu où la condition de victoire était le nombre de ressources récoltées, nous avons conçu trois nouvelles cartes. Dans ces trois cartes, on retrouve une zone violette qui représente la zone où les ressources apparaîtront et des maisons de différentes couleurs représentant les positions des différentes bases.

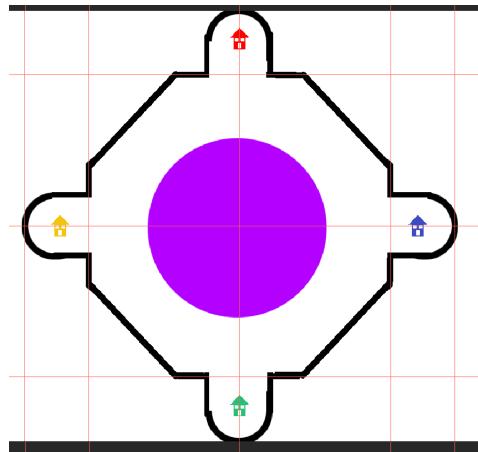


FIGURE 4.11 – Carte 1 mode ressource

Cette carte est la carte la plus simple. Les bases sont dans des espaces clos assez difficile d'accès pour centraliser l'action dans une seule zone : le centre. La zone centrale est en forme d'octogone afin de limiter les zones où il n'y a rien d'intéressant toujours dans l'intention de focaliser l'attention des agents sur la zone d'apparition des ressources.

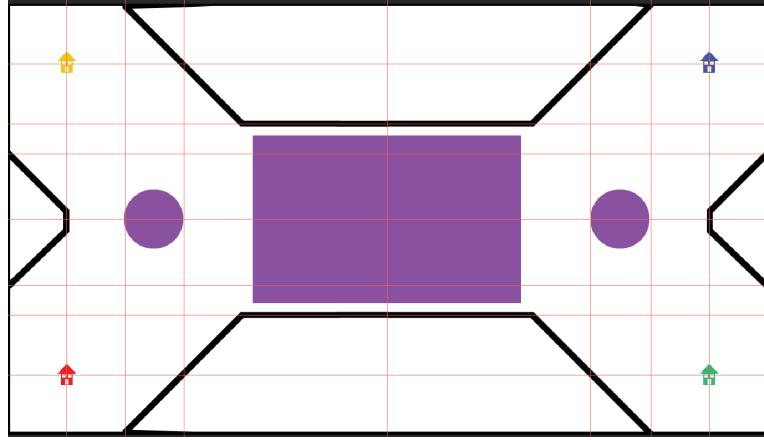


FIGURE 4.12 – Carte 2 mode ressource

Ici, on tente de séparer les zones d'intérêt en ajoutant deux zones d'apparition de ressources séparés de la zone central pour provoquer des affrontements entre les deux équipes disposé sur le même côté de l'écran. Ces zones sont volontairement petite pour faire en sorte que les agents préfèrent aller dans la zone centrale qui serait bien plus abondante en ressource.

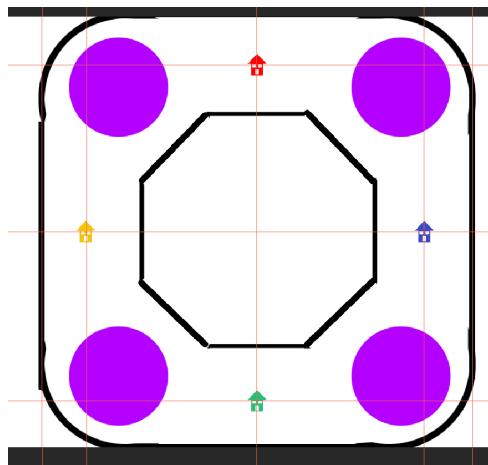


FIGURE 4.13 – Carte 3 mode ressource

Pour cette dernière carte, le but était de séparer les affrontements dans plusieurs zones pour obliger les affrontements entre deux équipes pour trancher avec les autres cartes où les quatre équipes s'affrontent au même endroit, d'où la zone centrale inaccessible.

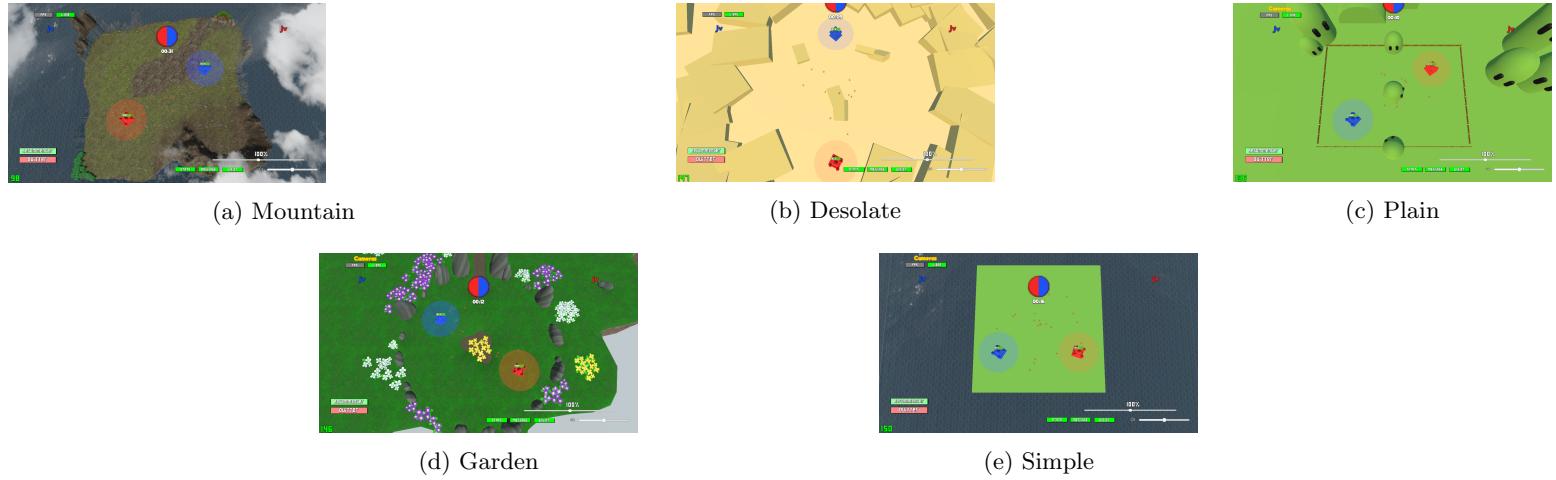


FIGURE 4.14 – Les cartes jouables de WarBot

L'équipe moteur s'est occupée de la création des cartes actuellement présentent dans le projet. La carte Simple était au départ la carte qui servait à faire nos tests dans un environnement où peut d'éléments risquaient de perturber le comportement des agents. Les cartes montagne et Desolate ont été conçues pour permettre plus de possibilités de collision avec le décor. Quand à la carte garden, c'est un model 3D trouvé sur Inernet qui a été ajouté pour vérifier la possibilité d'ajouter des cartes n'étant pas prévues spécifiquement pour notre moteur au départ.

#### 4.4.4 Equilibrage

L'équilibrage est la phase où on ajuste les caractéristiques de chaque unité afin que chacune d'elles possède ses forces et ses faiblesses. Si une unité est trop forte, les joueurs la privilégieront alors que si une unité est trop faible, les joueurs ne voudront pas perdre leur temps à créer un comportement pour une unité qui ne rivalise pas contre ses adversaires. La diversité de comportements possibles serait alors amoindri.

Dans le cas de WarBot, les statistiques se modifient via un fichier texte sous ce format :

```

Units:
  WarBase:
    MaxHealth: 300
    MaxInventory: 200
    PerceptionRadius: 120
    SpawnDelay: 1.5
  WarTurret:
    MaxHealth: 30
    MaxInventory: 20
    PerceptionRadius: 70
    ReloadTime: 0.1
    Cost: 20
  WarExplorer:
    MaxHealth: 45
    MaxInventory: 100
    PerceptionRadius: 80
    Speed: 65
  
```

```

Cost: 10
WarEngineer:
    MaxHealth: 80
    MaxInventory: 100
    PerceptionRadius: 50
    Speed: 30
    SpawnDelay: 4.0
    Cost: 15
WarHeavy:
    MaxHealth: 100
    MaxInventory: 50
    PerceptionRadius: 85
    Speed: 55
    ReloadTime: 1.0
    Cost: 30
Ressources:
    HealCost: 30
    HealValue: 20
    TakeCount: 20
    GiveCount: 20
    GiveDistance: 30
    MaxRessources: 20

```

Pour rendre le calcul des statistiques plus simples, nous avons créé un tableur. Pour se faire, nous nous sommes servi de la configuration de la version Java et nous avons calculé un ratio pour chaque statistique de chaque unité en prenant l'explorateur comme référence :

	WarExplorer	WarEngineer	WarHeavy	WarTurret	WarBase		WarBullet	WarRocket
Ratio Java :						Vitesse		
Rayon perception	1	0.833	0.833	1	2	1	0.6	
Distance de vue	1	0.5	0.5	0.833	1.33	1	1	
Coup création	1	5	2.5	6		Radius Explo		
Vie max	1	7.5	4	20	30	Degat	1	5
Inventaire	1	1	1	2.5	250000			
Vitesse	1	0.5	0.4					
Rechargement				1	1			

FIGURE 4.15 – Tableau des ratio

Nous modifions ensuite les valeurs dans le tableau “valeur par défaut”, ce qui modifie automatiquement les valeurs dans le tableau “calcul via ratio” qui nous sert de référence pour remplir le tableau “Rééquilibrage Unity” dans lequel nous modifions les valeurs à la main pour équilibrer au mieux la version actuelle.

Valeur défaut :	Rayon perception	360		Vitesse	400	
	Distance de vue	80		Lifespan	3	
	Coup création	20		Radius Expl		
	Vie max	200		Degat	10	
	Inventaire	100				
	Vitesse	65				
	Rechargement	1				
Calcul via ratio :	Rayon perception	360	299.88	299.88	360	720
	Distance de vue	80	40	40	66.64	106.4
	Coup création	20	100	50	120	0
	Vie max	200	1500	800	4000	6000
	Inventaire	100	100	100	250	25000000
	Vitesse	65	32.5	26	0	0
	Rechargement	0	0	1	1	0
Rééquilibrage Unity :	Rayon perception	360	360	360	360	360
	Distance de vue	80	40	70	90	100
	Coup création	20	100	50	100	
	Vie max	200	200	160	200	1000
	Inventaire	100	100	40	40	500
	Vitesse	65	50	45		
	Rechargement			1	0.1	

FIGURE 4.16 – Calcul des statistiques

La seule unité pour laquelle nous ne calculons pas les statistiques est le WarLight. En effet, cette unité s'est vu rajoutée au cours du projet et n'était donc pas présente lors de la création du tableau. Ses caractéristiques sont donc choisies de manière subjective en se basant sur les statistiques des autres unités. Nous souhaitons que cette unité soit un peu plus lente que les explorateurs. Elle doit également être de résistance moyenne, faire peu de dégâts mais avoir une cadence de tir élevée.

Une fois les unités modifiées, des tests avec des équipes adoptant des stratégies différentes sont effectués. Lors de nos test, nous nous sommes rendu compte que l'ajout d'une nouvelle unité rendaient l'équilibrage délicat car cela demande de reconsiderer le rôle de toutes les autres unités. Le fait de ne pas disposer de toutes les possibilités disponibles dans la version Java rend également l'équilibrage difficile car nous ne pouvons pas réellement nous servir d'elle comme référence.

Au départ, le WarHeavy semblait presque inutile. En effet, la particularité de cette unité dans la version Java était de pouvoir tirer au-delà de son champs de vision grâce à ses coéquipiers qui pouvaient lui transmettre des coordonnées. L'unité s'orientait alors dans cette direction et faisait feu sur une cible qu'elle ne distinguait pas. Cependant, cette fonctionnalité n'étant pas présente dans notre version, le WarHeavy possédait des caractéristiques trop faibles pour justifier son coût de création. Le plus gros ennui était sa distance de vue trop petite mais en augmentant simplement cette caractéristique, on risquait de créer une unité trop forte qui rendait les WarLight inutiles.

Nous avons donc décidé de réduire l'angle de perception et d'augmenter la portée de son champs de vision. Ainsi, les WarHeavy détectent mieux les ennemis mais sont plus vulnérables aux attaques qui arrivent de côté.

Les autres réglages ont plus été des règlements mineurs sur les autres unités pour obtenir des parties plus dynamique. On a par exemple légèrement augmenté les dégâts des unités offensives pour que les parties se terminent plus rapidement.

# **Troisième partie**

# **L'avenir du projet**

## Chapitre 5

# Amélioration possible

### 5.1 Interpréter

Bien sûr de nombreuses choses restent à mettre en place et à améliorer dans ce projet. Maintenant que les développeurs ont une base permettant d'ajouter facilement des modes de jeux, des cartes et des unités, les possibilités d'améliorations de MetaBot très grandes.

Au sein du travail de l'équipe interpréteur, de grosses modifications sur la condition de victoire “Course aux ressources” sont possibles, afin de le transformer comme un vrai mode de jeu, et pas simplement un changement rattaché au mode de jeu Warbot.

L'ajout de nouvelles caméras est bien sûr faisable si l'imagination des groupes suivant trouve une caméra intéressante.

L'interpréteur devra lui être mis à jour avec l'ajout de possibilités dans le langage, mais il suffira pour l'ajout de primitives telles qu'elles sont dans le projet actuel.

Il sera aussi nécessaire comme indiqué plus haut, de remettre en place une fenêtre pour afficher les matchups entre équipes, de manière à ne pas surcharger l'affichage. Il serait aussi bien d'étendre le fonctionnement des ELO point pour le faire fonctionner à 3 ou 4 joueurs, ce qui est pas forcément dans l'idée initiale du système ELO, mais qui sera plus cohérent par rapport au projet

Enfin, le maintien à jour des traductions sera nécessaire au fur et à mesure du développement, et bien sûr l'ajout de nouvelles langues facilement dès que cela est nécessaire.

Pour la gestion de la fin du jeu, un travail restera nécessaire pour rendre l'initialisation des éléments d'interfaces plus propre. De plus, actuellement la fin du jeu fonctionne sur la condition de victoire car nous n'avons pas deux modes différents, mais il faudra juste changer l'appel au test et à l'exécution de la fin de jeu dans l' “Update” du script.

## **5.2 Game Design**

Bien sûr de nombreux bugs vont subsister lors de la modification et l'ajout de fonctionnalités que nous ne voyons pas encore, mais nous pourrons répondre aux questions des prochaines équipes travaillant sur le projet si besoin.

# **Chapitre 6**

## **Conclusion**

# **Quatrième partie**

## **Annexe**

**6.1 Code important**

**6.2 Documents**