Rapport

Mathieu Taillandier
Rony Murat
Jordan Valnet
Maxime Binetruy
Sébastien Macher
Alexandre Huynh

14 janvier 2010

Table des matières

1. Introduction **4**

2. Conception **5**
2.1 Organisation de l'équipe . . . 5
2.2 Établissement du planning . . . 6
2.2.1 Planning prévisionnel . . . 6
2.2.2 Planning réel . . . 7
2.3 Organisation du travail . . . 8
2.3.1 Mise en place de quelques conventions . . . 8
2.3.2 Outil de travail collaboratif . . . 8
2.4 Architecture logicielle . . . 9
2.4.1 Diagramme de classe - première partie . . . 10
2.4.2 Diagramme de classe - seconde partie . . . 11

3. Réalisation **12**
3.5 Graphismes et principes d'animation . . . 12
3.5.1 Thématique retenue et manifeste . . . 12
3.5.2 Format 3D et contraintes posées sur la modélisation . . . 12
3.5.3 Structure de l'animation . . . 13
3.5.4 Texturage . . . 13
3.5.5 Estimations et complexité . . . 14
3.5.6 Problèmes liés au rendu graphique et solutions apportées . . . 15
3.6 Les loaders . . . 16
3.6.1 Loader OBJ . . . 16
3.6.2 Loader MD2 . . . 17
3.6.3 Problèmes rencontrés . . . 19
3.7 Physiques de l'arène . . . 20
3.8 Mouvements . . . 21
3.9 Collisions . . . 21
3.10 Les projectiles . . . 23
3.11 Intelligence artificielle . . . 24
3.11.1 Fonctionnement . . . 24
3.11.2 Graphe de déplacement . . . 26
3.11.3 Déplacement d'un noeud à un autre . . . 26
3.11.4 Amélioration de l'IA . . . 28
3.12 Gestion du son . . . 31

3.12.1 Mise en place du son 31
3.12.2 Choix des sons 31
3.12.3 Améliorations possibles 32
3.13 Principe du jeu 33
3.13.1 Guide utilisateur 33

4. Conclusion **34**

5. Bibliographie **35**

6. Annexes **36**
A.1 Quelques captures d'écran 36
A.2 Pseudo-code de l'IA 37

1. Introduction

Le jeu vidéo connait son premier âge d'or dans les années 80 grâce à des jeux comme Pac-Man ou Space Invaders et n'a depuis cessé de se développer à tous les niveaux, si bien qu'il est aujourd'hui en passe de devenir un média à part entière, au même titre que la télévision ou le cinéma.

Il était donc important pour les étudiants en multimédia que nous sommes de pouvoir envisager le fonctionnement et la réalisation d'un jeu vidéo, ce que le projet Smashstein a été l'occasion de faire puisqu'il proposait de réaliser un jeu vidéo en 2.5D (3D où les déplacements du joueur sont cantonnés à un plan) doté d'une intelligence artificielle.

Dans ce but nous avons procédé en deux phases principales, avec en premier lieu la **conception théorique**, comprenant la **répartition des tâches** au sein du groupe, l'**établissement d'un rétroplanning**, la réalisation de l'**architecture logicielle** et la **mise en place des conventions** de programmation. La seconde phase concerne la **réalisation** à proprement parler, à savoir la **modélisation des décors et personnages** et la **programmation en C++ de l'ensemble des fonctionnalités du jeu**, avec toutes les **difficultés techniques** qui l'accompagnent.

2. Conception

2.1 Organisation de l'équipe

Jusqu'à maintenant, la plupart des projets que nous avions eu à faire à l'Imac se déroulaient par binôme ou au maximum trinôme. Dans ces cas-là, la gestion du groupe s'en trouvait simplifiée car la répartition du travail était simple. Dans le cas d'une équipe de six personnes, il est nécessaire de bien connaître les aptitudes, les passions et motivations de chaque membre pour en tirer le maximum pour le bien de l'équipe (*plus on aime ce que l'on fait, plus on le fait bien*). De plus, nous avons eu la chance - légèrement provoquée - que chaque membre de notre équipe provient de filières très différentes et possède des connaissances et talents très complémentaires.

Un projet de programmation comme Smashstein demande en effet beaucoup de connaissances et de travail dans ce domaine, c'est pourquoi la grande majorité de l'équipe s'est attelée à cette tâche à un moment ou à un autre de l'avancement. Cependant, la programmation est à un projet comme Smashstein le ciment est à un bâtiment, il est nécessaire de créer de l'enrobage et de la matière autour pour lui donner une forme plus esthétique et agréable. C'est dans cet optique que nous avons concentré la seconde partie de l'équipe dans la réalisation des éléments graphiques tel que les modèles 3D ou les décors. Un jeu sans attrait graphique peut vite perdre de son intérêt. Le dernier membre de l'équipe s'est entièrement dédié à l'univers sonore car cette fonction est bien souvent mise en défaut alors que sans elle, un jeu peut bien vite sembler fade et triste.

Ainsi, nous avons su trouver dans cette équipe toutes les compétences nécessaires au bon déroulement de ce projet afin qu'il aboutisse de la meilleure des façons malgré le peu de temps qui nous était imparti.

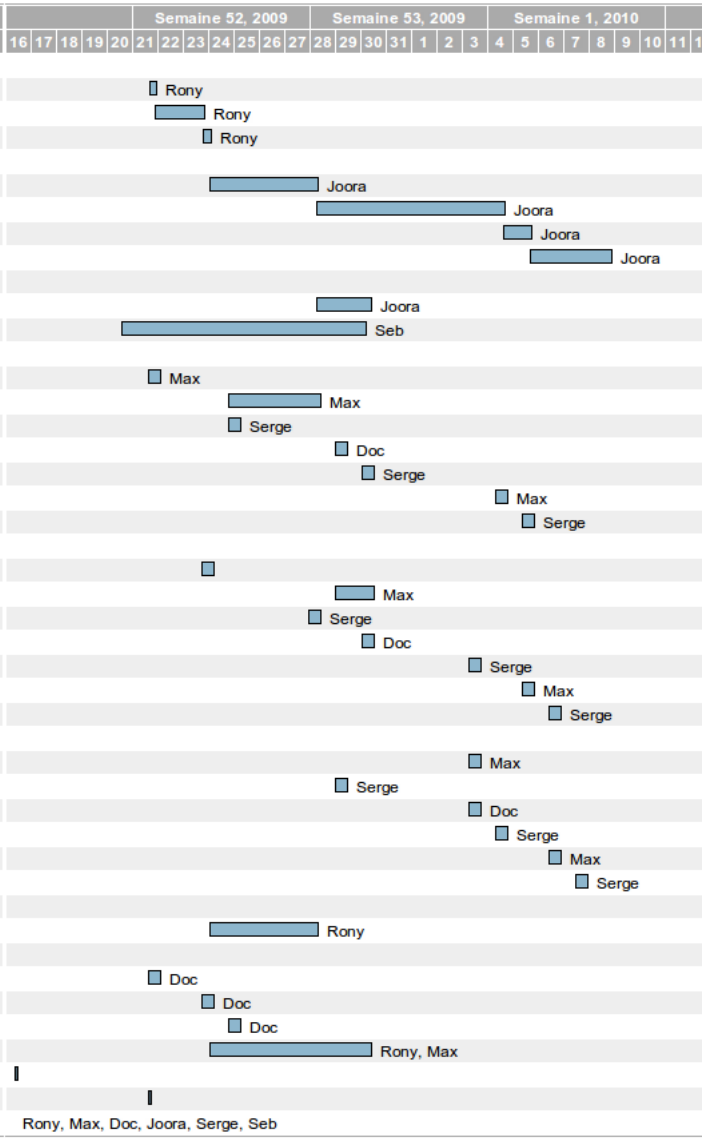
2.2 Établissement du planning

2.2.1 Planning prévisionnel

Avant de démarrer le projet, il nous a semblé nécessaire de nous réunir afin de partager nos idées. Les phases de réflexion ne sont évidemment pas à négliger, puisque c'est le tremplin pour partir d'une même base. Nous savions par la suite ce qui devrait être plus ou moins fait. Autrement dit, nous nous sommes répartis les tâches selon nos affinités avec tel ou tel domaine. C'est ainsi qu'un rétroplanning a été aussitôt mis en place. Ce planning prévisionnel permet de fixer quelques dates clés, d'avoir une vue d'ensemble des tâches à effectuer et d'avoir un repère dans la progression de ce projet.

Diagramme de Gantt

TPÉ	Nom	Travail
1	**Affichage de l'arène**	**1d 5h**
1.1	Loader OBJ	4h
1.2	parseur XML	4h
1.3	Intégration loader md2	5h
2	**Gestion de l'arène**	**8d**
2.1	Gestion du radar	2d
2.2	Gestion de l'inertie et de la gravité	2d
2.3	Gérer les collisions	1d
2.4	Effets spéciaux	3d
3	**Gestion du son**	**7d**
3.1	gestion du son (code)	2d
3.2	création sonore	5d
4	**Création des modèles 3D**	**7d**
4.1	Boxman	1d
4.2	Beach	1d
4.3	Castor	1d
4.4	Opera	1d
4.5	Terminal	1d
4.6	Bandicoot	1d
4.7	Pinguin	1d
5	**Création des animations 3D**	**7d**
5.1	Anim Boxman	1d
5.2	Anim Beach	1d
5.3	Anim Castor	1d
5.4	Anim Opera	1d
5.5	Anim Terminal	1d
5.6	Anim Bandicoot	1d
5.7	Anim Pinguin	1d
6	**Texturing**	**6d**
6.1	Texture Beach	1d
6.2	Texture Castor	1d
6.3	Texture Opera	1d
6.4	Texture Terminal	1d
6.5	Texture Bandicoot	1d
6.6	Texture Pinguin	1d
7	**Application Qt**	**4d**
7.1	Codage des caméras	1d
7.2	**Interface Qt**	**3d**
7.2.1	page titre	1d
7.2.2	choix personnages	1d
7.2.3	choix stage	1d
8	Création de l'IA des bots	6d
9	Réunion : réflexion	1h
10	Réunion : répartition des tâches	1h
11	Rédaction du rapport	

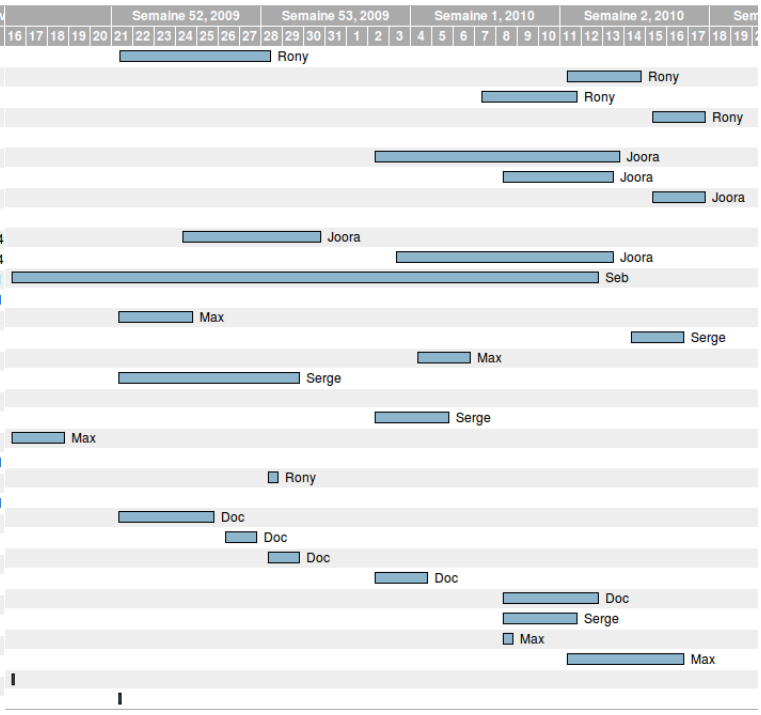


2.2.2 Planning réel

Après quelques semaines de travail, le planning réel a été dressé. Ce qui est intéressant, c'est de constater les changements par rapport au planning initial. Par exemple, nous avons mis en retrait la modélisation et animation de quelques personnages. Ou encore, certaines tâches ont pris plus de temps que prévu. Les fêtes de fin d'année y sont probablement pour quelques choses.

Diagramme de Gantt

TPÉ	Nom	Trav
1	Mise en place des loader / parseur XML	6d
2	Mise en place de l'IA / Graphes de déplacement	4d
3	Création du diagramme de classe et mise en place des design pattern	3d
4	Mise en place des projectiles – Particules – Effets spéciaux	3d
5	**Gestion de l'arène**	**8d**
5.1	Arène: physique / mouvement / animation	4d
5.2	Gestion de l'avatar : commande au clavier	1d
5.3	Gestion des collisions	3d
6	**Gestion du son**	**6d**
6.1	Installation/implémentation du son	3d 4
6.2	Gestion du son dans le jeu	2d 4
7	Recherche bruitages/musiques	12d
8	**Création des modèles 3D (Modelage/animation/texturing)**	**17d**
8.1	Boxman	3d
8.2	Castor	3d
8.3	Opera	3d
8.4	Bandicoot	8d
9	**Création des maps**	**7d**
9.1	Rome	4d
9.2	CerisierLand	3d
10	**Application Qt**	**12d**
10.1	Codage des caméras	1d
10.2	**Interface Qt**	**11d**
10.2.1	Fenêtre principale avec différentes vues	4d
10.2.2	Fenêtre choix personnages	2d
10.2.3	Fenêtre choix stage	2d
10.2.4	Diverses boîtes de dialogue	3d
11	Apprentissage LaTeX, mise en forme du rapport	3d
12	Création des graphiques (UML/Planning/...)	2d
13	Recherche et création du Logo	1d
14	Algorithme IA (pseudo Code)	6d
15	Réunion : réflexion	1h
16	Réunion : répartition des tâches	1h



2.3 Organisation du travail

2.3.1 Mise en place de quelques conventions

Avant de se lancer dans la programmation du jeu, nous nous sommes mis d'accord sur la façon de présenter le code. En effet, la mise en place de quelques règles d'écriture était nécessaire. Ceci permet d'avoir un code cohérent mais surtout plus compréhensible par tous les membres du groupe et donc plus facilement maintenable.

Voici ces quelques règles :

- Le nom des classes, méthodes, attributs, variables, fonctions... en anglais
- Le nom des classes commencent par une majuscule
- Les autres noms commencent par une minuscule et le mot composé qui suit par une majuscule
- Les accolades des blocs d'instructions sont alignées verticalement :

```
void method()
{
    ...
}
```

2.3.2 Outil de travail collaboratif

Deux outils de travail ont été mis en place dès le départ afin de faciliter la communication entre nous et la coordination des travaux réalisés :

- une liste de diffusion privée
- un système de gestion de versions

Tous les échanges de courriers électroniques se sont fait à travers cette adresse de diffusion : `list-smashtein@pandaco.net`. Étant donné que nous n'étions que six dans cette liste, nous nous sommes permis de l'utiliser sans retenue. Ainsi, quelque soit la discussion, tous les membres se tenaient au courant.

Concernant le choix du système de versionning, nous avons opté de mettre en place un dépôt centralisé de type Subversion. Nous l'avons déjà utilisé par le passé, nous étions donc plus ou moins aisés face à cet outil bien pratique. De ce fait, tous nos travaux étaient constamment synchronisés avec le dépôt. Afin d'éviter les conflits, nous nous mettons toujours d'accord sur quelles parties nous allons travailler.

2.4 Architecture logicielle

Dans l'optique de ce projet, nous avons décidé de réaliser un diagramme de classe qui répondrait aux principales fonctionnalités du programme de la façon la plus idéale possible. Pour ce faire, nous avons utilisé différents design patterns pour répondre à des situations bien précises. Nous n'allons bien sûr pas décrire la totalité de nos choix pour ce diagramme de classe dans ce rapport. Cela serait inutile et redondant. Cependant, nous expliciterons tout de même nos choix en ce qui concerne les principaux design pattern mis en place.

L'arène dans ce projet est l'élément central. C'est un objet unique et nous utilisons le design pattern « **Singleton** » pour le représenter. Elle utilise une **Factory** via la classe *Robots* pour générer les bots qui bénéficient d'une IA[1] et l'avatar contrôlé par le personnage.

L'arène utilise une *Collection* pour stocker les objets 3D chargés via les .OBJ[2]. Elle gère deux collections : les objets du décor non-interactif *DecorMesh* et les objets interactifs qui rentrent dans la gestion des collisions *IOMesh*. Bien entendu, il existe des itérateurs pour parcourir ces différents objets dans les collections.

L'arène et les bots qui possèdent une IA répondent au design pattern **Observateur**. En effet, chaque robot peut s'abonner à l'arène qui lui indique à partir de quel moment il peut effectuer une nouvelle action et lui communique les ennemis qui l'ont frappés...

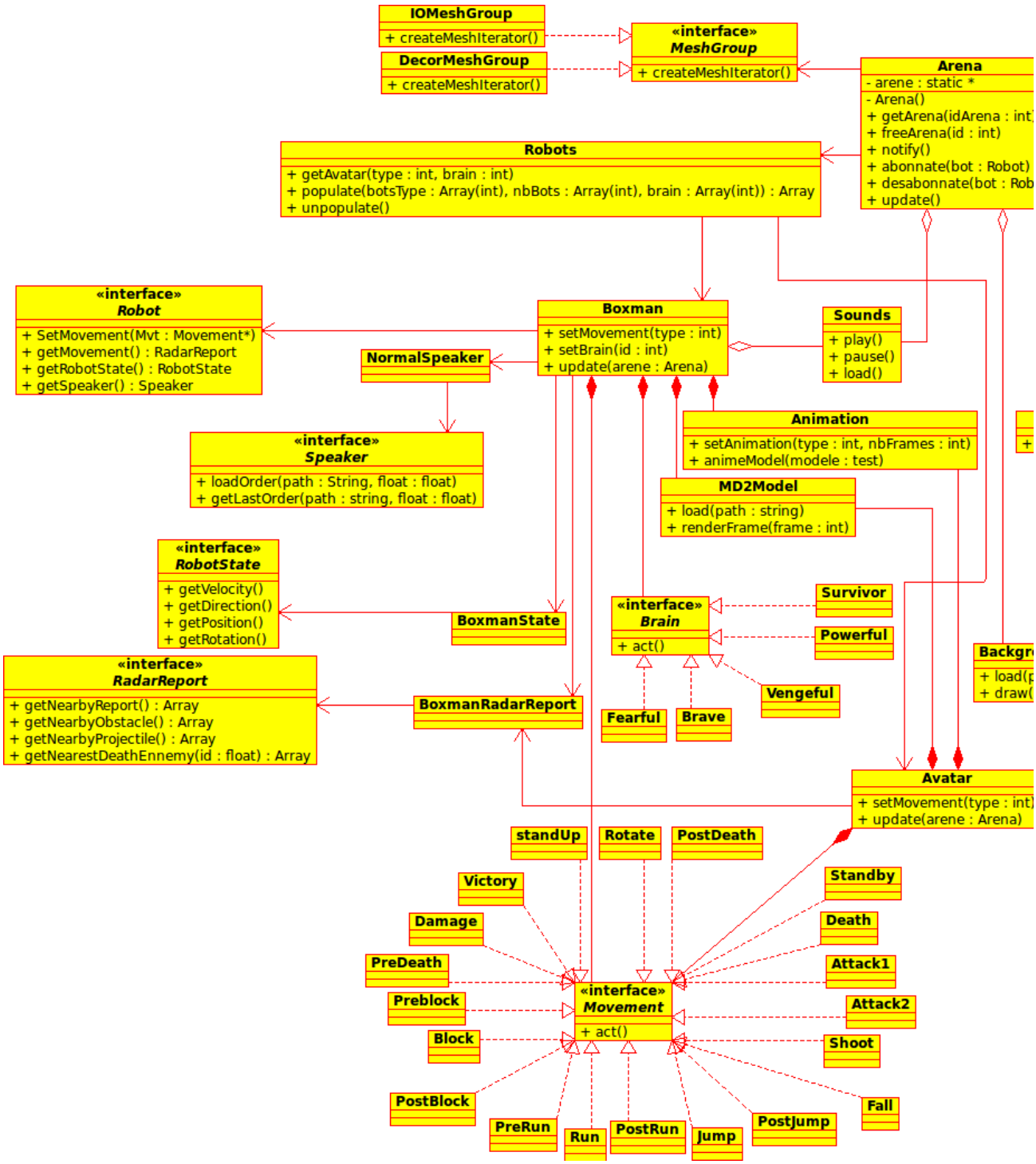
Une fois un bot créé, il faut être capable de lui attribuer certaines caractéristiques tel que son modèle md2, ses animations, ou encore son intelligence artificielle. Dans deux de ces cas, nous utilisons un autre design pattern pour les mettre en place. Le design pattern **Strategy** a été utilisé. L'interface *Movement* en effet, permet de gérer toutes les animations du modèle 3d (run, jump, attack...) et l'interface *Brain* gère, quant à elle, les « cerveaux » du bot pour pouvoir modifier dynamiquement les comportements qui vont influencer les actions du robot dirigé par l'ordinateur.

Dans de nombreux autres cas, nous avons mis en place des interfaces qui vont faciliter la gestion des classes et des objets. Tels que tous les objets interactifs sont des obstacles. L'interface projectile pour gérer les projectiles de l'arène.

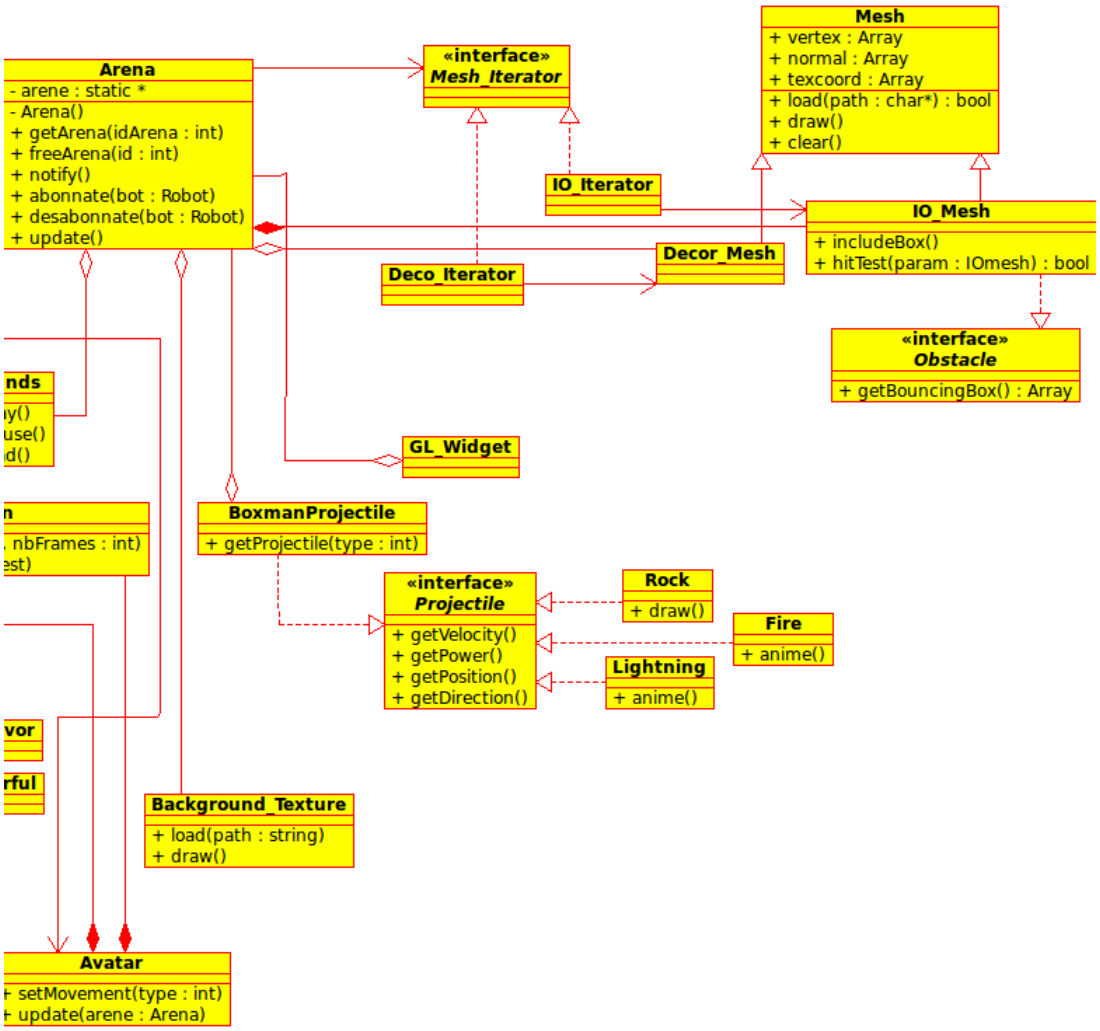
Chaque *Robot* implémente l'interface robot et possède des éléments tels que leur *speakers* qui implémentent, eux aussi, d'autres interfaces. Nous nous sommes efforcés de mettre tout ce qui nous paraissait commun en interface afin de faciliter la gestion et la compréhension du programme.

[1] Intelligence Artificielle
[2] Voir Section 3.6.1

2.4.1 Diagramme de classe - première partie



2.4.2 Diagramme de classe - seconde partie



3. Réalisation

3.5 Graphismes et principes d'animation

3.5.1 Thématique retenue et manifeste

Étant conscient que le temps réservé au projet ne nous permet d'effectuer des recherches graphiques approfondies, nous avons renoncé à produire une réalisation graphique soignée et cohérente. Il serait plus juste de dire que notre recherche ne sera pas "pensée" en terme de composition, d'harmonie chromatique et de ce qui fait traditionnelement figure de règles d'or des productions graphiques ordinaires.

Notre thématique retenue sera donc : GARBAGE.

1. Par GARBAGE, nous entendons le chaos le plus total, l'énergie brute de la témérité et de l'impulsivité.
2. Les éléments essentiels de notre capharnaum seront le saugrenu, les collisions de choses paradoxales, l'aléatoire et l'absurde.
3. Les rencontres visuelles improbables, l'hétéroclite, les associations créées seront notre poésie surréaliste.
4. Nous ne prétendons à rien, nous ne sommes que rebuts et brouillons.
5. Nous rejetons la logique du rationnel et privilégirons les méandres de l'esprit onirique et les liens qu'ils créent.
6. Nous rejetons les cinq premiers points et nous ferons ce que nous pourrons libres de toute contrainte...

3.5.2 Format 3D et contraintes posées sur la modélisation

Après un débat entre MD2 et MD5, nous avons opté pour le premier format plus facile à mettre en oeuvre. En utilisant ce format-là, nous avons besoin de beaucoup plus de mémoire pour pouvoir stocker la liste de points du modèle pour chaque frame d'animation. Nous nous exposons aussi aux problèmes de normales et aux sacades inévitables des animations. Pour apporter une solution à ces problèmes, nous avons choisi de réaliser des modèles 3D en low-polygones. Nous nous sommes donc imposés **un nombre maximum de 1000 polygones pour les personnages** et de **8000 polygones pour les arènes**.

Nous avons également souhaité développer dans un premier temps plusieurs personnages (de l'ordre de 5), mais nous avons réalisé que nous ne pourrions pas en modéliser et

surtout animer autant. Plutôt que d'en faire beaucoup, nous avons donc décidé de nous **concentrer sur trois** d'entre eux et de **développer davantage leurs animations**.

3.5.3 Structure de l'animation

Nous avons tout d'abord défini quels mouvements pourront être effectués par les personnages dans le jeu. Puis nous avons défini des priorités aux animations et dressé la liste finale que voici :

```
#standby 1      // le personnage est en attente.
#prerun 2       // le personnage prend son impulsion et commence à courir.
#run 3          // -- boucle-- le personnage cours.
#jump 4         // le personnage prend une impulsion pour sauter, saute et atteint le sommet de la courbe.
#fall 5         // -- boucle-- le personnage chute après un saut.
#postjump 6     // le personnage ce réceptionne.
#attack1 7      // le personnage attaque au corps à corps selon le mode 1.
#throw 8        // le personnage attaque à distance.
#damage 9       // le personnage encaisse un coup.
#preblock 10    // le personnage se prépare à parer un coup (ou bien activer un bouclier de défense).
#block 11       // --boucle-- le personnage de protège de toute attaque.
#postblock 12   // le personnage cesse de se protéger.
#predeath 13    // le personnage encaisse un coup puissant qui l'envoi voler au loin.
#death 14       // --boucle-- le personnage en vol plané suite à un coup puissant.
#postdeath 15   // le personnage se réceptionne (tombe à terre) suite à la chute provoquée par un coup puissant.
#standup 16     // le personnage se relève pour poursuivre le combat.
#attack2 17     // le personnage attaque au corps à corps selon le mode 2.
#rotate 18      // le personnage pivote sur lui même pour changer de direction.
#postrun 19     // le personnage s'arrête brutalement après avoir courru.
#victory 20     // le personnage se manifeste au joueur de manière exubérante en cas de
                   victoire ou bien d'inactivité prolongée.
```

Le terme –boucle– indique lorqu'une animation a été prévue pour être jouée en boucle. Celle-ci s'accompagne souvent d'une "**pré-animation**" et d'une "**post-animation**". Nous avons segmenté de cette façon pour pouvoir ajuster plus finement les transitions entre animations et réduire le saut visuel provoqué par l'enchaînement d'images clés (keyframe) n'ayant rien en commun. Nous espérons ainsi limiter leur impact visuel (tenter de passer sous le seuil de perception) et augmenter la fluidité générale des mouvements des personnages.

3.5.4 Texturage

Afin d'améliorer le rendu graphique des modèles en *low-polygone*, nous nous somme penchés sur la réalisation de textures très détaillées. Cette solution ayant montrée ses limites (arêtes saillantes impossibles à faire disparaître...), nous avons décidé de mettre en oeuvre le "**normal mapping**".

Une "**normal map**" est une textures en RVB qui définie une géométrie illusoire. Celle-ci est généralement créée à partir d'un modèle en *high-polygone* qui va donner ensuite ses propriétés de réflexion de lumière au modèle *low*. Les principaux écueils de cette méthode sont :

– la nécessité de réaliser un modèle *high* (qui ne sera pas utilisé, il servira juste à la *normal map*), ce qui prend beaucoup de temps.

– la difficulté d'ajuster le modèle *low* et *high-poly* pour que la *normal map* corresponde bien (risque de générer des erreurs de texturage).

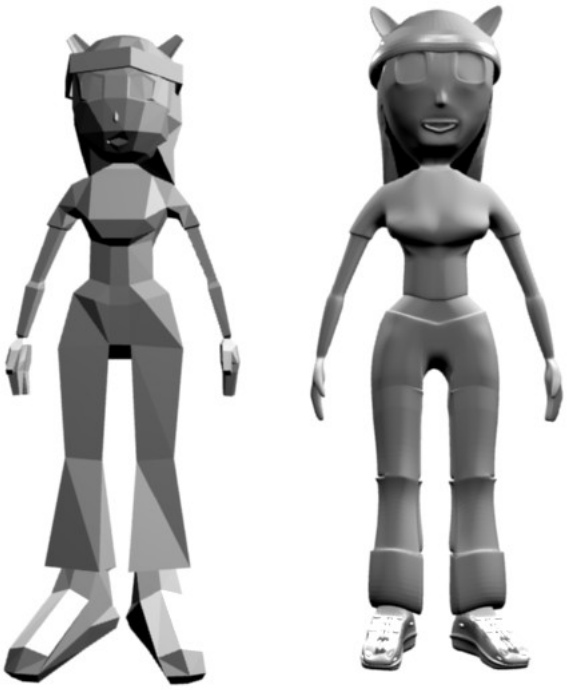


FIG. 3.1 – Modèle en low-poly et high-poly

3.5.5 Estimations et complexité

L'essentiel des opérations coûteuses en temps de calcul se font au moment du chargement.

Voici quelques-unes de nos estimations au départ du projet qui d'ailleurs ont été revues à la baisse au final :

– 1 point peut avoir au maximum 10 normales
– 1 triangle est composé de maximum 30 vertex
– 1 personnage est composé d'environ 500 triangles
– 1 personnage est composé d'environ 20 animations
– 1 animation est composé de 5 frames
– 1 personnage est composé de 15000 vertex
– 1 arène est composée de maximum 4 personnages
– 1 arène est composée de 140.000 triangles

- 5.500.000 de vertex stockés dans la carte graphique
- Environ 45 milions de float dans la carte graphique

3.5.6 Problèmes liés au rendu graphique et solutions apportées

Implémentation des normales

Le premier problème est un problème de conception concernant l'implémentation des normales : chaque vertex de nos modèles peut avoir plusieurs normales. Lorqu'il se trouve sur une arête par exemple, il appartient à deux polygones différents et possède donc deux normales mais notre implémentation des vertex n'en autorise qu'une seule.

Plusieurs solutions ont été proposées : stocker un vertex par normale quitte à répéter plusieurs fois le même point : position et coordonnées de texture. Cette solution présente l'avantage d'être compatible avec la structure des Vertex Buffer Object (VBO) mais implique également un nombre de vertex stockés multiplié par quatre (estimation en moyenne), et une difficulté d'implémetnation accrue au moment de leur construction dans le loader.

3.6 Les loaders

3.6.1 Loader OBJ

Dès le départ, nous voulions avoir un rendu visuel convenable. Il nous a paru alors nécessaire de créer des arènes à partir de modèles objets en 3D modélisés et texturés via un logiciel adapté (Blender, 3DSMax). Notre choix s'est porté sur le format OBJ qui est un format simple et facile à implémenter permettant de charger facilement des objets en 3D avec des textures. En effet, chaque OBJ peut etre associé avec un MTL. Ce dernier fournit des informations sur les textures utilisées et les couleurs ambiantes et diffuses de chaque modèle. Les textures sont chargées à partir d'images PPM.

La version actuelle de notre loader ne gère pas de multi-texturing sur un seul OBJ mais nous espérons avoir le temps d'implémenter la version gérant ce multi-texturing pour le rendu final.

Les OBJ sont chargés dans les classes Mesh qui possèdent 2 sous-classes. En effet, afin de gérer les collisions, les Mesh sont séparés en 2 types. Les éléments de décor stockés dans les classes DecorMesh et les éléments interactifs qui sont ceux avec lesquels le joueur et les bots, pourront rentrer en collision stockés dans les classes IOMesh. Chaque IOMesh possède donc des Enveloppes pour détecter les collisions.

Le chargement des Mesh se fait via un fichier XML qui donne les OBJ à charger. Le formatage du fichier est le suivant : Toutes les informations sur un niveau sont contenues dans une balise `<map>`. Dans cette balise se trouvent quatre types de balises :

- la balise `<graph>` correpond au fichier décrivant le graphe de déplacement à utiliser pour que l'IA[3] puisse se déplacer.

  path : chemin vers le fichier.

- la balise `<background>` correspond à la texture de fond du niveau. Elle a comme attributs :

  path : chemin vers la texture de fond du niveau.

  width, height : respectivement les largeurs et hauteurs de cette texture.

  x, y, z : respectivement les positions en x, y et z du centre de la texture.

- les balises `<element>` renseignent les OBJ des éléments à charger dans les classe DecorMesh. Elle possède comme attributs :

  path : chemin vers le fichier en .obj.

  x, y, z : positions en x, y et z du modèle.

[3]Intelligence Artificielle

xrot, yrot, zrot : angles de rotation en degrés de l'élément respectivement sur les axes x, y et z.

- les balises `<mesh>` renseignent les OBJ des éléments à charger dans les classes IO-Mesh. Elle possède les mêmes attributs que la classe élément mais donne en plus :

  xBox, yBox, widthBox, heightBox : respectivement la position en x, y et la largeur et la hauteur de la boîte qui englobe cet objet utile pour la détection des collisions.

Exemple :

```
<map>
   <graph path="obj/graph.txt" />
   <background path="obj/textures/clouds.ppm" width="100" height="50" x="-50" y="-15" z="-10.0"> </background>
   <mesh path="obj/map_Rome_allege.obj" x="0" y="-0.2" z="0" xrot="0" yrot="0" zrot="0" >MAP_ROME_ALLEGEE</mesh>
</map>
```

3.6.2 Loader MD2

Toutes les informations sur un bot sont données dans la balise `<bot>` Dans cette balise se trouve plusieurs types de balises :

`<name>` : nom du modèle de bot.
`<brain>` : nom du plugin d'IA à utiliser.
`<body>` : informations sur le bot et sur le modèle MD2 à choisir. Cette balise possède comme attributs :
- 'velocity' : vitesse du bot.
- 'resistance' : résistance de ce dernier.

Elle contient également d'autres balises spécifiques au modèle MD2 :

`<model>` : nom du binaire MD2 à utiliser. Ses attributs :
- 'x', 'y', et 'z' : respectivement les positions x, y et z du modèle par rapport à l'origine du dessin.
- 'xrot', 'yrot' et 'zrot' : respectivement les angles de rotation en degré par rapport à l'origine du dessin.
- 'scale' : changement d'échelle à effectuer sur les 3 axes pour dessiner le modèle convenablement.

`<texture>` : chemin vers la texture à utiliser pour afficher le modèle.
`<anime>` : chemin vers le fichier d'animation qui donne les différentes animations du modèle.

Exemple :

```
<bot>
  <name>boxman</name>
  <brain>libAINormal.so</brain>
  <body velocity="1.0" resistance="1.0">
    <model x="0.0" y="-0.2" z="0.0" xrot="0" yrot="-90" zrot="90" scale="0.013">p_boxman.md2</model>
    <texture>p_boxman.ppm</texture>
    <anime>p_boxman_anim.txt</anime>
  </body>
</bot>
```

Les lignes de commentaires commencent par '#'. Dans le fichier[4] sont donnés dans l'ordre les animations :

`<type_animation> <frame_début> <frame_fin> <fps>`

Exemple :

```
#MD2 Anim p_bandicoot
#Total : 550 frames
#Decoupe moyenne : 5 frames per key
#Estimation pour md2 : 70 * 2

#standby 1
#prerun 2
#run 3
#jump 4
#fall 5
#postjump 6
#attack1 7
#throw 8
#damage 9
#preblock 10
#block 11
#postblock 12
#predeath 13
#death 14
#postdeath 15
#standup 16
#attack2 17
#rotate 18
#postrun 19
#victory 20 // Indication : faire une pause à : 525 ; 530 ; 535 ; boucler de 540 à 550

1 1 40 35
2 40 50 40
3 50 70 40
4 80 105 40
5 105 120 40
6 120 160 50
7 160 200 40
8 200 230 40
9 230 260 40
10 260 270 40
11 270 280 40
12 280 290 40
13 290 310 40
14 310 325 40
15 325 365 40
16 365 435 40
```

[4]Voir exemple dans l'annexe

```
17 435 465 40
18 465 480 40
19 485 520 50
20 520 550 45
```

3.6.3 Problèmes rencontrés

Le problème que nous avons rencontrés et que nous n'avons pas compris tout de suite, vient du type de fichier utilisé pour les textures. En effet le format PPM donne une image miroir de l'image réelle. Or appliquer la texture sur le modèle MD2 requiert une image miroir alors que les OBJ utilisent l'image réelle. Étant donné que les textures des deux modèles étaient chargées en mode miroir, les textures ne s'appliquaient pas correctement. Il a donc fallu changer le mode de chargement selon le modèle sur lequel il fallait l'appliquer.

Par rapport au chargement des textures, il nous était impossible de charger une et une seule fois les textures pour toutes les vues (modes porsuite, global, debug "sphérique" et libre). Chaque vue étant contenue dans un widget, il a fallu donc charger quatre fois ces mêmes textures. Ce mode de chargement nous a semblé bien plus qu'aberrant. De plus, la consommation mémoire s'en voit alourdie, ce qui bien embêtant.

3.7 Physiques de l'arène

L'arène possède certaines lois physiques auxquelles on soumet chaque robot ou avatar. Le modèle physique qui a été mis en place est le suivant :

Un robot possède (via la classe **BoxmanState**) :

- une direction (vecteur3f)
- une accélération (vecteur3f)
- une vitesse (vecteur3f)
- une position (vecteur3f)
- une vitesse maximale en x, y et z (stockée dans un vecteur3f)

L'arène, quant à elle, possède :

- une force de gravité (vecteur3f négatif en y)
- une vitesse maximale pour la chute (double)

La physique est appliquée à chaque frame, avec une intensité proportionnelle au temps écoulé depuis la dernière frame afin d'être visuellement constante malgré des variations de fps.

Elle se comporte globalement d'une façon réaliste un peu modifiée :

```
accélération += direction
vitesse      += accélération + gravité
vitesse      /= force de frottement
vitesse      =  max(vitesse, vitesse max)
position     += vitesse
```

L'accélération et la direction sont remises à zéro.

La direction correspond à une impulsion dans un certain sens, provoquée par un ordre de l'IA/avatar. L'avantage de fonctionner ainsi permet d'avoir un démarrage progressif. Cependant trois vecteurs auraient suffi pour le même modèle.

Cas particuliers :
Sur le sol, on ne tombe plus. La force de gravité annulée. La force de frottement est élevée sur le sol, et quasi nulle en l'air. Cependant des mouvements ajoutent leurs propres lois et comportements en plus du modèle général. Par exemple un saut possède des lois physiques internes non réalistes pour permettre de diriger (et le plus aisément possible) le personnage en vol.

3.8 Mouvements

Les mouvements des robots et avatars sont nombreux et plus ou moins complexes. Pour simplifier la gestion des mouvements nous avons utilisé le design pattern *État* : chaque mouvement est défini dans une classe à part qui hérite de l'interface Movement, qui implémente une méthode `act()`.

Un robot possède un seul mouvement (en pointeur) qui peut changer via la méthode `setMovement(Movement*)`.

A chaque frame, la méthode `act()` du mouvement du robot est appelée. Le changement de mouvement se fait par le mouvement lui-même, dans la méthode `act()`.

Illustrons par un exemple :

Le robot a tout d'abord un mouvement "*StandBy*", c'est ce mouvement-là qui est effectué. Quand le robot donne l'ordre d'avancer dans une direction, le mouvement *StandBy* l'interprète et change le pointeur mouvement du robot en un nouveau mouvement "Run". La frame suivante, ce sera le mouvement *Run* qui sera joué.

Remarque sur le design pattern choisi :
Il est très approprié à la situation et a bien simplifié la gestion complexe des mouvements. Cependant, il n'évite pas certaines redondances de code lorsqu'on doit gérer les mêmes cas depuis plusieurs mouvements différents.

Rôle - chaque mouvement peut ou doit :

- gérer l'animation 3D du robot,
- interpréter les ordres reçus par les robots,
- gérer les déplacements des robots
- changer le prochain mouvement

3.9 Collisions

Si rien ne se touche, il n'y a pas de jeu... Les personnages ne doivent pas traverser le décor et pouvoir se toucher entre eux. C'est le but du jeu !

Nous avons donc ajouté des **boites englobantes**[5] autour des objets principaux : le sol, les plateformes, certains obstacles et bien sûr les robots/joueurs.

Comme tout se joue sur un seul plan, il n'y a que quatre côtés à tester par objet.
Un test est effectué pour chaque robot sur tous les objets du décor, indiquant le cas échéant quelle face est touchée. Par exemple, si c'est la face du dessous, alors le robot est "sur le sol" et ne tombe plus ! De même, les robots peuvent se "traverser" mais des tests

[5] Voir figure ci-dessous

sont effectués lors des coups : ai-je heurté quelqu'un ? De quel côté ? Ensuite sont gérées les conséquences (vol plané, dommages...).
Il en sera de même pour les projectiles lorsque nous les implémenterons.

FIG. 3.2 – Les boîtes englobantes visibles en blanc sur cette image permettent de gérer les collisions.

3.10 Les projectiles

A l'heure où nous écrivons ces lignes, nous n'avons pas encore commencé la gestion des projectiles. Mais nous pouvons donner une description des fonctionnalités à implémenter.

Les projectiles seront modélisés via des double billboards et des particules dans un premier temps et par des OBJ également, si nous en avons le temps. C'est la classe Arene qui gère les projectiles qui sont instanciés via la *factory BoxmanProjectile*. Le lancement d'un projectile s'effectue par la création d'un billoard et/ou d'un système de particules. Une *factory* sera utilisée pour générer des modèles de particules et billboards qui seront prédéfinis dans la classe afin d'éviter d'avoir à fournir les paramètres des systèmes de particules par exemple.

Un projectile est créé à chaque fois qu'un bot ou que l'avatar souhaite en utiliser et chaque type de bot possède un type de projectile qui lui est associé. Le paramétrage des particules et des billboards de chaque bot sera fait en dur dans la *factory* et pourra, si nous avons le temps, être défini via le fichier XML définissant un bot.

Selon le type de projectile la physique pourra ou pas être appliquée. Des projectiles très rapides ne subiront pas la physique alors que d'autres plus lent y seront soumis.

Les système de particules seront également utilisés pour donner de la crédibilité au jeu. Lors de l'arrêt d'un personnage, on peut jouer de la fumée pour montrer la glissade ou faire une petite explosion quand un projectile heurte un bot ou lorsqu'un bot donne un coup.

3.11 Intelligence artificielle

3.11.1 Fonctionnement

NB : La mémoire se base sur la récupération d'un argument de "main" pour le profil du joueur et la construction d'arbres de probabilités personalisés et stockés dans des fichiers textes.

L'IA est appelée lorsqu'elle reçoit le radar. Nous avons réfléchi aux actions qu'elle devait obligatoirement exécuter lors de son processus de prise de décision. De fait, nous devions avoir fixer un gameplay pour pouvoir imaginer le comportement de l'intelligence artificielle. Voici une liste des actions qui ont finalement été retenues :

- Direction droite
- Direction gauche
- Sauter
- Coup1
- Coup2
- Projectile
- Bloquer

À partir de ces éléments, l'IA peut : se déplacer pour s'approcher du joueur ou au contraire le fuir, atteindre des plateformes en sautant, choisir d'attaquer au corps à corps ou bien à distance. Elle peut également décider de se défendre et parer certains coups.

Mais comment l'IA allait-elle décider de l'action à effectuer ?

Après réflexion, nous nous sommes rendu compte que personne ne réagirait de manière identique car nous avons chacun des caractères différents. D'un côté, un agira prudemment, testant les défenses de l'adversaire, le jaugeant avant d'attaquer tandis qu'un autre foncera, ne laissant aucune seconde de répit à son adversaire... Afin de rendre tous ces comportements possibles, nous avons décidé d'employer le design pattern "*stratégie*" qui nous permet de redéfinir de manière dynamique le comportement de notre IA. Aisni notre classe "**Brain**" est liée à une interface qui s'occupe de redéfinir sa methode `act()`. Cette implémentation a également l'avantage d'être facilement mise à jour (on peut sans grande difficulté ajouter un nouveau comportement / caractère). L'idée d'avoir un "*Brain*" par comportement a été écarté, car elle impliquait une incohérence au niveau de la conception (un bot ne change pas de cerveau pendant la partie ! Il change juste sa manière d'être).

Implémenter les comportements a donc été la première étape, mais notre IA n'était pas encore capable de prendre une décision. Que lui manquait-il ?

Afin d'effectuer un choix, quelqu'un doit disposer des données du problème. Pour notre IA, ces données lui sont fournies par le radar, indirectement par l'arène donc. Il nous restait donc à déterminer quelles informations allait communiquer le radar et de quelle façon notre IA allait les analyser.

En se positionnant du côté du joueur, quelles sont les informations dont nous disposons ?

$\Longrightarrow$ La position de tous les robots adverses, la position de notre avatar, l'état de santé de tous les personnages, les quelques derniers mouvements effectués par les adversaires (soit une tendance générale). À l'exception du dernier élément, tout est assez simple à récupérer pour notre IA (toujours via l'arène).

Et la prise de décision ?

À partir de tout ça, notre IA peut se débrouiller. Suivant les informationss reçues, elle se fixe une cible, c'est-à-dire un autre robot sur la scène. Ses critères de sélection sont affectés selon son caractère. Elle décide ensuite de l'action à accomplir[6].

NB : les notions liées à la mémoire et les parties commentées ci-dessous sont relatives aux améliorations de l'IA décrites plus loin dans le rapport.

[6] voir le pseudo-code en annexe

3.11.2 Graphe de déplacement

Lors du chargement de l'arène, un graphe de déplacement est créé. Nous n'avons pas encore mis en place ce système mais il sera fait pour le rendu final. L'intelligence artificielle se déplace dans la map grâce à ce graphe. Les points importants de l'arène, les positions des plates-formes ainsi que les pièges (trous, bords de l'arène) sont référencés dans celui-ci. L'intelligence artificielle peut donc savoir quelles sont les zones où elle doit se rendre en fonction de l'ennemi qu'elle a choisi de fuir ou d'attaquer. L'IA fait des interpolations sur les positions sur les positions à atteindre pour savoir s'il faut sauter ou avancer afin de rejoindre un point.

Le graphe est maintenu à jour de telle sorte que l'IA peut savoir où trouver chaque bot en parcourant le graphe. Nous pensons utiliser boost-graph[7] afin d'implémenter les graphes.

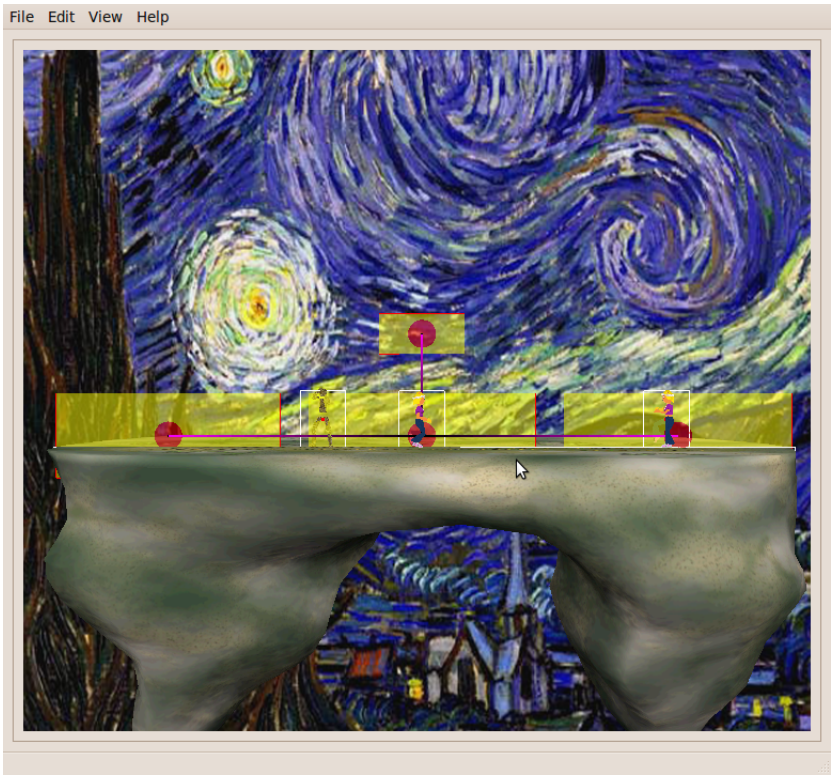


FIG. 3.3 – Graphe de déplacement : les disques rouges correspondent aux noeuds

3.11.3 Déplacement d'un noeud à un autre

Avant de procéder à quoi que ce soit, l'IA se posera une série de question afin de savoir quelle action effectuer. Voici ci-dessous la liste des questions :

[7]Bibliothèque permettant le parcours et la manipulation de graphe

1. **Dans quel noeud je suis ?**
2. **Dans quel noeud se trouve mon objectif ?**
   → On cherche à la fois où le bot se trouve dans l'arène (noeud A) et où l'objectif du bot se trouve dans l'arène (noeud B) : parcours du graphe.

   Algorithme par position des noeuds

```
debut parcours
  * initialise un noeud temporaire bot //(position infinie)
  * initialise un deuxième noeud temporaire target //(position infinie)
  * pour chaque noeud
    * calcul de la distance (positionObjectif, positionNoeud) ;
    * la distance est-elle plus petite que celle du noeud temporaire bot ?
      - si oui : stockage du noeud dans noeud temporaire bot et continue
      - si non : continue
    * calcul de la distance (positionCible, positionNoeud) ;
    * la distance est-elle plus petite que celle du noeud temporaire target ?
      - si oui :
        * le noeud est-il une zone dangereuse (un trou par exemple) ?
        // Ici le caracère de l'IA peut modifier les probabilités
        // afin de définir qu'est ce qui est dangereux ou qui ne l'est pas
          si oui : continue
          si non : stockage du noeud dans noeud temporaire target
      - si non : continue
  fin pour
fin parcours
```

3. **Comment me rendre de A à B ?**
   → Recherche du plus court chemin dans le graphe avec l'algorithme de Dijkstra en renvoyant une liste de noeuds

   a) Hypothèse 1
   Dans ce cas, l'action "goTo" reste en tête de file tant qu'elle n'est pas accomplie.
   Problème : risque de croiser le joueur vers lequel il se dirigeait et de l'ignorer jusqu'à avoir atteint l'ancienne position.
   Solution : des intervalles entre les zones clés pas trop éloignés.

   b) Hypothèse 2 (redéfinition de B)
   L'action "goTo" est mise à jour à chaque rafraîchissement de radar mais pas défilé (seules les coordonnées d'arrivée changent). Elle est défilée seulement lorsque les coordonnées d'arrivée sont atteintes.
   Problème : lorsque l'objectif est d'éviter la cible / joueur, il faut définir une cible (zone-clé) qui soit la plus éloignée possible de lui. Cela revient à faire un parcours en profondeur en plus pour touver le noeud le plus éloigné.
   Solution : l'implémentation d'une action "flee" différenciée de "goTo" qui parcours le graphe non pas pour trouver le chemin le plus court entre A et B mais pour trouver le chemin le plus long en partant de B et passant par A (sans jamais repasser par le même noeud). Puis de retourner seulement le chemin à partir de A.

   c) Hypothèse 3 (B non ponctuel)
   L'action "goTo" n'est pas implémentée telle quelle mais plutôt par des rayons d'ac-

tion centrés sur la cible qui agiraient comme suit : fuir (rayon maximal), se rapprocher (rayon minimal, ou en tout cas à portée d'attaque), se rapprocher pour tirer (rayon égal à la portée d'un projectile), se maintenir à une certaines distance de la cible (un certain rayon). L'action "goTo" disparaît alors de la file comme les autres, quand l'objectif est atteint.
Problème : que faire si le bot ne peut pas atteindre le cercle (si le joueur tombe et que le bot doit l'attaquer par exemple) ?
Solution 1 : le demi cercle subit constamment des variations de rayon aléatoires obligeant le bot à se remettre en mouvement.
Solution 2 (ou complémentaire) : lorsque le bot est resté inactif (obligation d'implémenter une action "attente") un certain nombre de frame dépendant de son caractère, il change de stratégie (attaque à distance par exemple).

3.11.4 Amélioration de l'IA

Le fait de n'effectuer qu'une seule action puis une autre puis une autre sans lien entre ces dernières ne semble pas très "réaliste". Lorsqu'on joue à un jeu de baston, les meilleurs sont ceux qui établissent des stratégies (même si celles-ci se résument à appuyer sur tous les boutons frénétiquement dans l'espoir de toucher son adversaire). Un stratégie signifie des actions qui ont un lien entre elles. Ce choix est justifié par le nombre des actions à disposition : un bot pourra n'attaquer qu'à distance pendant un certain temps pour affaiblir son adversaire tout en se tenant éloigné de ses coups puis passer à l'attaque au corps à corps, mais un autre pourra au contraire décider de privilégier une série d'attaques au corps à corps. Dans les deux cas, on se rend compte qu'un simple random sur les actions du bot ne suffit pas à simuler une intelligence. La solution proposée est simple : **plutôt que de choisir au hasard avec des probabilités homogènes la prochaine action à effectuer, les probabilités sont affectées**. Prenons un exemple :

Soit Bob un bot au caratère plutôt défensif. Bob se fait attaquer par un joueur plutôt belliqueux. Bob se protège d'un coup. Quelle est sa prochaine action ? Il peut :

- Fuir à droite (20%)
- Fuir à gauche (20%)
- Attaquer (20%)
- Lancer un projectile (20%)
- Continuer à se protéger (20%)

Pour le moment, toutes les actions ont 20% de chance de se réaliser mais à l'évidence certaine de ces propositions subiront un malus à leur probabilité. Bob est défensif, la probabilité qu'il attaque ou qu'il lance un projectile est faible et est donc divisée par deux (passant à 10%) tandis que la probabilité qu'il se protège ou tente de fuir augmente. Bob s'est protégé à l'action précédente, si la situation n'a pas changé et que le joueur belliqueux est toujours à portée, le choix pour l'action précédente est toujours valide, la probabilité de "se protéger" reçoit donc un nouveau bonus. Le tirage aléaoire se fera donc avec les probabilités suivantes :

- Fuir à droite (25%)

- Fuir à gauche (25%)
- Attaquer (10%)
- Lancer un projectile (10%)
- Continuer à se protéger (30%)

On se rend compte qu'en réalité deux passes ont été faites : la première modifie les probabilités en fonction du caractère, la seconde en fonction de la situation (a-t-elle évoluée de manière radicale ?).

Bien sûr, cette methode diffère peu de la première mais le bot semble posséder tout de même une once d'intelligence. Si l'on veut poursuivre dans cette voie, il est facile de multiplier les facteurs intervenant sur les probabilités pour affiner le choix final et le rendre d'une certaine façon plus cohérent. Il serait bon pour éviter à une action de se répéter excessivement de fixer un pallier de probabilité à 50%.

Revenons à la barette d'action de la classe "Brain". Pourquoi avoir une barette d'actions ?
Ce tableau organisé comme une pile dans un premier temps, puis comme une file dans un second, contient un nombre prédéfini d'actions qui aiderons / orienterons le choix de l'IA de manière plus précise.

```
// on a rempli la liste au démarrage de manière aléatoire
// (parmi les actions liées au caractère du bot)
* à chaque fois que l'IA reçoit le radar
  * initialise une liste de taille x
  * détruit le plus ancien maillon
  * pour tous les éléments de la barette d'action
    * modification des probabilités en conséquence
  fin pour
  // Prise effective de la décision
  * ajoute l'action issu de la décision dans la nouvelle liste
  * la nouvelle liste prend la place de l'ancienne
fin chaque
```

La methode qui aurait été implémentée si nous avions eu le temps parait la plus aboutie des trois (même s'il est difficile de l'évaluer correctement sans l'avoir observée en pratique) est la suivante : l'**implémentation d'une mémoire**.

En effet là où nous, joueur, pouvons observer les tendances des adversaires, l'IA ne peut rien. Nous avons donc décider de la doter d'une mémoire afin de rajouter un maillon essentiel au processus de décision qui est : l'évaluation des conséquences de ses choix précédents. Le principe est simple : si un chien fait une bêtise il sera grondé, il saura qu'il a fait une erreur et ne recommencera pas, au contraire si on lui donne un sucre suite à son double salto arrière, alors il le fera plus souvent et en épatera plus d'un ! Notre IA doit donc être capable de se souvenir si tel ou tel de ses actions ont eu de bonnes répercussions ou non (dans notre jeu et pour simplifier, si le bot a fait des dégâts ou bien en a encaissé).

L'implémentation est réalisée comme suit : l'IA possède deux arbres formés de toutes les combinaisons possibles, un pour l'attaque, un pour la défense. La taille des arbres est

liée au nombre d'actions disponibles et à la capacité de sa mémoire, dans notre cas et avec des enchainements qui ne dépassent pas plus de cinq actions, la complexité reste raisonnable. Chaque fois que l'IA est sollicitée, elle fait le bilan de son précédent enchaînement et incrémente en fonction les probabilités dans l'arbre. Certains enchaînements se verront donc privilégiés par rapport à d'autres (note : pour augmenter la difficulté d'une IA, augmenter la taille des enchaînements permet une mesure plus précise et un choix d'autant plus adapté à la situation). Il est bien sûr possible en poursuivant le raisonnement de stocker ces arbres dans des fichiers si l'IA est amené à rencontrer plusieurs fois le même adversaire afin qu'elle "apprenne" son style de jeu (nécessite l'implémentation de profils de joueurs).

3.12 Gestion du son

3.12.1 Mise en place du son

Il nous a semblé inconcevable de faire un jeu sans son, l'ambiance sonore étant presque aussi important que l'image pour l'immersion dans le jeu.

Nous avons donc fait le choix d'une bibliothèque pour le son. Il en existe plusieurs, notamment *SDL_mixer* et *fmod*, les deux étant aussi performantes, de complexité d'utilisation similaire. N'utilisant pas déjà *SDL*, nous avons choisi arbitrairement *fmod*.

Après quelques difficultés d'installation, la bibliothèque était opérationnelle, fonctionnant sous GNU/Linux 32 ou 64 bits.

Nous avons créé une classe **Sound** adaptée pour le jeu, utilisant certaines fonctions de *fmod*, afin de pouvoir jouer des sons aux moments appropriés dans le jeu de façon simple et pratique.

Cette classe permet les actions suivantes :

- Initialisation et création d'un système sonore qui sera utilisé pour tous les sons (méthode statique). En cas d'échec, le jeu ne dois pas être perturbé donc la classe est conçue pour continuer à fonctionner mais sans son
- Chargement d'une musique depuis un fichier, qui sera jouée en streaming pour ne pas surcharger la mémoire
- Chargement d'un petit son depuis un fichier, qui sera lu dans la mémoire pour pouvoir être lu rapidement (correspond aux voix et effets qui sont joués à plusieurs reprises durant le jeu).
- Au chargement, on précise si le son sera joué en boucle ou non
- Lecture / Arrêt d'un son
- Ajout d'un effet (écho, flanger, lowpass, highpass, etjenpass...)
- Ajustage du volume d'un son seul
- Ajustage du volume de tous les sons de type « musique »
- Ajustage du volume de tous les sons de type « effets/voix »
- Ajustage du volume général

Nous avons mis en place un réglage du volume général, des effets et de la musique via une fenêtre « préférences » accessible depuis le menu QT.

3.12.2 Choix des sons

Nous avons choisi de lancer une musique globale qui se répète en boucle tout le long du jeu. Cette musique vient du jeu vidéo *Super Smash Bros Brawl* et correspond bien à l'ambiance souhaitée, en particulier pour l'arène « Rome » actuellement utilisée. Ensuite, pour chaque personnage, nous utilisons des bruitages ou voix qui sont joués lors de certains événements, par exemple, donner un coup ou s'en prendre un. Des sons seront aussi choisis pour être ajoutés aux effets spéciaux, par exemple lors de certaines attaques, victoire, etc.

Nous envisageons aussi d'utiliser des effets sonores sur les sons ou la musique lors de certains événements.

3.12.3 Améliorations possibles

Il aurait été bien d'approfondir la gestion sonore, en ajoutant des effets plus évolués à certains moments, en utilisant plus de sons différents, en créant des sons nous-mêmes pour certaines voix par exemple.

Des sons pourraient aussi être joués à d'autres moments que ce qui a été fait.

Il serait par ailleurs intéressant de mettre plus de musiques, différentes selon les arènes, et différentes selon qu'on joue, qu'on est en pause ou qu'on est sur un menu.

Enfin des bruitages pourraient être ajoutés comme ambiance pour les arènes (pluie, vent, oiseaux, applaudissements) pour une meilleure immersion.

3.13 Principe du jeu

3.13.1 Guide utilisateur

Rappel du principe du jeu :

Smashtein Garbage est un jeu qui s'inspire fortement du célébre jeu *Super Smash Bros Brawl*.

Le but du jeu consiste à se taper dessus jusqu'à éjecter un joueur de l'arène ou jusqu'à épuisement de ses points de vie. Chaque robot possède un certain nombre de vies. Lorsqu'il n'y a plus de vie, le robot est éliminé. Le vainqueur sera donc le dernier restant.

Au démarrage du jeu, un écran d'introduction apparaît permettant de présenter le logo de notre jeu. L'écran suivant affiche le choix de l'avatar ainsi que de l'arène. Attention, que le combat commence !

Contrôlez votre avatar avec les touches directionnelles de votre clavier. Via la barre de menu, il est possible de changer de vue. Des raccourcis claviers y sont associés :

- ▷ Caméra poursuite (`CTRL+F1`) : suit un personnage
- ▷ Vue globale (`CTRL+F2`) : vue d'ensemble de tous les personnages
- ▷ Caméra debug (`CTRL+F3`) : permet de parcourir l'environnement en mode sphérique
- ▷ Caméra libre (`CTRL+F4`) : permet de parcourir l'environnement en vue FPS[8]
- ▷ Mode multi-vue (`CTRL+F5`) : affiche les 4 vues précédentes simultanément

Il est possible de **régler le niveau du son général**, de la **musique** et des **effets sonores** via le panneau de préférences. Ces préférences sont accessibles depuis la barre de menu ou depuis le raccourci `CTRL+P`

[8]First Person Shooter

4. Conclusion

Pour conclure, la réalisation du projet Smashstein s'est révélée comme une expérience très instructive et enrichissante en tout point.

Premièrement, elle nous aura permis de réaliser un véritable jeu vidéo dans des conditions proches de celles du travail en entreprise puisqu'au sein de la relativement grande équipe de ce projet, chacun a pu réellement se spécialiser afin de valoriser au mieux ses compétences propres et travailler de manière d'autant plus efficace.

Par ailleurs ce projet a été l'occasion d'approfondir nos connaissances et de nous aguerrir dans divers domaines. Bien entendu, la programmation C++ a joué un rôle central et nous avons pu grandement pratiquer ce langage en découvrant au passage de nouvelles bibliothèques. L'utilisation des logiciels de modélisation *3D* que sont *Blender* et *3DS MAX* a également été cruciale. Cependant, l'aspect le plus intéressant reste la découverte des mécanismes de combinaison de ces domaines qui n'avaient été abordés en cours que séparément jusqu'à présent.

Enfin, s'il fallait émettre un avis sur notre travail, nous pourrions dire que, même si le résultat est encore perfectible sur certains points, *Smashtein Garbage* remplit toutes les fonctionnalités demandées de manière satisfaisante et en apporte même de nouvelles. Nous sommes donc plutôt satisfaits du résultat, même si nous aurions aimé passer plus de temps sur la partie esthétique du jeu afin d'aboutir sur des décors et personnages plus jolis, mais aussi pourquoi pas de composer nos propres sons et mélodies à l'aide d'un logiciel comme *Rosegarden*.

5. Bibliographie

[1] Documentation officielle QT 4.0 [en] : http://doc.trolltech.com/4.0/

[2] J. Siek, L. Lee, A. Lumsdaine, The Boost Graph Library, User Guide and Reference Manual, Addison-Wesley, 2002

[3] FAQ LaTeX [en] : http://www.tex.ac.uk/cgi-bin/texfaq2html?introduction=yes

[4] Aide LaTeX [fr] : http://latex.developpez.com/faq/

6. Annexes

A.1 Quelques captures d'écran

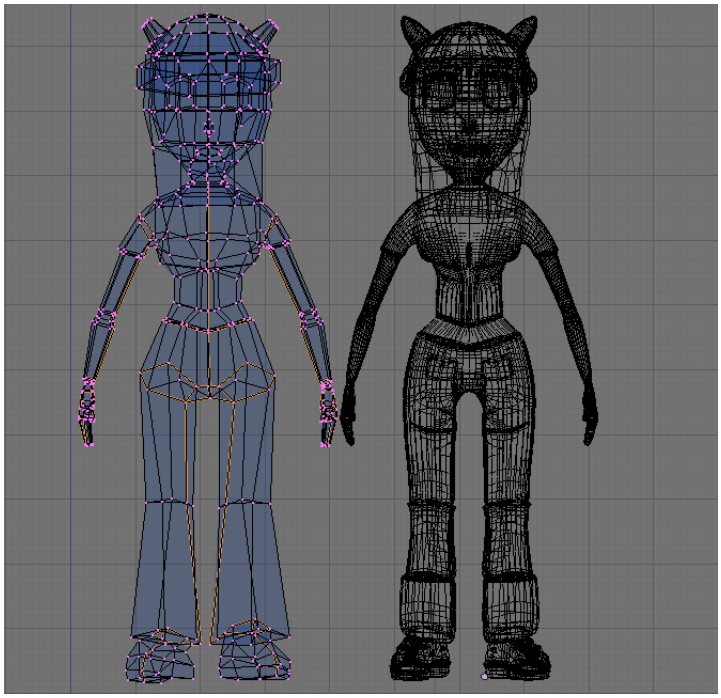


FIG. A.4 – Visualisation des meshes

FIG. A.5 – Test de rendu d'un personnage

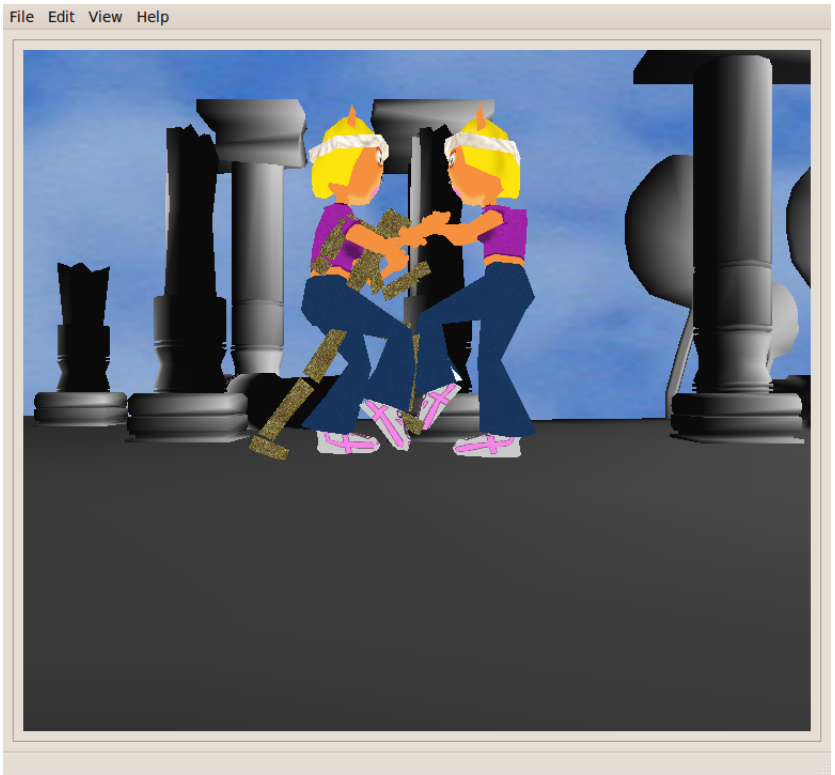


FIG. A.6 – Vue globale de l'arène

A.2 Pseudo-code de l'IA

```
Toutes les x frames : regarde s'il change de comportement (modulé par le comportement)
```

```
actions possibles :

direction droite
direction gauche
sauter
coup1
coup2
projectile
bloquer

-------------------------------------------------------------

structures de données :

arbre attaque
arbre defense
barette mémoire (file) "enchainement player" (même nb de barettes que de joueur)
barette mémoire (file) "enchainement bot"
barette action (file)

recupération d'un argument de main pour le profil du joueur
stockage des arbres de probabilités dans les fichiers textes

-------------------------------------------------------------

RECUPERATION DU RADAR + ACCES A LA MEMOIRE

qui est mon ennemi ?
  * y a-t-il plus d'un ennemi ?
    -> si non :
      * le choix est fait
    -> si oui :
      * choix entre différentes propositions
      -> quel est mon caractère ? (agressif, peureux, vengeur, etc...)
      /*
      Critères de selection
      si ennemi est très très bas en PV alors l'option correspondante recevra un bonus enorme
      si un ennemi a infligé + de 50 % de dégâts (pourcentage modulé par le caractère) alors il cherche a se venger
      si un ennemi a reçu + de 50 % de dégâts (idem) alors il va chercher à l'achever
      si les probabilités sont proches entre elles alors l'option "plus près"
      */

        -> le plus près
        -> le plus mal en point
        -> celui qui a infligé le plus de dégât
        -> celui qui a été le plus tapé

remplit la mémoire sur l'action précédente
  * quelle a été l'action du player ?
    * si la barette contient X (un seuil pédéfini) actions alors reset de la barette
    * si la dernière entrée de la barette date d'au moins X frame alors reset de la barette
    * enregistrement de l'action dans une barette mémoire temporaire "enchainement player"
  * le bot a-t-il fait des dommages ?
    -> si non :
      * si la barette contient X (un seuil pédéfini) actions alors reset de la barette
      * si la dernière entrée de la barette date d'au moins X frame alors reset de la barette
      * enregistrement de l'action précédente dans une barette mémoire temporaire "enchainement bot"
    -> si oui :
      * mise à jour du graphe d'attaque en parcourant selon l'enchainement effectué
  * le bot a-t-il encaisser des dommages ?
    -> si oui :
      * récupération de la barette temporaire "enchainement player"
      * mise à jour du graphe de defense en parcourant selon l'enchainement effectué (malus)
    -> si non :
      * récupération de la barette temporaire "enchainement player"
      * la dernière entrée de la barette date de moins X frame (obsolète)
        -> si oui :
          * reset de la barette
        -> si non :
          * mise à jour du graphe de defense en parcourant selon l'enchainement effectué (bonus)
```

```
quel est mon état ? choix :
  -> quel est mon caractère ? (agressif, peureux, vengeur, etc...)

  /*
  Notes comportement
  Certains possèdent des conditions ("instructions prioritaires")
  Exemple : agressif ne doit pas laisser de repos à son adversaire
  (pas attendre avant d'attaquer) et doit se trouver à middle range de sa cible
  */

    -> attaque
    -> défense

ai-je suffisamment de probabilités pour cet ennemi ?
// seuil fixé dans les arbres de proba et modulé par le caractère
  -> si non :
    * comportement passe à "recherche information"
    // c'est-à-dire remplir l'arbre de probabilité correspondant)

    /*
    Notes "recherche information"
    Parcours de l'arbre correspondant.
    Si une inconnue est rencontré sa priorité devient maximale,
    si plusieurs inconnues existent alors choix aleatoire.
    */

    -> si oui :
      * alors parcours de l'arbre de probabilité correspondant
        -> comportement X

mise à jour de la barette d'action
  * remplir la barette avec les actions données par la comportement
    -> y a-t-il un événement majeur ?
    // apparition super bonus, point de vie proche de zéro, se maintenir à middle range
      -> si oui :
        * insérer cette action en tête de barette stratégie

  * vérifié la validité de l'action (si elle n'a pas déjà été faite)
  * lancer la première action de la barette
```