

Documentation technique

Celerity

Angèle GERAUD, Baptiste LE GOFF,
Boris LABBE, Vincent DOUILLET

Table des matières

Introduction	1
1 Les logiciels utilisés	3
1.1 Présentation des outils	3
1.2 La place des outils dans le processus de création	3
1.3 Organisation des <i>Assets</i>	4
2 L'utilisation d'Unity	5
2.1 Prefab et GameObject	5
2.2 Notions sur l'utilisation de l'API d'Unity	6
3 Le First Person Controller	7
3.1 Gestion des collisions	7
3.2 Saisir le mobilier	8
3.3 Les armes	9
3.3.1 La batte de Baseball	10
3.3.2 Le lanceur de balles	10
4 Les portes	12
5 Les ennemis	14
5.1 Les paramètres	14
5.2 Les fonctions	15
6 Les effets de la TRR	17
6.1 Le First Person Controller	17
6.2 Les objets immobiles	17
6.3 Les objets mobiles	17
7 MiddleVR	19
7.1 Présentation de MiddleVR	19
7.2 Interfaçage avec Unity	19
7.3 Le serveur VRPN	20

Introduction

Cette documentation a pour but d'expliquer la manière dont l'application ludique *Celerity* à été construite.

Celerity permet à l'utilisateur de se déplacer dans un monde virtuel soumis aux lois de la Théorie de la Relativité Restreinte. L'immersion de l'utilisateur dans le monde virtuel est assurée par l'exploitation de périphériques de réalité virtuelle¹.

Les logiciels ayant servis seront présentés pour commencer, ainsi que leur utilisation dans le cadre de l'application. Ensuite, le fonctionnement d'Unity sera expliqué très brièvement, après quoi la conception des principaux composants de l'application sera commentée. Finalement, l'intégration de la TRR et de la RV seront détaillées.

1. Les termes « Théorie de la Relativité Restreinte » et « réalité virtuelle » seront respectivement abrégés « TRR » et « RV » dans la suite du document

Chapitre 1

Les logiciels utilisés

1.1 Présentation des outils

Blender est un logiciel libre de modélisation 3D. Blender est très complet et gère la création des modèles, l'application de textures sur ces derniers ou encore leur animation. Blender a été utilisé en version 2.65.

Unity est un environnement de développement de jeux vidéos. Il fournit un moteur graphique, un moteur physique ainsi qu'une interface graphique et une API adaptées pour exploiter ces deux moteurs. Unity a été utilisé en version 4.1.2f1.

MiddleVR est un plugin compatible avec plusieurs applications 3D, dont Unity. Son rôle est d'ajouter la gestion de périphériques de RV à ces applications.

1.2 La place des outils dans le processus de création

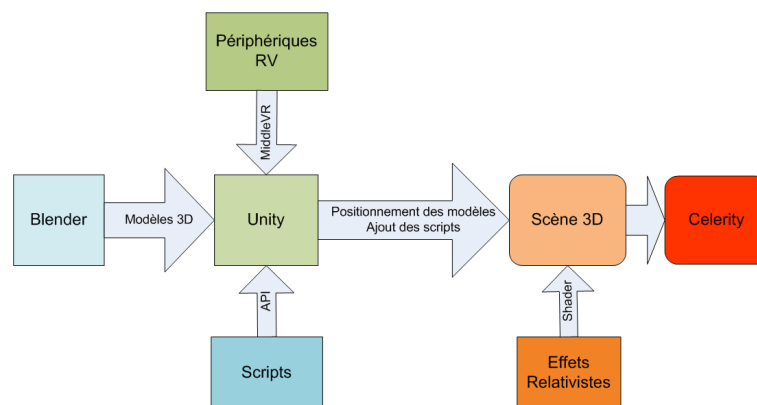


FIGURE 1.1 – L'utilisation de Blender et Unity

La manière dont ces logiciels ont été utilisés est représentée par la Figure 1.1.

1. Blender a servi à créer les modèles 3D. Les *Materials* de Blender ont également été utilisés pour ajouter des couleurs ainsi que des textures aux modèles. Aucune animation n'a été réalisée avec Blender.
2. Unity est le composant le plus important. Les modèles créés sous Blender y ont été importés pour devenir des *Assets* réutilisables un nombre quelconque

de fois dans Unity. Il existe différents type d'*Assets* : les modèles 3D sont des *Prefabs* et les matériaux associés à ces modèles dans Blender sont importés dans Unity sous la forme de *Materials*.

3. MiddleVR a été utilisé conjointement avec le serveur VRPN, permettant ainsi l'utilisation de WiiMotes pour contrôler le joueur.

1.3 Organisation des *Assets*

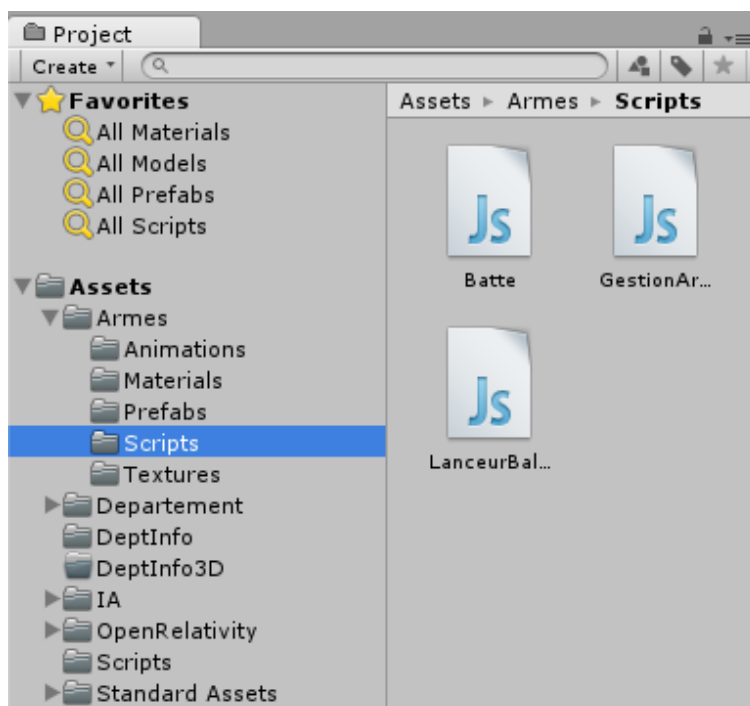


FIGURE 1.2 – L'organisation des *Assets*

Le problème de l'organisation des *Assets* peut paraître trivial mais celui-ci prend son importance lorsque leur nombre devient grand.

La Figure 1.2 montre l'arborescence dans laquelle les *Assets* ont été classés. Les différents dossiers portent des noms explicites et leur contenu est structuré de façon similaire avec les dossiers suivants, présents au besoin : Scripts, Materials, Prefabs, Sons, Textures. Les *Assets* attenants aux catégories délimitées par ces dossiers sont alors classés dans le sous-dossier adéquat. Les dossiers « Standard Assets » et « DeptInfo » sont gérés automatiquement par Unity et ne respectent donc pas cette organisation.

Dans le dossier « Department » se trouvent plusieurs *Prefabs*, dont `batimentstructure` qui correspond à toute la structure du bâtiment, sans le mobilier toutefois. Le mobilier, justement, se doit d'être un objet séparé de `batimentstructure` pour pouvoir lui appliquer des comportements ciblés tels que les interactions avec l'utilisateur. C'est donc avec Unity qu'il faut ajouter et positionner le mobilier sur la scène, et non pas directement avec Blender dans le fichier `batimentstructure`.

Chapitre 2

L'utilisation d'Unity

Les notions qui suivent concernent Unity et permettent de comprendre les descriptions techniques qui suivent. Cette documentation n'a toutefois pas vocation à expliquer en détails le fonctionnement d'Unity.

2.1 Prefab et GameObject

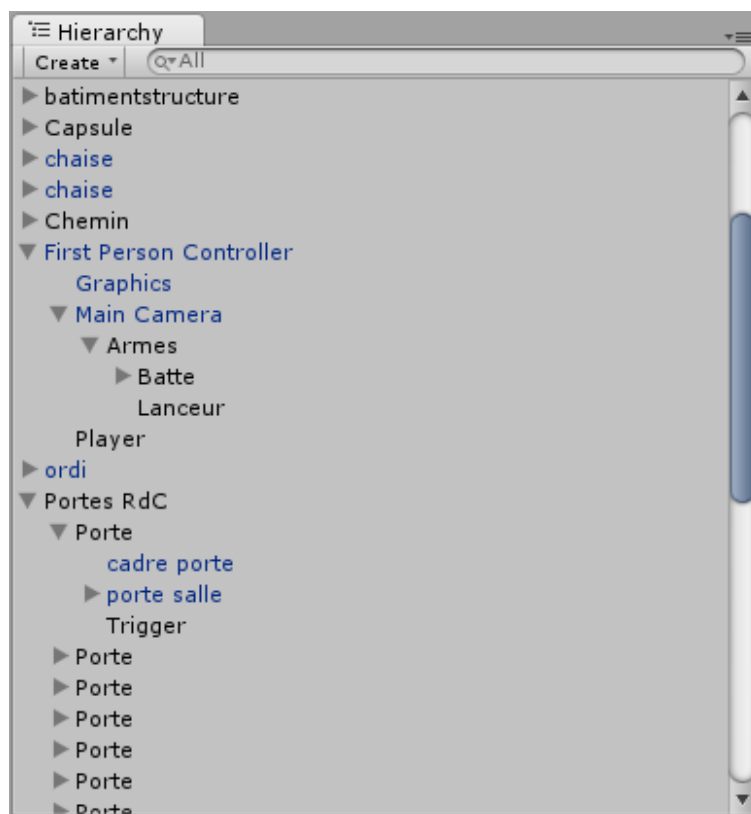


FIGURE 2.1 – La hiérarchie des GameObject présents sur la scène

Une fois les Prefabs disposés sur la scène d'Unity, ils deviennent alors des *GameObjects*. Les *GameObjects* sont listés dans la fenêtre « Hierarchy » (voir Figure 2.1) de l'interface d'Unity. Leur comportement en cours de jeu peut être spécifié avec des scripts, un autre type d'*Asset* qu'il est possible d'attacher à un *GameObject*.

Pour écrire ces scripts, l'API d'Unity a été exploitée.

Il est intéressant de noter que si plusieurs *Prefabs* exploitent le même *Material*, sa modification impacte directement tous les *Prefabs* l'utilisant.

Les *GameObject* peuvent posséder plusieurs composants. Les *Collider* servent à détecter les collisions. Les *RigidBody* servent à donner à un *GameObject* un comportement physique (ajout d'une masse, d'une inertie...). Dans le cas de *Celerity*, les objets mobiles et seulement ceux-ci possèdent un *RigidBody*. Un *GameObject* peut aussi être le fils d'un autre *GameObject*. Une telle relation est réalisée par glisser-déposer du fils sur son parent dans la fenêtre « Hierarchy ».

2.2 Notions sur l'utilisation de l'API d'Unity

L'API d'Unity est accessible par trois langages : le Boo, créé spécialement pour Unity, le Javascript et le C#. Dans le cas de *Celerity*, le Javascript a été utilisé pour toute la gestion du comportement du monde virtuel alors que le C# a servi pour l'implémentation des effets de la TRR.

Plusieurs fonctions peuvent être définies dans les scripts sans jamais être appelées explicitement, puisque c'est en réalité Unity qui les appelle automatiquement, aux instants suivants :

Start() : Lancement de l'application.

Update() : Au début du rendu de chaque image. L'intervalle de temps entre les appels est donc variable.

FixedUpdate() : Au début de chaque étape du moteur physique, à intervalles réguliers.

L'appel à d'autres fonctions caractéristiques peut être déclenché par des colliders par exemple.

Les scripts sont le plus souvent associés ou attachés à un *GameObject*. Il est alors possible d'accéder aux attributs de ce *GameObject* dans le script.

Chapitre 3

Le First Person Controller

Le First Person Controller est un asset standard fourni par Unity et représente, comme son nom l'indique, le point de vue qu'a l'utilisateur sur le monde virtuel. Il a donc un rôle central dans l'application puisque c'est par lui que sont gérées les collisions, la possibilité d'attraper les objets, les armes, mais aussi une partie des effets de la TRR. Tous ces éléments vont maintenant être détaillés.

Il est à noter toutefois que le *GameObject* « Player » attaché au First Person Controller est quant à lui détaillé dans la partie 6 (page 17) portant sur la TRR.

3.1 Gestion des collisions

```
1 var forceAppliquee = 2.0;
2
3 // Cette fonction est appelee lorsque le joueur entre en collision
  avec un objet
4 // L'objet est alors pousse
5 function OnControllerColliderHit (hit : ControllerColliderHit) {
6     var body : Rigidbody = hit.collider.attachedRigidbody;
7     // Pas de rigidbody
8     if (body == null || body.isKinematic)
9         return;
10
11     // Il ne faut pas pousser les objets en dessous du joueur
12     if (hit.moveDirection.y < -0.3)
13         return;
14
15     // Calcul de la direction vers laquelle deplacer l'objet,
16     // La poussee s'effectue sur les cotes, pas vers le haut / bas
17     var pushDir : Vector3 = Vector3 (hit.moveDirection.x, 0, hit.
        moveDirection.z);
18
19     // Application du deplacement
20     body.velocity = pushDir * forceAppliquee;
21 }
```

Listing 3.1 – Collision.js

La simulation de collisions réalistes est assurée par le script `Collision.js` attaché directement au First Person Controller. Ceci permet d'accéder, au sein du script, au *Collider* du First Person Controller pour détecter une collision avec tout autre objet.

C'est la fonction `OnControllerColliderHit` (ligne 5) fournie par l'API d'Unity qui est appelée lorsqu'une collision se produit. Cette fonction nous donne accès à

un objet de type *ControllerColliderHit* nous donnant accès aux informations utiles concernant la collision en cours comme le collider de l'objet, le rigidbody associé à l'objet (ligne 6), les coordonnées du point d'impact... Le script *Collision.js* se charge donc d'appliquer une force sur l'objet (ligne 20) avec lequel le FPS est entré en collision pour « pousser » l'objet dans la direction opposée à la collision.

3.2 Saisir le mobilier

```

1 function saisirObjet() {
2     var direction:Vector3;
3     direction = transform.TransformDirection(Vector3.forward);
4     // Lancement d'un rayon devant la camera
5     if(Physics.Raycast(transform.position, direction, hit, portee)) {
6         // Si on rencontre un rigidbody, il devient l'objet attrape
7         if(hit.rigidbody) {
8             rigidbodyAttrape = hit.rigidbody;
9             rigidbodyAttrape.useGravity = false;
10
11             // Calcul de la distance a laquelle doit rester l'objet pour ne
12             // pas gener le joueur dans ses déplacements
13             // On utilise une approximation de la taille de l'objet
14             var vecteurTaille = rigidbodyAttrape.collider.bounds.size;
15             // Selectionner le max des composantes de la taille avec un
16             // minimum de 2
17             distance = System.Math.Max(System.Math.Max(vecteurTaille.x,
18                 vecteurTaille.y),System.Math.Max(2,vecteurTaille.z));
19             distance *= 0.80;
20             objetSaisi = true;
21             SynchroTir.attraper();
22         }
23     }
24 }
```

Listing 3.2 – fonction saisirObjet() de Attraper.js

Cette fonctionnalité est implémentée avec le script *Attraper.js* attaché à la Main Camera, un fils du First Person Controller exposant la direction vers laquelle est tourné l'utilisateur (ligne 3). Cette direction est utile car le script *Attraper.js* lance un rayon en face de l'utilisateur, dans la direction où il regarde. Ce lancer de rayon est rendu possible par la fonction *Physics.Raycast* de l'API d'Unity. Cette fonction permet notamment de définir la longueur du rayon (variable *portee*) et donne accès à une information importante : la présence ou non d'un collider sur le chemin du rayon.

C'est sur cette information que se base le script, dans la fonction *saisirObjet* (voir Listing 3.2) : si un collider a été trouvé en face du joueur, à une distance inférieure à la portée spécifiée, alors le GameObject auquel ce collider appartient est asservi à la Main Camera, grâce à la fonction *asservirObjet*.

```

1 function FixedUpdate() {
2     // Si un objet est saisi
3     if(objetSaisi)
4     {
5         // Si on maintient Fire2, l'objet reste saisi
6         if(Input.GetButton("Fire2")) {
7             // Si on appuie sur "Fire1" l'objet est lance
8             if (Input.GetButton("Fire1")) {
9                 lancerObjet();
10            }
11        }
12    }
13 }
```

```

10     }
11     // Sinon il suit le joueur
12     else {
13         asservirObjet();
14     }
15 }
16 // Sinon il est libere
17 else {
18     libererObjet();
19 }
20 }
21 // Si un objet n'est pas saisi, on essaie d'en saisir un quand il y
   a appui sur "Fire2"
22 else
23 {
24     // Utilisation de tirPossible() pour eviter de pouvoir attraper de
       nouveau un objet venant d'etre lance.
25     if(Input.GetButton("Fire2") && SynchroTir.tirPossible()) {
26         saisirObjet();
27     }
28 }
29 }

```

Listing 3.3 – fonction FixedUpdate() de Attraper.js

C'est dans la fonction `FixedUpdate` (voir Listing 3.3) que le comportement de l'objet attrapé est spécifié. Si la touche « Fire2 » est maintenue alors l'objet continue de suivre la caméra. Sinon, l'objet est simplement libéré par l'appel à la fonction `libererObjet`. Toutefois, si un objet est saisi et que l'utilisateur envoie la commande « Fire1 » alors l'objet est lancé en avant (fonction `lancerObjet` suivant le même principe que celui utilisé pour les collisions : une force en avant lui est appliquée. Si aucun objet n'est en main et que l'utilisateur presse la touche « Fire2 » alors la fonction `saisirObjet` est appelée pour tenter d'attraper un objet. L'appel à la fonction `tirPossible` du script `SynchroTir.js` évite de rattraper un objet qui vient d'être lancé grâce à l'utilisation d'un timer.

3.3 Les armes

Les armes sont des fils de l'*Empty GameObject* « Armes », qui est lui même fils de la Main Camera. Un *Empty GameObject* est un *GameObject* n'ayant aucun impact sur le monde virtuel mais servant simplement à hiérarchiser les éléments présents sur la scène 3D.

```

1  function Update()
2  {
3      if (Input.GetKeyDown("1") || Input.GetKeyDown("c"))
4      {
5          SelectWeapon(0);
6      }
7      else if (Input.GetKeyDown("2") || Input.GetKeyDown("v"))
8      {
9          SelectWeapon(1);
10     }
11 }
12
13
14 function SelectWeapon(index : int)
15 {

```

```

16   for (var i=0;i<transform.childCount;i++)
17   {
18       // Activer l'arme selectionnee
19       if (i == index) {
20           transform.GetChild(i).gameObject.SetActiveRecursively(true);
21       }
22       // Desactiver toutes les autres
23       else {
24           transform.GetChild(i).gameObject.SetActiveRecursively(false);
25       }
26   }
27 }

```

Listing 3.4 – Extrait du script GestionArmes.js

Le script `GestionArmes.js` (voir Listing 3.4) sert à changer l'arme courante. Il repose sur la fonction `GetChild` (lignes 20 et 24) de l'API Unity qui sert à récupérer le *GameObject* fils dont le numéro est passé en paramètre. Une boucle se charge ensuite d'activer l'arme voulue et de désactiver toutes les autres, avec la fonction `SetActiveRecursively`.

Lorsqu'une arme est activée, elle apparaît à l'écran et peut être utilisée. Pour cela, chaque arme dispose d'un script attaché qui associe l'action attendue lors de l'activation de l'arme à l'appui sur la touche « Fire1 ».

3.3.1 La batte de Baseball

Dans le cas de la batte de Baseball, cette action correspond simplement à jouer une animation de frappe avec la batte. Une animation associée à un *GameObject* peut être facilement jouée à l'aide de la fonction `animation.Play` fournie par Unity.

3.3.2 Le lanceur de balles

```

1  var projectile : Rigidbody;
2  var speed = 40;
3  var dureeDeVie = 5;
4
5  function Update(){
6      if( Input.GetButtonDown( "Fire1" ) && SynchroTir.tirPossible() ){
7          var instantiatedProjectile : Rigidbody = Instantiate(
8              projectile, transform.position, transform.rotation );
9              instantiatedProjectile.velocity = transform.TransformDirection
10                 ( Vector3( 0, 0, speed ) );
11                 instantiatedProjectile.AddForce (0, 10, 0);
12                 Physics.IgnoreCollision( instantiatedProjectile.collider,
13                     transform.root.collider );
14             }
15
16             if(instantiatedProjectile){
17                 //Destruction de l'instance de la balle
18                 Destroy (instantiatedProjectile.gameObject, dureeDeVie);
19             }
20         }

```

Listing 3.5 – LanceurBalleRebondissante.js

Le lanceur de balles instancie le prefab correspondant à la balle (ligne 7) et lui donne une vitesse vers l'avant. Tout se passe ici dans la fonction `Update`. Au bout d'une durée de vie spécifiée par la variable `dureeDeVie`, la balle disparaîtra par l'appel à la fonction `Destroy`.

L'appel à la fonction `Physics.IgnoreCollision` (ligne 10) au moment de l'instanciation de la balle l'empêche d'entrer en collision avec le collider du First Person Controller.

Chapitre 4

Les portes

Les portes sont constituées de trois composants : le cadre, la porte en elle-même et un trigger (voir Figure 2.1). Le trigger est un *GameObject* de type *Cube* dont le composant *Mesh Renderer* a été désactivé, le rendant effectivement invisible. Le collider du trigger est également spécifié comme *IsTrigger*, ce qui signifie que ce collider ne provoque plus de collisions mais les détecte simplement. C'est le script `Porte.js`, attaché au trigger, qui va effectuer la rotation de la porte.

```
1  var amplitude : float = 90;
2  private var openAngle: float;
3  private var closedAngle : float;
4
5  private var open : boolean = false;
6  private var trigger : boolean = false;
7
8  private var targetValue : float = 0;
9  private var currentValue : float = 0;
10 private var easing : float = 0.038;
11
12 var porte : GameObject;
13
14 function Start(){
15     /* Initialisation des valeurs pour que la porte conserve sa position
16        initiale au lancement */
17     currentValue = porte.transform.eulerAngles.y;
18     targetValue = currentValue;
19
20     /* Calcul des valeurs de l'angle pour une porte ouverte et fermee */
21     closedAngle = porte.transform.eulerAngles.y;
22     openAngle = closedAngle + amplitude;
23 }
24
25 function Update () {
26     currentValue = currentValue + (targetValue - currentValue) * easing;
27     porte.transform.eulerAngles.y = currentValue;
28
29     if(trigger && Input.GetKeyDown("f"))
30     {
31         if(open)
32         {
33             fermerPorte();
34         }
35         else
36         {
37             ouvrirPorte();
38         }
39     }
40 }
```

```
37     }  
38   }  
39 }
```

Listing 4.1 – fonctions `Start()` et `Update()` de `Porte.js`

Pour les rotations, les angles que doit avoir la porte dans sa position ouverte et fermée sont calculés dans la fonction `Start` et la position de la porte est mise à jour dans la fonction `Update` (voir Listing 4.1 lignes 25 et 26), en fonction de deux variables `currentValue` et `targetValue` représentant les angles courant et à atteindre, respectivement. Le facteur `easing` sert à fluidifier le mouvement de la porte.

```
1  function OnTriggerEnter(other : Collider) {  
2    if(other.tag == "Player")  
3    {  
4      trigger = true;  
5    }  
6  }  
7  
8  function OnTriggerExit(other : Collider) {  
9    trigger = false;  
10 }
```

Listing 4.2 – fonctions `OnTriggerEnter()` et `OnTriggerExit()` de `Porte.js`

La fonction `OnTriggerEnter` (voir Listing 4.2) du script `Porte.js` est appelée lorsque le First Person Controller entre dans le volume délimité par le trigger. Cette fonction active la possibilité d'interagir avec la porte avec le booléen `trigger`. De façon symétrique, la fonction `OnTriggerExit` désactive les interactions avec la porte. Ensuite, dans la fonction `Update`, si l'utilisateur appuie sur « f » alors la fermeture / ouverture de la porte est déclenchée, en fonction de son état actuel mémorisé avec le booléen `open`.

Les fonctions `fermerPorte` et `ouvrirPorte` appliquent une rotation à la porte en modifiant les variables `currentValue` et `targetValue`.

Chapitre 5

Les ennemis

Le comportement des ennemis est spécifié par une intelligence artificielle assez basique. Ils suivent un chemin prévu à l'avance, et s'ils détectent le joueur proche d'eux ils vont le suivre et lui tirer dessus. Si ils perdent le joueur de vue alors ils retournent continuer leur chemin d'origine.

5.1 Les paramètres

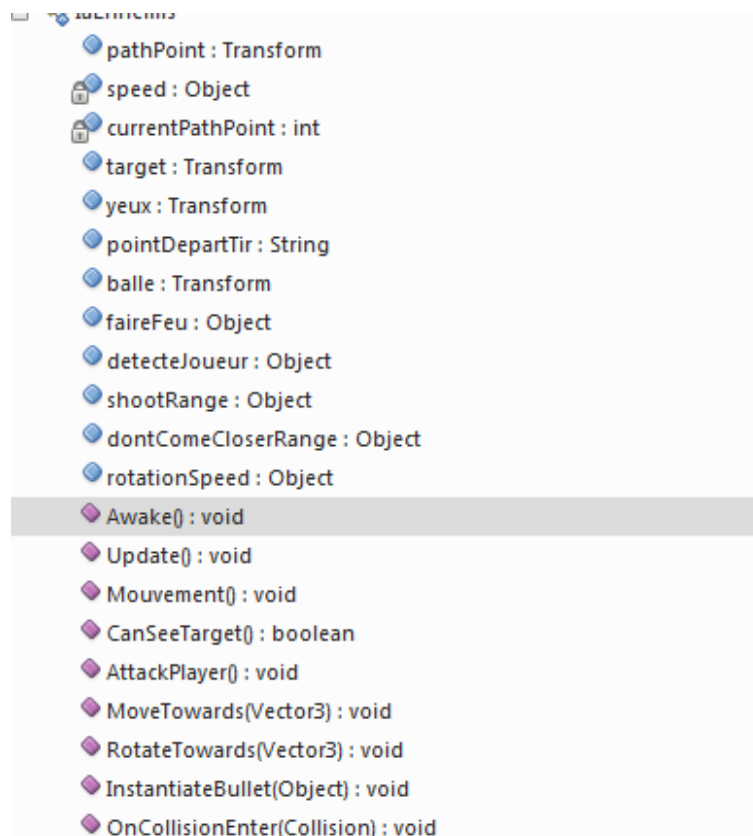


FIGURE 5.1 – Les différentes variables et fonctions utilisées pour l'IA

Voici une description des paramètres (voir Figure 5.1) utiles aux IA :

- `pathPoint` (paramètre) : Un tableau de *GameObject* vide représentant des

jalons (le premier élément `pathPoint[0]` est toujours vide). L'ennemi va suivre les différents jalons contenus dans le tableau.

- `speed` : Un entier qui définit la vitesse de déplacement de l'ennemi. Bien sûr rien n'empêche de le passer en paramètre pour associer une vitesse différente à chaque ennemi.
- `currentPathPoint` : Le point courant prédéfini (cette variable ne pourra pas contenir le joueur, qui lui est accessible avec `target`) vers lequel se dirige actuellement l'ennemi.
- `target` (paramètre) : La cible à repérer et à exterminer (le joueur ici).
- `yeux` (paramètre) : Indique la position des yeux de l'ennemi, c'est entre les yeux et le joueur que le script va vérifier s'il y a des obstacles, et donc déterminer si l'ennemi peut voir le joueur.
- `pointDepartTir` (paramètre) : Le point de départ des tirs.
- `balle` (paramètre) : Le *Prefab* utilisé pour la balle.
- `faireFeu` : Variable interne pour gérer la vitesse de tir.
- `detecteJoueur` : Booléen qui passe à vrai quand le joueur est détecté, faux sinon.
- `shootRange` : Constante qui définit la portée du tir de l'ennemi.
- `dontComeCloserRange` : Constante qui définit la distance minimale entre le joueur et l'ennemi (pour éviter que les ennemis arrivent au corps à corps, une valeur à 0.0 permettra à l'ennemi de se coller contre le joueur). Arrivé à cette distance, l'ennemi va viser le joueur.
- `rotationSpeed` : Vitesse de rotation de l'ennemi pour viser le joueur.

5.2 Les fonctions

Les paramètres précédents sont utilisés dans les fonctions suivantes (voir Figure 5.1) :

- `Awake()` : Fonction d'initialisation, initialise le tableau `pathPoint` ci-dessus.
- `Update()` : Appelle la fonction `Mouvement()` au début de chaque rendu d'image. Met à jour le booléen `detecteJoueur` grâce à la fonction `CanSeeTarget()`.
- `Mouvement()` : Si le joueur est repéré, cette fonction appelle `AttackPlayer()`, sinon fait bouger l'ennemi entre les points prédéfinis.
- `CanSeeTarget()` : Vérifie si l'ennemi voit le joueur ou non. Elle vérifie d'abord si la distance absolue entre le joueur et l'ennemi n'est pas trop grande. Ensuite elle vérifie quel est le premier élément avec lequel on peut entrer en contact entre le contenu de la variable `yeux` et `target`, si ce contenu correspond au joueur (`resultat = target`) alors l'ennemi a détecté le joueur.
- `AttackPlayer()` : Vérifie que l'ennemi est à portée de tir, sinon il avance. La fonction vérifie ensuite si l'ennemi est à la bonne distance pour tirer, et commence à viser le joueur. Si jamais l'ennemi ne voit plus le joueur à la fin de l'action décidée (avancer, viser ou tirer), l'ennemi retourne sur son chemin.
- `MoveTowards(position)` : Avance l'ennemi vers la position passée en paramètre.
- `RotateTowards(position)` : Effectue une rotation vers la position passée en paramètre (vise la position).
- `InstantiateBullet(temps)` : Instancie une nouvelle balle tous les `temps` secondes.
- `OnCollisionEnter(other)` : Fonction appelée lorsque l'ennemi entre en collision avec un objet. S'il entre en collision avec une arme ou un élément mobile

du décor, l'ennemi est détruit.

Chapitre 6

Les effets de la TRR

Plusieurs objets sont affectés par l'ajout de la TRR mais ils demandent des modifications différentes en fonction de leur rôle au sein de l'application.

6.1 Le First Person Controller

L'*Empty GameObject* « Player », fils du First Person Controller, rassemble tous les scripts nécessaires à la TRR pour le First Person Controller.

VelocityFPC.cs : Calcule la vitesse du First Person Controller en la lissant. Cette vitesse est utilisée par les scripts ci-dessous pour les calculs des effets relativistes

GameState.cs : Permet au First Person Controller d'être cohérent avec les lois de la TRR et de mémoriser son état pour réutilisation dans les autres scripts.

InfoScript.cs : Affiche à l'écran la vitesse de déplacement actuelle du First Person Controller et la vitesse de la lumière.

SpeedOfLightManager.cs : Permet d'augmenter ou de réduire manuellement la vitesse de la lumière en appuyant sur les touches « n » et « m » respectivement.

6.2 Les objets immobiles

Pour les objets immobiles, seul le matériau doit être modifié en *ColorShift* (voir Figure 6.1). Ceci correspond à l'utilisation d'un shader ajoutant les effets relativistes à l'objet. Il faut ensuite renseigner la texture à appliquer dans le champs « Base (RGB) ».

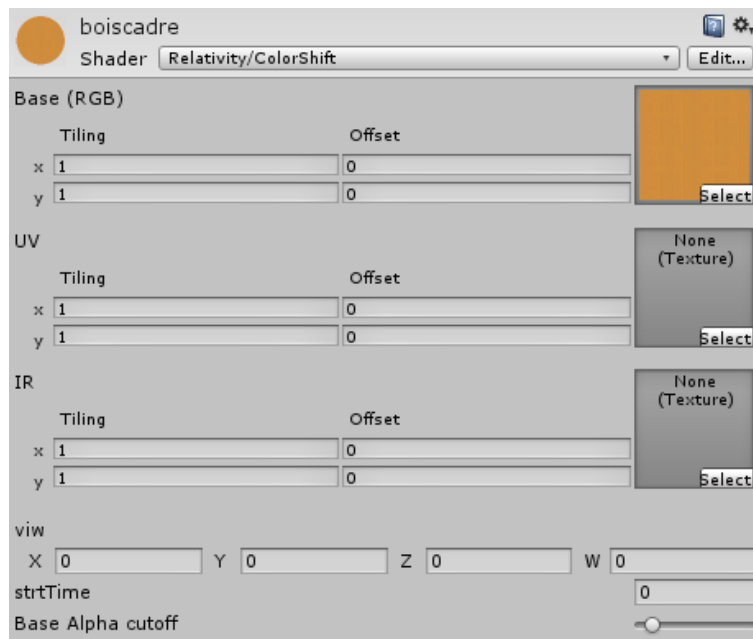
6.3 Les objets mobiles

En plus de modifier le matériau comme pour les objets immobiles, il faut ajouter trois scripts aux objets mobiles :

VelocityFPC.cs : Calcule la vitesse du First Person Controller en la lissant. Cette vitesse est utilisée par les scripts ci-dessous pour les calculs des effets relativistes

RelativisticObject.cs : Sert à calculer les transformations adéquates à appliquer à l'objet en fonction de sa vitesse.

SpeedSynchro.cs : Synchronise la vitesse de l'objet entre les deux scripts précédents, sans quoi les effets relativistes ne seront pas activés quand ce n'est pas le First Person Controller qui bouge mais l'objet lui-même.

FIGURE 6.1 – L'application du *shader* relativiste

`SpeedLimiter.cs` : Limite la vitesse de l'objet pour qu'elle ne dépasse pas celle de la lumière, car cela est impossible selon la TRR.

Chapitre 7

MiddleVR

7.1 Présentation de MiddleVR

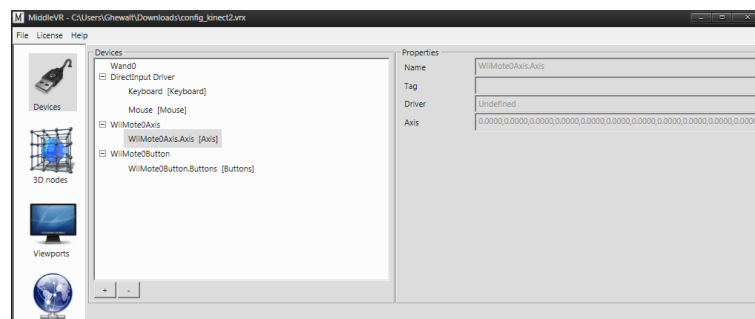


FIGURE 7.1 – Le logiciel MiddleVR

MiddleVR (Figure 7.1) est un plugin puissant qui permet aux applications d'utiliser la Réalité Virtuelle. Avec son plugin compatible Unity il permet aux applications ludiques de recréer l'impression de profondeur mais aussi d'utiliser différents périphériques de RV comme la WiiMote ou la Kinect. Pour notre projet nous allons seulement utiliser la Wiimote, nous allons donc détailler le fonctionnement de l'interaction entre la WiiMote et Unity.

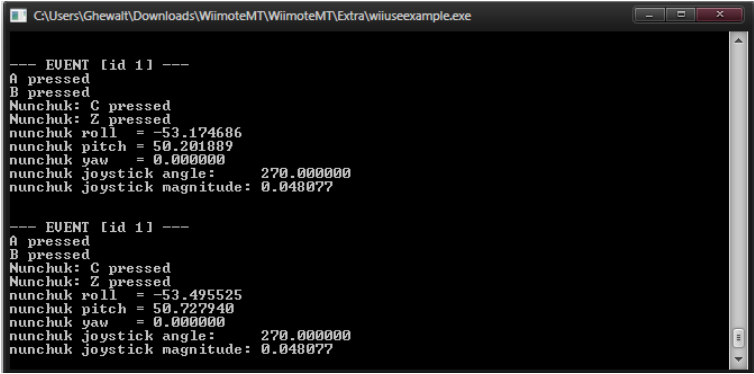
7.2 Interfaçage avec Unity



FIGURE 7.2 – Chaîne des logiciels entre Unity et la WiiMote

La Wiimote se connecte au PC comme un simple appareil bluetooth, mais Windows ne peut pas l'utiliser directement, un logiciel tiers doit être présent pour pouvoir détecter les différentes actions effectuées à l'aide de la manette (appui d'un bouton, accéléromètre...). Pour effectuer cela MiddleVR utilise VRPN (voir Figure 7.2), une bibliothèque libre fournissant une interface entre une application et les dispositifs de RV.

7.3 Le serveur VRPN



```
C:\Users\Ghewalt\Downloads\WiiMoteMT\WiiMoteMT\Extra\wiiuseexample.exe

--- EVENT [id 1] ---
A pressed
B pressed
Nunchuk: C pressed
Nunchuk: Z pressed
nunchuk roll = -53.174686
nunchuk pitch = 50.201989
nunchuk yaw = 0.000000
nunchuk joystick angle: 270.000000
nunchuk joystick magnitude: 0.048077

--- EVENT [id 1] ---
A pressed
B pressed
Nunchuk: C pressed
Nunchuk: Z pressed
nunchuk roll = -53.495525
nunchuk pitch = 50.727940
nunchuk yaw = 0.000000
nunchuk joystick angle: 270.000000
nunchuk joystick magnitude: 0.048077
```

FIGURE 7.3 – Un serveur VRPN reconnaissant une WiiMote

Grace à VRPN Windows peut reconnaître les pressions sur les touches, et les variations de position de la WiiMote (voir Figure 7.3). Pour finir il faut faire le lien avec Unity 3D grâce à un script. Les étudiants de 4^{ème} année nous ont fourni ce dernier, qui permet un contrôle de notre personnage à la WiiMote directement dans Unity.