Projet d'application N° 231
Rapport

Noms des élèves :
COLLETTE Baptiste
DEPIERRE Amaury
LI Zepeng
MARTIN Thomas
SAKHO Momar
Commanditaire :
AEECL
Tuteurs scientifiques :
CHALON René
MULLER Daniel
Date du rapport :
14/04/2014

Remerciements

Nous tenons à remercier Monsieur MULLER et Monsieur CHALON pour nous avoir supervisés pendant ce projet. Nous remercions également le Learning Lab de Lyon pour avoir financé l'Oculus Rift et la version de Unity 3D pro que nous avons utilisés. Nous tenons à remercier Unity pour nous avoir fourni une version étudiante de leur logiciel, ainsi que les modèles trouvés sur l'Asset Store de Unity que nous avons utilisés. Nous remercions aussi les personnes qui nous ont aidés pendant les bêta-tests de notre jeu. Enfin, nous tenons à remercier les étudiants des autres projets liés à celui-ci pour nous avoir apporté de l'aide et des points de vue externes.

Résumé

Ce projet d'application avait pour but de reprendre un PE de l'an dernier portant sur la réalisation d'un jeu sur le campus de l'École Centrale de Lyon. La grande différence portait sur le fait que le jeu serait réalisé cette année en 3D via le moteur de jeu Unity. En reprenant des éléments du précédent jeu, nous avons cherché à améliorer l'expérience du joueur en proposant un jeu de rôles avec un scénario, des ennemis présents sous la forme de zombies et des niveaux pour la montée en puissance du joueur. Le jeu était également prévu pour être utilisé avec un casque de réalité augmentée Oculus Rift et il devait incorporer des éléments d'autres PA et PE gravitant autour tels que des minijeux ou une prise en compte de l'émotion du joueur.

Nous avons décidé de ne pas repartir de l'ancienne version du jeu, mais de recommencer à partir de zéro car le passage de la 2D à la 3D est délicat. Nous avons utilisé pour cela plusieurs logiciels tels que Unity, Blender ou MakeHuman. Les explications de notre réalisation sont détaillées dans le présent rapport. Au terme de ce PA nous avons réussi à réaliser un jeu fonctionnel d'une durée de vie de 1 heure, incluant une trame principale, plusieurs ennemis et un campus modélisé en 3D. Nous pouvons suggérer plusieurs améliorations sur le jeu actuel. Le campus n'est pas entièrement modélisé et plusieurs bâtiments restent à créer. De même, certains personnages ne sont pas encore modélisés. Une grande fonctionnalité prévue initialement était les relations avec les différents laboratoires de Centrale Lyon qui n'ont pas pu être développées en détail (un système d'allégeance était initialement prévu). Le projet pourra donc être repris par des élèves l'an prochain, d'autant plus que nous avons écrit une documentation permettant d'aider ceux consultant notre code.

Contenu

Table des figures

Introduction

Ce rapport présente les principaux résultats du projet d'application n°231 de l'année 2014-2015 de l'Ecole Centrale de Lyon. Le but de ce projet est de créer un jeu vidéo en 3D se passant sur le campus de l'Ecole Centrale de Lyon, en utilisant le moteur de jeu Unity 3D.

Notre projet s'intitule "ECLementary2", référence à "ECLementary", nom du projet d'étude duquel ce projet est la continuation. L'idée de ce jeu vidéo est de permettre aux étudiants, aux professeurs et au personnel de l'Ecole de découvrir le campus à travers ce jeu. Pour permettre plus d'interactions, le modèle de jeu adopté est le RPG (Role Playing Game, type de jeu dans lequel le personnage principal accomplit des quêtes dans un univers ouvert).

Comme ce projet suit le PE31 de l'année 2013-2014, nous avons souhaité conserver l'idée d'un joueur qui chercherait à libérer le campus de monstres qui l'auraient envahi. Nous avons rajouté à ce scénario initial plusieurs améliorations présentées dans ce rapport : une plus grande liberté de mouvement pour le joueur, plusieurs armes, et une possibilité d'interagir avec d'autres personnages. Ce projet a deux problématiques : la modélisation du campus de l'Ecole Centrale de Lyon, et l'implémentation du jeu vidéo en lui-même.

Dans une première partie, nous expliquerons le contexte du projet et nos objectifs. Dans une deuxième partie, nous présenterons les différents outils que nous avons exploités pendant ce projet. Enfin, dans la dernière partie, nous définirons les mécanismes du jeu que nous avons créé.

I - Présentation du projet

1) Contexte

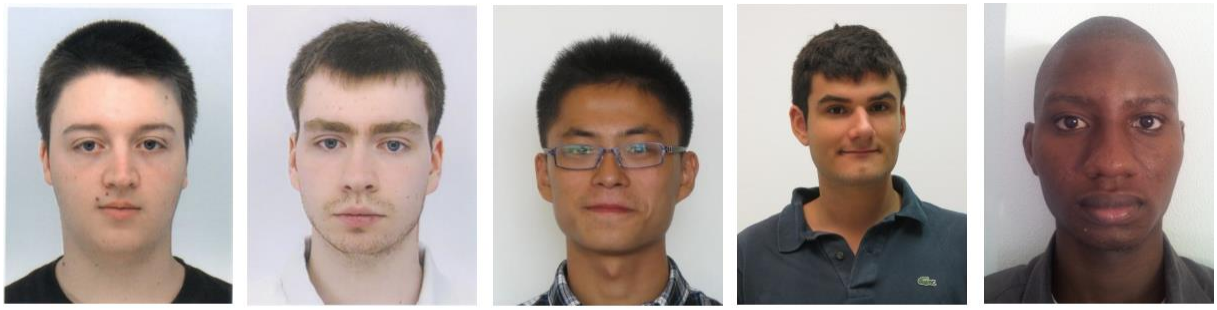
Le Projet d'Application 231 se situe dans la continuité du Projet d'Étude 31 de l'année précédente (2013-2014). L'objectif de ce PE était de créer un jeu vidéo se déroulant dans l'enceinte du campus de l'École Centrale de Lyon et de l'EMLyon. A l'issue de l'année de travail sur le PE, le jeu se composait de différentes scènes en deux dimensions, dans lequel le joueur pouvait se déplacer et interagir. Ce PA a eu pour principal but de s'inspirer de ce jeu afin d'en faire une version en trois dimensions. Pour cela, nous avons utilisé le moteur de jeu Unity 3D afin de construire une représentation du campus en 3D et d'y insérer les caractéristiques du jeu. Cette structure permet facilement l'intégration dans le jeu de différents modules qui pourront être développés dans le futur.

A l'heure actuelle, le secteur des jeux vidéo est en plein essor. Il convient donc de spécifier les caractéristiques du jeu attendu pour répondre aux attentes des joueurs et des commanditaires. Le jeu créé est principalement destiné aux étudiants et personnel de l'ECL. Nous avons donc pris en compte la cible à qui ce jeu est destiné lors de sa conception.

2) Présentation de l'équipe et répartition des tâches

Organisation de l'équipe

L'équipe du projet est constituée de 5 personnes : Baptiste Collette, Amaury Depierre, Zepeng Li, Thomas Martin et Momar Sakho. Leur fonction au cours du projet est précisée sur la figure 1.



Baptiste Collette	Amaury Depierre	Zepeng Li	Thomas Martin	Momar Sakho
Chef de projet	Responsable Unity	Responsable IA	Responsable modélisation	Responsable C#

Figure 1 : Trombinoscope de l'équipe

Nous avons opté pour l'utilisation d'un GANTT général pour tout le projet afin de voir dans quel ordre exécuter les tâches (des extraits figurent dans la partie ci-dessous). En pratique, nous avons opéré en méthode agile ; toutes les semaines, nous testons le logiciel pour savoir quels bugs sont apparus et quels ajouts étaient à faire.

Organigramme des Responsabilités et Gantt associé

Le projet s'est divisé en deux grandes tâches : la modélisation du campus sous Unity 3D et Blender, et l'implémentation du principe de RPG. La figure 2 présente les différentes tâches de la modélisation du campus, ainsi que les responsables pour chacune d'elles. La figure 3 quant à elle présente la répartition temporelle de ces tâches.

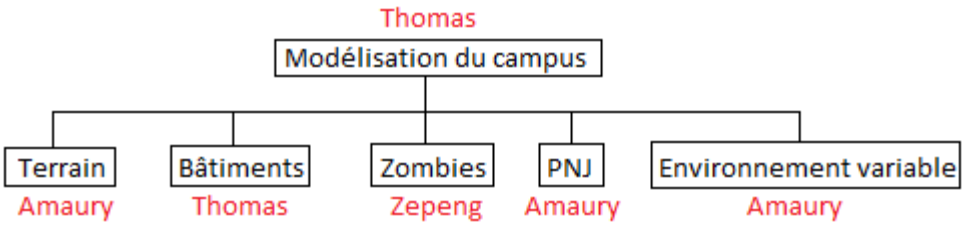


Figure 2 : Organigramme des responsabilités pour la modélisation du campus

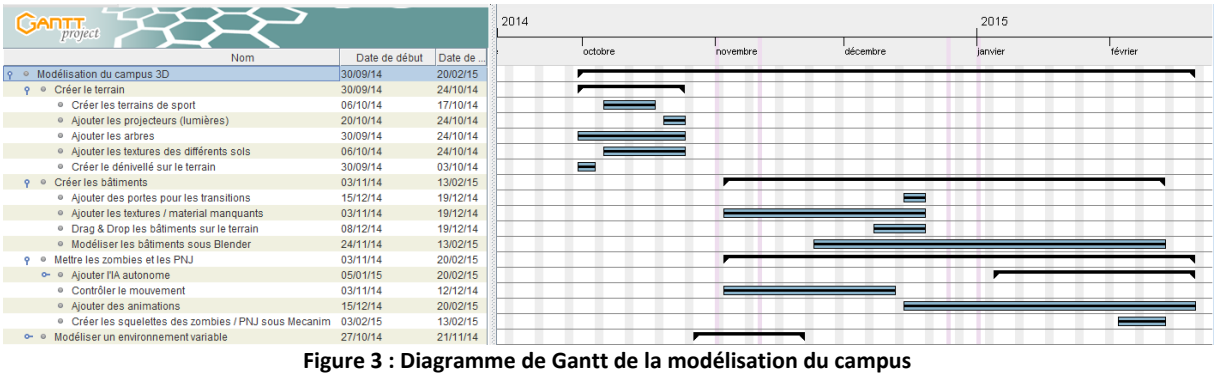


Figure 3 : Diagramme de Gantt de la modélisation du campus

La modélisation du campus se fait en cinq étapes : créer le terrain sur lequel le joueur va évoluer, modéliser les bâtiments accessibles, modéliser et faire bouger les zombies, créer les personnages non jouables avec lesquels le joueur va interagir, et modéliser un environnement variable selon l'avancement du scénario et l'heure.

La figure 4 présente les différentes tâches nécessaires à l'implémentation du RPG, ainsi que les responsables pour chacune d'elles. La figure 5 présente le diagramme de Gantt prévisionnel correspondant à ces tâches.

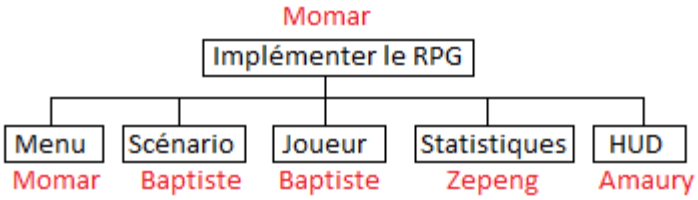


Figure 4 : Organigramme des responsabilités pour l'implémentation du RPG

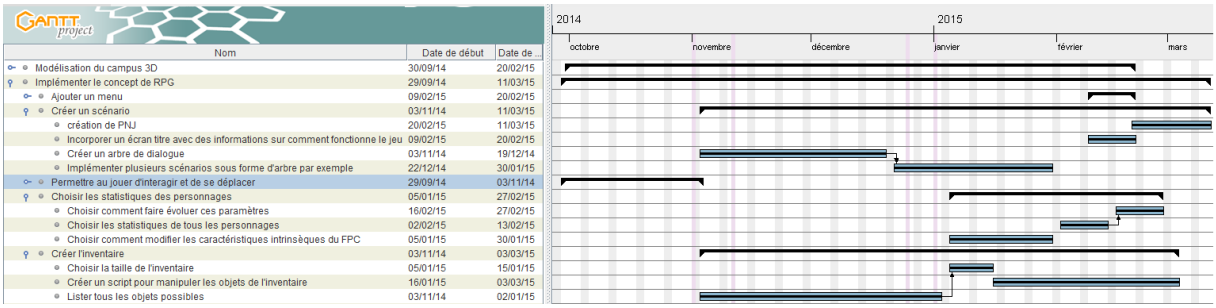


Figure 5 : Diagramme de Gantt pour l'implémentation du RPG

L'implémentation du principe de jeu de rôle se divise elle aussi en cinq étapes : créer un scénario, créer un menu pour que le joueur puisse notamment sauvegarder, créer un joueur que l'utilisateur contrôle, implémenter un système de statistiques pour que le joueur puisse s'améliorer au fil du scénario, et afficher à l'écran un ensemble d'informations (ATH) telles que la barre de vie, d'endurance, de mana, d'expérience, l'arme équipée actuellement ou encore un radar.

3) Cahier des charges

Le code nécessaire à la mise en relation des personnages, des objets et des niveaux a été écrit en langage C# afin de permettre une intégration future de différentes fonctionnalités annexes (par exemple pour pouvoir ajouter le contenu du travail d'autres projets).

Nous avons décidé de développer un jeu de type Role Playing Game (RPG) en vue à la première personne. Le scénario sera linéaire et le joueur pourra avancer en réalisant différentes quêtes, tout en faisant évoluer son personnage.

Les livrables sont multiples. Le logiciel (format .exe ou .app) fourni comprend une partie du campus de l'ECL modélisée en trois dimensions, l'implémentation d'un scénario basique avec plusieurs quêtes à réaliser, un système d'évolution basique du joueur, d'autres personnages avec lesquels interagir (Personnages Non Jouables, PNJ en abrégé) ainsi qu'une partie du jeu compatible avec la technologie Oculus Rift. Les assets créés pour ce projet à l'aide de Blender ainsi que le code source que nous avons fait sont délivrés sous la licence CC BY-NC-SA 4.0 (https://creativecommons.org/licenses/by-nc-sa/4.0/).

En plus du logiciel, et du présent rapport détaillant le déroulement du projet, un document expliquant plus en détail comment réaliser les différentes opérations nécessaires à la réalisation d'une fonctionnalité du projet est fourni en annexe.

4) Objectifs

L'objectif de notre jeu vidéo, baptisé ECLementary 2, est de proposer une aventure de type jeu de rôles (combat contre des ennemis et montée en puissance de notre personnage) dans un campus modélisé en 3 dimensions.

Le joueur devait pouvoir suivre des quêtes non obligatoires et être confronté à plusieurs choix tels que l'allégeance à un laboratoire en particulier. Un autre objectif était de pouvoir jouer au jeu avec un casque de réalité augmentée Oculus Rift. Cela aurait été utile pour la coopération avec les autres projets jeu vidéo.

En effet, ce projet est lié à 6 autres projets : le PE 26, qui reprend le code et l'idée du PE 31 de l'année dernière, et qui doit entres autres rajouter du contenu multimédia (photos, mini-jeux) tout en étant libre d'apporter des changements. Les autres projets étaient :

- Un PE créant un mini-jeu qui sera incorporé dans notre projet lorsqu'il sera fini.
- Un PAi s'intéressant aux mesures des émotions dans le jeu vidéo.
- Un PAr cherchant comment modifier un environnement de façon dynamique selon les émotions et actions du joueur.
- Un PAr étudiant les sentiments de peur de joie, etc… chez le joueur avec des encéphalogrammes.
- Un PAr cherchant à créer des dialogues réalistes basés sur des personnages réels (ici les doctorants et les professeurs de l'École Centrale de Lyon.

Trois réunions ont été tenues cette année entre tous les PA, pour décider comment les autres PA allaient s'incorporer à notre projet. Cependant, ces réunions n'ont pas abouti à une implémentation complète des autres projets au notre ; seule la scène du labyrinthe, que nous avons créée, a été utilisée par les autres projets.

Tous les objectifs n'ont pas pu être tenus. La modélisation du campus reste par exemple limitée car nous ne disposons pas de tous les bâtiments. Cependant le jeu reste pleinement jouable et possède une durée de vie approximative d'une heure, ce qui est suffisant pour proposer une vraie expérience au joueur. L'Oculus Rift n'a pu être utilisé que dans une seule scène du jeu (le labyrinthe) car les autres possédaient des objets modélisés sous Blender qui ne peuvent pas être visualisés avec l'Oculus à cause de problèmes de compatibilité.

II - Outils et méthodologie utilisés

1) Gestionnaire de version et système de commentaire

Pour une collaboration efficace et un bon suivi des changements apportés au projet, nous nous sommes tournés vers le gestionnaire de version Git. Écrit par Linus Torvalds, Git est un puissant logiciel, libre et gratuit. Il fonctionne selon l'architecture peer-to-peer, mais pour des besoins d'uniformisation des versions, nous avons aussi utilisé la plateforme Github, qui héberge un serveur git.

Git n'est pas le seul gestionnaire de version du marché. Il compte notamment Svn, Bazaar et Mercurial parmi ses concurrents. Cependant, svn, bien qu'encore très utilisé, est de plus en plus considéré comme dépassé alors que Mercurial et Bazaar n'ont rien à envier à Git.

Le code est commenté en utilisant la convention Doxygen, afin de générer une documentation du code à l'aide du logiciel doxywizard. Cette convention est la suivante : dans chaque fichier créé, on spécifie quand le fichier est créé, par qui, et surtout dans quel but. De même,

pour toutes les fonctions et méthodes créées, on spécifie comment et pourquoi la fonction marche, quels sont les paramètres qu'on lui donne en entrée, et quel paramètre est renvoyé en sortie.

Le logiciel doxywizard et le gestionnaire d'images GraphViz permettent de générer une documentation du code du projet, permettant notamment de savoir comment les classes sont liées entre elles et quelles sont toutes les fonctions créées.

2) Unity 3D

Le projet utilise le moteur de jeu Unity 3D (qu'on appelle plus brièvement Unity). Au début du projet, la version la plus récente du logiciel était la version 4.5. Aujourd'hui, la version utilisée est la version 4.6.1f1. C'est sur celle-ci que le code fournit pourra fonctionner (pour éviter tout problème de compatibilité).

Le système de GameObject

Un GameObject est le composant primaire sur lequel se base Unity. Un GameObject peut représenter un personnage, un élément du décor, ou encore un scénario. Un GameObject en lui-même n'est rien : c'est un container sur lequel on peut attacher d'autres objets, parmi lesquels :

- Des lampes : Unity propose plusieurs types de lampes : des lampes directionnelles, qui éclairent la scène dans une direction à la manière du soleil ; des spots lumineux, qui agissent comme des projecteurs d'un certain rayon ; et des lampes ponctuelles, qui émettent dans la lumière autour d'elles dans un certain rayon.

- Des colliders : C'est le corps rigide de l'objet. C'est sa représentation physique. C'est avec ce collider qu'on va pouvoir étudier les collisions du personnage. Les colliders peuvent être de deux types : rigides, ou trigger. Un collider rigide ne peut pas être traversé, il se "cogne" aux objets de la scène. Un collider "trigger" est une zone invisible qu'on peut traverser.

- Des meshs : Ce sont les représentations graphiques des objets. C'est dans ce composant qu'on dit au GameObject à quoi il va ressembler. Cependant, ce n'est parce qu'un objet ressemble à une chaise qu'il va agir comme tel ; il est possible de donner comme mesh une chaise et comme collider un cube.

- Des rigidbody : C'est le composant qui dit au moteur physique de Unity de calculer l'évolution de la position d'un objet. Un objet ne subit d'actions extérieures que s'il contient cet objet. De ce fait, une collision entre deux objets ne sera considérée par Unity que si ces deux objets ont un collider mais également un rigidbody. Un rigidbody permet également de définir comment l'objet va subir les actions extérieures. Il est possible de dire si oui ou non l'objet va subir la gravité (sachant qu'un objet sans rigidbody ne la subira pas), il est également possible de bloquer la rotation / translation d'un objet selon chacun de ses axes, mais également de définir un objet comme kinematic. Un objet kinematic ne subira aucune force, mais pourra subir des collisions. C'est une façon pratique de permettre à un objet dans les mains du joueur de subir des collisions sans lui faire subir les forces liées au corps du joueur.

- Des sources audio : Un objet qui a une source audio pourra émettre du son. Une audioSource ne produit pas de son en elle-même : il faut lui définir un clip audio (audioClip) qu'elle pourra jouer.

- Des systèmes de particules : Un objet qui a ce composant va générer des particules selon un schéma défini par l'utilisateur. Il peut choisir la couleur des particules, leur taille, le temps pendant lequel elles seront générées... Ce procédé est utilisé dans le rendu d'effets physiques tels que des flammes ou de la fumée.

- Des caméras : C'est sur un GameObject qu'on met la caméra par laquelle le joueur va voir les scènes.

- Des scripts : Les scripts dérivant de la classe MonoBehaviour de Unity peuvent être attachés à des gameObjects et modifier leur comportement. Il s'agit du « code brut » du jeu.

Choix du langage

Le langage imposé pour ce projet est le C#. Tous les scripts (hors programmes annexes) sont donc faits dans ce langage. Unity propose deux autres langages : Javascript et Boo, sachant qu'il est relativement compliqué d'utiliser deux langages pour un même projet. En effet, certains scripts cherchent à accéder à d'autres scripts (car ce sont eux qui définissent le comportement des gameObjects) ; si ces deux scripts ne sont pas faits dans le même langage, ils ne pourront pas communiquer.

Au début du projet, peu de tutoriaux étaient faits en C#. Certains sont proposés en annexe. La documentation est une façon rapide d'en apprendre plus sur les méthodes de Unity : c'est la principale source d'information.

Le principe de séparation en scènes

Unity fonctionne en un système de scènes. Le jeu est donc séparé en plusieurs scènes que le joueur arpente selon le scénario. Il y a de nombreuses scènes dans le jeu, notamment la scolarité, le gymnase ou le campus. La figure 6 présente un exemple de scène, l'intérieur du gymnase.

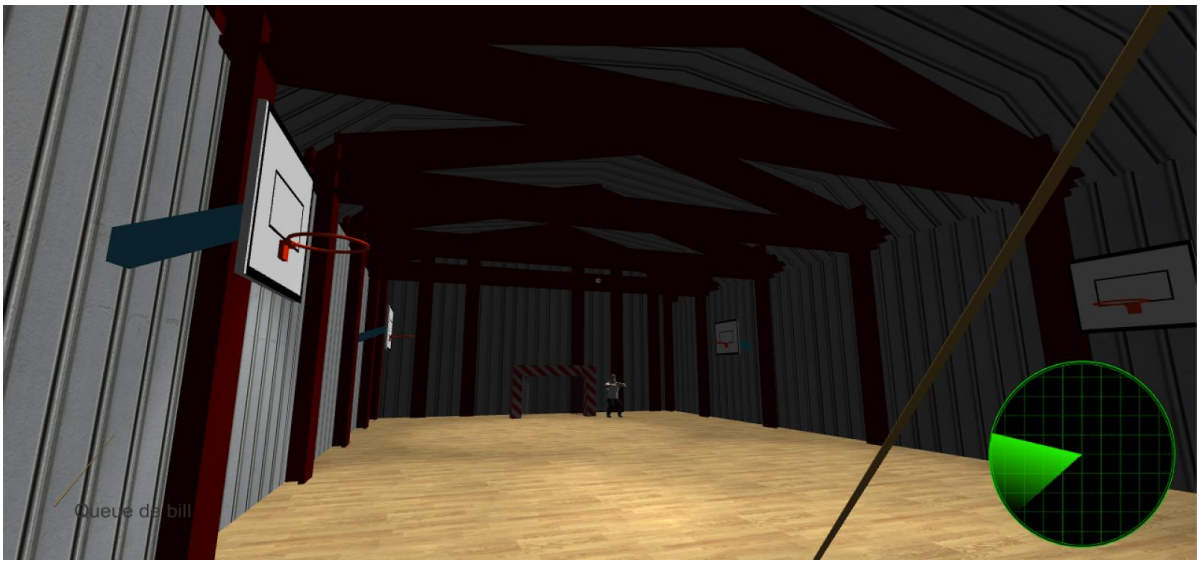


Figure 6 : Intérieur du gymnase

Chaque scène contient différents objets et personnages qui permettent au joueur d'avancer dans le jeu. Certaines scènes ne sont pas accessibles tout de suite : le gymnase par exemple n'est accessible qu'en en ayant la clé.

Le problème avec la présence de différentes scènes est que la transition entre deux scènes n'est pas aisée. En effet, tout objet présent dans une scène est détruit lors du passage à une autre scène. Le chargement d'une scène efface donc la précédente.

Heureusement, Unity permet de dire à certains objets de ne pas être détruits ; il faut cependant bien savoir lesquels, pour ne pas par exemple avoir deux joueurs dans la même scène, ou aucune interface graphique dans une autre scène.

Nous avons choisi de ne créer qu'un seul objet qui ne serait pas détruit lors d'un changement de scène. A cet objet, on greffe tous les autres qu'on doit conserver ; il sert donc de container pour tous les objets conservés d'une scène à l'autre. On compte parmi eux le joueur, les éléments d'affichage graphique, le calcul de l'heure actuelle dans une scène...

Le passage entre scènes est géré par une State Machine, qui utilise une arborescence d'états comme montré dans la figure 7.

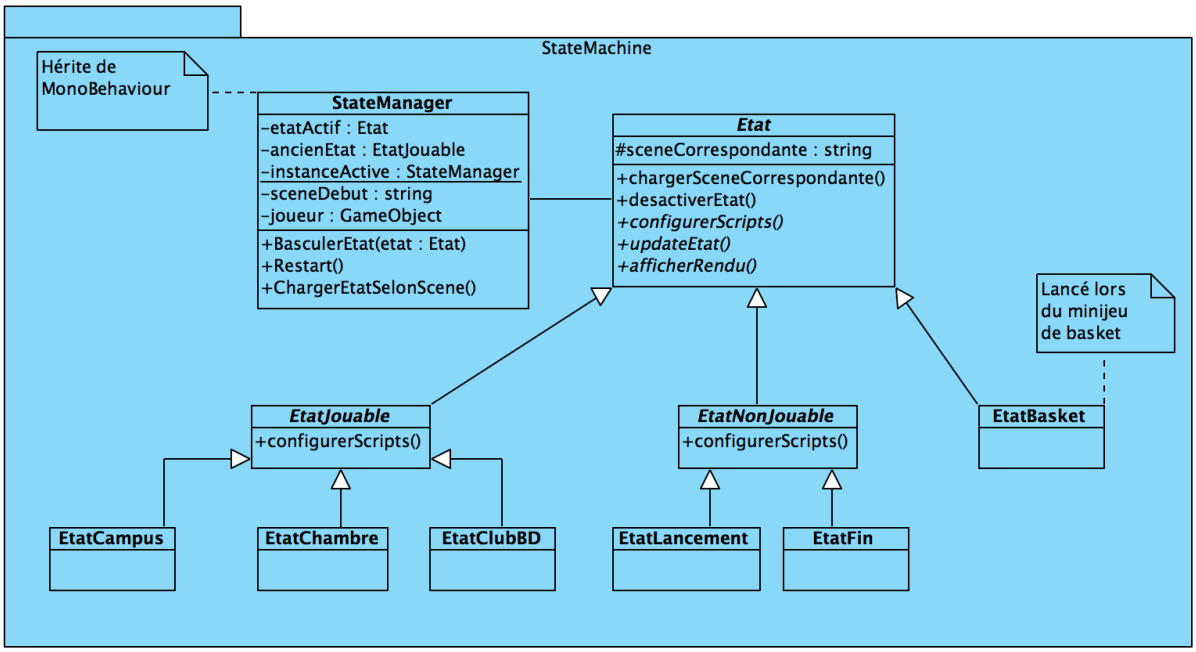


Figure 7 : La state machine

L'intérêt des scènes est de ne pas calculer tout le campus à chaque génération de scène. Il est inutile de générer l'amphi 2 lorsqu'on se trouve dans une chambre sur le campus. Utiliser différentes scènes est donc obligatoire pour utiliser convenablement Unity.

On a donc choisi de séparer le jeu en plusieurs scènes : le campus, le laboratoire LIRIS, chacune des chambres, le foyer, le bâtiment de la scolarité, le bureau du directeur, l'amphi 2. Chacune de ces scènes a donc ses propres spécificités et est générée individuellement.

3) Blender

Blender est un logiciel de modélisation, d'animation et de rendu 3D open source. Il est donc complètement gratuit et modifiable à l'envie, ce pourquoi nous l'avons choisi. Pour plus d'informations sur son utilisation, la documentation explique comment réaliser des objets et les importer. Unity est prévu pour importer des fichiers .blend contenant des objets réalisés sous Blender, mais aussi des materials (la couleur de l'objet sur lequel on peut appliquer des textures) et des UV map (le patron de l'objet déplié pour savoir comment poser les textures dessus). La grande majorité des objets présents dans le jeu (bâtiments, armes, objets à ramasser etc…) a été modélisée

sous Blender et importée. Les PNJ tels que les zombies ou le WEIman ont été importés depuis l'Asset Store de Unity gratuitement.

Le campus est fait sous Blender. Tous les bâtiments ne sont pas modélisés : ceux qui l'ont été sont les résidences Adoma et Comparat, le W1 et le W1bis. Seul l'extérieur de ces bâtiments est modélisé : nous n'avons pas eu le temps et les ressources nécessaires pour en faire plus. En revanche, les chambres, l'amphi 2 et le gymnase ont un intérieur modélisé et texturé. Certains bâtiments non modélisés sont nécessaires au scénario : pour y accéder, nous avons créé des portails que le joueur traverse. Si le projet est repris par la suite, il sera possible avec plus de temps de faire en sorte que tous les bâtiments soient modélisés pour se passer de cet artifice.

Une autre fonctionnalité de Blender est de réaliser puis importer des animations. Cela a été fait notamment pour les animations du combat contre la boule ou les flexions du professeur de sport. Unity incorpore un animator pour lier les animations entre elles et former une séquence, régler leur vitesse ou les exécuter à l'envers. Les figures 8, 9 et 10 présentent respectivement des captures d'écran des bâtiments Comparat et Adoma, de l'intérieur de la scolarité et de l'amphi 2 modélisé sous Blender puis importés dans Unity.

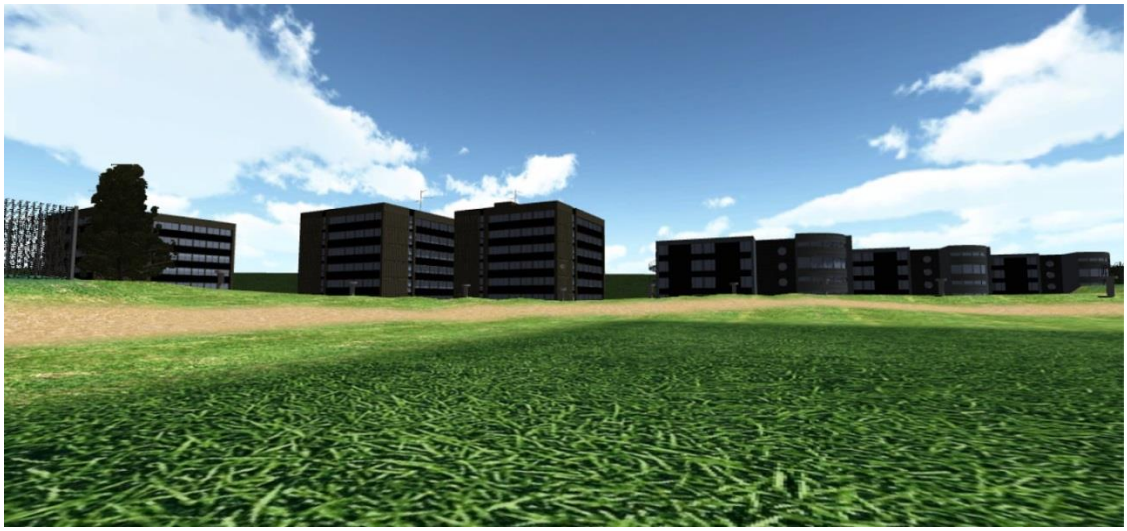


Figure 8 : Les bâtiments Comparat et Adoma faits sous Blender et importés dans le jeu

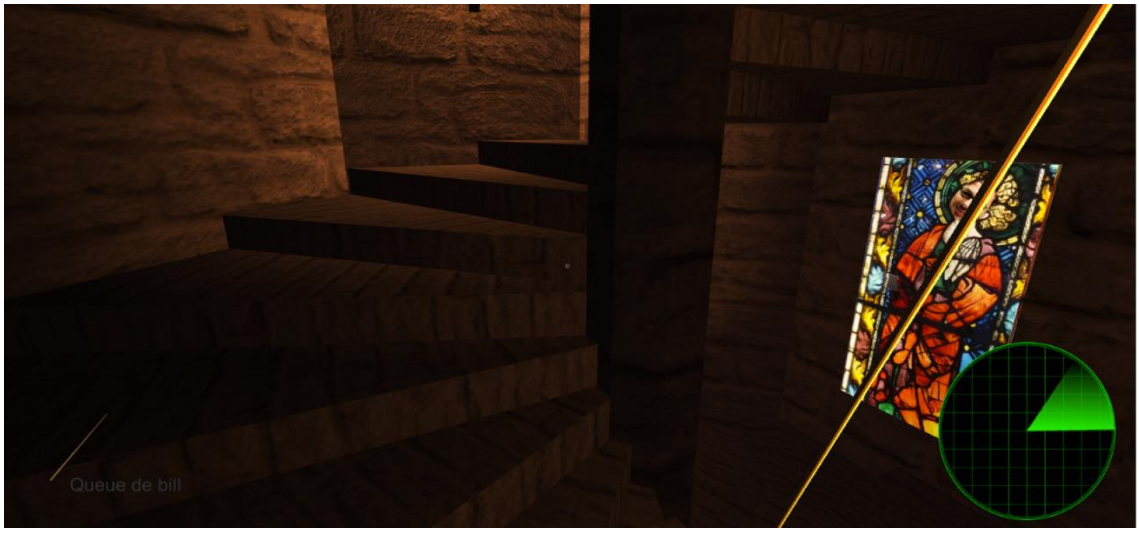


Figure 9 : Intérieur de la scolarité

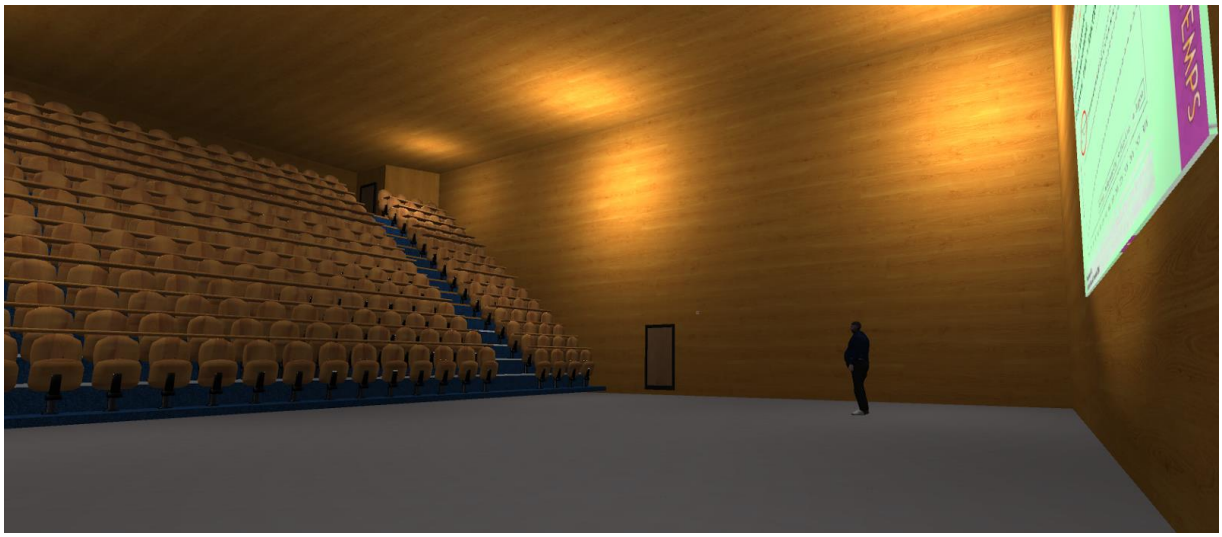


Figure 10 : Intérieur de l'amphi 2

4) Makehuman et FaceGen Modeller

Afin de modéliser les personnages utilisés dans le jeu, plusieurs logiciels ont été utilisés : les maillages des corps ainsi que les vêtements ont été réalisés avec Makehuman, les quelques visages l'ont été avec FaceGen Modeller et le tout a été assemblé sous Blender.

Makehuman permet de générer des maillages de personnages réglables. Ce logiciel possède également la particularité d'avoir un module d'interface spécial avec Blender, ce qui permet une très bonne compatibilité entre les deux logiciels. Pour pouvoir modéliser un maillage, Makehuman se base sur un personnage de base. De nombreux paramètres permettent ensuite de modifier l'aspect général du maillage, ce qui permet de réaliser la plupart des modèles souhaités. De plus, Makehuman permet de choisir des vêtements parmi une petite base de données qui s'ajustent alors automatiquement au maillage du corps (quelques retouches sur Blender sont cependant toujours nécessaires, comme on peut le voir sur la figure 11).

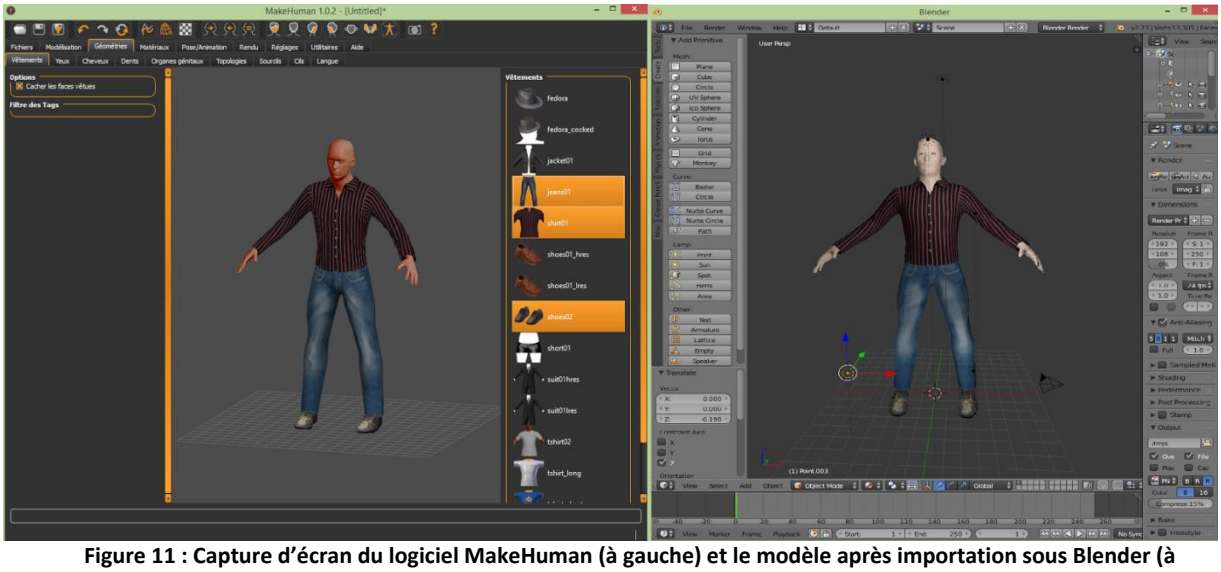


Figure 11 : Capture d'écran du logiciel MakeHuman (à gauche) et le modèle après importation sous Blender (à droite)

Makehuman possède également de nombreux paramètres permettant de modéliser les visages. Cependant, il est compliqué de les utiliser afin de représenter le visage d'une personne réelle. C'est pour cela que nous nous sommes penchés vers le logiciel FaceGen Modeller.

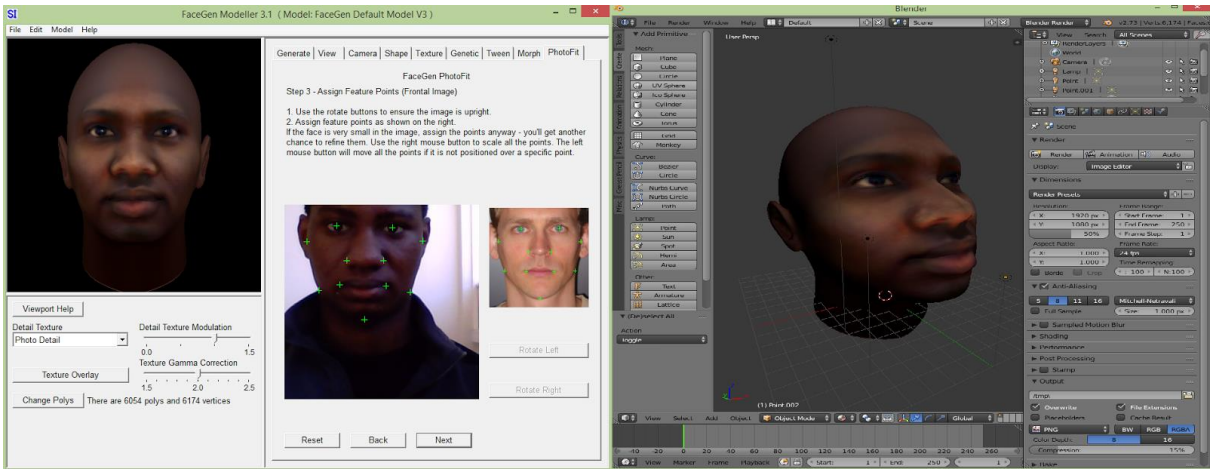


Figure 12 : Capture de l'interface de FaceGen Modeller (à gauche) et du modèle une fois importé sous Blender (à Droite)

En effet, ce logiciel permet, à partir de trois photographies (une de face et deux de profil), d'obtenir une modélisation plus ou moins réaliste (en fonction du modèle et de la qualité des photos) de la tête d'un sujet. La figure 12 présente l'interface du logiciel FaceGen Modeller : on peut voir la zone de "photofit" (qui permet d'ajuster les marqueurs du logiciel sur les photos du visage à modéliser) ainsi que le modèle créé à partir des photos. Cependant, la version gratuite du logiciel reste limitée, et il est assez complexe d'appareiller les têtes sur les corps modélisés sous Makehuman avec Blender. Nous avons donc décidé de ne pas y recourir systématiquement pour les visages, et de nous contenter des visages paramétriques fournis par Makehuman pour une grande partie des personnages du jeu.

Une fois les corps et les visages terminés, il faut encore modéliser les cheveux et les accessoires (comme par exemple des lunettes) et les ajouter au modèle. Nous avons pour cela utilisé le logiciel Blender. Il existe deux grandes techniques de modélisation des cheveux : soit chaque cheveu est modélisé, ce qui implique de très nombreux points sur le mesh et donc une baisse importante des performances du jeu; soit les cheveux sont modélisés par une surface représentant le cuir chevelu sur laquelle une texture imitant les cheveux est plaquée. C'est cette solution (beaucoup plus performante) que nous avons retenue pour ECLementary 2. Un exemple de résultat est donné par la figure 13.

Figure 13 : Capture d'un personnage totalement modélisé (corps, tête et cheveux)

5) Qt

Le framework Qt, basé sur le langage C++, permet d'avoir simplement une interface graphique. Nous l'avons utilisé dans un programme annexe du jeu, pour faciliter la gestion des dialogues. Ce programme offre une interface graphique simple pour créer des dialogues, les modifier et les enregistrer. La figure 14 présente une capture d'écran de cet outil.

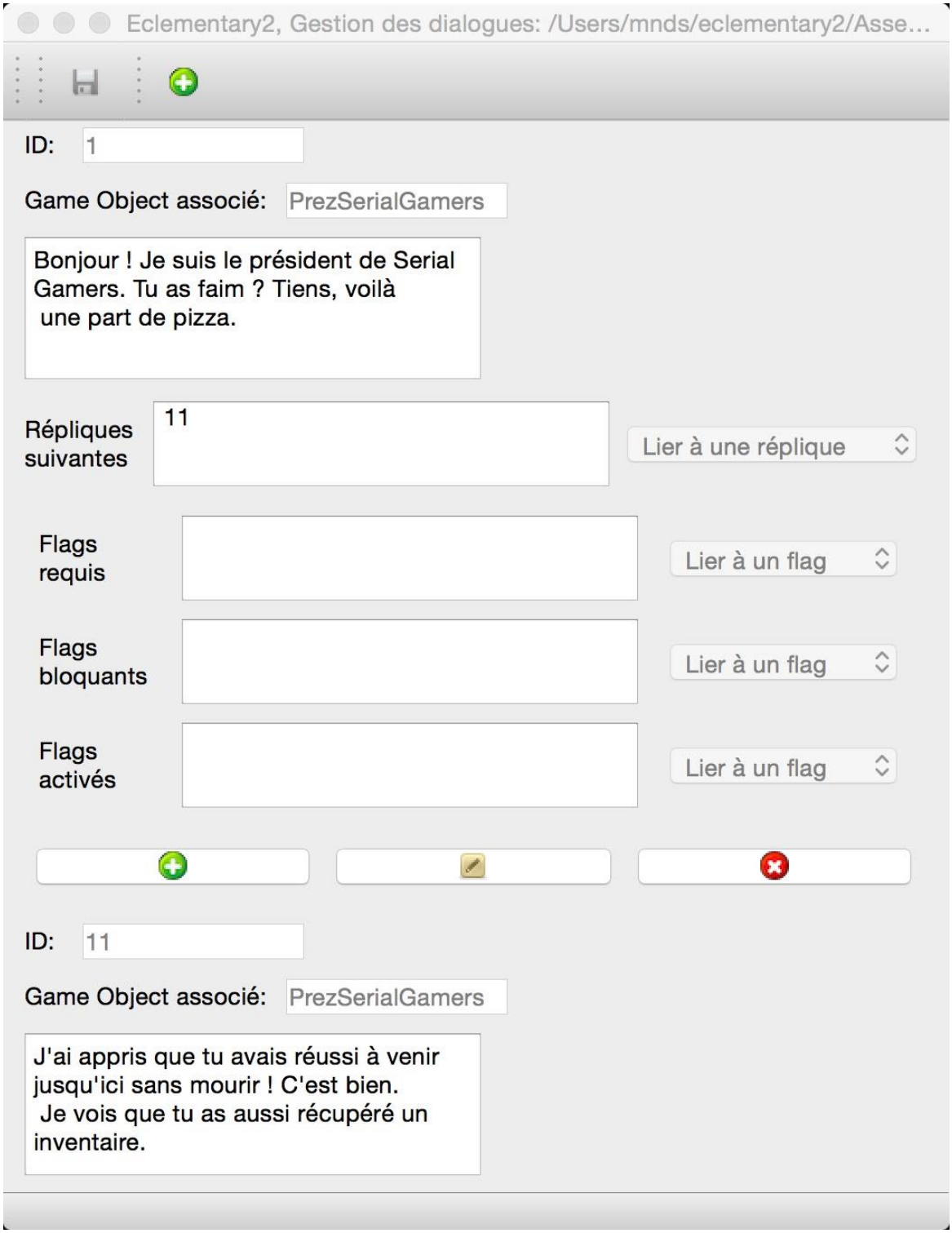


Figure 14 : Interface graphique permettant de modifier un dialogue

6) Architecture du code

Le code du jeu est organisé en fonctionnalités. En effet, les scripts liés au dialogue, aux évènements, à la santé, aux états, aux messages, et aux flags sont organisés différemment. Cela est dû en partie à l'utilisation de Unity, dont les scripts héritent pour la plupart de la classe *MonoBehaviour*. En effet, tout script devant être attaché à un objet Unity (GameObject) doit hériter de cette classe. Ainsi, il pourra être directement accessible à partir du GameObject associé.

Prenons l'exemple de deux fonctionnalités: la gestion des états et celle des dialogues joueur-PNJ. Les états décrivent une hiérarchie de classe, héritant du script *Etat*, et forment une "State machine" avec le script *StateManager*. Le patron de conception StateMachine est utilisé ici afin de pouvoir gérer les changements d'états (et donc de scènes du jeu), à partir d'une seule instance du StateManager, dont la durée de vie est égale à celle d'une partie. Dans ce système, seul le stateManager est directement rattaché à un objet Unity, et hérite donc de MonoBehaviour.

Les dialogues, eux, réalisent l'interface *Interactif* (nécessaire à tout objet avec lequel le joueur peut interagir) et implémentent le design pattern du composite, un *Dialogue* étant composé de *Replique*. Le script Dialogue hérite aussi de *MonoBehaviour* (il s'attache directement aux personnages). Les détails de la conception de ces deux fonctionnalités sont donnés dans les sections qui leur sont réservées.

Le scénario est divisé en nombreuses étapes pour savoir quoi faire à quel moment. Une clé peut n'apparaître qu'à la fin d'une certaine quête, quand un personnage peut disparaître après avoir été tué par un zombie. Pour gérer toutes les branches du scénario, on introduit un système de "flags". Un flag correspond à une action du scénario que le joueur a accomplie. Selon quels flags sont activés ou non, des scènes, personnages, armes, objets sont accessibles ou pas. La gestion des flags est assurée par un script (*FlagManager.cs*), qui vérifie quels flags peuvent être activés ou non, et qui à chaque activation de flag modifie la scène dans laquelle se trouve le joueur en conséquence. Par exemple, si un personnage vient d'être tué par un zombie, il ne va plus jamais apparaître.

Les différents modules sont donc :

- Dialogue : Permet d'ajouter, de supprimer, de modifier les répliques associées aux différents personnages présents dans le jeu.
- Zombies : Modélise les monstres que le joueur affronte.
- Caractéristiques : L'une des caractéristiques fondamentales du type de jeu que nous avons choisi est de permettre aux joueurs comme aux ennemis d'acquérir de l'expérience ; plus un personnage a d'expérience, plus il est puissant. Ce module gère cette progression.
- Objets : Contrôle tous les objets du jeu, qu'ils soient armes, décor ou magie. Ce module gère également l'inventaire du joueur.
- Informations contextuelles : S'occupe de gérer tout ce qui s'affichera à l'écran : radar, inventaire, caractéristiques du personnage, dialogues, messages d'information… Ce module est donc fortement lié aux autres.
- Personnage principal : Permet au personnage contrôlé par l'utilisateur de marcher, sauter, courir...
- Scénario : Lie tous les modules entre eux selon un scénario prédéfini. Ce module gère l'interaction entre les différents niveaux du jeu, les "flags" présentés précédemment, ou encore les messages d'information qui aideront l'utilisateur à savoir où aller s'il est perdu.

Le module Scénario est donc celui qui lie tous les autres modules entre eux. En effet, les dialogues doivent s'appliquer à des moments précis de l'histoire présentée dans le jeu ; les objets doivent être disponibles à certains moments du jeu pour accomplir certaines quêtes...

7) Méthodologie de test

Les tests des différentes fonctionnalités du jeu ont été réalisés en deux parties. Une fois qu'une partie du code était terminée, une personne du projet (différente de l'auteur du code en question) était chargée de la tester et de remonter au responsable de cette partie les éventuels bugs. Une correction était alors apportée, suivie d'une nouvelle phase de tests; et ainsi de suite jusqu'à la validation définitive du code en question. Cette méthode nous permet ainsi d'éviter qu'un mauvais code se propage et persiste pendant trop longtemps dans la structure du jeu.

Dès que nous avons eu une première version "jouable" du jeu, une phase de bêta test a été lancée en demandant à plusieurs Centraliens volontaires de tester le jeu. Un formulaire permettant de nous faire parvenir avec précision les descriptions des bugs rencontrés et leurs différentes remarques sur le jeu a également été fourni aux testeurs. Ces retours nous ont ainsi permis d'avoir un regard extérieur sur le projet et les différentes mécaniques du jeu. Les bugs signalés ont également été corrigés dans le jeu.

III - Mécanismes du jeu

1) Personnages et ennemis

Personnage principal

1. Contrôles

Le jeu se joue au clavier. Les touches choisies sont :

- Avancer : Z
- Reculer : S
- Aller à gauche : Q
- Aller à droite : D
- Courir : maintenir Shift gauche
- Ouvrir/fermer l'inventaire : I
- Ouvrir/fermer l'interface des caractéristiques : K
- Changer d'arme : Molette de la souris ou B/N
- Utiliser l'objet équipé : Clic gauche
- Lancer : Clic droit
- Interagir : E
- Bouger la camera : souris
- Lire un message : L
- Sauter : Espace
- Pause : Echap

Unity a des fonctions permettant de savoir quelles touches sont appuyées par l'utilisateur. Il suffit ensuite de récupérer les touches ci-dessus qui sont pressées et de faire les actions qui en découlent. Les mouvements du personnage et de la caméra sont gérées par un script attaché au GameObject représentant le joueur. A l'aide du composant de Unity "CharacterControler", qu'on attache au joueur, on peut le faire se déplacer en utilisant le moteur physique de Unity. Il est possible de déplacer un objet qui possède ce composant dans une direction donnée à une certaine vitesse par simple appel à une fonction de Unity.

2. Armes

Les modèles des armes sont modélisés sous Blender et importés sous Unity. Les armes sont ensuite placées devant le joueur, pour lui montrer quelles armes il tient en main. Le joueur ne peut en revanche porter qu'une arme à la fois. Les armes ont plusieurs actions possibles : certaines permettent de frapper devant soi en infligeant des dégâts aux ennemis à portée, certaines permettent de se soigner, quand d'autres servent à tirer des boules de feu.

La plupart des armes peuvent être jetées : le joueur peut les ramasser de nouveau. Les armes au sol peuvent de manière générale être ramassées par le joueur : il suffit d'interagir avec lorsqu'il en est proche. Les armes peuvent être changées en utilisant la molette de la souris ou les touches B et N. L'arme courante ne peut pas être changée pendant qu'un coup est donné.

3. Expérience

Le jeu avait pour but d'être un RPG ; un système de progression du personnage a été pensé. A chaque fois qu'un ennemi est vaincu, le joueur se voit donner des points d'expérience. En fonction du nombre de points d'expérience qu'il possède, on associe au joueur un niveau. Plus ce dernier est élevé, plus les caractéristiques du joueur sont élevées. Celles-ci sont :

- Ses points de vie : plus le joueur en a, plus il pourra survivre longtemps aux assauts des zombies.
- Ses points de mana : chaque attaque magique draine des points de mana. Un sort ne peut pas être lancé si le joueur n'a pas assez de mana.
- Son attaque : c'est un bonus qui est octroyé à chaque coup avec une arme. Plus le bonus est élevé, plus chaque attaque est puissante.
- Son attaque magique : pareil que pour l'attaque, mais pour les attaques magiques.
- Sa défense : c'est une réduction des dégâts subis par le joueur. Plus la défense est élevée, moins les coups qu'il subit sont douloureux.

4. Magies

Le jeu incorpore également un concept assez basique de magie. Il se manifeste sous la forme de deux armes utilisables, le parchemin de feu et le parchemin de foudre, et une jauge supplémentaire dans les caractéristiques du joueur, le mana. Lancer un sort à l'aide d'un parchemin consomme du mana et celui-ci ne remonte pas à moins de boire de la Kroca Cola. Le joueur dispose actuellement de deux sorts par parchemin (4 au total), tous étant très forts et pouvant terrasser les zombies en une fois. L'idée initiale était d'implanter la magie comme une conséquence de l'allégeance à un laboratoire. Selon le laboratoire choisi le joueur débloquait un seul et unique parchemin utilisable et maniant une magie similaire au champ d'étude du laboratoire : foudre pour le laboratoire Ampère, eau pour le laboratoire de mécanique des fluides etc… Malheureusement le seul laboratoire implanté dans le jeu est le LIRIS, cette fonctionnalité a donc été retirée et les parchemins cachés dans le jeu.

Personnages non jouables

Les Personnages Non Jouables, ou PNJ, sont des personnages dans le jeu distincts du joueur. Nous en avons distingué 2 types :

1. Les "alliés"

Ce sont les PNJs servant d'indicateurs au joueur, comme le WEIMAN et le GarsDuFoyer. Ils sont placés dans des endroits spécifiques du campus, normalement ils ne se déplacent pas. Ils interagissent avec le joueur en lançant des dialogues, et ils donnent des informations importantes au joueur, pour que ce dernier puisse avancer dans le jeu.

2. Les Ennemis

Ce sont des zombies qui rôdent dans le campus, ils peuvent attaquer le joueur. Etant donné que l'on peut trouver des modèles de zombies gratuits sur Internet, nous n'en avons pas modélisé et avons directement pris 2 modèles sur Asset Store. Les figures 15 et 16 présentent les 2 modèles de zombie utilisés dans le jeu.

Figure 15 : Zombie de type 1

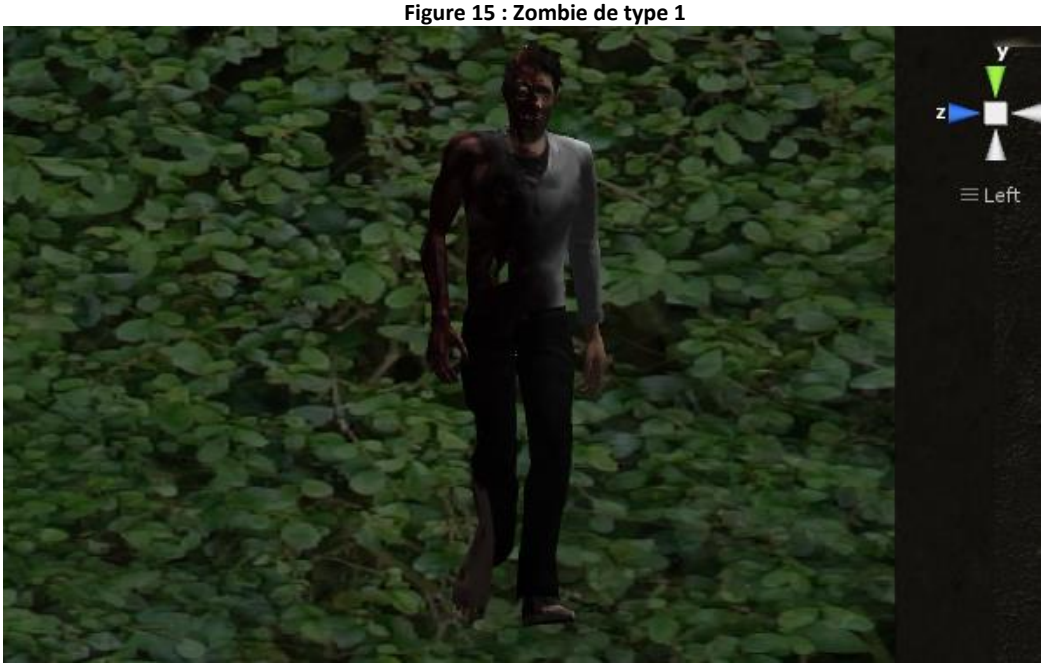


Figure 16 : Zombie de type 2

Sur le campus, nous avons placé des points de création de zombies où ces derniers apparaissent. Ils sont indiqués par une flèche rouge sur la figure 17.

Figure 17 : Points d'apparition des zombies marqués par des flèches rouges

Le nombre de zombies dans le campus change en fonction de la difficulté du jeu. Chaque fois qu'un zombie est tué, un nouveau réapparaît, de sorte à garder son nombre total.

Comportements détaillés des zombies:

Dans notre jeu, les zombies sont dotés des fonctionnalités suivantes :

1) Différencier leurs mouvements selon leur état. En effet, pour les zombies ordinaires qui rôdent se déplacent de manière aléatoire sur le campus tant que le joueur ne s'approche pas d'eux. Cependant, une fois que le joueur s'approche à une certaine distance d'eux, ils repèrent ce dernier et le poursuivent. Et quand ils rattrapent le joueur ils lancent une attaque.
2) Lancer des attaques. En mode d'attaque, les zombies lancent des attaques qui diminuent les points de vie du joueur.
3) Subir des attaques. Inversement, quand le joueur les attaque avec ses armes, ils peuvent perdre des points de vie, jusqu'à ce que ceux-ci deviennent 0 et dans ce cas-là les zombies meurent.
4) Différencier leurs propriétés en fonction de leur niveau. Les propriétés (Points d'attaque, de défense, et points de vie) des zombies sont différentes selon leur niveau. Nous avons fixé et introduit le niveau d'un zombie spécifique, et on a introduit 2 sortes de zombies de niveaux différents qui ont également des apparences différentes.
5) S'animer. En effet, tous les comportements du zombie présentés ci-dessus sont accompagnés d'une certaine animation du zombie. Quand le zombie se déplace sur le campus par exemple, il représente une animation de marche normale; une fois qu'il détecte le joueur, il se met tout de suite à courir pour le suivre; il possède aussi des animations d'attaque et de mort. La synchronisation entre les animations et les états des zombies est réalisée par une state machine qui est implémenté par le composant « animator » dans unity.

6) Processus de « Pathfinding » des zombies. Nous nous sommes servis d'un paquetage d'animation gratuit sur Unity. Pour certaines animations des figures (l'animation d'attaque par exemple), nous n'avons pas trouvé des réalisations appropriées sur l'Asset Store de Unity, et nous avons dû en créer une.

Quand le zombie détecte et suit le joueur, il essaie de trouver le plus court chemin disponible vers lui. Pour ce faire nous avons appliqué l'algorithme A* qui est très pertinent pour les recherches de chemin. En l'occurrence nous avons directement mis en œuvre une solution fournie par Unity, c'est-à-dire le composant « Navigation » qui permet de créer une carte du terrain expliquant quelles zones sont accessibles aux zombies et lesquelles ne le sont pas.

2) Dialogue et scénario

Dialogue

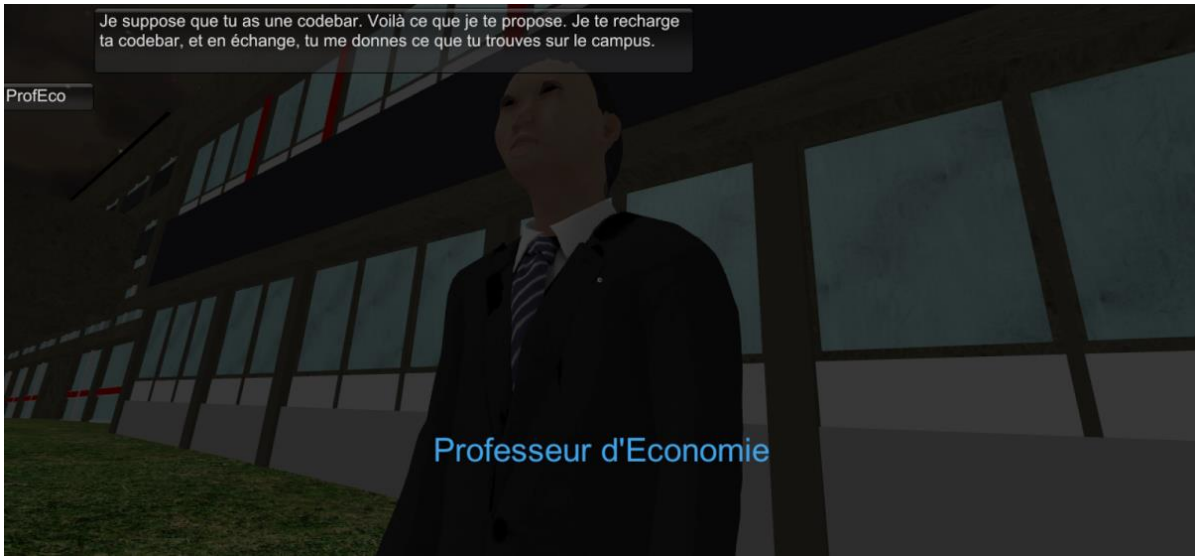


Figure 18 : Exemple de dialogue avec un PNJ

Afin de faciliter l'ajout, la suppression et la modification de dialogues, nous avons utilisé le format JSON pour stocker les répliques des dialogues. Le parser SimpleJSON nous permet d'analyser et de lire/écrire dans les fichiers au format JSON.

C'est donc la valeur des attributs des objets répliques qui est stockée dans les fichiers JSON. Cependant, écrire un dialogue peut vite devenir un travail fastidieux, vu le nombre de répliques qu'il peut y avoir et les attributs à modifier. Une erreur dans l'écriture d'un dialogue peut provoquer un bug lors de l'exécution du dialogue. Il est donc devenu évident pour nous de créer un outil de gestion de dialogues, écrit en C++, et qui à l'aide d'une interface Qt, facilite grandement l'écriture de dialogues. Les figures 18 et 20 présentent un exemple de dialogue avec le professeur d'économie et un autre avec le WEIman. La figure 19 récapitule la structure du système de dialogue du jeu.

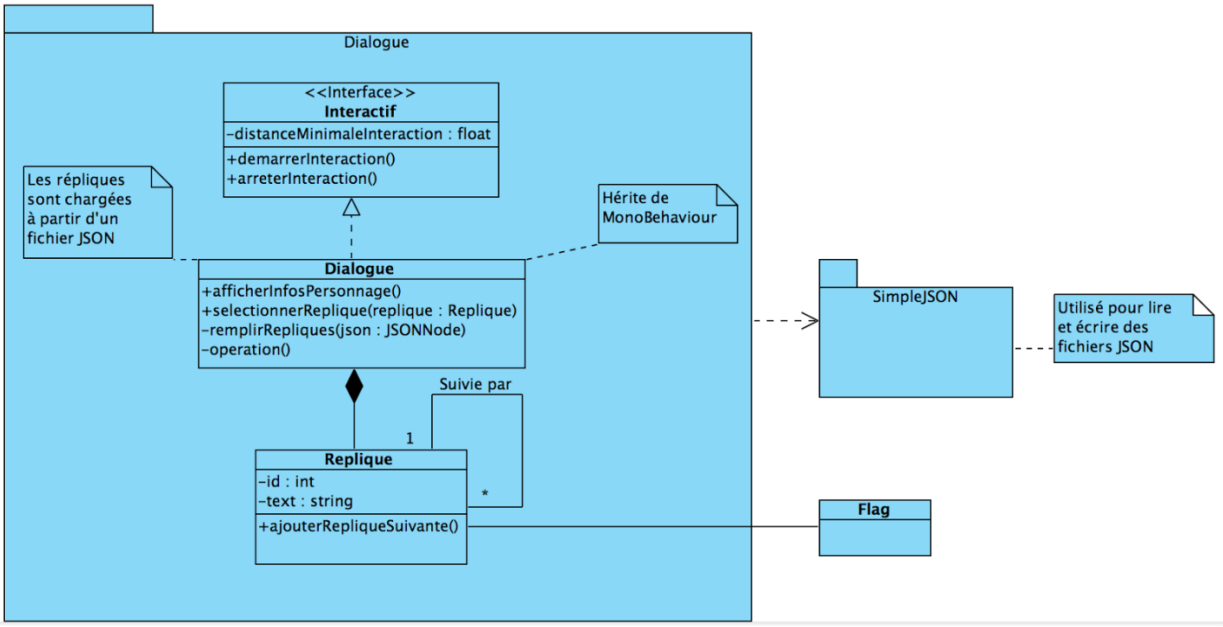


Figure 19 : Système de dialogues

Figure 20 : Dialogue à choix multiple

Scénario

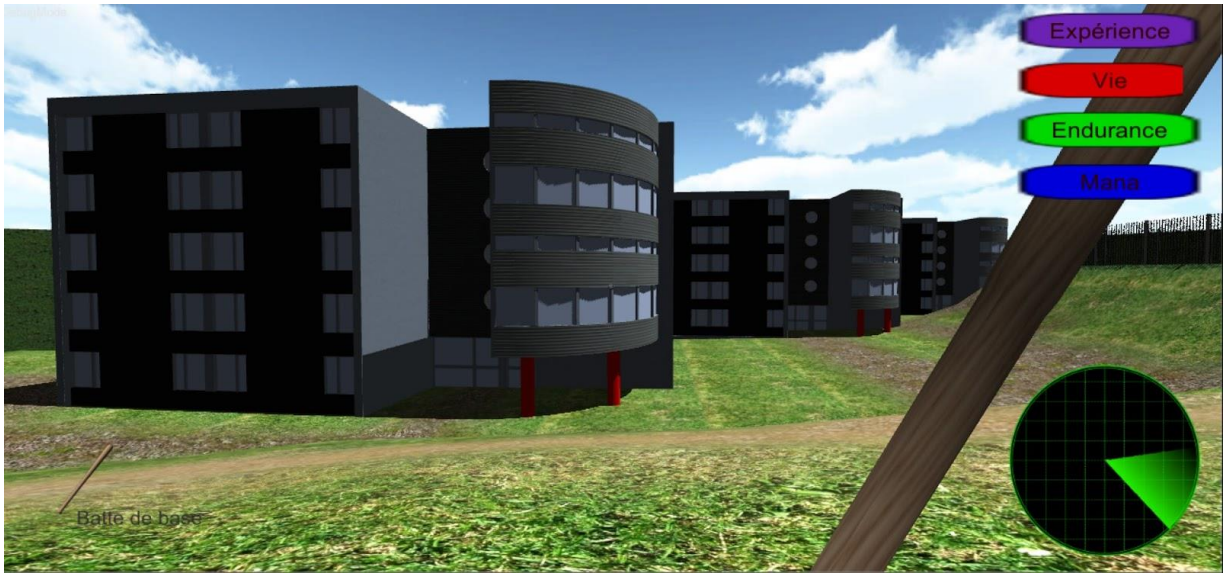


Figure 21 : Bâtiments Adoma

Un aspect important dans un jeu vidéo est le scénario. Nous avons déterminé assez tôt dans le projet une ébauche de scénario à laquelle nous nous sommes tenus. Le scénario devait permettre de rester dans le cadre du jeu que nous avions imaginé : un jeu qui se déroulerait sur le campus de l'Ecole Centrale de Lyon, en vue à la première personne, et qui permettrait au joueur d'accomplir des quêtes. Pour avoir un peu plus d'intérêt de la part du joueur, nous avons décidé que le joueur serait confronté à une invasion de zombies sur le campus, l'obligeant à agir pour se sauver.

Le jeu commence donc par le jour du week-end d'intégration. Après celui-ci, le joueur se retrouve au milieu du campus et rentre chez lui. La figure 21 présente le bâtiment d'Adoma dans lequel le joueur doit aller. La figure 22 présente la chambre dans laquelle le joueur arrive lorsqu'il rentre dans ce bâtiment.

Figure 22 : Intérieur d'une chambre

En se levant, il découvre un campus désert. En s'approchant du foyer, il rencontre un autre personnage qui lui explique que des créatures étranges ont envahi le campus ; à lui de le sauver ! Ce personnage donne la clé de sa chambre au joueur, où celui-ci apprend qu'il devrait aller parler aux

professeurs du laboratoire LIRIS. Après les avoir aidé à retrouver des composants électroniques sur le campus, le joueur se voit doté de nouvelles compétences : il peut transporter des objets et accéder à son inventaire. Fort de ces nouvelles aptitudes, le joueur peut aller converser avec d'autres Centraliens au foyer, où il se verra confier de multiples quêtes pour libérer Centrale. Il va par exemple devoir libérer le gymnase de l'influence d'un professeur infecté comme on peut le voir sur la figure 23, visiter le bâtiment de la scolarité de la figure 24, ou survivre dans un labyrinthe pour sauver le directeur.

Figure 23 : Combat contre le professeur de sport

Par souci de temps, seule une trame principale a été faite pour ce projet. En revanche, le système de scénario, de progression du personnage et de dialogue est fait de telle sorte qu'il est facile d'ajouter de nouvelles quêtes. De ce fait, si l'intrigue est très linéaire dans la version présentée, un scénario plus étoffé est faisable en reprenant le projet.

Figure 24 : Sommet du bâtiment de la scolarité

Certains choix lors de ce même scénario n'ont pas été implantés. A un moment, le joueur est censé pouvoir choisir avec quel laboratoire de Centrale il ira combattre les zombies. Or, seul le laboratoire LIRIS existe sur la version présentée ; le joueur n'a donc pas le choix. De même, le professeur d'économie se trouvant devant le W1 (représenté sur la figure 25) devait proposer divers

articles au joueur, qui devait aussi posséder de l'argent. Tout cet aspect achat et vente d'objets n'a pas été fait mais peut l'être pour rallonger la durée de vie du jeu.

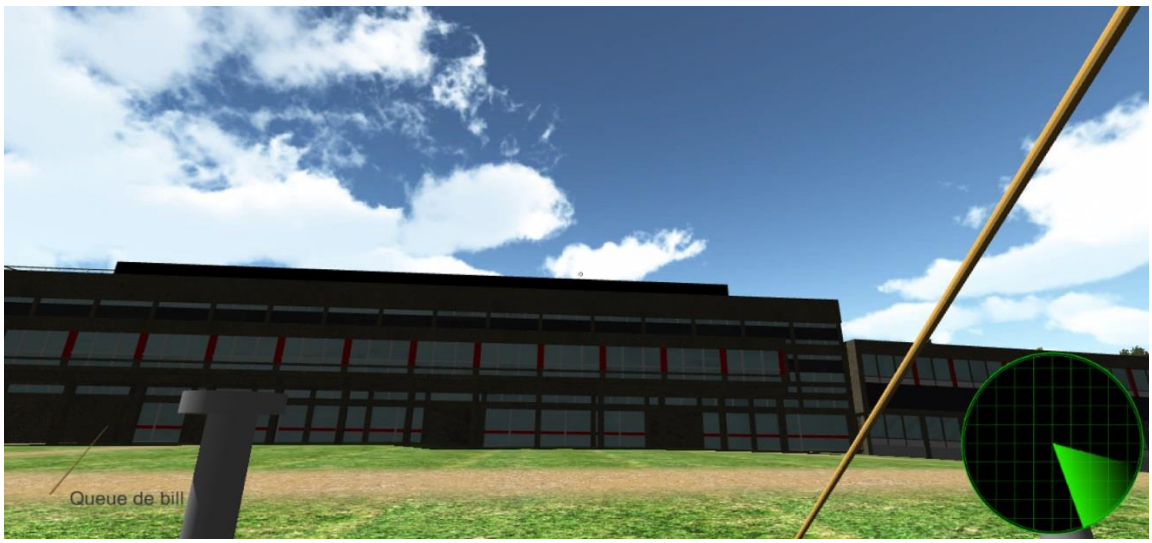


Figure 25 : Bâtiments W1 & W1bis

Sauvegarde

Le joueur peut à tout moment dans le jeu sauvegarder sa progression en se rendant dans sa chambre, pour la reprendre plus tard. Plusieurs éléments doivent donc pouvoir être enregistrés pour pouvoir reprendre le jeu dans l'état où il a été quitté. Parmi ceux-là nous pouvons citer le nombre de points de vie du joueur, le nom de l'état, les armes dans l'inventaire, les états des flags, etc...

Pour cela, le script *GameData.cs* a été créé pour regrouper ces données en son sein. Lors de la sauvegarde et du chargement d'une partie, c'est une instance de cette classe qui est sérialisée (dans le fichier sauvegarde.bin), respectivement désérialisée. Après le chargement du *gameData*, les données récupérées sont ensuite « rendues » à leurs différents propriétaires.

Interface graphique

Afin de pouvoir communiquer certaines informations du jeu, il a été nécessaire d'implémenter plusieurs interfaces fournissant des renseignements au joueur. La figure 26 montre ces différents affichages formant l'"ATH" (affichage tête haute).

Il y a tout d'abord quatre barres représentant les différentes caractéristiques du joueur (expérience, points de vie, endurance et mana). Ces barres sont présentes en permanence (dans le coin en haut à droite de l'écran) à l'exception des phases de dialogue avec les PNJ ainsi que lors de la mise en pause du jeu.

La case en bas à gauche de l'écran est également activée en même temps que ces barres, et permet au joueur de savoir l'objet qui est actuellement actif (objet d'attaque ou consommable). Les deux carrés centraux que l'on voit (toujours sur la figure 26) représentent les interfaces d'inventaire et de caractéristiques, que le joueur peut respectivement ouvrir avec les touches I et K. La première permet au joueur de visualiser ce qu'il possède sur lui et la seconde reprend les données des quatre barres et lui permet d'avoir accès à ses autres caractéristiques (niveau du joueur, attaque, défense …).

Et enfin, on peut trouver en bas à droite un radar permettant au joueur de détecter dans un rayon variable les ennemis (points rouges) et les PNJ (points jaunes). Ce radar se met à jour à chaque image du jeu, et permet donc une détection en temps réel afin de ne pas être surpris.

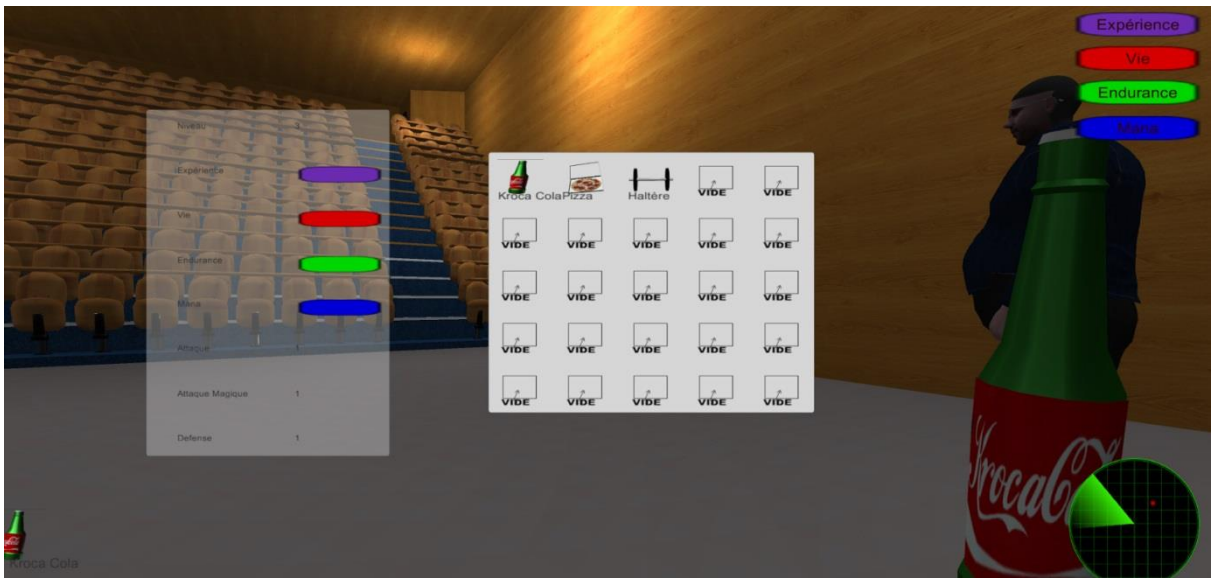


Figure 26 : Illustration des différentes fenêtres de l'ATH

Ecran Pause

Un écran pause est activé lorsque l'on appuie sur la touche Echap. Il permet au joueur soit de revenir au jeu soit de quitter le jeu. La figure 27 montre une capture d'écran de l'écran de pause.

Figure 27 : Écran de pause

Quelques variations sur l'écran pause: Quand on rentre dans cet état, les dialogues ne s'affichent plus (S'ils sont déjà activés avant), le volume de la musique est baissé, et les mouvements du joueur sont bloqués.

3) Musiques et textures

Les musiques et textures constituent des points essentiels d'un jeu vidéo, malheureusement nous ne disposions dans notre équipe d'aucun membre capable de réaliser des textures de qualité ou de composer des musiques, il a donc fallu les prendre sur Internet.

Musiques

Le fait que le jeu soit soumis à une licence nous empêche d'utiliser des musiques soumises à des droits d'exploitation. Mais pendant le développement le campus était désespérément silencieux ; c'est pourquoi nous avons décidé d'incorporer différentes musiques soumises à des droits d'auteur dont la liste détaillée se trouve dans la bibliographie. Bien entendue nous ne respectons pas le droit d'utilisation de ces musiques, un critère important à prendre en compte si le projet est repris. Ces musiques ont été choisies pour instaurer une ambiance et se lancent donc dès l'arrivée du joueur dans une scène. Il y a au minimum une musique par scène et certaines (notamment la scène campus extérieur qui constitue le cœur du jeu) en possèdent plusieurs pour former une playlist choisissant le morceau de manière aléatoire.

Certaines musiques se déclenchent également lorsqu'un certain flag, c'est à dire un certain événement de l'histoire, est activé. C'est le cas du boss de fin : la musique se lance après le dialogue au début du combat afin de rendre l'expérience du joueur plus dramatique et épique.

Textures

Une texture est une image appliquée sur un objet et qui lui donne sa couleur et son apparence. Un exemple de texture est présenté sur la figure 28 avec le ballon de basket du mini-jeu.

Figure 28 : Présentation du jeu de basket

Les textures utilisées dans le jeu sont toutes issues d'Internet et majoritairement de CGTextures.com (cf bibliographie pour une liste exhaustive). Le grand avantage de ce site est qu'il propose des photos d'excellente qualité gratuitement et libres de droit (même utilisables dans un contexte commercial). L'incorporation des textures dans le jeu consistait majoritairement à les retoucher un peu avec des logiciels de traitement d'images tels que GIMP ou Paint.NET. En effet, de nombreuses textures servent pour les décors et doivent être répétées un grand nombre de fois sur la surface de l'objet. Le problème est que l'on observe souvent des problèmes de raccord à la bordure de la même texture dupliquée côte à côte. La solution consiste à appliquer un filtre Seamless de GIMP à toutes ces textures avant de les importer sous Unity.

Les logiciels étaient aussi utilisés pour modifier la teinte, la saturation ou l'intensité d'une texture afin de la rendre plus belle et plus adaptée à la scène. Mais la véritable utilité de GIMP réside dans la possibilité d'obtenir des normal map d'une texture.

Pour faire simple, là où une texture est une simple image de base appliquée sur un objet en 3 dimensions pour lui donner de la couleur, la normal map permet de simuler un relief à la surface d'un objet pourtant représenté par un simple plan. Ainsi, l'infographiste désirant modéliser un mur de briques dans un jeu vidéo ne le réalisera pas brique par brique. A la place, il prendra un plan sur lequel il appliquera une texture "diffuse" (l'image de base) puis une normal map afin de simuler le relief à la surface du mur pourtant plat.

Pour calculer l'éclairement en un point d'une surface, Unity fait le produit scalaire entre le rayon lumineux incident et la normale de la surface. Une normal map permet tout simplement de modifier artificiellement l'orientation de cette normale sans modifier la structure de l'objet.

Unity incorpore un Material de type Bumped Diffuse prenant en entrée une texture diffuse et une normal map, nous l'avons donc utilisé sur une bonne partie de nos matériaux. Il existe également 2 autres maps utilisés par les professionnels du jeu vidéo : la displacement map et la specular map. Du fait de leur complexité et de leur gourmandise en matière de performances, nous n'y avons pas touché mais leur incorporation reste possible sous Unity. La figure 29 présente la différence entre une texture et sa normal map.

Figure 29 : A gauche, texture "diffuse" ; à droite, la normal map associée

Conclusion

Les objectifs du projet étaient de modéliser le campus de l'Ecole Centrale de Lyon en 3D et d'y implémenter les éléments basiques d'un jeu vidéo de type jeu de rôle. Le logiciel produit lors de ce projet permet à un utilisateur de jouer dans une partie du campus et d'y accomplir une unique trame pendant environ une heure, posant ainsi bien les bases d'un jeu vidéo comme nous l'avions souhaité. Les résidences étudiantes, le foyer, le W1 et le W1bis sont entièrement modélisés, sans compter des chambres, plusieurs objets transportables et l'amphithéâtre 2 de l'Ecole Centrale de Lyon. Le joueur peut combattre différents types d'ennemis, utiliser des armes variées et maîtriser la magie tout en parcourant un scénario faisant de lui le sauveur du campus. L'utilisateur voit son personnage progresser et communiquer avec d'autres personnages et se voit doté d'un ATH contenant le radar, l'inventaire et les caractéristiques du joueur.

Le projet est donc la base d'un jeu de rôle basé sur le campus de l'Ecole Centrale de Lyon. Il peut être continué de deux façons : soit en étendant la zone de jeu à d'autres écoles du Campus Lyon Ouest – l'EMLyon étant initialement prévue, soit en ajoutant de nouvelles quêtes, de nouveaux bâtiments et d'autres objets. L'implémentation de la technologie de l'Oculus Rift peut également être améliorée (actuellement, seule la scène du labyrinthe est compatible). Néanmoins l'objectif principal de ce projet – la réalisation d'un jeu basé sur le campus en 3D – a été respecté et nous espérons que ce projet sera repris l'an prochain et amélioré.

Annexes

Bibliographie

« Unity - Game Engine ». Consulté le 17 septembre 2014. http://unity3d.com/

« CGTextures ». Consulté le 07 octobre 2014. http://www.cgtextures.com/

« Good Textures ». Consulté le 07 octobre 2014. http://www.goodtextures.com/

« Unity – 18 free substances ». Consulté le 23 octobre 2014. https://www.assetstore.unity3d.com/en/#!/content/1352

« blender.org - Home of the Blender project - Free and Open 3D Creation Software ». Consulté le 17 septembre 2014. http://www.blender.org/.

« Doxygen Manual: Doxywizard usage ». Consulté le 17 septembre 2014. http://www.stack.nl/~dimitri/doxygen/manual/doxywizard_usage.html.

« Graphviz | Graphviz - Graph Visualization Software ». Consulté le 31 mars 2015. http://www.graphviz.org/.

« ▶ Unity - How to Change Terrain Height In-Game (C#) - YouTube ». Consulté le 27 septembre 2014. http://www.youtube.com/watch?v=1j9McAMK4lE.

« A* Pathfinding Project ». Consulté le 2 octobre 2014. http://arongranberg.com/astar/features.

« How can I perform some action based on the terrain texture currently under my player? - Unity Answers ». Consulté le 4 novembre 2014. http://answers.unity3d.com/questions/14998/how-can-i-perform-some-action-based-on-the-terrain.html.

« Modifying terrain height under a gameobject at runtime - Unity Answers ». Consulté le 27 septembre 2014. http://answers.unity3d.com/questions/11093/modifying-terrain-height-under-a-gameobject-at-run.html.

« quill18creates - YouTube ». Consulté le 3 octobre 2014. https://www.youtube.com/channel/UCPXOQq7PWh5OdCwEO60Y8jQ.

« Quill18's Unity 3d & Blender Programming Tutorials ». Consulté le 2 octobre 2014. http://quill18.com/unity_tutorials/.

« ▶ Amnesia: The Dark Descent OST - Ambience 12 - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=TaDZmP6f0-M&index=13&list=PL19BE6702DBB45367.

« ▶ Bach: Toccata and Fugue in D minor, BWV 565 - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=IVJD3dL4diY.

« ▶ C'est pas sorcier - Musique Générique Complète - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=VrsOZKYrVdA.

« ▶ Davy Jones Music Box music (NO ORGANS) - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=hGzvkySJKbM.

« ▶ Final Destination - Super Smash Bros. Brawl - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=PLJuT8zPmvA.

« ▶ Irish tavern music - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=SBATrLRWySg.

« ▶ League of Legends - Thresh Theme - Music - Extended HD - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=hvKGHEpdR9Y.

« ▶ Pasodoble clasico taurino - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=jPE33jwbTy0.

« ▶ Professor Layton and the Curious Village: Puzzle theme extended - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=L6AMdFYY_A4.

« ▶ Rocky theme song - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=DhlPAj38rHc.

« ▶ Scary horror music!! - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=TUWi9ytJhDc.

« ▶ The Fellowship of the Ring Soundtrack-02-Concerning Hobbits - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=_pGaz_qN0cw.

« ▶ Survivor - Eye Of The Tiger - YouTube ». Consulté le 24 mars 2015. https://www.youtube.com/watch?v=btPJPFnesV4.

« Makehuman | Open source tool for making 3D characters ». Consulté le 20 février 2015. http://www.makehuman.org/.

« Animator 动画器 [圣典百科] ». Consulté le 10 octobre 2015. . http://wiki.ceeger.com/script:unityengine:classes:animator:animator.

« 【Unity3d 小程序篮球发射游戏源码适合初学者】-unity 项目源码-u3d 游戏开发者社区【游戏蛮牛】unity3d 官网 ». Consulté le 25 septembre 2015. http://www.unitymanual.com/thread-3176-1-1.html.

« Unity3D 手机游戏开发 ». Consulté le 25 septembre 2015. . http://www.douban.com/note/293772040/.

« Stack Overflow ». Consulté le 13 février 2015. http://stackoverflow.com.

« Débutez dans la 3D avec Blender ». Consulté le 12 janvier 2015. . http://openclassrooms.com/courses/debutez-dans-la-3d-avec-blender.

« Main Page - BlenderWiki ». Consulté le 12 janvier 2015. http://wiki.blender.org/.

« Unity-UI ». Consulté le 12 mars 2015. http://unity3d.com/learn/tutorials/modules/beginner/ui.

« Makehuman | Open source tool for making 3D characters ». Consulté le 20 février 2015. http://www.makehuman.org/.

Documentation

Note préliminaire

Ce document est destiné aux personnes qui souhaitent reprendre le projet. Il n'a pas pour but d'expliquer le fonctionnement de Unity mais de dire comment réutiliser les scripts fournis par le projet. Il faut voir ce document comme un tutoriel permettant d'utiliser le code du projet, et non pas comme une description technique de la méthodologie employée pour résoudre les problèmes auxquels nous avons fait face. Pour une documentation du code en lui-même, regarder la documentation fournie par Doxygen dans Documentation/Documentation Doxygen/html/index.html. Le code du projet est disponible à l'adresse https://github.com/mnds/eclementary2/.

OBJETS

De manière générale, le mieux est de copier un objet qui marche et de le modifier un peu. La plupart des objets sont constitués d'un gameObject et d'une partie graphique ; cette technique permet de modifier plus rapidement l'aspect graphique d'un objet.

Créer un objet de l'inventaire

How to :

Importer de Blender un mesh de l'objet (exemple : une clé). Créer un GameObject vide qui portera le nom de l'objet (exemple : CléCampus) et mettre le mesh importé de Blender comme enfant de ce GameObject vide. Mettre les coordonnées de l'enfant à (0,0,0).

Si cet objet est un objet qui permet d'attaquer, placer le GameObject vide comme enfant de la MainCamera. Copier le Transform d'une des armes déjà présentes et en coller les valeurs sur ce GameObject vide.

Maintenant, sans toucher au GameObject vide (le script d'attaque fait translater ce GameObject vide entre deux positions de l'espace précises), bouger les coordonnées de Graphique pour qu'il soit placé convenablement par rapport au joueur (adapter avec la vue Scène). Une fois qu'il est bien mis, ajouter un BoxCollider au GameObject vide et le placer convenablement.

Ensuite, attacher le script ContactArmeScript sur l'objet et cliquer sur le bouton dans l'inspecteur. Le cas échéant, bien remettre le collider s'il est décalé.

Le collider sur l'objet sera toujours orienté selon les axes globaux ; le boxcollider sera peut-être mal orienté. Ne pas en tenir compte, faire au mieux.

Changer au besoin les scripts attachés sur cet objet "arme" (Lancer, Attaquer, Soigner...).

Faire un prefab de cet objet dans Armes (pas en scène).

Prendre ce prefab, le remettre dans la vue, mettre les rotations du Graphique à 0. Mettre le script ObjetEnSceneScript sur le GameObject vide, le lancer dans l'inspecteur (cliquer sur le bouton). Remettre bien le collider. Ajouter le script GlowSimple à Graphique. Faire de cet objet un autre prefab destiné à être placé en jeu. Pour le placer, changer cette fois la rotation du GameObject vide, pas de Graphique.

Les ajouter ensuite dans le script Inventaire de Joueur comme les autres armes déjà placées.

Pourquoi :

L'inventaire est géré dans le script Inventaire, qui est mis sur le GameObject Joueur. Celui-ci prend la liste de tous les objets qui peuvent être mis dans l'inventaire, de façon à pouvoir les activer et les désactiver selon l'arme équipée. C'est aussi ce script qui sait combien de munitions il reste pour chaque objet. Il faut donc dire à Inventaire quels sont les objets dont il doit gérer l'activation et la désactivation. Attention à bien mettre dans cette liste les armes de la vue Scene et non les prefabs des armes.

Les objets, eux, sont attachés à la MainCamera. De cette façon, les objets restent toujours affichés au même endroit ; les objets sont toujours "devant" le personnage.

Les objets ont ensuite plusieurs comportements possibles, cumulables (sauf Attaque et Soin) :

→ Attaquer : lorsqu'on fait clic gauche, l'arme se balance de droite à gauche, infligeant des dégâts, puis revient à sa position initiale. Ce balancement se fait entre deux points de l'espace précis, définis dans le script éponyme, d'où l'importance de bien placer l'arme au début avec les scripts présentés ci-dessus.

→ Lancer : l'arme est lancée devant le joueur après un clic droit. Elle quitte son inventaire, et peut être ramassée après avoir été lancée (sauf dans certains cas). L'arme lancée est un clone d'un prefab différent (voir paragraphe suivant).

→ Soigner : l'arme soigne le joueur après un clic gauche.

→ Attaque magique : clics gauche et droit font certains effets différents.

Caractéristiques de l'objet prefab

How to :

Cet objet servira aux instanciations. Il doit donc avoir au moins le script Pickable activé. Le script Attaquer doit voir son bypass à true, sinon un objet placé sur la scène ou une de ses instances pourraient réagir à l'appui sur la touche d'attaque. Son collider ne doit pas être sur IsTrigger car l'objet doit être tangible.

Ces objets sont prêts à être lancés ; il est intéressant de bien faire attention à ce que le champ indiquant si l'objet doit disparaitre après un certain temps soit bien renseigné.

Pour le créer, on importe son graphique dans Unity. On le place dans Graphique. Puis, on ajoute le script ContactPrefabScript qui organise directement tous les scripts. Il ne reste qu'à faire bouger le graphique de façon à avoir bien un collider et un graphique qui vont ensemble.

Pourquoi :

Cet objet est utilisé de deux façons : dans les scènes, où le joueur interagit avec ; et ce prefab est instancié quand le joueur veut jeter un objet de son inventaire. Dans le premier cas, l'objet est en scène et peut être ou pas ramassé. Dans le deuxième cas, pendant un certain temps, l'objet est en l'air, puis ensuite retombe en sol.

Lorsqu'un objet est dans l'air, on considère que cet objet ne peut pas être ramassé. Son comportement est régi par ObjetLance. C'est ici qu'on dit si l'objet doit être détruit en touchant le sol (imaginons une grenade, elle ne doit pas rester sur le sol, elle doit exploser).

Lorsqu'un objet est sur le sol, il doit avoir le script Pickable. Ce script dit que l'objet peut effectivement être ramassé par le joueur. Le test est fait dans le script Inventaire : lorsqu'on appuie sur E, on vérifie si l'objet devant nous a un script Pickable, et on le ramasse si oui.

ATTENTION : ces objets ne doivent pas avoir les scripts Attaquer, Lancer ou Soigner !

Caractéristiques de l'objet attaché à la caméra

Les objets associés à l'inventaire sont attachés à la caméra pour toujours les avoir en face de soi. La caméra à laquelle ils sont associés n'a pas d'importance car les différentes caméras possibles

(Oculus ou non) sont au même point de l'espace. On les met donc sur la MainCamera par commodité.

Les objets sur la caméra doivent avoir un Rigidbody Kinematic (pour être déplacés par Attaquer) et ne pas subir la gravité (sinon ils tombent au sol). Le script Attaquer ne doit pas avoir son bypass à true (sinon impossible d'attaquer), et avoir ses champs remplis (placements initial et final). Il ne doit pas être en train d'attaquer. Les scripts Lancer et Soigner ne doivent pas être en bypass, et lancer doit avoir un objet réel (le prefab) et un point de départ. Pickable doit être en bypass pour éviter de ramasser l'objet qu'on a déjà dans les mains. De même, l'objet n'est pas lancé donc le bypass de Objet Lance est à true.

Cette fois, le centre de l'objet a une réelle importance. En effet, tous les objets attaquent de la même manière : du côté haut droit de l'écran au côté bas gauche. On fait donc bouger le gameObject par script entre deux points codés directement dans le script Attaquer. Il ne reste plus qu'à mettre bien le Graphique pour qu'il suive le gameObject tout le long. Le script ContactArmePrefab, une fois actualisé dans le gameObject, fait une partie du travail. C'est à l'utilisateur de faire bien attention à ce que le collider et le renderer aillent bien ensemble. Regarder les exemples déjà présents.

Caractéristiques d'un objet sur le sol

Les objets sur le sol ne doivent (sauf cas particulier) pas être détruits après un certain temps. Par précaution, il faut désactiver ou supprimer les scripts Attaquer, ObjetLance et Lancer pour qu'ils ne réagissent pas. Selon si l'objet doit ou non subir la gravité, on configure le rigidbody comme voulu. Pour qu'il ne bouge pas, on coche IsKinematic et on décoche UseGravity ; on fait exactement le contraire s'il doit subir la gravité.

Le script Pickable doit être activé si l'objet est fait pour être ramassé. Le bypass n'est pas nécessaire ; si l'objet ne peut pas être ramassé au début de la scène, le booléen IsPickableDebut se charge d'empêcher les interactions comme un bypass. Il ne faut pas oublier de configurer la distance de ramassage.

Il n'y a pas de script actuellement pour faire ces opérations.

ATTENTION : UN OBJET AVEC LE SCRIPT PICKABLE NE DOIT JAMAIS AVOIR DE PARENTS. EN EFFET, LORSQU'UN OBJET AVEC PICKABLE EST RECUPERE, SON PLUS HAUT PARENT EST DESACTIVE !

CONDITIONS DU JOUEUR

Récupération du joueur principal

Le gameObject JoueurPrincipal contient un script prévu spécialement à cet effet (SetJoueurPrincipal.cs). Ce script indique au ControlCenter (classe statique) que le gameObject est bien le joueur principal. On utilise donc dans n'importe quel script ControlCenter::GetJoueurPrincipal pour récupérer le joueur principal.

Animations d'un personnage

Les animations sont placées dans un Animator. A l'aide du module Mecanim de Unity, il est possible de relier différentes animations créées sous par exemple Blender par script. L'Animator est une interface graphique qui permet de dire comment passer d'une animation à une autre à l'aide de variables accessibles par des méthodes de la classe de Unity Animator.

Caractéristiques d'un joueur

Les caractéristiques d'un objet correspondent aux valeurs Attaque, Défense, Mana, Points de vie (PV)... Elles symbolisent la partie RPG du projet.

Les différentes caractéristiques et statuts d'un personnage sont accessibles dans le script Caracteristiques.cs. Les conditions sont explicitées dans ce même script dans une liste.

Formules de dégâts

La formule de dégâts se trouve dans le script ControlCenter. Cette formule diffère selon si on veut prendre en compte les caractéristiques de l'attaquant et/ou de l'objet qui subit les dégâts.

Elles sont utilisées dans les scripts suivants :

- Zombie_comportement
- Attaquer
- ObjetLance

SCENARIO

Le scénario se trouve dans le fichier Scénario. Si vous ne trouvez pas le fichier, ça n'a pas d'importance, tout le scénario est déjà incorporé dans le jeu.

Stockage du nom des scènes

Le nom des scènes est très important car c'est avec ce nom qu'on peut changer de scène (Application.LoadLevel(nomDeLaScene)). Il faut donc les changer le moins possible. Si toutefois c'est nécessaire, il faut également le changer dans ControlCenter. En effet, ControlCenter permet d'accéder au nom réel d'une scène à partir de sa description.

Exemple : si je veux savoir le nom de la scène qui représente le campus, il me suffit de faire ControlCenter.nomDeLaSceneDuCampus.

Ecriture des dialogues

Le personnage principal est représenté dans les dialogues sous le nom "Joueur". Prendre garde à ce que les autres personnages du jeu soient nommés dans les dialogues par le nom du GameObject correspondant dans la scène.

Dans les répliques des dialogues, prendre en compte le fait que le saut à la ligne ne se fait pas automatiquement dans les GUIText utilisés. Ainsi, il faut bien penser dans les dialogues à mettre des sauts à la ligne (« \n ») tous les 70 caractères à peu près de façon à être sûr que les dialogues soient bien lisibles. Prendre d'autres dialogues déjà faits (comme ceux du PrezSerialGamers) comme référence.

Attention : le système de json oblige à écrire les accents de manière particulière (en unicode) !

Flags

Les flags sont une façon de savoir où le joueur se trouve dans le scénario. En effet, selon l'avancement de sa quête, le joueur ne pourra pas aller partout. Il faut donc pouvoir savoir si telle ou telle action a déjà été faite. Pour ça, on utilise le principe de flag. Lorsqu'une quête est finie, on active un flag qui signifie "la quête a été faite". Tous les flags sont déclarés dans le script FlagManager. Chaque flag a un identifiant unique pour le repérer, des prédécesseurs (on ne peut pas activer le flag "ouvrir la porte" si on n'a pas "obtenu les clés") et des bloquants. Un flag peut s'activer si aucun flag bloquant n'est activé, et si tous ses prédécesseurs sont activés.

Les flags peuvent s'activer grâce aux dialogues, en passant des triggers, en cliquant sur des objets... Chaque réplique d'un dialogue a un champ permettant de dire quels flags peuvent s'activer, et quels flags doivent être activés pour lire la réplique.

ActiverFlagOnTriggerEnter se met sur une trigger. Lorsque le joueur principal passe dedans, on essaie d'activer le flag correspondant (qu'on change dans l'inspecteur).

ActiverFlagInteractif hérite de l'interface interactive. De ce fait, quand le joueur va appuyer sur E devant un objet qui a ce script, le flag s'active.

ActivationSelonFlags est un script qui, attaché à un objet, le rend visible si certains flags sont activés et d'autres désactivés. L'actualisation des objets se fait à travers ControlCenter, qui contient la liste de tous les objets qui ont ce script (cette déclaration se fait dans ActivationSelonFlags::Start). ControlCenter reçoit l'information de FlagManager à chaque changement de flags ; il peut donc tester tous les objets potentiellement activables.

Sécurité pour les flags

Il y a plusieurs sécurités pour empêcher le joueur de sauter des étapes.
> Les répliques des dialogues sont conditionnées par les flags activés.
> ActivationSelonFlags active les objets qui portent ce script selon les flags activés.
> ChangementSceneFlags ne permet le changement de scène que si les flags sont bien activés.
> Les flags en eux-mêmes contiennent des indications sur ce qui doit ou pas être débloqué.

Tests de certains passages

Les flags précédents celui duquel on veut commencer doivent être à true (à cause de ActiverFlag qui regarde les prédécesseurs). Savoir quels flags activer est assez délicat. Si on veut accéder à une scène, il faut trouver dans la scène Campus par exemple où se trouve le portail vers cette scène. (ex : PortailLIRIS) Puis, on regarde le script ActivationSelonFlags et on met à true/false les flags qu'il faut. Ensuite, on regarde s'il y a un script de changement de scène (ChangementSceneFlagOnTrigger ou ChangementSceneFlagInteractif), et on met à true/false les flags associés. Ensuite, on va dans EcranTitre, on lance une partie et on va chercher le portail.

Changements de scènes

Le changement de scène peut se faire via une activation de flag (par événement par ActivationFlag::FaireActionsConnexes) ou par un autre script. Le problème vient du fait qu'une transition peut parfois déclencher des flags, et des fois non, comme lorsque le joueur revient dans sa chambre. On ne sait pas si les flags auront le temps de s'enclencher avant le changement de scène.

On utilise donc ChangementSceneFlag. On donne une liste d'id à activer avant de changer de scène. On active tous ces id avant le changement de scène. Dans ce cas, il ne faut pas mettre dans les événements de ces id un changement de scène, sinon il pourrait y avoir conflit.

Dans StateManager, on fait la liste des états correspondant à chacune des scènes pour simplifier le changement de scène. Chaque scène nouvelle et son état doivent être déclarés dans StateManager. Les noms des scènes sont écrits une seule fois dans ControlCenter : si le nom des scènes change, il suffit de le modifier dans ce script pour que ça se répercute partout. En revanche, les noms des scènes sont toujours appelés manuellement dans les scripts ChangementSceneFlag. Des erreurs peuvent être repérées par la recherche de scènes non existantes dans la console Debug.

Pour les transitions, le Joueur doit toujours aller à un SpawnPoint (comprendre : un GameObject avec juste le script SpawnPoint placé à un endroit particulier de la scène) particulier. Le nom de ce spawnPoint est très important parce que les scripts de transition vont toujours chercher le GameObject qui a le nom renseigné dans le champ de ChangementSceneFlagInteractif (par exemple).

Pour mieux comprendre, regardez un objet comme PortailLIRIS (dans la scène Campus). La scène dans laquelle on doit aller est évidente (le LIRIS, déclaré dans ControlCenter - voir après). On définit les flags selon ce qui a été fait dans FlagManager. Ensuite, on doit déclarer un SpawnPoint.

SpawnPoints

ControlCenter contient une string qui correspond au nom du GameObject sur lequel le joueur doit se téléporter à l'entrée d'une scène. Le Joueur va prendre la position et la rotation de ce GameObject. Ce dernier doit avoir le script SpawnPoint.

Le nom de ce spawnPoint est très important. C'est grâce à ce nom qu'on saura où se téléporter lors d'un changement de scène (voir plus haut).

Ajout d'une scène

Créer une nouvelle scène, lui donner un nom. Dans ControlCenter, sous la liste des noms de scènes, ajouter le nom de la scène et un alias, de façon à pouvoir changer le nom de la scène Unity dans un seul script au cas où. Désormais, pour accéder au nom de cette scène depuis n'importe où, faire ControlCenter.nomNouvelleScene, où nomNouvelleScene est le nom choisi.

Ajouter également dans ControlCenter un alias pour cette scène dans l'enum Scenes. Créer un état lié à cette scène (donc un nouveau script) : le joueur entrera dans un état différent à chaque scène. Prendre le même modèle que les états liés aux scènes déjà créées.

Il existe deux types d'états : les états jouables et les états non jouables, qui héritent respectivement des classes EtatJouable et EtatNonJouable. Une scène est associée à un état jouable lorsque le joueur peut jouer dedans (se déplacer, utiliser ses armes, etc.). Dans le cas contraire, la scène est liée à un état non jouable (exemple de l'écran de lancement, de l'écran de mort). La scène et l'état sont associés en mettant le nom de la scène comme valeur de la variable nomSceneCorrespondante de l'état. L'état jouable auquel le joueur a accès en premier est EtatDepart.

Pour les transitions, créer des SpawnPoints (GameObject vide => ajouter SpawnPoint). Les noms sont aussi importants. Pour aller vers une autre scène, utiliser un GameObject sur lequel est placé un script ChangementSceneFlagOnTrigger ou ChangementSceneFlagInteractif. Dire quels flags seraient activés en changeant de scène. ActivationPossibleDesFlags doit être de même taille que ListeDesFlagsPouvantEtreActives et ne doit avoir que des true. NomsDesScenesAdmissibles contient tous les couples scènes/spawnPoints qui peuvent être chargés par cette transition, selon les flags activés ou nom. Les spawnPoints sont référés par noms, d'où l'importance de bien les nommer au sein d'une scène. Le nom de la scène où aller est aussi écrit ici.

Les noms exacts des scènes peuvent être changés dans l'onglet Project, puis dans ControlCenter. Toutes les scènes sont déclarées dans ControlCenter.Scenes, puis utilisées dans les scripts permettant de changer de scènes. Il est quand même déconseillé de modifier le nom des scènes et surtout des spawnPoints qui eux ne sont pas stockés dans un script global.

Activation de gameObjects selon les flags activés

Certains PNJ par exemple ne peuvent exister dans une scène que si certains flags sont ou non activés. Le problème est de gérer cette activation ou non.

A chaque entrée de scène, on va donc chercher tous les objets qui pourraient devoir changer d'état activé/désactivé. Ceux-ci porteront un script nommé ActivationSelonFlags. De cette façon, même s'ils se désactivent, les scripts seront toujours accessibles grâce à cette recherche.

A chaque entrée de scène, celui-ci va récupérer tous les gameObjects qui ont ce script et les placer dans ControlCenter. Chaque ActivationSelonFlags va contacter ControlCenter et lui dire qu'il existe. ControlCenter aura donc toujours sur lui la liste de ActivationSelonFlags de la scène, qu'ils soient activés ou désactivés.

Dans FlagManager, on va dans ActiverFlag et avant chaque return (et après chaque changement d'état d'un flag), on demande à ControlCenter de regarder tous les objets de la scène qui peuvent changer d'état et de vérifier s'ils doivent être modifiés.

Notes sur les changements de scènes

Dans FlagManager sont écrits les emplacements de chaque activation de flags. Dans MazeWallManager on retrouve les transitions liées au labyrinthe.

Trouver le nom de la scène associé à un état

StateManager.changerEtatSelonScene fait le travail (prend le nom d'une scène telle qu'elle est déclarée dans ControlCenter.Scenes et renvoie le "vrai" nom de la scène sous forme de string.)

AFFICHAGE D'INFORMATIONS

Texte

Le script ActivationFlag propose d'afficher, via l'objet nommé Texte (un GUIText), du texte, selon que l'objet est activable (comprendre : que les flags sont aux bonnes valeurs pour permettre l'interaction avec l'objet) ou non. Comme son nom l'indique, ce script contient à la base le code permettant d'activer un flag en interagissant avec un objet. En revanche, il peut être détourné en affichage de texte en mettant comme valeur du flag '0', qui n'existe donc pas. Pour afficher du texte, utiliser AffichageTexteRaycast. Les tests d'affichage de texte se font dans le script "SetTexte".

Surbrillance

La surbrillance est obtenue en changeant le type de shader sur le renderer d'un objet. Le shader self-illumin donne un effet de surbrillance. Le script GlowSimple change simplement le shader lorsque l'objet est sélectionnable (c'est-à-dire que le script Inventaire reconnait l'objet comme "récupérable").

SAUVEGARDE

Changements selon flags

Premier cas : Activer/désactiver des objets selon les flags

On attache ActivionSelonFlags. On dit quels flags doivent être activés et désactivés. Le script va dire à ControlCenter que le gameObject associé contient ce script. De ce fait, FlagManager, à chaque changement de flags, demande à ControlCenter de vérifier tous les objets qui ont "ActivationSelonFlags". Pour ne pas tester des objets NULL, on dit aussi à ControlCenter dans quelles scènes sont les objets.

Deuxième cas : Objets attachés au Manager

Par exemple, les musiques d'ambiance. Celles-ci changent selon les scènes et les flags. On crée la classe ActionSelonFlags qui de la même façon dit à ControlCenter quels objets vont voir leur comportement modifié par les flags. A chaque changement de flags ces objets sont réévalués.

BOSS

Nous avions prévu deux boss : un prof de sport et le boss final.

Boss final

La première phase consiste en un changement de la taille du boss pour qu'il prenne des proportions gigantesques. Dans la deuxième phase, il poursuit le joueur. Celui-ci doit rester à distance sinon il meurt. Des dégâts d'écrasement sont simulés en mettant au niveau des pieds du modèle de boss des triggers, qui si elles sont déclenchées par un objet, tuent cet objet (en lui infligeant 10000 dégâts).

Le principal problème est de faire en sorte que les dalles infligent des dégâts au joueur. En effet, le joueur a un CharacterController : donc il n'actionne pas les événements OnTrigger/CollisionEnter. Il faut donc dans un nouveau script regarder constamment quelle est la texture sous le joueur et agir en conséquence. Il faut alors ajouter sur le joueur, à ses pieds, un nouveau trigger. Celui-ci "appuiera" sur les cases quand le joueur marchera dessus.

Comme le boss a déjà ces triggers pour l'écrasement, les cases du sol n'auront qu'à implémenter OnTriggerEnter pour infliger des dégâts.

JOUEUR

Déplacement du joueur

Le déplacement du joueur se fait grâce au script FPCClassic. On attache un CharacterController sur le prefab du joueur. C'est ce Controller qu'on fait déplacer dans le script. FPCClassic permet de contrôler le mouvement et la souris. C'est aussi dans ce script qu'on bloque les mouvements si besoin.

Trigger externe

Le CharacterController n'est pas reconnu comme une trigger et n'engendre pas de collisions. Si on mettait sur le sol un script qui dit "si quelqu'un me touche, lui infliger des dégâts", le joueur ne serait pas détecté. De ce fait, on ajoute une trigger en enfant à Joueur pour qu'il puisse se faire détecter. Ce système sert surtout dans le dernier combat.

SonSol

Ce prefab contient juste une AudioSource localisée aux pieds du Joueur. Elle sert notamment lors de la mort du Joueur dans le labyrinthe.

TerrainSoundManager

Ce prefab contient une AudioSource qui permet, lorsque le joueur marche sur un certain sol, de jouer un son. Les textures et les clips audio sont mis dans deux listes qui doivent correspondre. Le script regarde tout le temps sur quelle texture le joueur se trouve et joue un clip audio s'il se déplace.

LABYRINTHE

Le labyrinthe suit son propre écosystème. Les monstres qui y sont ne peuvent pas être touchés par le joueur. La mort de ce dernier se déclenche lorsqu'un monstre s'approche à une certaine distance de lui. Contrairement aux autres morts, le joueur n'est pas expédié dans l'écran de mort ; il revient juste à son point de départ.

Le labyrinthe est généré aléatoirement à chaque chargement de scène. Des murs sont alignés selon une grille et tournés de façon aléatoire selon un angle proportionnel à 90°. On enlève ensuite les murs qui se seraient superposés dans le processus pour obtenir tout le labyrinthe.

Ensuite, à chaque « case » du labyrinthe se trouve une trigger. Chaque trigger est actionnée par le joueur lorsqu'il passe dans une case. Les monstres sont prévenus à chaque mouvement du personnage de sa position. Lors de la génération du labyrinthe, on fait par un système de raycast une reconnaissance de la carte du labyrinthe. De ce fait, les monstres savent où est le joueur, où ils sont, et la carte du labyrinthe.

Pour éviter que le jeu ne soit trop injuste, les zombies ne prendront pas le chemin le plus court vers le joueur. Ils se déplacent en effet selon un algorithme à essais successifs basique (ils cherchent un chemin, mais pas le meilleur). Les zombies peuvent donc avoir des mouvements incohérents, ce qui est voulu dans le cas de ce labyrinthe où seuls les monstres savent où se situent les murs et le joueur.

CONTROLE

_Controle est un prefab mis dans Manager. Manager n'est pas détruit lors des changements de scène, donc _Controle est commun à toutes les scènes et commence dans la scène EcranTitre.

Horloge

Contient un script qui contient l'heure actuelle. Celle-ci est remise à 13h30 lorsqu'on rentre dans la scène Campus.

DebugManager

Contient le script Debug. Il permet de récupérer les inputs faits par l'utilisateur et lui permet d'entrer des codes. Le plus intéressant est d'entrer la commande "/debug" (puis Entrée) pour activer le mode debug. Attention, toutes les touches comptent lorsqu'on entre une commande ; si ça ne marche pas, recommencer.

Le mode Debug est signifié par un texte en haut à gauche. Entrer un int permet de savoir si le flag associé est à true ou false. La touche M donne une vitesse de course surhumaine.

LecteurMusiques

Le script LireMusiques est celui qui gère les musiques d'ambiance des scènes. Le mettre sur un GameObject avec un audioSource. Un GameObject (LecteurMusique dans Manager>_Controle) porte ce script. Pour ajouter une musique, signifiez dans quelle scène elle doit se jouer, et quels scripts doivent être activés ou non. Mettre plusieurs musiques permet de les faire défiler aléatoirement.

REMARQUES PROPRES A UNITY

Bug dus à la désactivation d'objets

Faire attention lorsqu'on désactive des objets à l'état des scripts qui y sont attachés. Par exemple, lors de l'évaluation d'un dialogue, imaginons que la dernière réplique désactive le personnage en activant un flag. L'activation du flag se fait avant l'appel à ArreterInteraction. Donc, sans rien faire, le PNJ disparaîtrait juste avant de permettre au joueur de bouger de nouveau.

Pour résoudre ça, on utilise la fonction MonoBehaviour::OnDisable(). Celle-ci est appelée lorsqu'un objet est désactivé mais aussi lorsqu'il est détruit. OnDestroy() est appelé lorsqu'un objet est détruit uniquement.

Oculus

Pour mettre l'Oculus, il faut rajouter OVRCameraRig dans la scène, obtenue grâce aux scènes d'essai. Le dossier OVR doit être présent dans Assets, et les plugins qui y sont doivent être dans Assets>Plugins. La version DirectToRift marche si le jeu est compilé sous la version pro. ControlCenter contient un booléen qui dit si on souhaite activer ou non l'Oculus.

ATTENTION : Les fonctions OnGUI sont inutiles sous Oculus ; les boutons créés ne s'affichent pas. Il est donc impossible de par exemple lancer le jeu car les boutons au démarrage sont faits par OnGUI ! De même pour les dialogues, qui seront invisibles.

PETIT TUTORIEL POUR BLENDER (OBJETS, ANIMATIONS & EXPORTATIONS)

Ce petit guide a pour but d'expliquer comment créer de A à Z un objet sous Blender, comment lui ajouter des matériaux et comment l'importer sous Unity sans problème.

Tout d'abord il vous faut disposer d'une version fonctionnelle de Blender disponible à l'adresse suivante : http://www.blender.org/. Une fois installée et ouverte, vous arrivez devant une interface déstabilisante. Quelques commandes basiques pour naviguer :

- Molette dans la vue 3D : zoomer/dézoomer
- Clic sur la molette dans la vue 3D : rotation de la vue 3D
- Shift+molette dans la vue 3D : défilement horizontal de la vue 3D
- Ctrl+molette dans la vue 3D : défilement horizontal de la vue 3D
- Pavé num. 7 : met la vue au-dessus
- Pavé num. 1 : vue selon l'axe X
- Pavé num. 3 : vue selon l'axe Y
- Pavé num. 4/6 : rotation de la vue 3D horizontalement
- Pavé num 8/2 : rotation de la vue 3D verticalement
- Pavé num. 5 : perspective cavalière ou perspective conique
- Pavé num. 0 : vue via la caméra

Prenez le temps de vous familiariser avec ces commandes basiques. Créons maintenant un objet. Vous remarquerez un cube dans la scène de base, vous pouvez vous en servir comme base ou le supprimer avec Suppr ou X. Blender différencie le mode "objet" du mode "edit". Pour modifier la structure de votre objet (ou maillage, voire mesh parlons anglais que diable), il faut basculer en edit mode. Attention vous ne pouvez en modifier qu'un objet à la fois et c'est celui entouré d'un joli contour. Voilà une nouvelle floppée de commandes pour créer une forme de base et pouvoir la modifier basiquement :

- Shift+A : ajouter une forme de base. Fonctionne en edit mode comme en object mode, mais si vous êtes en edit mode la forme créée sera dans le même objet que vous modifiez. Vous noterez en bas à gauche de la fenêtre que vous pouvez personnaliser un peu votre objet (nombre de points pour un cercle, cercle fermé ou ouvert etc…), mais seulement au moment de sa création, si vous le bougez vous ne pourrez plus !
- Clic droit : Pour sélectionner quoique ce soit dans la vue 3D (Objet, point, face, caméra, etc…), il faut TOUJOURS faire un clic droit, pas un gauche.
- Clic gauche : Place le curseur 3D à l'endroit du clic. Le curseur est ce réticule rouge de la vue 3D. Quand vous ajoutez un objet, il arrive à la position du curseur. Généralement vous ne voulez pas qu'il bouge puisqu'il décale la position d'arrivée de vos objets.
- N : fait apparaître la barre des propriétés. Les 2 utilités de ce menu déroulant sont la possibilité de régler manuellement l'endroit du curseur (sous 3D cursor, généralement vous le ferez car vos objets apparaissent n'importe où) ou la position de la sélection actuelle (dans Transform, tout en haut). Une utilité bonus de ce menu est la possibilité d'importer une image en arrière-plan lorsque vous calez la vue selon un axe avec le pavé numérique ; il faut pour cela cocher Background images. La majorité du temps, vous n'aurez pas besoin de ce menu, fermez le donc en rappuyant sur N.
- La petite barre sous la vue 3D vous permet de faire bouger votre sélection : translation, rotation et homotétie selon les 3 axes. Marche en edit mode pour bouger des points ou en object mode pour bouger des objets. Vous pouvez utiliser les touches G, R et S respectivement pour avoir le même résultat en bougeant la souris. Si vous appuyez ensuite sur X, Y ou Z, vous bloquerez la transformation selon l'axe souhaité

- Tab : basculer en edit mode, vous ne pouvez alors modifier qu'un seul objet, celui sélectionné. N'oubliez pas, on sélectionne avec clic droit.
- Z : activer la transparence. Utile pour sélectionner des points cachés.
- Suppr ou X : ouvrir le menu de suppression, vous pouvez choisir quoi supprimer en détail et ça c'est bien.

Astuces pour sélectionner plusieurs points rapidement

- Shift+ Clic droit : ajoute le point sur lequel vous cliquez à la sélection. Permet donc de manipuler plusieurs points en même temps, pratique ! Si vous le faites sur un point déjà sélectionné, ça le désélectionne.
- B + rectangle validé avec Clic gauche : sélectionne tous les points à l'intérieur du rectangle de sélection. Pensez à activer la transparence si vous voulez ceux derrière ! Remarquez qu'une fois n'est pas coutume, on utilise le clic gauche.
- B + rectangle validé avec un clic sur la molette : **dé**sélectionne tous les points à l'intérieur du rectangle de sélection. Pensez à activer la transparence si vous voulez ceux derrière ! Remarquez qu'une fois n'est pas coutume, on utilise le clic gauche.
- Ctrl + + : si vous avez au moins un point sélectionné, sélectionne ceux adjacents à lui. L'inverse se fait avec Ctrl + -.
- Si vous voulez sélectionner des faces ou des arêtes plutôt que point par point, utilisez le menu suivant sous la vue 3D.

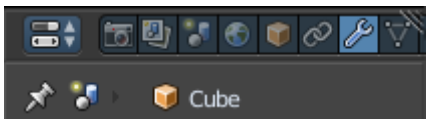
- A : tout sélectionner. Si un ou plusieurs points sont déjà sélectionnés, désélectionne tout à la place.

Ajouter des points sur un maillage

- Ctrl + Clic gauche : ajoute un point à l'emplacement du curseur. Si vous aviez déjà un point sélectionné, il les relie entre eux.
- Shift + D : duplique la sélection, l'équivalent d'un copier-coller. Penser à valider avec un clic gauche à la fin.
- E : extrude la sélection, difficile à expliquer regardez et vous verrez. Se valide avec clic gauche.
- F : relie les points sélectionnés entre eux avec soit des arêtes soit des faces. N'oubliez pas que 4 points ne sont pas forcément coplanaires. Si vous sélectionnez trop de points, soit Blender vous dit nan soit vous obtenez quelque chose d'abominable issu d'un des 9 plans infernaux.
- Shift + A : rajoute une forme de base sur le maillage à l'emplacement du curseur 3D.

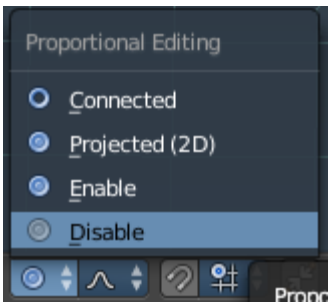
Enfin, voilà quelques commandes utilitaires :

- En object mode (on change de mode avec Tab) vous pouvez ajouter des modifiers à vos objets. Rendez-vous dans le menu modifier à droite (cf photo). Les plus utiles sont Array, Mirror, Boolean et Subdivision Surface (tous dans la catégorie Generate de la liste des modifiers). Internet vous apprendra comment vous en servir en détail.



- Vous pouvez manipuler les points en edit mode de manière proportionnelle. La commande se trouve sous la vue 3D et correspond à un menu déroulant dont le symbole est un petit rond. Cochez enable ou disable, je ne me sers que de ceux-là. Vous pouvez choisir la manière dont se déplacent les points à côté de la sélection en choisissant le type de courbe à droite.

Lorsque vous manipulez une sélection de points (rotation, translation ou homotétie), les points alentour seront influencés. Le rayon d'influence apparaît alors comme un cercle blanc et se règle avec la molette.



- Blender incorpore une banque d'add-ons. Si vous vous voulez modéliser un engrenage sans y passer une heure, allez dans file, user preferences, onglet add-on et vous n'avez qu'à cocher l'ajout souhaité.

Une fois votre somptueuse création terminée, il est temps de lui donner de la couleur ! En mode objet, allez dans l'onglet materials représenté par une boule. Il se trouve à droite de la vue. Si votre objet comporte plusieurs couleurs ou plusieurs textures, il faut lui associer plusieurs matériaux. Cliquez sur le plus dans l'onglet materials puis sur new, ça en créera un autre. Pour attribuer des faces de votre objet à un matériau, passez en edit mode, sélectionnez les faces concernées puis Assign qui se trouve dans l'onglet materials. De base, tout l'objet porte le même matériau, le premier de la liste. Lorsque vous serez sous Unity, les matériaux seront conservés et vous pourrez leur mettre des textures.

Importer votre création sous Unity

Suivez scrupuleusement ces étapes pour ne pas que votre création sur laquelle vous avez passé des heures arrive horriblement mutilée dans le jeu.

- Si vous avez plusieurs objets, faites les étapes suivantes pour chacun d'entre eux
- Tout d'abord, sélectionner votre objet et Ctrl + A, Scale.
- Passez en edit mode avec Tab. Sélectionnez tous les points avec A.

- Faites W puis remove doubles dans le menu venant de s'ouvrir. Cela supprimera d'éventuels doublons créés par accident.
- Faites Ctrl + N, cela recalculera la direction des normales de votre objet vers l'extérieur. Si une face apparaît invisible en arrivant sous Unity, vérifiez que vos normales sont orientées dans le bon sens. Internet vous aidera si ce n'est pas le cas.
- Faites U puis Smart UV Project, OK. Cette commande déplie le patron de votre objet pour que Unity détermine comment appliquer la texture. Si vous ne le faites pas, l'objet importé possédera une teinte fade moche.
- Sauvegardez bien votre fichier, vous avez fini ! Il vous reste à l'importer sous Unity.

Fabrication d'une animation sous Blender :

Pour faire une animation sous Blender, il faut tout d'abord ouvrir le mesh de l'objet que l'on souhaite animer sous Blender, créer un squelette qui servira à l'animation par la suite avec Blender (ou insérer un squelette tout fait) et le positionner correctement par rapport au mesh. Pour créer un squelette, Object Mode, Shift +A, Armature puis extruder (avec E) le premier os pour créer la hiérarchie. On extrude toujours en edit mode.

Une fois le mesh placé correctement par rapport au squelette, il faut dire quels points sont attachés à quels os. Pour cela on sélectionne tous les objets, puis le squelette et on utilise Ctrl+P et l'option « with automatic weights ». On peut ensuite voir si les affiliations sont bonnes en bougeant les os. Si oui, on peut enregistrer le .blend et passer aux animations. Si non, on peut annuler avec Ctrl+Z et essayer soit de bouger un peu les os, soit d'utiliser les weights pas automatiques (bien plus long et difficile).

Pour les animations à proprement parler, on sélectionne la frame où l'on veut travailler et les os que l'on souhaite faire bouger, on appuie sur I pour insérer une Keyframe et on sélectionne ce que l'on veut (location, rotation, scale). On sélectionne ensuite une autre frame, on bouge les os à la position d'arrivée, on insère une autre frame (avec I) et le logiciel calcule les déplacements intermédiaires.

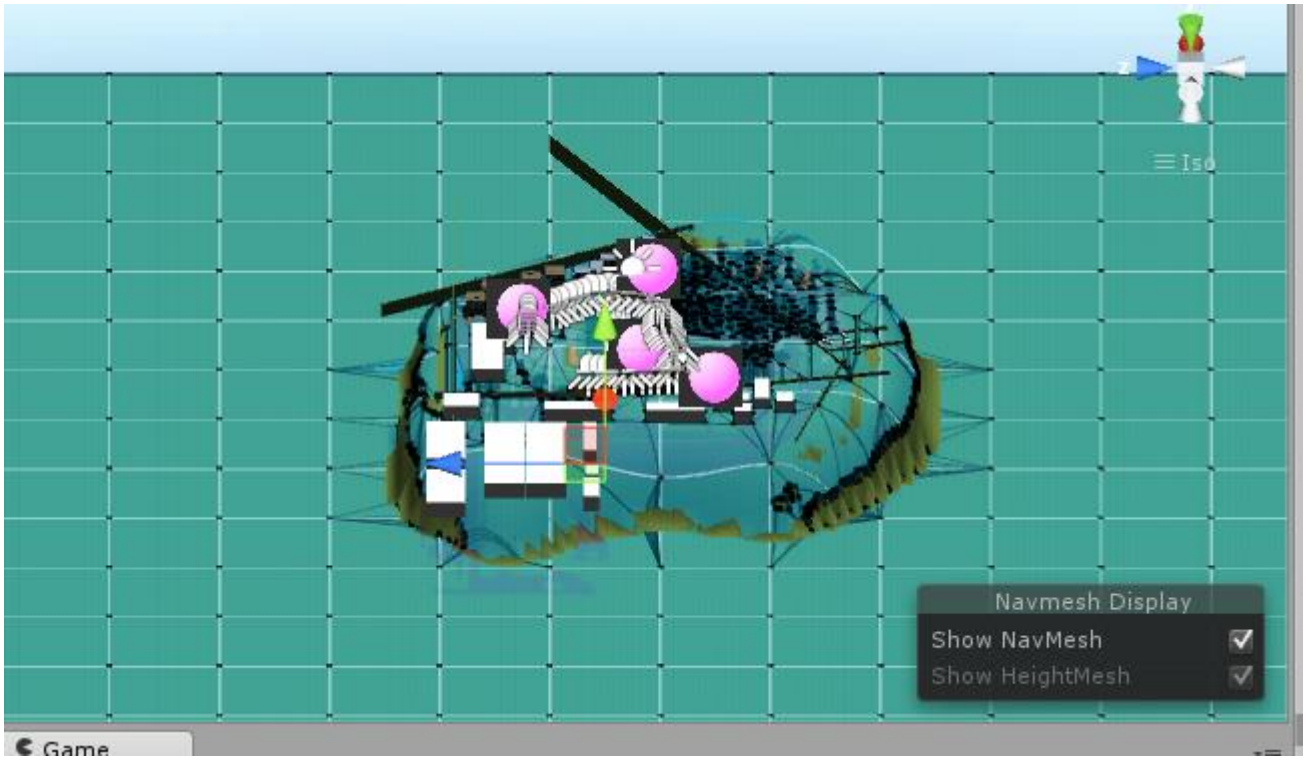
On peut faire plusieurs animations à la suite, on pourra ensuite les découper sur Unity. Une fois fini, on enregistre le .blend, et on peut l'importer sous Unity. On remet les textures sur les bons matériaux. Sous Unity, dans l'onglet « rig » du .blend soit on prend un « Humanoid » si on a un humain, soit un « generic, create from this model » si l'on a autre chose (une boule par exemple). Dans l'onglet animation, on voit le « default take » qui correspond à toutes les images de blender, on peut en sélectionner une partie et en faire une animation à part entière que l'on peut utiliser dans un animation controller, mais c'est une autre histoire !

COMMENT METTRE EN OEUVRE LE PATH-FINGDING SOUS UNITY

Dans la scène Campus, on a doté les zombies des capacités de trouver le chemin vers une destination. En effet, on a appliqué l'algorithme A* fourni dans unity par le composant "Navigation". Bien que les développeurs de Unity arrivent à trouver un millier de documentations sur Internet précisant le fonctionnement de ce composant, on va vous faire une brève explication:

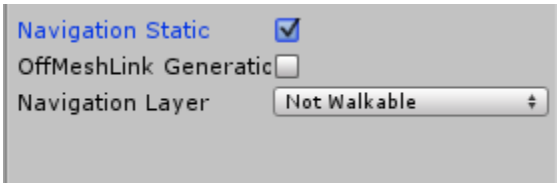
1. Bake the terrain.

En français "calculer le terrain". Clique sur "Window - Navigation" pour ouvrir la fenêtre de gestion de navigation. Vous pouvez voir dans la scène change de forme:



C'est parce que on avait déjà calculer le terrain. Sinon, si vous voulez refaire le calcul, cliquez sur "clear - bake", et vous verrez le processus de calcul.

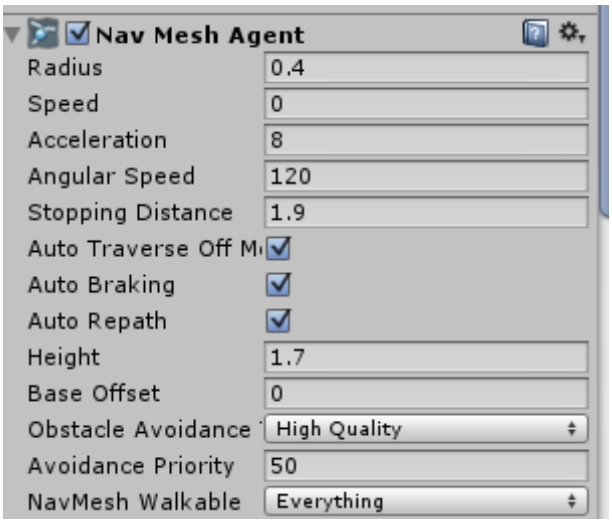
Les résultats du "calcul de terrain", fournissent en fait les points accessibles par les personnages. Pour assurer un bon fonctionnement, on devrait éliminer les terrains où il y a les bâtiments. Pour ce faire, changer les propriétés des "mesh" des bâtiments dans l'onglet "Object" de navigation comme ci-après:



Après avoir refait un "Bake", et les terrains des bâtiments seront éliminés.

2. Ajouter le composant "Nav Mesh Agent" aux zombies

Ce sont les zombies qui vont chercher les destinations dans notre jeu. Cliquer sur le zombie et cliquer "Component - Navagation - Nav Mesh Agent".



Pour les explications détaillées des paramètres de ce composant, se référer à la documentation officielle de Unity.

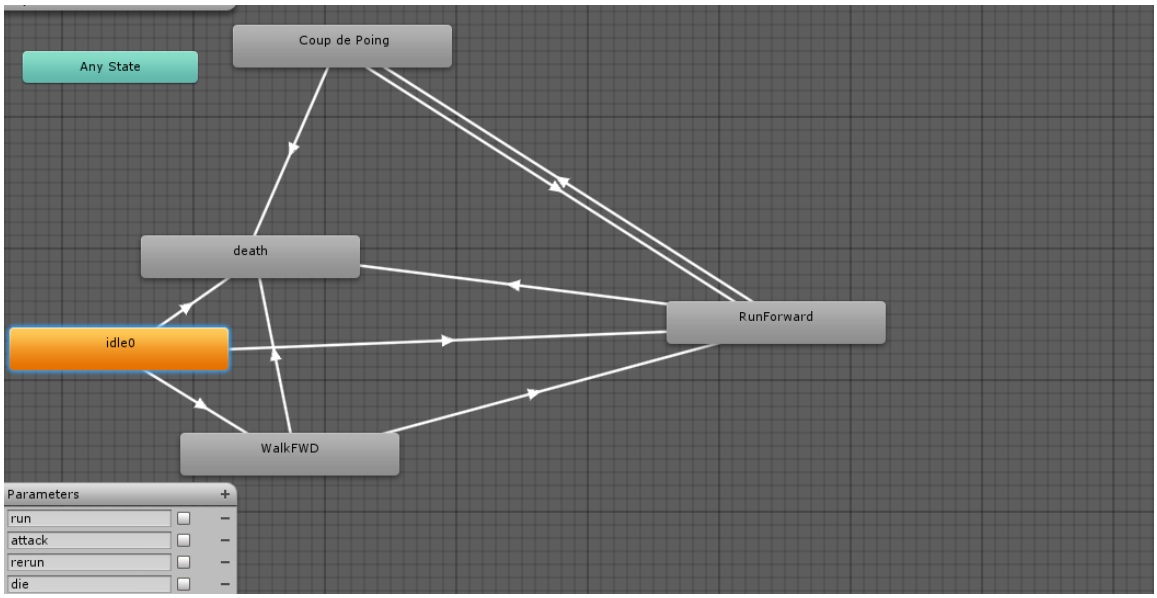
3. Détermimer la destination de "PathFinding"

Il faut le faire à travers des scripts. Dans notre jeu on l'a réalisé dans le script "Zombie_comportement". Explication de conception des comportements des zombies:

Les zombies sont créés par les 4 points d'apparition dans la scène. C'est le script attaché à "Creation_zombie" qui gère ce fait.

i) A leur apparition, les zombies se baladent de manière aléatoire sur le campus tant que le joueur ne s'approche pas d'eux. En effet, dans le script "Zombie_comportement", on accorde au zombie une destination aléatoire sur le campus toutes les 10 secondes. Cependant, il y a des chances que cette destination soit au sein du terrain occupé d'un bâtiment, ce qui fait que le zombie n'arrive pas à trouver un chemin vers cette destination (car "non walkable") et se bloque sur sa position. Pour régler ce problème, on teste si le zombie arrive à trouver un chemin (calculé automatiquement par Unity) au bout de 2 secondes, si ceci n'est pas le cas, le zombie change de destination; et ainsi jusqu'à ce qu'il trouve une destination aléatoire accessible.

ii) Une fois que le joueur s'approche à une certaine distance d'eux, ils repèrent ce dernier et courent après lui. Et quand ils rattrapent le joueur il se met en mode d'attaque. Pour réaliser ces fonctionnalités, on a fait une correspondance entre les animations et l'état de zombie, à travers le composant "Animator" de zombie et le script "Zombie_comportement". On doit créer au préalable des animations correspondant aux différents états du zombie dans le Animator comme suivant:



Quand le zombie est en état de "balade aléatoire", son animation de "Walk" est activée, la vitesse du zombie est 1. Une fois que le zombie s'approche à 3 mètres de joueur, il se met en état "isSuivreJoueur" et sa destination de PathFinding est fixée sur la position du joueur jusqu'à sa mort. Dans cet état-là, l'animation est basculée à "run" et la vitesse du zombie est mise à 3. Tout ce processus est effectué dans le script. Quand les zombies meurent, l'animation est basculée à l'animation "death" et le zombie est détruit quand l'animation est finie.

Addendum

Architecture des ressources utilisées

Tous les scripts et modèles doivent être placés dans le dossier Assets généré par Unity. C'est le contenu de ce dossier qui est présenté dans la vue Project de Unity. Nous détaillons dans la suite l'intérêt de chaque dossier.

a) _IMPORTED

Un des avantages de Unity est de mettre à disposition des utilisateurs l'Asset Store, où on peut trouver des assets divers et variés. Tous les assets qui ont été téléchargés sont placés ici (que ce soient ceux trouvés sur l'Asset Store ou ceux importés des fichiers par défaut de Unity, les Standard Assets).

b) Bâtiments

C'est là que sont mis les bâtiments faits sous Blender. Les Materials sont ceux importés directement depuis Blender. Les objets importés le sont sous forme de Prefab.

c) Blender

On y trouve les autres choses faites sous Blender : prefab d'armes directement importés de Unity, personnages faits sous Blender après être passés par MakeHuman et FaceGenModeler.

d) Campus

Animation : On y trouve les animations liées aux objets du campus (mais pas les zombies : on y trouve par exemple l'animation de déploiement de la boule).
Arbres : Les materials qui permettent de faire des arbres, ainsi que leurs modèles.
Ennemis : Prefab des ennemis qu'on ne trouve que dans le campus, donc la Boule notamment.
Materials : Materials utilisés dans la scène : ceux des portails, de la haie, de la lune...
Models : Modèles de certains objets : anciennes portes, lampes, Comparat, et leurs matériaux. Certains modèles ne sont pas utilisés.
Scripts : Certains scripts utilisés dans le campus ; allumage des lampes et script qui fait défiler le temps sur le campus.
Sounds : C'est là qu'on mettrait les sons qui sont produits en marchant sur certaines textures de sol.
Terrain : Le terrain qui est utilisé dans la scène Campus.
Textures : Textures de la Skybox et du terrain.

e) Dialogue

C'est un dossier important. C'est dedans qu'on trouve tous les .json qui sont nécessaires aux dialogues et aux messages à afficher sur l'écran. Ce dossier est celui dans lequel le logiciel récupère les messages à afficher lorsqu'il est lancé. Il faut donc le copier et le coller dans le dossier Data qui est généré après compilation.
Tous les .json des dialogues des personnages ainsi que les .json des messages à afficher à l'écran sont ici.

f) Écran

Contient les animations, materials, prefabs et scripts nécessaires aux affichages des écrans de mort, de titre et de fin.

g) Editor

C'est un dossier propre à Unity dans lequel des scripts sont placés. Chaque script permet de changer l'affichage dans la vue Inspector pour un autre script.

h) FPS

Le jeu est également un jeu à la première personne, d'où l'existence de ce dossier alors que ce n'est pas vraiment un jeu de tir à la première personne.

On trouve dans ce dossier les modèles de la balle de basket et du curseur qui s'affiche au milieu de l'écran, les scripts attachés aux armes et aux magies, le script qui permet de faire apparaître le joueur sur la carte et de le faire bouger.

i) Gizmos

Ce sont des extensions de Unity (comme Editor) qui permettent d'afficher de nouveaux symboles dans la vue scène.

j) ATH

On y trouve les materials, prefabs et scripts liés à l'inventaire, au radar et à l'affichage à l'écran des caractéristiques.

k) Jeu BasketBall

Assets liés au jeu de basketball activable en inspectant l'un des paniers du campus.
Détails : La barre en dessous de l'écran contrôle la force de lancement; on ne peut lancer qu'un seul ballon toutes les secondes, mais on peut lancer une infinité de balles. La balle lancée se détruit (disparaît) au bout de 5 secondes, afin d'éviter l'encombrement des balles dans la scène. Un "Collider" est ajouté au panier dans le but de déterminer si la balle traverse le panier.

l) Labyrinthe

Prefabs et scripts liés au labyrinthe. Le labyrinthe est une scène spécifique du jeu où le joueur doit sortir d'un labyrinthe généré aléatoirement. Dans ce labyrinthe, le joueur est poursuivi par des monstres invincibles qui tuent le joueur lorsqu'ils s'en approchent.
Les monstres connaissent le labyrinthe et l'emplacement du joueur ; pour rendre le jeu jouable, ils se déplacent de façon non optimale et peuvent donc faire de grands détours.

m) Materials

Materials divers, dont ceux des armes importées de Blender.

n) Mecanim

Animations et prefabs des personnages du jeu.

o) Musique

Toutes les musiques du jeu, qui sont ensuite utilisées dans le GameObject LecteurMusique présent sur la scène Ecran titre.

p) OVR et Plugins

Dossiers obtenus de la scène Tuscany d'Oculus Rift. Ces dossiers doivent être placés directement dans Assets pour que l'Oculus Rift fonctionne. Celui-ci ne peut être utilisé qu'après compilation avec une version professionnelle de Unity (lancer ensuite le .exe "directToRift"). La caméra utilisée pour l'Oculus Rift est spéciale (elle est placée au même niveau que la MainCamera), et doit être activée pour fonctionner. Pour gérer ceci, dans ControlCenter, on a un paramètre qui signifie si la caméra doit être celle de l'Oculus. Avant compilation, mettre ce paramètre à true. Attention : aucune boite de dialogue créée par 'OnGUI' ne s'affiche avec l'Oculus, notamment les boutons qui permettent de démarrer le jeu. Tel quel, le logiciel ne permet pas de jouer à l'Oculus. En changeant l'écran titre et toutes les boites de dialogue en OnGUI, il sera possible d'utiliser l'Oculus Rift.

q) Prefabs

Tous les prefabs du jeu. On voit les prefabs des armes et des objets de deux façons : ceux attachés au personnage, qui peuvent infliger des dégâts et être lancés, et ceux sur le sol, qu'on peut ramasser.

r) RPG

Contient les scripts de Caracteristique et de Niveau, pour gérer les éléments de RPG.

s) Scenes

Contient toutes les scènes.

t) Zombie

Les scripts, animations et prefabs des zombies, ainsi que le prefab qui les crée.

u) Generic

Tous les scripts, prefabs... qu'on peut utiliser partout. Tout ce qui peut être utilisé partout est rangé dans ce dossier.

Avec cette organisation, il était facile de répartir les tâches sans risquer de conflits via Git : les personnes qui s'occupent des zombies ne vont pas toucher aux scripts qui gèrent l'inventaire par exemple. Sans cette organisation, les scripts notamment auraient été trop dispersés et l'usage d'un gestionnaire de versions aurait été moins efficace.