Projet Unity 2017 :
Zombies May Cry

Bortier Christophe

Bura Roland

Table des matières

Description du jeu

Introduction

Notre jeu est un Survival horreur qui se déroule dans une grotte sombre. Des zombies apparaissent par vague et le joueur doit tous les tuer pour finir le niveau et passer au suivant. Le nombre de zombies ne fait qu'augmenter à chaque niveau, leur point de vie aussi et il est de plus en plus dur de survivre. Le jeu s'arrête quand le joueur meurt. Le but est donc de survivre le plus longtemps possible pour faire le meilleur score, il n'y a pas d'autre fin que la mort du joueur. Pour varier un peu les niveaux, la carte sur laquelle se passe le jeu change à chaque fois.

Mécaniques implémentées

- Carte générée aléatoirement
- Déplacement au clavier
- Tir de projectiles
- Visée du tir à la souris
- Caméra centrée sur le joueur
- Gestion du score et d'un high score persistant
- Gestion de la vie du player et des ennemis
- Déplacement "intelligent" des ennemis
- Gestion des dégâts
- Gestion de "loot" aléatoire après la mort des ennemis
- Changement de niveaux et progression de la difficulté
- Gestion de bonus persistant d'un niveau à un autre
- Apparition aléatoire des ennemis et du joueur
- Pooling des objets pour limiter l'instanciation abusive

Organisation

Répartition du travail et estimation du temps de travail

Christophe

- Carte générée aléatoirement (12 heures)
- Gestion de la vie du joueur et des ennemis (4 heures)
- Gestion des dégâts (2 heures)
- Gestion de "loot" aléatoire (4 heures)
- Changement de niveaux et progression de la difficulté (12 heures)
- Gestion de bonus persistant d'un niveau à un autre (2 heures)
- Transition d'une scène à l'autre (1 heures)
- Apparition aléatoire des ennemis et du joueur (10 heures)
- Pooling des objets pour limiter l'instanciation abusive (1 heures)
- Gestion de la lumière (2 heures)

Roland

- Déplacement au clavier(3h)
- Tir de projectiles(12h)
- Visée du tir à la souris(8h)
- Caméra centrée sur le joueur(4h)
- Gestion du score et d'un high score persistant(3h)
- Déplacement "intelligent" des ennemis(8h)
- Création des menus et des UI(6h)
- Ambiance "horreur"(3h)
- Dégâts continues des ennemies(1h)

A toutes ces tâches doivent s'ajouter de très nombreuses heures de debug, mise en commun et balancing. Estimation : 15-20 heures par personnes.

Outils

Comme nous ne travaillons qu'à deux, aucun outil de gestion de tâches n'a été utilisé et nous nous sommes juste tenu au courant de l'avancement et de la répartition de ces dernières verbalement ou par messages. Pour la mise en commun du travail, nous avons pris une approche très modulaire. C'est à dire que nous nous sommes forcé à travailler chacun sur des parties très différentes du projet et de ne pas toucher aux mêmes objets et scriptes en même temps. De ce fait, aucun outil de mise en commun du code ne s'est avérer nécessaire. Nous avons mis à profit les heures de cours pour mettre en commun notre travail respectif par simple remplacement de fichiers avec une vérification que tout se passait toujours bien. Sur un projet plus large ou avec plus de personnes nous aurions évidemment choisi une autre approche. La mise en commun se faisait sur l'ordinateur de Christophe et pour continuer à travailler sur le même projet ainsi que pour une certaine sécurité en cas de perte de donnée, un GIT a été utiliser pour stocker la dernière version du projet et permettre la récupération par Roland ou comme point de retour en cas d'erreur ou de crash important.

Description des fichiers importants

MapGeneration/MeshGenerator

Ces deux-là vont ensemble, ils permettent de créer une carte aléatoire 3D. Ils sont aidés par les scripts Coord et Room qui représentent respectivement les coordonnées de chaque case de la map et les pièces de "vide".

MapGeneration : Ce script crée une table à deux dimensions (width et height) et la remplit de 1 et de 0 selon une proportion dictée par un "randomFillPercent" un pourcentage qui représente la quantité de 1 par rapport au 0. 1 représente un mur et 0 du vide.

Une fois les 1 et les 0 placé dans la table, une opération de "smoothing" à lieu à plusieurs reprises. C'est à dire que chaque chiffre va être comparé à ses voisins et, si trop peu sont similaire a lui, ici 4 sur les 8, il change et devient un 1 s'il était un 0 ou inversément. Après plusieurs itérations, les chiffres se retrouvent donc groupés ensemble, formant des pièces de vide ou des bouts de murs. Il faut encore s'assurer que les bords seront toujours fermés donc des murs sont placés sur les contours.

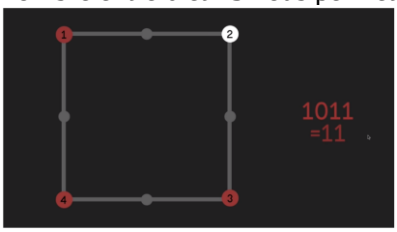
Ensuite, il faut "Process" la map, la parcourir et supprimer en transformant les pièces qui sont plus petite qu'une certaine taille en murs et la même chose pour les ilots de murs en pièces. Les pièces et

les murs survivant sont enregistrer dans une liste et triée par taille, la plus grande est nommée “pièce principale”.

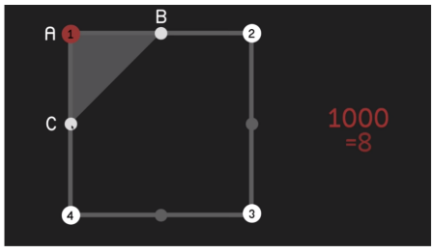
Il faut maintenant s’assurer que toutes les pièces soient accessibles de partout. Pour cela on va parcourir la liste des pièces, regarder quelle est la pièce la plus proche et les connecter ensemble. Maintenant toutes les pièces sont connectées à une autre mais pas forcément toutes entre elles (il pourrait y rester des pièces connectées l’une à l’autre mais pas avec un autre groupe), donc il faut continuer à parcourir et chercher à les connecter toutes, directement ou par l'intermédiaire d’une autre à la pièce principale.

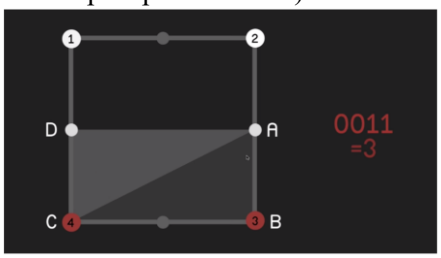
La connexion se fait en repérant la case de chaque pièce qui sont les plus proches et de tracer une ligne entre, une fois la ligne tracée, toutes les cases (de murs) traversées par cette ligne sont transformées en “vide” et un passage est créer.

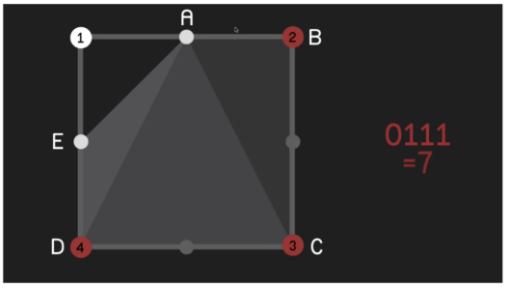
MeshGenerator : Avec la MapGeneration nous avons une carte remplie de carrés, 0 et 1. En utilisant le processus “Marching Squares” nous allons transformer ces carrés en meshes qui peuvent être utilisé dans le jeu.
Tous les “blocs”, case de 1 ou 0 vont représenter les coins d’un carré. Pour transformer cela en hexadécimale, nous allons dire que si un coin est un mur il vaut 1 sinon 0 dans le nombre 0000 en commençant par le coin en haut à gauche et en continuant dans le sens des aiguilles d’une montre. Ce nombre entre 0 et 15 nous permet de savoir quelle est la configuration de ce carré.



Comme les meshes sont constitué de triangles, il faut maintenant relier les coins “allumés” du carré entre eux. Si par exemple seul un coin du carré était allumé, il ne serait pas possible de tracer de triangle ce qui n’est pas idéal, nous allons donc relier les milieux des côtés adjacents à ce coin afin d’harmoniser nos triangles. (Quelques exemples pour illustrer).







Une fois les meshes de toute la carte créé, il faut encore créer des murs. Pour cela il va d’abord falloir trouver et relier tous les bords des triangles qui sont en contact avec du “vide”.
Une fois cette opération finie, il suffit de créer des meshes perpendiculaires à notre carte et les murs sont en place.

SpawnEnemies

Assez similaire à SpawnPlayer, SpawnEnemies utilise la liste des pièces créer dans MapGeneration pour connaître les coordonnées de toutes les cases de la carte ou il n'y a pas de murs. Une case aléatoire est choisie et si le joueur n'est pas trop proche, qu'il n'y a pas déjà un ennemi dessus ou très proche et que cette case n'est pas le long du mur de la pièce pour éviter qu'un ennemi imposant apparaissent partiellement dans le mur, la case est choisie comme lieu d'apparition de l'ennemi, une méthode s'occupe de transformer les coordonnées de la case sur la carte en coordonnées dans le jeu en lui-même.

Le script est aussi responsable du type d'ennemi qui va apparaître et de la vie de ce dernier ainsi que de l'instancier en allant le chercher dans l'Objet Pool s'il y en a déjà un. Comme les ennemis sont des préfabs, il faut aussi leur assigner un événement de manière programmatique à leur instanciation.

Ce script est également responsable de compter le nombre d'ennemis restant et d'informer notre Game Manager qu'il n'y a plus d'ennemis et que le niveau peut se finir.

Un des plus grands problèmes liés à ce script fut une incohérence entre la position théorique d'apparition d'un ennemis et sa position réelle. En effet le navmesh des ennemis posait problème et déplaçait parfois l'ennemis, ce dernier se retrouvant donc parfois dans les murs. Ce problème a été réglé grâce à navmesh.warp().

Manager

Le manager est le script responsable pour le changement de niveau, la progression de la difficulté et de la persistance des données entre chaque scène. Il est également partiellement responsable du changement de scène avec le Scene Loader qui prend le relais lorsque le Manager est détruit lorsque la partie recommence.

La méthode InitLevel() s'occupe de trouver le SpawnEnemies et de lui transmettre les instructions pour le niveau actuel : difficulté, nombre d'ennemis, intervalle entre chaque apparition d'ennemi.

Il s'occupe de retenir et d'appliquer les changements fait au player, ce dernier étant instancier dans SpawnPlayer à partir d'un prefab à chaque niveau, il modifie la vie, les munitions... au début du niveau et enregistre ce qu'il s'est passé pendant la partie à la conclusion de celui-ci.

Enfin, il est responsable de sauvegarder le High Score du joueur.

Déplacement "intelligent" des ennemis

Pour ce faire nous avons utilisé NavMesh d'Unity ainsi que deux scripts (LocalNavMeshBuilder et NavMeshSourceTag) travaillant ensemble dans le but de bake à nouveau le NavMesh lorsqu'un changement de décor sur la map est effectué.

- LocalNavMeshBuilder :

Nous permet de définir une zone dans laquelle pour tout changement (ajout, suppression, modification) d'un composant ayant le script "NavMeshSourceTag" le navMesh est automatiquement mis à jour en modifiant les NavMeshData ,qui permet la création d'instances de NavMesh, en lui ajoutant les NavMeshBuildSource récupérés grâce au script NavMeshSourceTag.

- NavMeshSourceTag :

Récupère le mesh-filter (ce qui permet d'avoir un rendu à l'écran) de l'objet pour le passer au constructeur du NavMesh en l'ajoutant au NavMeshBuildSource.

Mouvement et orientation (visée) du joueur :

Pour commencer nous avons régler la drag et angular drag dans le rigidbody du player à Infinity pour éviter un ralentissement au fil du temps. Le déplacement et la rotation du joueur est géré dans le script KeyboardMoves.

KeyboardMoves

Nous permet de déplacer le joueur grâce à la physique de manière smooth en continue en utilisant MovePosition() sur le rigidbody du joueur. Il faut faire attention à normaliser le mouvement pour ne pas se déplacer plus vite en diagonale. Cela se fait simplement en faisant un normalized sur le vecteur.
Pour ce qui est de la rotation, nous utilisons un ray allant de la caméra jusqu'au sol à la position de la souris pour orienter le joueur vers notre souris. Pour ce faire nous avons un int floorMask qui contient le layerMask du sol et qui nous permet de nous assurer que le ray ne prendra en compte que la collision avec ce sol et non le reste. Une fois le ray obtenu il faut encore le cast pour en retirer une direction (vecteur) une fois ce vecteur en poche il faut passer par Quaternion pour le transformer en rotation et particulièrement la fonction LookRotation de celui-ci qui permet de changer "l'axe avant" de l'objet ici de l'axe Z à la position de la souris. Pour finir il faut l'appliquer au RigidBody du player en utilisant MoveRotation() qui permet une orientation smooth en continue de l'objet.

Conclusion

- Ce que nous voulions réaliser au départ c'était un Survival horreur en vue isométrique avec des cartes différentes entre chaque niveau. Avec la possibilité d'utiliser différentes armes et pouvoir ramasser des bonus variés.

- Et c'est ce que nous avons à l'exception près que notre jeu a une vue du haut et non isométrique. Cela rend mieux avec notre level design et permet une meilleure visibilité générale en jeu. Il reste bien entendu une longue liste d'améliorations qui peuvent être envisagées. D'abord pour rendre le jeu plus propre : des assets pour ne pas faire un carré attaqué par des sphères mais de vrais zombies, une meilleure gestion de la lumière car elle ne réfléchit pas contre les murs et traverse ces derniers, une indication visuelle lorsque le joueur est attaqué car ce n'est pas très visible. Ensuite, et beaucoup plus nombreuses, pour rendre le jeu plus chouette : diversité dans les ennemis et dans les armes, améliorer encore les bruitages et l'ambiances, un score qui dépendrait de la difficulté et du temps mis à nettoyer le niveau. Et en voyant encore plus grand : le moyen de mettre les High Score en ligne pour se comparer à ses amis voire même un mode multijoueur, un système de progression de personnage plus avancé pour des éléments de RPG. Les possibilités d'amélioration sont grandes mais étant donné le temps qui nous était imparti et des défis techniques que nous nous sommes lancés, nous sommes heureux de l'avancement de notre projet et pouvons dire que nous avons retiré beaucoup de ce projet. Principalement au niveau de notre maîtrise des concepts Unity abordé pendant le cours et maintenant acquis mais également en poussant un peu plus loin et en apprenant des choses plus avancées grâces à des tutoriels.

- Pour ce qui est des problèmes rencontrés nous en avons eu plusieurs :

    - Les principaux problèmes rencontrés ont été des incompréhensions du comportement de certaines choses. Le comportement inattendu entre le positionnement des ennemis et le

déplacement fait par le navmesh avant l'utilisation du warp() (référer plus bas pour plus de détails) , le comportement de certains objets lorsqu'on les ressortait de l'ObjectPool : le fait que l'abonnement aux événements fait de manière programmatique n'apparaisse pas dans l'inspecteur et se produise deux fois lorsque cet abonnement est ajouté une deuxième fois à l'instanciation de l'objet ou les problèmes lorsque l'ObjectPool persistant de scène en scène essayent de retourner des objets qui ont été détruit lors du changement de scène ne sont que quelque exemples.

- Détails du problème de l'apparition des ennemies sur la carte. Ceux-ci avaient tendance à apparaître à des positions différentes de celles qu'on leurs assignait. Avec l'aide du professeur Jean Gobert du Coster, nous avons découvert que c'était à cause du NavMeshAgent que cela se produisait car il empêchait l'ennemi d'être "téléporté" à la position assignée. Pour pallier à ce problème nous avons essayé de désactiver le NavMeshAgent sur les ennemis et de le réactiver après leurs apparitions. Malgré le fait que cela semblait une bonne idée cela a conduit à un bug qui faisait que les ennemis ne suivaient pas toujours le joueur ou parfois ils s'arrêtaient à une bonne distance de celui-ci. Pour finir nous avons utilisé le NavMesAgent.Wrap pour définir la position de l'ennemis au moment d'apparaître sur la carte.

- Notamment pour ce qui est de la collision des bullets avec les murs. Le collider sur les murs étant très fin la collision n'était pas toujours détectée par Unity il a fallu donc paramétrée la collision sur une détection continue et activer l'interpolation qui permet d'atténuer l'effet de la physique à une fréquence d'images fixe. Avant de trouver cette solution nous avons essayé de gérer les impacts grâce à des trigger mais ceci posait problème car notre jeu se situe dans un contexte concave et donc nos balles étaient détruites dès leurs création.

- Un autre problème rencontré était celui de la progression des balles dans l'espace. Nous avons dû utiliser la vélocité pour les faire avancer en ligne droite car avec les autres méthodes nous avons eu des problèmes de mise à jour quant à la direction dans laquelle on voulait tirer (3-4 frames de retard) ou alors les balles qui partent dans des directions complètement aberrantes

- Un "bug" qui persiste toujours à ce jour est celui de la caméra qui met un certain temps à trouver le joueur pour se caler sur ses déplacements ce qui produit un message d'erreur disant que le script ne trouve pas le joueur sur la carte. Il est à noter que cela ne gêne pas l'expérience de jeu.

- Pour ce qui est des choses que nous aurions faits autrement n'étant pas des experts en Unity il n'y a qu'une chose concernant le jeu qui nous vient actuellement à l'esprit et ce serait de gérer les tirs avec des Ray, RaycastHit, LineRenderer pour ne pas devoir créer à chaque fois des objets pour chaque balle tirée et avoir un "meilleur" effet visuel. Pour ce qui est de la mise en commun du projet celle-ci n'a pas été problématique puisque nous avons toujours travailler sur des fichiers et objets différents mais lorsque des bugs étaient trouvés et réglés, il fallait sans cesse s'envoyer la dernière version du scripts ou juste la ligne problématique corrigée. Je pense que si le projet avait été plus long il aurait fallu le faire de toute façon donc cela aurait été une bonne pratique de le mettre en place dès le début.

Annexe

Toutes les textures et assets utilisés ont été repris du projet SpaceShooter 2017

Tutoriels suivis :

- Aide pour la caméra : https://unity3d.com/fr/learn/tutorials/projects/survival-shooter/camera-setup?playlist=17144
- Déplacement et rotation du joueur : https://unity3d.com/fr/learn/tutorials/projects/survival-shooter/player-character?playlist=17144
- Autobake NavMesh : https://github.com/Unity-Technologies/NavMeshComponents
- Génération de la map aléatoire : https://unity3d.com/es/learn/tutorials/s/procedural-cave-generation-tutorial

GitHub du projet : https://github.com/tof113/GIT