Castle Of Demise

SQRT(100)

Paul **LACANETTE**

Jean **HERAIL**

Roni **YILDIZ**

Clément **GRONDIN**

Amadéo **HÉAULME**

Rapport de Projet

18/06/2024

Table des matières

1. Introduction **5**
1.1. Présentation du rapport 5
1.2. Présentation de l'entreprise 5
1.3. Comparaison de nos tâches accomplies avec les objectifs initiaux du cahier des charges 7
1.4. Notations utilisées lors de ce rapport de projet 7
2. Projet **9**
2.1. Présentation du projet 9
2.2. Origine et nature du jeu 9
2.2.1. Histoire qui a inspiré le projet 9
2.2.2. Début de la création 10
2.3. État de l'art 11
2.3.1. Influences techniques avec ID Software 11
2.3.1.1. Wolfenstein (1981) 11
2.3.1.2. DOOM (1993) 11
2.3.1.3. Quake (1996) 12
2.3.2. Influences Artistiques et Environnementales 12
2.3.2.1. Ultrakill (2020-Présent) 12
2.3.2.2. Castlevania (1986) 13
2.4. Organisation du Projet 13
2.4.1. Répartition et description des tâches 13
2.4.2. Outils et logiciels utilisés 16
3. Moteur de jeu **17**
3.1. Présentation de Godot 17
3.2. Pourquoi Godot ? 18
3.3. Difficultés rencontrées 18
3.3.1. Godot 3.5.3 18
4. Interfaces et mécaniques du Jeu **20**
4.1. Mécaniques de jeu 20
4.1.1. Kinematic Body 20
4.1.2. Déplacement 21
4.1.3. Les armes 21
4.1.4. Ressources Récoltables 22
4.1.5. Le son 23
4.1.5.1. Effets sonores (Sound SFX) et Musiques 23
4.2. Interfaces 24
4.2.1. Menus hors jeu 24
4.2.1.1. Menu Principal 24

4.2.1.2. Menu Options....25
4.2.1.3. Menu de Sélection du mode de jeu....27
4.2.1.4. Menu Multijoueur....27
4.2.1.5. Menu d'Hébergement....27
4.2.2. Menus fonctionnels en jeu....28
4.2.2.1. ATHs....28
4.2.2.2. Menu Pause....29
4.2.2.3. Menu de Mort....29
4.2.2.4. Menus de Victoire....30
5. L'environnement de jeu....30
5.1. Descriptif de l'environnement de jeu réalisé....30
5.1.1. Les cartes....30
5.1.2. Les monstres....33
5.1.3. Les Modèles 2D (lutins et textures)....33
5.1.4. Modèles 3D....34
5.1.5. Lutins joueurs....34
6. Le mode multijoueur....35
6.1. Descriptif de l'état actuel du mode multijoueur....35
6.1.1. Le système multijoueur....35
6.1.1.1. Encodage....36
6.1.1.2. Détection des joueurs....37
6.1.1.3. Lancement de la partie....38
6.1.1.4. La synchronisation des joueurs....39
6.1.2. Lancement du jeu....41
6.1.2.1. Les différents modes de jeu....41
6.1.2.2. La partie....41
6.2. Difficultés rencontrées....43
6.2.1. La synchronisation et transition vers Godot 4.2....43
6.2.2. L'incident du dictionnaire....44
6.2.3. Les caméras....45
6.2.4. Incompatibilité multijoueur et non multijoueur....45
7. Le site web....46
7.1. États antérieurs du site web....46
7.2. Etat actuel du site web....46
8. L'intelligence artificielle....47
8.1. Pathfinding....47
8.2. Gestion des signaux des attaques....47
9. Conclusion....48
9.1. Crédits....48
9.2. Ressenti personnel....48

9.2.1. Amadéo Héaulme....48
9.2.2. Paul Lacanette....49
9.2.3. Clément Grondin....50
9.2.4. Roni Yildiz....50
9.2.5. Jean Herail....51
9.3. Conclusion de ce rapport....51
10. Annexe....52

1. Introduction

1.1. Présentation du rapport

Ce rapport est le compte-rendu final de notre projet, donc du jeu Castle of Demise, présenté par le groupe sqrt(100). Il a pour objectif de comparer ce que nous avons accompli avec ce que l'on avait prévu lors du commencement du projet et depuis mars. Ce rapport a aussi comme but de présenter la version finale et fonctionnelle du jeu et de donner une vision d'ensemble sur le projet et sa réalisation.

1.2. Présentation de l'entreprise

Notre entreprise se nomme sqrt(100) et est composée de cinq membres.

Jean HERAIL assume le rôle de chef d'équipe au sein du projet. Doté d'une solide expérience en programmation et d'une familiarité avec l'intelligence artificielle, il a apporté une expertise technique essentielle pour la réalisation du jeu. De plus, Jean démontre un intérêt pour la musique, ce qui a énormément enrichi l'aspect sonore du jeu.

Paul LACANETTE combine une solide culture vidéoludique avec un intérêt prononcé pour la musique et le développement de jeux vidéo. Son expérience préalable avec Godot, ainsi que sa passion pour les jeux de tir à la première personne, ont fait de lui un atout majeur dans la réalisation du projet. Il s'est consacré à l'approfondissement de ses compétences et à la réalisation d'un jeu vidéo de qualité. Ces compétences se sont retrouvées indispensable dans la réalisation du multijoueur et l'implémentation de l'IA. Paul a également créé les ennemis et ajouté la majorité des textures.

Amadéo HEAULME en tant que membre clé de l'équipe de développement du jeu, son rôle prépondérant est la création du mode multijoueur, une tâche cruciale pour garantir une expérience de jeu amusante et stimulante. Par ailleurs, Amadéo a apporté une touche ludique au projet en ajoutant des éléments surprenants et divertissants pour

les joueurs. Il s'est également chargé de l'enrichissement du jeu en proposant des fonctionnalités additionnelles. En tant que créateur du dépôt Github, Amadéo veille à ce que toutes les mises à jour soient bien documentées et communiquent efficacement avec l'équipe via Discord, garantissant ainsi une collaboration fluide et transparente.

Clément GRONDIN possède une solide connaissance de la culture vidéoludique et nous a apporté une perspective rafraîchissante au développement du jeu. En qualité de membre de notre équipe, son rôle consiste notamment à l'implémentation de certaines mécaniques et s'occupe de la cohérence des cartes. Clément s'est chargé de la création du menu Option, et de sa sauvegarde. Sa participation active a été source d'inspiration pour l'ensemble de l'équipe, contribuant de manière significative à l'évolution positive de notre projet.

Roni YILDIZ, le consultant culturel vidéoludique, apporte une richesse de connaissances et une passion profonde pour les jeux vidéo au sein de l'équipe. Fortement imprégné de la culture du jeu vidéo depuis son plus jeune âge, il a offert des références précieuses pour le projet Castle of Demise. Son rôle consiste également à développer l'aspect créatif du jeu, notamment en élaborant l'univers, l'histoire servant de base au jeu, et en supervisant la modélisation des éléments du jeu. La contribution de Roni assure une profonde immersion dans le monde du jeu et garantit une expérience captivante pour les joueurs. Roni s'est ainsi chargé de la création des cartes.

1.3. Comparaison de nos tâches accomplies avec les objectifs initiaux du cahier des charges

Tâches	1ère Soutenance	Achevé
Mécaniques de Jeu		
Déplacements	100%	100%
L'arme	75%	100%
Le Son	80%	100%
Interfaces	80%	100%
L'environnement de Jeu		
Les cartes	45%	100%
Les monstres	80%	100%
Multijoueur	35%	100%
Site Web	40%	100%
IA	0%	100%

Figure 1 : Tableau d'avancement

Bien que la majorité des objectifs que nous nous étions fixés soient maintenant accomplis, certains le sont d'une manière différente de ce qui était prévu lorsque nous les avons pensés. Par exemple, nous avions imaginé plusieurs armes mais avons abandonné cette idée pour se concentrer sur une seule, rendant son système plus complet que ce qui était prévu, mais cela sera expliqué plus en profondeur plus loin.

1.4. Notations utilisées lors de ce rapport de projet

Afin d'avoir une meilleure compréhension des termes que nous allons aborder dans ce rapport, nous avons décidé de les ajouter dans cette partie, avec leur traduction et une définition. Lors de ce rapport, nous parlerons donc de:

- *HUD* (heads-up display), en français *ATH* (affichage tête haute) est un affichage qui se superpose à la caméra dans les jeux vidéo. Ils montrent des statistiques et des variables, comme ici dans notre cas où nous montrons la vie du joueur et les munitions qu'il possède.
- *Sprites*, en français *Lutins*, sont les textures des objets, monstres et joueurs que nous avons. Dans notre cas, ces sprites sont orientés en permanence vers nous.
- *Joueur "Serveur"*, principalement employé dans la partie sur le multijoueur, représente le joueur qui a décidé d'héberger la partie actuelle. Toutes les opérations de synchronisation sont effectuées par lui.
- *Joueur "Client"*, principalement employé dans la partie sur le multijoueur, représente le joueur qui a décidé de rejoindre la partie actuelle. Il envoie ses données au joueur "Serveur", qui va les traiter, et qui renvoie tout.
- *Localhost*, qui ne comporte pas spécialement de traduction en français, et l'adresse IP correspondante au réseau local. Elle est universelle et égale à "127.0.0.1". Elle est principalement employée dans la partie sur le multijoueur.
- Le FPS (First-Person Shooters), en francais jeu de tir à la première personne, est le style que nous avons choisi pour notre jeu. En effet, le jeu se déroule à la première personne.
- Le *Fast-FPS* (jeu de tir à la première personne rapide) , est un sous-genre du FPS dont les mécaniques sont majoritairement héritées de Doom. Parfois également appelé *Boomer Shooter* , ce sont des jeux de tirs avec une jouabilité fluide et qui misent sur les réflexes du joueur. C'est le genre de notre jeu.
- LAN (Local Area Network) est un type de multijoueur consistant à relier les appareils se trouvant sur le même réseau WiFi.

- Les *shaders*, en francais *nuanceurs* , sont des programmes permettant de calculer une meilleure absorption et une meilleure réflexion de la lumière car la trajectoire de cette dernière est calculée en temps réel par lesdits nuanceurs. Ces nuanceurs permettent d'obtenir un environnement 3D plus réaliste que ça soit pour de l'animation ou pour le jeu vidéo.
- Le *steampunk*, en *francais uchronie* vaporiste est un mouvement culturel ainsi qu'un genre artistique et littéraire issu de la science fiction. Ce genre s'illustre par une projection dans le futur depuis le XIXème siècle pour imaginer comment serait le futur mais avec des machines à vapeurs au lieu d'utiliser de l'électricité. D'où sa traduction en français par uchronie (désignant une réalité alternative à un moment de l'histoire) vaporiste.
- Les logiciels *Open Source* sont des logiciels dont le code et la redistribution sont accessibles à tout le monde, sans devoir payer une licence.

2. Projet

2.1. Présentation du projet

Ce projet de fin d'année, avait pour but de développer de A à Z un jeu vidéo en équipe de 5 personnes avec pour contraintes un mode multijoueur et une implémentation d'Intelligence Artificielle. Pour ce faire nous nous sommes mis d'accord pour développer Castle Of Demise.

Castle Of Demise est un *FPS* (jeu de tir à la première personne). Notre jeu possède une campagne composée d'un niveau et précédé d'un tutoriel ainsi qu'un mode multijoueur en arène en 1 contre 1. Notre objectif dans ce projet a été de proposer une expérience plaisante à tous les joueurs parcourant notre jeu. Que ça soit à travers la jouabilité, l'architecture des niveaux ou la joie de partager une expérience avec un proche à travers le mode multijoueur.

2.2. Origine et nature du jeu

2.2.1. Histoire qui a inspiré le projet

Ce qui suit est l'intrigue et le contexte que nous avions initialement prévu pour le déroulement du jeu mais que nous n'avons malheureusement pas été en mesure d'implémenter.

L'intrigue du jeu se situe dans une uchronie vaporiste, et plus précisément dans un village isolé du monde, du nom de *Leidenstadt*. Ce village est en proie à des attaques de vampires tous les cent ans. La source ? Le château du Comte Dracula. Il envoie chaque siècle son armée constituée de créatures tout droit sorties des enfers. Et à leur tête ses lieutenants qui sont des vampires qu'il a lui-même converti.

Son armée décimait continuellement les villages alentour. Tous sauf à l'exception de *Leidenstadt* qui depuis la nuit des temps a compté en son sein une lignée de chasseurs de vampires qui ont repoussé avec succès les forces de Dracula mais qui n'arrivent jamais à l'occire définitivement ce dernier étant reclus dans son château dans lequel aucun n'a pu pénétrer avec succès.

Malheureusement lors du dernier assaut le dernier représentant de cette lignée a péri sans descendance ce qui fait que la magie inhérente à cette lignée à disparu. Fort heureusement la disparition de cette lignée correspond au début de la révolution industrielle. Les villageois ne se laissant pas abattre vont pendant 100 ans confectionner des armes basées sur les progrès de cette ère ainsi qu'une armure permettant de résister aux attaques des créatures démoniaques et des vampires.

À chaque fois que l'armure est touchée, elle perd de la vapeur. Lorsque la quantité de vapeur atteint zéro, le porteur de l'armure devient vulnérable à toute attaque. Les porteurs de ces armures sont appelés les Chasseurs des Enfers. Cependant, malgré tout le temps investi dans la création de ces avancées, seules deux unités ont pu être formées. Malheureusement ces unités ne s'entendent pas très bien et forment une rivalité qu'on souhaitait mettre en scène dans le jeu initialement.

Ainsi le jeu débute à l'entrée de *Leidenstadt* dans un niveau qui sert de tutoriel et nous sommes face à une forêt avec un brouillard épais d'une couleur orangeâtre pour indiquer le cataclysme imminent et nous laisse penser qu'un incendie s'est déclenché pas loin. Quand on arrive à *Leidenstadt* on se rend compte que le village est en train de subir une attaque de la part de Dracula, le but est alors de repousser cette attaque et de pénétrer le château pour se débarrasser de Dracula une bonne fois pour toutes. Malheureusement nous n'avons pas pu implémenter la suite mais nous allons quand même vous raconter ce qui était prévu initialement.

Une fois dans le château, le personnage incarné par le joueur était censé rencontrer son rival qui l'avait devancé et qui était parti pour tuer Dracula et ses lieutenants avant lui. On était censé prendre une route différente explorer le château pour en apprendre plus sur l'histoire des lieux et surtout pour occire Dracula et ses lieutenants et faire en sorte qu'ils ne reviennent plus jamais. Nous avions également prévu initialement que l'avant-dernier niveau soit un combat à mort entre le joueur et le rival en raison d'un conflit d'idéaux. À l'issue de ce combat, le protagoniste sort victorieux mais changé et obtient la capacité permettant de battre Dracula.

2.2.2. Début de la création

Au tout début du projet, ayant plusieurs membres passionnés par le genre du *jeu de tir à la première personne*, nous nous sommes rapidement mis d'accord sur le genre du jeu qu'on voulait réaliser. Initialement nous nous étions mis d'accord sur *RétroRage* comme nom de jeu mais une fois que nous avons eu une idée plus claire de l'ambiance qu'on voulait donner le jeu nous nous sommes accordés pour nommer *le jeu Castle Of Demise*. Ce nom nous est venu assez naturellement après avoir imaginé le synopsis et le contexte derrière le jeu étant donné que le château devait initialement être l'élément

central. *Demise* signifie le déclin en anglais et met bien en lumière l'ambiance du jeu. La raison de l'utilisation de l'anglais pour le titre vient principalement de nos inspirations vidéoludiques à savoir *Castlevania* et *Doom* (qui est d'ailleurs un synonyme de *Demise*). En parlant d'inspirations, la prochaine partie va en faire état.

2.3. État de l'art

2.3.1. Influences techniques avec ID Software

Le développement de Castle of Demise s'inspire profondément des pionniers du genre du FPS, notamment des œuvres emblématiques produites par *ID Software*, une société fondée par *John Romero*, *Tom Hall*, *John* et *Adrian Carmack*. Cette entreprise a marqué l'histoire du jeu vidéo en donnant naissance à des titres phares tels que *Wolfenstein* 3D, *Doom* et *Quake*, qui ont eu une influence durable sur le genre des FPS, à tel point que pendant un certain temps, les jeux de ce type étaient communément désignés sous le terme de "*Doom-likes*", dont nous nous sommes inspirés de l'appellation pour notre jeu.

2.3.1.1. Wolfenstein (1981)

Wolfenstein 3D, sorti en 1992, est le premier FPS de l'histoire. Il se distingue non seulement comme étant l'un des tout premier jeu de l'histoire à utiliser un moteur de jeu en 3D, mais aussi par sa capacité à immerger les joueurs dans un environnement tridimensionnel grâce à la vue à la première personne. Cela a marqué une véritable révolution dans le monde du jeu vidéo, et *Wolfenstein* 3D a joué un rôle crucial dans la démocratisation du genre FPS. Sur le plan technique, le jeu a été novateur à l'époque, car il a utilisé des textures, des animations et des sprites en 2D, tandis que le décor était modélisé en 3D, une stratégie qui a également été mise en œuvre dans *Doom*. Cette approche a permis une optimisation et une adaptabilité du jeu à une variété de plateformes quasi infinies.

La communauté s'est lancé le défi d'adapter le jeu que l'on va aborder par la suite sur le plus de plateformes physiquement imaginables

2.3.1.2. DOOM (1993)

Doom, sorti en 1993, a poursuivi le travail technique initié par *Wolfenstein 3D*. Il a amélioré le level design en ajoutant plus de verticalité, offrant des environnements, des décors et un bestiaire plus soigné et varié que son prédécesseur. Le moteur de *Doom* est si complet et polyvalent qu'il est toujours utilisé par la communauté pour créer de nouveaux jeux et des mods, témoignant de sa longévité. On peut citer *Selaco* et *Prodeus*, deux jeux récents se basant pourtant sur ce moteur. *Doom* a rencontré un succès commercial massif, avec 3,5 millions de copies vendues entre 1993 et 1999, ainsi qu'une estimation de 20 millions de joueurs sur la même période, basée sur le nombre de téléchargements du premier épisode gratuit du jeu. Ce jeu a grandement contribué à la popularisation des *FPS*, et sa stratégie d'utiliser principalement des éléments en 2D a également inspiré la conception de Castle of Demise.

2.3.1.3. Quake (1996)

Quake, sorti après *Doom*, a apporté une révolution technique majeure en offrant un moteur qui, contrairement à celui de son prédécesseur, était capable de produire un rendu 3D en temps réel. *Quake* a également introduit la possibilité de sauter et de regarder de haut en bas, ajoutant ainsi plus de verticalité au gameplay. Ce jeu a également été l'un des premiers *FPS* de l'histoire à proposer un mode multijoueur sous la forme d'un Deathmatch jouable soit en ligne soit en *LAN*, fonctionnalité ayant grandement inspiré le mode multijoueur de notre jeu, dont le LAN sera aussi l'interface primaire.

D'autres jeux nous ont inspirés sur le côté artistique de Castle of Demise. *Ultrakill* et *Castlevania* font partie de nos plus grandes sources d'inspirations.

2.3.2. Influences Artistiques et Environnementales

2.3.2.1. Ultrakill (2020-Présent)

Ultrakill, un jeu de type *Fast FPS*, privilégiant ainsi la mobilité, est disponible en accès anticipé depuis 2020 et se déroule dans un univers post-apocalyptique. Ce jeu se distingue par sa narration environnementale, les échanges dialogués étant d'une immense rareté, ce qui contribue à instaurer une atmosphère de jeu où la concision communicationnelle prévaut. Le joueur incarne un robot doté de la faculté d'absorber le sang de ses adversaires pour régénérer sa propre vitalité. La narration s'articule

principalement à travers l'environnement du jeu, des éléments scripturaux judicieusement disséminés dans les niveaux, et des conversations sporadiques avec les rares personnages présents. Cette approche narrative, conjuguée aux caractéristiques esthétiques des protagonistes, a constitué une source d'inspiration significative pour Castle of Demise.

2.3.2.2. Castlevania (1986)

La série de jeux *Castlevania* se distingue comme une pionnière dans la création d'un genre vidéoludique bien distinct, communément désigné sous le terme "metroidvania". Il s'agit d'une saga qui a marqué un jalon essentiel dans l'histoire du jeu vidéo, en introduisant l'horreur gothique dans un contexte interactif. Cette franchise s'est particulièrement démarquée en s'inspirant de manière profonde de l'univers du roman classique "*Dracula*" de *Bram Stoker*, allant jusqu'à emprunter des éléments substantiels de l'intrigue, des personnages, et de l'atmosphère de cet ouvrage emblématique. Toutefois, c'est l'univers immersif et le bestiaire emblématique de la série "Castlevania" qui ont constitué une source d'inspiration majeure pour la conception de notre propre jeu vidéo.

L'ensemble de ces jeux a posé les bases techniques et artistiques pour le développement de Castle of Demise, en visant à offrir une expérience de jeu soignée, qualitative et ambitieuse, tout en rendant hommage à l'héritage des pionniers du genre FPS ainsi qu'aux jeux qu'à d'autres jeux qu'on tient particulièrement à cœur.

2.4. Organisation du Projet

2.4.1. Répartition et description des tâches

Pour faciliter et fluidifier l'avancement du jeu, nous avons opté pour une mise en place de rendez-vous fréquents (à raison d'un par semaine environ, sur la plateforme *Discord* ou sur le campus directement).

Tâches techniques (Programmation et Godot)

Tâche	**Responsable(s)**
Création de la carte de Tests	Paul L.
Création du Head-Up-Display (Pour le débogage)	Jean H.
Création de la Map multijoueur	Paul L.
Création de la Map Leidenstadt	Roni Y.
Moteur sonore (SFX / Musiques)	Jean H.
Menu Pause en jeu	Jean H.
Menu options dans le menu	Clément G.
Sauvegarde locale des paramètres	Clément G.
Paramétrage de la limite des FPS	Clément G.
Mouvements (Caméra, sauts, champ de vision..)	Jean H.
Création des ennemis (classe, sprites..)	Paul L. et Jean H.
Création de l'IA des ennemis	Paul L.
Création de la Map de tutoriel	Roni Y.
Cinématique du tutoriel (Musique, lévitation)	Jean H.
Gestion réseau du multijoueur	Amadéo H.

Conception du convertisseur “Code/IP”	Amadéo H.
Gestion des scores en multijoueur	Amadéo H.
Gestion de la gravité et de la téléportation	Jean H.
Écrans de fin de jeu (mort, victoire)	Paul L.
Système d’armes du joueur (tirs, effets de tirs)	Paul L.
Système de santé du joueur (dégâts, mort, *healing*)	Paul L.
Création d’objets ramassables (packs de vie / de munitions)	Paul L.
Shaders graphiques (effet “VHS” dans le menu pause)	Jean H.
Design des boutons (dans les menus)	Jean H.
Migration de Godot 3 à Godot 4	Jean H.
Design des objets 3D (maisons, pancartes, murets)	Roni Y.

Tâches organisationnelles (soutenances, administration)

Tâche	**Responsable(s)**
Chef du groupe	Jean H.
Site web	Jean H.
Création du repository Github	Amadéo H.

2.4.2. Outils et logiciels utilisés

Pour réussir notre projet, nous avons principalement utilisé deux logiciels: GitHub et Discord.

Le premier a été utilisé pour mettre notre projet dessus, pour pouvoir se le partager et fusionner tous les avancements sans problèmes car tout était automatique. GitHub nous permet aussi de souligner les problèmes que nous avons sans problèmes.

Le second a été utilisé car nous connaissions bien l'objet, et que nous pouvions communiquer plus aisément sur Discord que sur GitHub.

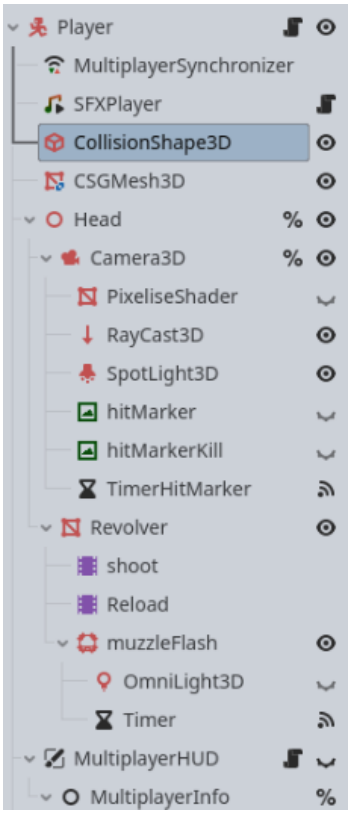
Afin de lier les deux services aisément, nous avons utilisé des *webhooks* GitHub que nous avons implémentés sur Discord. Nous avions donc en direct les informations sur chaque action que l'un d'entre nous faisait sur GitHub, leurs modifications, ce qu'ils ont envoyé, et si ça a été envoyé. Ça nous a été grandement utile durant toute l'année.

3. Moteur de jeu

3.1. Présentation de Godot

Godot est un moteur de développement de jeux *Open Source* donnant la possibilité de créer des jeux complexes et professionnels en 3D ou en 2D. Il vient avec un éditeur de code et sont langage propre, le *GDScript*. Ce langage est basé sur *Python* et est imaginée comme un outil simple est accessible pour les débutants, il intègre toutes les fonction de godot nécessaire au développement de jeu ainsi que d'une documentation intégrée. Nous avons cependant utilisé la version *.NET* de *Godot*, cette version nous permet de développer en utilisant le *C#*. Ce langage est bien plus professionnel et poussé. Bien que godot n'est pas était initialement prévue avec le *C#* en tête toute la fonction sont bien intégrée et la documentation d'une qualité similaire à celle en *GDScript*.

Godot est également unique dans son fonctionnement. Tout tourne autour des nœuds, chaque fonction d'un jeu est représentée avec un nœud, par exemple un modèle 3D sera un nœud *Mesh Instance 3D*, un son sera un *Audio Stream Player*, une collision est une *Collision Shape 3D*, etc. Les cartes , les ennemis comme les effets sont tous des groupes de nœuds, tous ces nœuds sont organisés hiérarchiquement en tant que père ou fils et ses groupes de nœuds sont appelés des scènes (des fichiers *.tscn*).



Ci-contre vous pouvez voir une partie de la hiérarchie des nœuds qui compose notre joueur. Chacun de ces nœuds possède plusieurs paramètres propres. Par exemple, un timer possède une durée, un paramètre dit "*OneShot*" faisant que le timer se répète ou au contraire, qui ne va se jouer qu'une seule fois.

Il est possible d'attacher des script a des nœuds dans le but de leur définir un comportement et des proprieté plus poussée. Par exemple, le nœud "*Player*" possède un script qui lui dicte comment se comporter, exemple lorsque "Z" est pressé tu dois avancer. Il est possible de donner un script à n'importe quel nœud.

Figure 2 : Noeud "*Player*"

Godot met également à disposition les signaux. Chaque nœud possède des signaux spécifiques qui peuvent être utilisés pour communiquer certaines informations au script. En reprenant l'exemple du Timer, nous voulons déclencher une certaine action une fois que ce timer a atteint zéro, pour ce faire nous utilisons le signal timeout() qui se définit comme une méthode classique mais qui communique avec le nœud Timer qui appelle cette méthode lorsqu'il a atteint zéro.

3.2. Pourquoi Godot ?

Godot est un moteur de jeux polyvalent et complet, le tout contenu dans un seul fichier .exe, il est possible de faire fonctionner godot sur n'importe quelle ordinateur ou téléphone portable.

De plus, Godot est un moteur qui devient de plus en plus populaire auprès des petits créateurs de jeux vidéo. De se fait un grand nombre de tutoriel récent est disponible, bien que la majorité utilise le GDScript il n'est pas dur de faire la "traduction" en C#.

Enfin, certains de nos membres étaient déjà familiarisés avec godot ce qui nous a donné une base de départ sur laquelle travailler.

Bien que Godot ne soit pas le moteur de jeu le plus répandu, nous avons choisi de l'utiliser pour sa simplicité et pour le fait qu'il est simple de le faire fonctionner sur la plupart des ordinateurs.

3.3. Difficultés rencontrées

3.3.1. Godot 3.5.3

Au début, nous avions opté pour la version 3.5.3 de Godot, nous pensions que prendre une version plus ancienne et donc plus stable était une bonne idée, car la version 4 de Godot était plus récente. Cependant au fil de l'avancement du projet et avec les conseils de notre professeur de programmation nous avons décidé de convertir notre projet de Godot 3.5.3 a 4.2.2. Ce changement n'était pas prévu et nous avons dû prendre le temps de réécrire quelques fonctions pour que notre jeu puisse fonctionner. Cependant ce changement a été bénéfique, tout d'abord nous en avons profité pour optimiser et réorganiser notre code pour rendre ce dernier un peu plus lisible. Ensuite

Godot 4 est une mise à jour qui apôtre plusieurs nœud liée au multijoueur très utile, comme les *MultiplayerSynchronizer*, *MultiplayerSpawner* et d'autres. Tous ces nouveaux nœuds nous ont été extrêmement utiles pour le développement du multijoueur.

4. Interfaces et mécaniques du Jeu

4.1. Mécaniques de jeu

4.1.1. Kinematic Body

Afin de pouvoir déplacer le joueur en trois dimensions, Godot met à disposition plusieurs nœuds permettant de manipuler la position d'une scène ou d'un nœud fils. La méthode que nous avons choisie est le *Kinematic Body*.

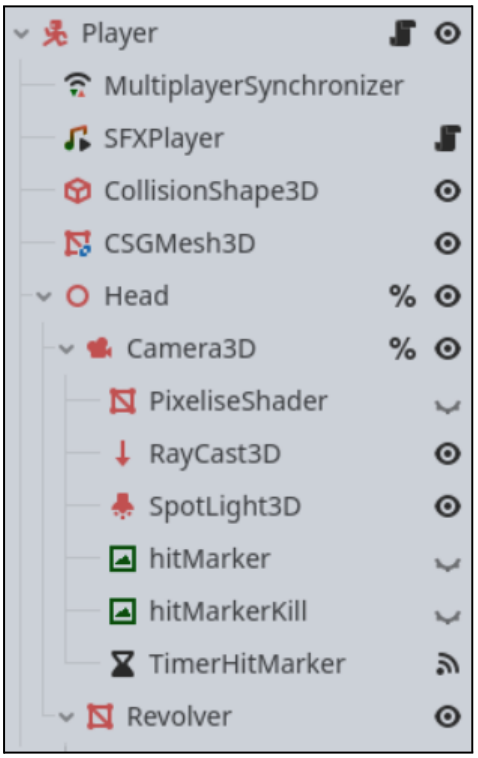


Figure 3 : Noeud *Player*"

Cette méthode possède plusieurs caractéristiques. Tout d'abord, contrairement au Rigid Body qui est contrôlé indirectement en appliquant des forces passant par le moteur physique de *Godot*, le *Kinematic Body* est entièrement contrôlé par du code. Grâce à ça il est bien plus adapté à un avatar contrôlé par l'utilisateur contrairement au *Rigid Body* plus compatible avec des objets avec lesquels le joueur peut physiquement interagir.

Pour implémenter les mouvements, tous les nœuds caractérisent le joueur (la caméra, le modèle, les *Raycast*, ...) sont tous des nœuds fils d'un nœud kinematic body nommée "*Player*" (comme vus sur l'image ci-dessous). De cette manière tous les nœuds fils de Player se déplacent en même temps et au même endroit que le Kinematic Body.

4.1.2. Déplacement

Le *Kinematic Body* étant contrôlé par code, nous avons implémenté plusieurs méthodes dédiées au contrôle du joueur.

Tout d'abord, vous pouvez voir ci-dessous la méthode gérant les appuis de touches de clavier qui font déplacer le personnage (ici Z, Q, S et D). Une fois que la touche appuyée est détectée par la méthode Input. Cette méthode incrémente ou décrémente le vecteur de l'axe Y ou l'axe X (ici nous nous penchons seulement sur les déplacements horizontal d'où l'utilisation d'un Vector2) avant d'appliquer le mouvement ce dernier est multiplié par l'orientation globale de la caméra, de cette façon le joueur avancera dans la direction ou il regarde.

Afin de pouvoir regarder les environs, nous avons ajouté la possibilité de tourner la caméra grâce à des mouvements de souris. De plus, afin de rendre le mouvement plus naturel, nous avons mis en place un système de champ de vision dynamique inspiré de Minecraft. Lorsque le joueur avance, le champ de vision s'agrandit, ce qui donne une sensation de vitesse. Pour les mêmes raisons, nous avons également ajouté une légère rotation de la caméra vers la gauche ou la droite en fonction des mouvements latéraux du joueur, inspirée du moteur *gldSrc* (*Gold Source*) développé par *Valve*, notamment utilisé pour le jeu *Half-Life*. En se basant sur la même inspiration nous avons fait en sorte que le joueur continue de glisser une fois la touche directionnelle relâchée en décrémentant la vélocité du joueur dans le sens inverse du mouvement implémentant ainsi une mécanique d'inertie.

4.1.3. Les armes

Notre jeu étant un jeu de tir nous avons dû ajouter un moyen de tirer. Nous avons choisi un système de rayon avec son origine au centre de la caméra du joueur, qui, contrairement à d'autres systèmes, rend le temps de parcours de la balle instantané et non soumis à la gravité, ce qui n'est pas important pour notre jeu basé sur des combats rapprochés. Nous avons donc choisi cette méthode pour sa simplicité et sa fiabilité. Afin de rendre le tir plus satisfaisant et intuitif nous avons ajouté un modèle de revolver, des impacts et des particules, de fumée ou de sang en fonction de la cible, apparaissant à l'impact d'une balle grâce au système de particule intégré à Godot, une animation de tir se jouant à chaque clic de souris et enfin un bruit de coup de feu.

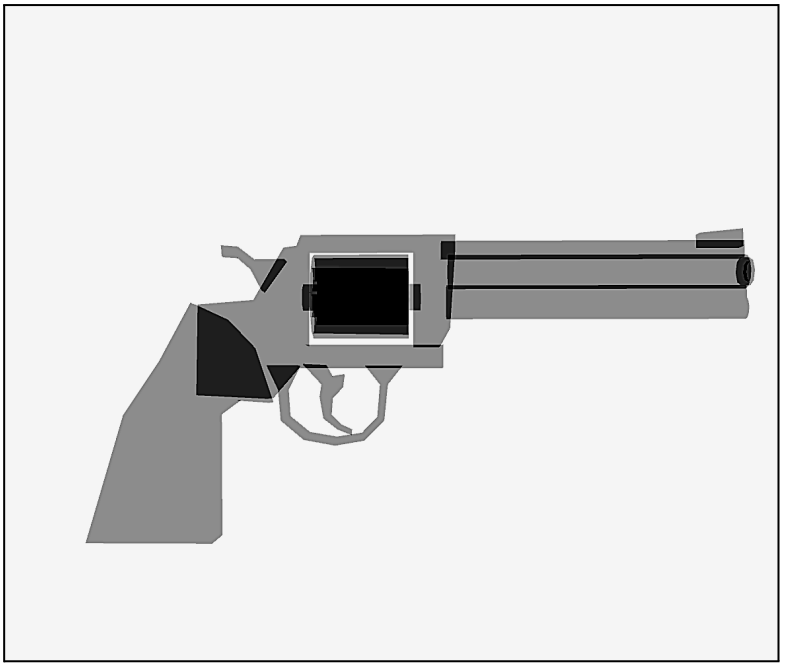


Figure 4 : Image transposée du modèle 3D du revolver

Cette animation de tir se constitue d'un tremblement de la caméra du joueur, d'un mouvement du pistolet et d'une émission de lumière au bout de celui-ci. Nous avons également ajouté un compteur de munitions qui empêche le joueur de tirer une fois qu'il a atteint zéro. L'arme étant un révolver, le compteur a une limite de six balles, mais cela fait peu, nous avons donc créé un système de recharge du revolver ainsi qu'un deuxième compteur pour savoir le nombre de munitions restantes en dehors du barillet. Ce système de recharge est, tout comme celui du tir, accompagné d'une animation et d'un effet sonore. Il est également impossible de recharger si le revolver contient déjà les six munitions maximales.

4.1.4. Ressources Récoltables

Lors des parties en solitaires, il est possible de ramasser deux types d'objets, un nous permettant de récupérer dix points de vie et l'autre nous donne douze munitions si nous ne sommes pas déjà au maximum de munitions, qui sont de soixante.

Figure 5 : Pack de soin

Figure 6 : Pack de munitions

4.1.5. Le son

4.1.5.1. Effets sonores (Sound SFX) et Musiques

Afin de pouvoir gérer uniformément les effets sonores du jeu, nous avons pris le parti d'implémenter, pour chaque nœud pouvant émettre un son, un nœud *AudioStreamPlayer* lié à une classe *SFXPlayer*, que nous avons créé.

La classe *SFXPlayer* permet de lier l'objet parent à un fichier audio, de le jouer et de gérer le volume des effets sonores. Cette centralisation des effets sonores permet de simplifier la gestion des sons dans le jeu, en évitant de devoir charger les fichiers audio par chemin absolu à chaque fois que l'on souhaite jouer un son.

La méthode PlaySFX permet de jouer un son en fonction de son type, en choisissant aléatoirement un fichier audio parmi ceux disponibles dans le dossier correspondant. La musique est gérée de manière similaire, avec un nœud *AudioStreamPlayer* dédié à la musique et assignée à son bus audio correspondant.

Voici l'arborescence du répertoire *SoundEffects* (figure 7) :

```
.
├── AmbientNight
│   └── AmbientNight.wav
├── DiesIrae
│   └── DiesIrae.wav
├── buttons
│   ├── focused
│   │   └── focused_button.wav
│   ├── pressed
│   │   └── pressed_button.wav
│   └── released
├── death
│   └── death.wav
├── gun
│   ├── cantshoot
│   │   └── CantShoot.wav
│   ├── gunshot
│   │   ├── gunShot01.wav
│   │   ├── gunShot02.wav
│   │   └── gunShot03.wav
│   ├── hit
│   │   └── hit.wav
│   └── reload
│       └── WeaponReload.wav
├── jump
│   └── Jump.wav
├── landed
│   └── Land.wav
├── pickupable
│   ├── ammo
│   │   ├── explosion.wav
│   │   └── hitHurt.wav
│   └── health
│       ├── healthPickupSound.wav
│       ├── powerUp (1).wav
│       └── powerUp.wav
├── playerhit
│   └── PlayerIsHurt.wav
└── steps
    ├── Step01.wav
    ├── Step02.wav
    ├── Step03.wav
    ├── Step04.wav
    └── Step05.wav
```

4.2. Interfaces

4.2.1. Menus hors jeu

4.2.1.1. Menu Principal

Parmi les menus hors partie que nous avons développés, le premier que l'on voit est le menu principal. Le nom de notre jeu est indiqué dans la partie haute de la fenêtre. Pour rendre la fenêtre d'accueil plus accueillante, nous avons également ajouté un fond flou représentant une image du jeu et placé notre logo en bas à gauche de la fenêtre.

Ces deux images forment l'arrière-plan de tous les menus hors partie. Le menu principal est constitué de quatre boutons principaux, le premier indiquant "*QUITTER*" nous permet de fermer le jeu, le deuxième indiquant "*OPTIONS*" nous permet de changer de scène et d'accéder au menu pour modifier les options, le troisième nommé "*JOUER*" permet d'accéder au menu permettant de choisir le mode auquel nous voulons jouer et enfin le dernier indiquant "*TUTORIEL*" mène au niveau de tutoriel. Lorsque l'on change de menu, on change la visibilité du menu actuel en caché et on passe le menu dans lequel nous allons en visible.

Lorsque l'on clique sur un bouton censé nous rendre sur une carte, on utilise la fonction "*ChangeSceneToFile*" et prend en paramètre le chemin de ladite carte dans notre projet.

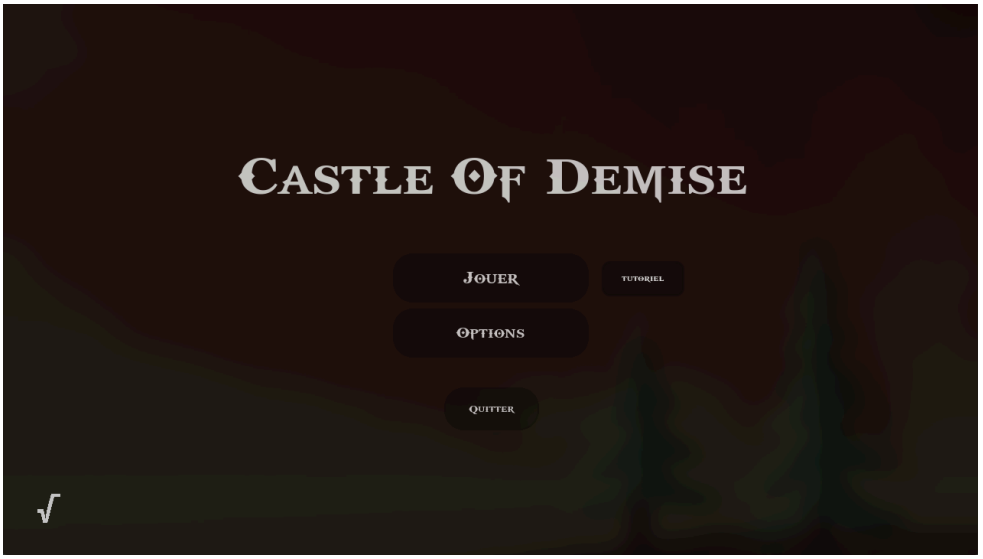


Figure 8 : Menu Principal

4.2.1.2. Menu Options

Le menu *Options* permet tout d'abord de revenir à l'écran d'accueil avec le bouton "*Retour*" mais, surtout il contient trois onglets dans lesquels il nous est possible de modifier différents aspects du jeu.

Dans l'onglet *Contrôles*, il est possible de modifier les touches nous permettant de nous déplacer, de recharger le pistolet, de faire pause ou reprendre la partie et de révéler ou cacher l'affichage tête haute du joueur.

Pour modifier une touche, il suffit de faire un clic gauche sur la touche à modifier puis de cliquer sur la nouvelle touche que l'on veut utiliser et cela se sauvegarde directement. On ne peut cependant pas modifier une touche par un bouton de la souris.

Figure 9 : Menu Options (Contrôles)

L'onglet Audio quant à lui, nous permet, grâce à des barres de son modifiables de changer les réglages sonores pour s'adapter aux envies du joueur. Il y a trois barres différentes, chacune servant à régler différents paramètres.

La première nommée "Master Volume" permet de modifier le volume de tous les sons du jeu sans exception. La deuxième est appelée "Musique Volume" et permet d'ajuster le volume des musiques présentes dans le jeu. Enfin la dernière nommée "SFX Volume" permet de changer le volume des différents effets sonores du jeu. Dans un souci de précision, à droite des trois barres de son se trouve un pourcentage indiquant

le niveau sonore actuel que la barre indique, cela permet de ne pas avoir une simple estimation du volume sonore.

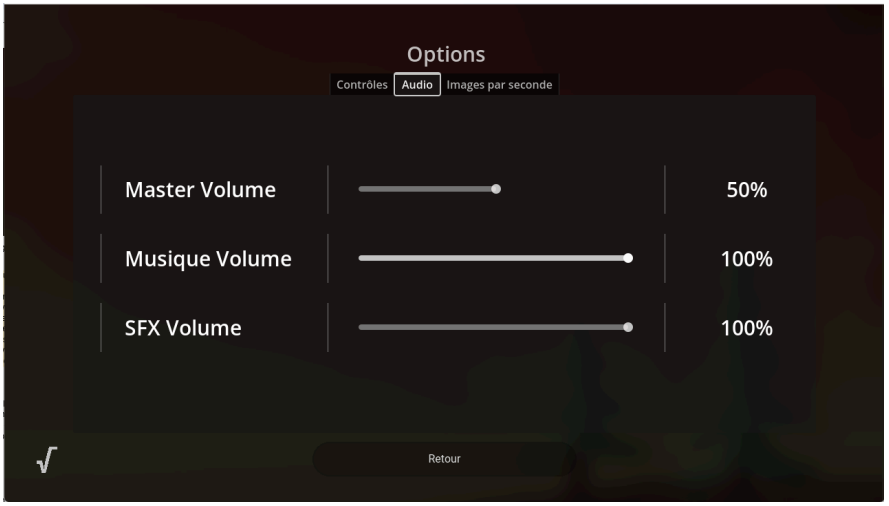


Figure 10 : Menu Options (Audio)

Le dernier onglet, l'onglet Images par seconde, nous laisse choisir le maximum d'images par seconde pouvant être atteint en jeu. Il se compose d'un unique bouton à choix multiples permettant de choisir entre un maximum de 120, 60 ou 30 images par seconde. Cela permet de s'adapter aux différentes générations d'ordinateurs.

```
[Diff]

diff=1.0

[FPS]

fps=1.0

[Audio]

Master=1.0
Musique=0.588
SFX=1.0

[keybinding]

key_z="Z"
key_s="S"
key_q="Q"
key_d="D"
key_space="Space"
key_escape="Escape"
hideHUD="F3"
key_r="R"
```

Tous les changements et modifications effectués dans le menu options sont enregistrés dans un fichier texte nommé "*settings*" situé dans un dossier dans l'ordinateur de l'utilisateur.

Cela permet de conserver ces modifications même lorsque l'on ferme le jeu et ils sont ensuite lus et chargés lors de démarrage du jeu. Cela permet de garder les préférences du joueur et ne pas forcer l'utilisateur à les modifier à nouveau à chaque fois qu'il redémarre le jeu.

Figure 11 : Fichier settings

4.2.1.3. Menu de Sélection du mode de jeu

Le bouton “JOUER” du menu principal nous envoie sur ce menu permettant de choisir les différents modes de jeu. Ce menu est composé de l’arrière-plan et de trois boutons. Le bouton “Retour” nous permet également de revenir en arrière sur le menu principal, le bouton “Jouer seul” nous envoie directement sur la carte de la campagne solitaire et enfin le bouton “Jouer à plusieurs” nous donne accès au menu multijoueur.

4.2.1.4. Menu Multijoueur

Ce menu permet de choisir d’accéder au menu d’hébergement, de revenir au menu précédent ou de rentrer le code de la partie que l’on veut rejoindre puis de cliquer sur le bouton “Rejoindre” avant que l’hôte lance la partie.

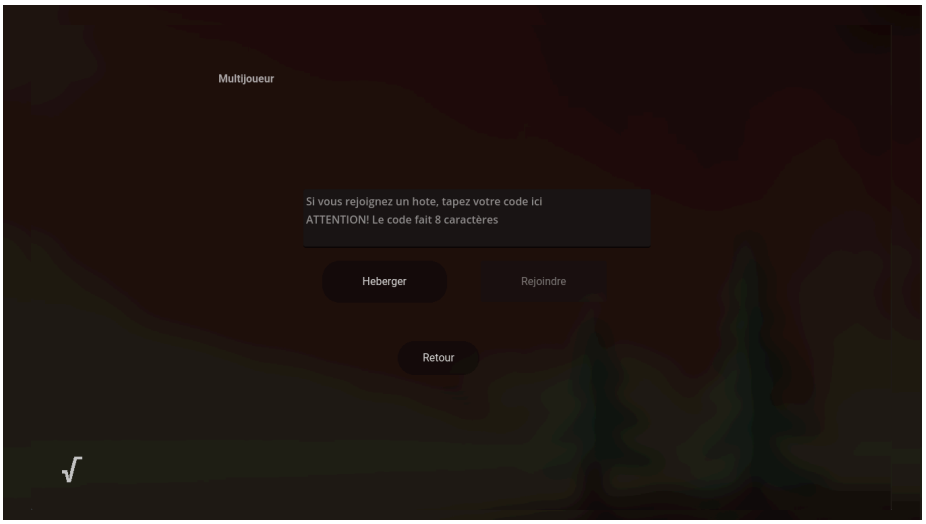


Figure 12 : Menu Multijoueur

4.2.1.5. Menu d’Hébergement

Comme les autres menus, ce dernier menu nous permet également de revenir au menu précédent. Il affiche aussi le code à sélectionner pour permettre à un autre joueur d’accéder à notre partie. Enfin, ce menu nous permet de sélectionner le mode

multijoueur Match à Mort ainsi que le score nécessaire pour gagner la partie puis de lancer la partie après que l'autre joueur l'ait rejoint, mais nous développerons l'explication dans la partie multijoueur.

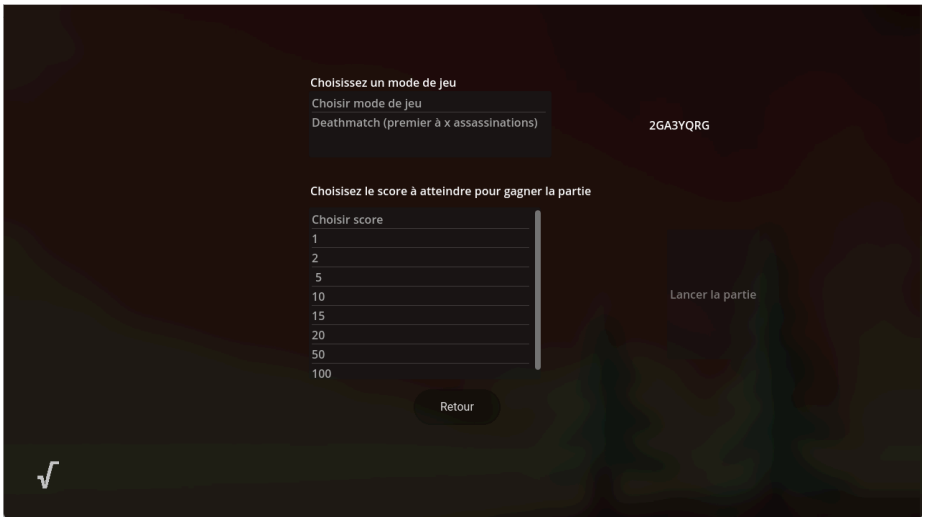


Figure 13 : Menu d'Hébergement

4.2.2. Menus fonctionnels en jeu

4.2.2.1. ATHs

Dans l'intégralité du jeu, les menus à Affichage Tête Haute sont présents au nombre de trois. Le premier permet d'afficher le nombre de points de vie du joueur et son nombre de munitions hors et à l'intérieur du chargeur. Le deuxième ATH est dans le multijoueur et affiche le score à atteindre pour gagner, le nombre de fois que l'on a tué son adversaire et le nombre de fois que l'adversaire nous a tué.

Le troisième et dernier ATH n'est visible que si l'on appuie sur la touche F3 ou une autre touche si elle à été modifiée précédemment dans le menu option. Cet ATH, lorsque affiché nous permet de voir la position du joueur, son accélération, son orientation, si l'on est en multijoueur ou non, la vie du joueur, le nombre d'images par seconde, le nombre de balles tirés, les dégâts effectués par un tir et enfin le nombre de monstres ou adversaires qui ont été tués. Ce menu a beaucoup servi durant le débogage.

4.2.2.2. Menu Pause

Lorsque l'on lance une partie, il nous est possible de mettre pause et donc d'afficher un menu fonctionnel durant la partie. Ce menu qui était déjà considéré complet lors de la soutenance de mars à tout de même été amélioré au niveau des graphismes et de la présentation.

Il permet d'effectuer plusieurs actions, telles que des nuanceurs customisées, comme le degré de pixellisation ou encore mettre le jeu en plein écran, qui donnent une immersion différente en fonction de nos préférences.

Le menu contient également des paramètres de son, nous permettant de modifier le volume de la musique et des effets sonores, tels que les tirs, les dégâts des ennemis, mais aussi le bruit des mouvements.

4.2.2.3. Menu de Mort

Si l'on est tué au cours de cette partie, un menu de mort s'ouvre. Ce menu nous indique que l'on est mort avec un texte "Vous êtes mort" et a pour arrière-plan une texture de mur entouré de rouge pour représenter le sang.

L'arrière-plan est pixelisé pour coller aux graphismes du reste du jeu. Il y a également un bouton permettant au joueur venant de perdre la partie de revenir au menu principal.

Figure 14 : Menu de mort

4.2.2.4. Menus de Victoire

Si il est possible de perdre en mourant, il est également possible de gagner, soit en tuant tous les monstres lorsque l'on joue en solitaire, soit en atteignant l'objectif avant votre adversaire en multijoueur. Lorsque cela arrive, le joueur est déplacé vers un menu de victoire.

Dans les deux cas, l'arrière-plan est identique aux autres menus hors jeu et nous permettent de revenir au menu principal. La seule différence entre ces deux menus est le texte affiché sur l'écran.

Figure 15 : Menu de victoire mode solitaire

5. L'environnement de jeu

5.1. Descriptif de l'environnement de jeu réalisé

5.1.1. Les cartes

Le jeu dispose de plusieurs cartes qui ont différentes fonctions.

Tout d'abord la carte de démo qui était initialement notre carte de débogage. C'est la toute première carte qu'on a modélisée pour tester les mécaniques de jouabilité, le moteur physique et le moteur de son. C'est une carte assez sommaire qui dispose des textures et des assets qu'on a créés pour les tester afin de vérifier qu'ils ne causent pas de soucis. Cependant au fil du développement nous avons peu à peu délaissé cette map en testant nos textures et modèles 3D directement sur les cartes concernées c'est pourquoi nous l'avons gardé en tant que carte démo.

Ensuite nous avons la première carte multijoueur. Cette carte a surtout servi en tant que carte de test et n'est plus présente dans le jeu final mais est toujours là dans les fichiers.C'est une carte assez petite qui dispose d'un pilier son centre, ce pilier dispose d'un tunnel à quatres entrées qui permet de brouiller son adversaire.Tout autour de ce pilier sont disposé d'autres piliers ainsi que des rampes qui permettent de sortir de la carte et de sauter dans le vide.Ce qui est une bonne stratégie, sachant que sauter dans le vide provoque une inversion de la gravité qui nous fait revenir à la surface. Tout autour du pilier , est également disposée une petite forêt qui n'a aucune collision qui permet uniquement de se masquer de la vue de l'adversaire.

Cette carte nous a également permis de créer la deuxième en tant qu'amélioration de celle-ci pour qu'elle soit plus grande et plus amusante en multijoueur. Cette carte fait le double de la première et reprend tous les éléments de celle-ci. Elle dispose néanmoins de plus de piliers et de rampes mais aussi de plateformes pour pouvoir sortir de la carte ainsi que d'autres forêts servant la même fonction que dans la première. Le manque d'obstacle contraint au mouvement constant. Cette carte garde la mécanique d'inversion de gravité une fois dans le vide. Cette mécanique peut certes permettre de se cacher, mais elle force néanmoins le joueur en dedans à chercher le joueur adverse. Une fois qu'ils sont tous les deux dans le vide ils ne peuvent plus se viser correctement et sont donc contraint à revenir sur la carte pour continuer le combat.

La quatrième carte est la carte de tutoriel. C'est la première carte du mode solo et elle était initialement prévue en tant que prologue au jeu pour immerger dans

l'ambiance de celui-ci, tout en apprenant au joueur les mécaniques de Castle Of Demise. Cette carte est constituée d'une sorte de corridor, un chemin entouré par la forêt, qui mène au premier vrai niveau. Ce chemin est pavé de pancartes indiquant au joueur les touches qu'il doit utiliser à un instant T. On peut voir par exemple dans l'image qui suit, un muret qui bloque la progression du joueur et une pancarte qui indique qu'on peut utiliser la touche espace pour le franchir.

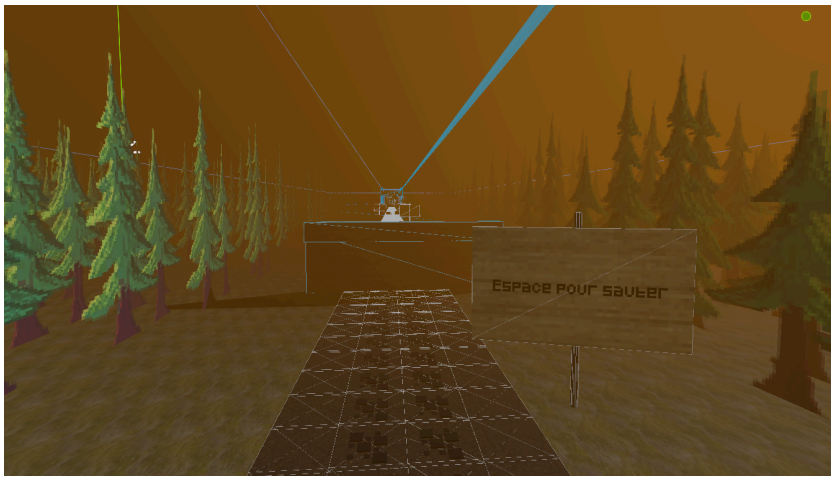


figure 16 : Capture d'écran du tutoriel

Cette carte sert donc d'initiation au joueur tout en le mettant dans l'ambiance par le brouillard orangé et les bruits de fonds qui sont un enregistrement fait de nuit en pleine nature. À la fin du chemin se trouve une pancarte qui nous accueille dans le village du début de jeu avant que la caméra ne prenne son envol pour une cinématique qui sert d'intro au jeu.

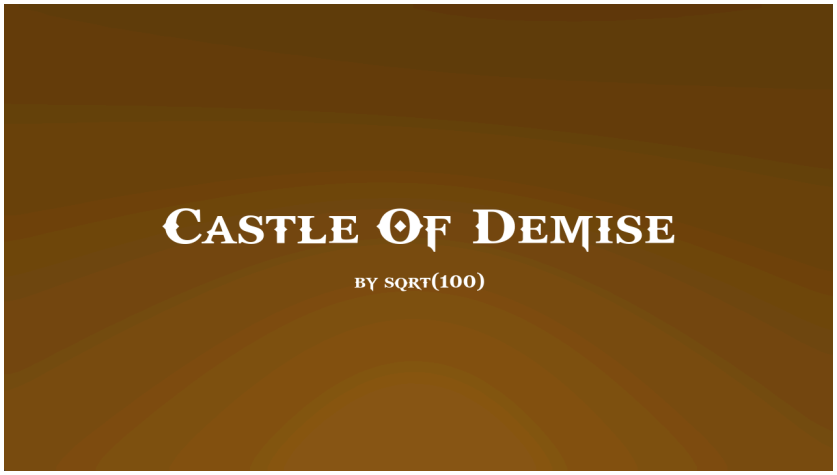


Figure 17 : Capture d'écran durant cinématique

Enfin, nous avons la carte du premier vrai niveau de la campagne du jeu, à savoir *Leidenstadt*. Le nom veut dire ville de la souffrance en allemand (et c'est également une référence à une chanson de *Jean-Jacques Goldman*). Cette carte reprend la fin de la carte de tutoriel (une partie du chemin et la pancarte) et nous lâche directement dans le village avec des chauves-souris qui nous agressent. Elle est constituée de plusieurs maisons, certaines sont accessibles et ont les fenêtres transparentes et d'autres ont les fenêtres embrumées et ne sont pas accessibles. Cette carte dispose également d'un puits dont le toît peut nous permettre de nous couvrir et d'accéder au toît des maisons environnantes. Les maisons accessibles contiennent des munitions et des kits de soin ainsi que des monstres qui nous en empêchent l'accès. Le but est de tuer tous les ennemis. Ce qui nous renvoie à l'écran titre.

Nous avons vu que *Leidenstadt* a été envahi par des monstres. Mais comment ont été créés ces monstres?

5.1.2. Les monstres

Dans notre projet, les ennemis sont représentés par des monstres. Ces monstres ont un modèle en deux dimensions. Ces modèles sont constamment orientés vers le joueur grâce à une option interne de *Godot* ce qui permet d'avoir une incrustation convenable d'éléments 2D dans un univers 3D. Nous avons deux modèles de monstres, un modèle de zombie et un de chauve-souris. Ces deux modèles ont une boîte de collision, des particules de sang lorsqu'ils se font toucher et une animation de mort. Les différents modèles de monstres ont un nombre de points vie défini en fonction de leur classe. Dans un premier temps pour que les ennemis nous suivent nous avons employé un système de *tracking* qui faisait que les monstres suivait systématiquement le joueur sans qu'il y ait une reconnaissance des obstacles.Mais par la suite nous avons opté pour un système de pathfinding qui est un type d'intelligence artificielle qui sera détaillé plus loin dans le rapport. Il aurait été possible de faire une plus grosse variété de monstres avec des patterns de déplacement plus complexe, mais nous avons estimé qu'il vaudrait mieux de présenter quelque chose de simple mais fonctionnel.

5.1.3. Les Modèles 2D (lutins et textures)

En termes de modèles 2D, nous avons les monstres dont on a parlé dans la partie précédente qui ont des lutins et des animations distincts. Pour ces lutins nous nous sommes fortement inspirés de modèles trouvé sur internet que nous avons retexturé.

Voici ci contre les différents dessins pour les zombies et chauves souris :

Figure 18 : Lutin de zombie

Figure 19 : Lutin de chauve-souris

Nous avons également les textures pour les murs, sols et pour les pancartes qui ont été réalisées en s'inspirant d'images en gros plans des textures dont on avait besoin. On a repris ces images puis on a baissé substantiellement leurs définitions d'images pour donner un effet pixelisé.

5.1.4. Modèles 3D

Nous avons quelques modèles 3D que nous avons récupérés comme le revolver ou les pierres incrustés dans les chemins, mais nous avons modélisé les autres modèles en utilisant la possibilité de créer des scènes 3D de *Godot*. Notamment les maisons qu'on peut voir dans *Leidenstadt* utilisant des textures que nous avons déjà importées pour le jeu, et un habile mélange de différentes formes géométriques. Nous pouvons également citer les tables présentes dans lesdites maisons modélisées de la même manière mais aussi le puits présent également à *Leidenstadt*. Les chemins ont également été réalisés de la même manière.

6. Le mode multijoueur

La partie multijoueur de Castle Of Demise est une partie que nous pouvons incontestablement définir comme une des parties les plus difficiles de ce projet.

Cette partie décrit l'entièreté de l'avancée du multijoueur de Castle Of Demise. Nous l'avons divisé en 2 sous-parties. Nous verrons en premier le descriptif du fonctionnement du multijoueur. Puis, nous décrirons les diverses difficultés affrontées durant le développement du mode multijoueur.

6.1. Descriptif de l'état actuel du mode multijoueur

Actuellement, nous avons un système multijoueur fonctionnel. Nous pouvons recenser les diverses parties qui composent le mode multijoueur, qui sont l'interface de connexion et de lien *Serveur/Client*, le lancement de la carte multijoueur et la synchronisation des joueurs, et finalement, le jeu en lui-même.

Le descriptif de la partie multijoueur est ainsi composé de deux parties: le système multijoueur qui assure la connexion entre les joueurs au niveau technique, et la partie qui assure la jouabilité de notre jeu.

6.1.1. Le système multijoueur

Commençons par parler du système multijoueur. Cette partie recense plusieurs fichiers, *Player.cs* pour la gestion du joueur dans l'environnement multijoueur, *MultiplayerMenu.cs* pour la gestions des ports ENet, *GameManager.cs* pour la gestion des joueurs, *IPParser.cs* pour la gestion de l'encodage, *SetupGameAshost.cs* pour la gestion de la jouabilité et *mpMap02.cs* pour l'apparition des joueurs.

Nous allons suivre, lors de cette partie, ce qui se passe dans votre ordinateur lorsque vous sélectionnez le multijoueur, en commençant par l'encodage.

6.1.1.1. Encodage

L'encodage est la première étape de notre voyage dans le multijoueur, et est l'étape la plus facile. Lorsque un joueur décide de lancer la partie en tant qu'hôte, le jeu va prendre l'adresse IP privée du joueur, et la transformer en code à 8 caractères composé de lettres et de chiffres, puis va le donner à ce joueur. Pour plus de clarté, nous allons appeler ce joueur "*Serveur*" par la suite. Ensuite, l'autre joueur, que nous nommerons donc "*Client*", va devoir rentrer le code, puis cliquer sur le bouton "Rejoindre". Ce qui lancera l'étape suivante, la détection des joueurs. Mais avant cela, voyons plus en détail comment fonctionne cet encodage, et son utilité.

Le fait d'utiliser de l'encodage pour notre jeu à la place de mettre un multijoueur efficace présente 2 avantages: il y a un contrôle sur qui peut venir, si vous ne donnez pas votre code, il sera plus difficile de rejoindre la partie, mais aussi d'effectuer plusieurs parties en même temps.

Ainsi, on peut avoir deux personnes, que nous nommons "*Serveur1*" et "*Serveur2*" qui vont chacune ouvrir leur serveur, et qui pourront inviter respectivement "*Client1*" et "*Client2*" sans problèmes de conflits.

Lorsque un joueur-serveur va créer sa partie, son adresse IP sera convertie en un nombre entre 0 et 10^{12}. Ce nombre passera alors en base 36, c'est-à-dire un mot composé des 10 chiffres et des 26 lettres de l'alphabet. Par exemple, l'adresse "192.168.62.175" deviendra "2GA3WHML", pendant que l'adresse "127.0.0.1", aussi appelée "*localhost*" deviendra "1MCCIR5T".

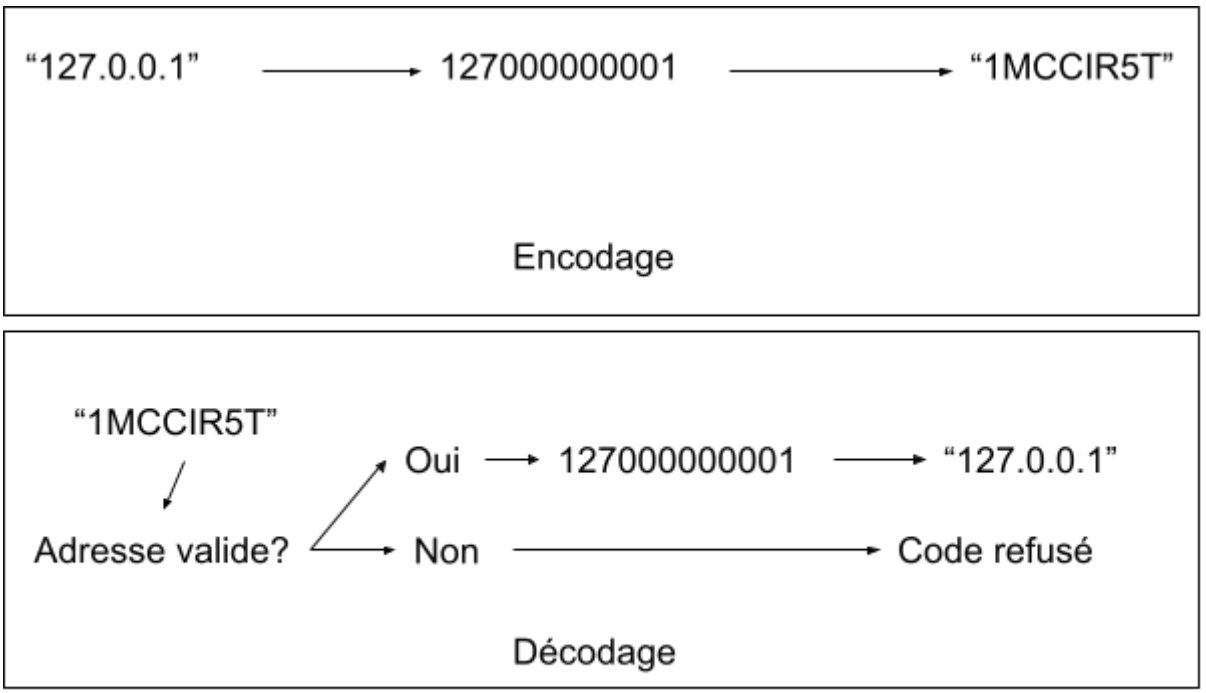


Figure 20 : schéma de l’encodage du multijoueur

Ensuite, un joueur “*Client*” pourra entrer ce code lorsqu’il se trouve sur le menu multijoueur. Nous avons de plus intégré une sécurité qui vérifie que le code fait la bonne longueur, c'est-à-dire 8 caractères, sans quoi le joueur n’aura pas la possibilité de cliquer sur “Rejoindre”.

Une fois ce bouton cliqué, le code entrée passe un niveau de vérification. Tout d’abord, il va mettre tous les caractères en majuscule, c’est-à-dire que le code “1MCCIR5T” correspond à la fois à “1MccIr5t” et à “1mccir5t”. Ensuite, une vérification de sécurité est passée: les adresses IP sont réglementées, et doivent appartenir à une classe, A, B ou C, qui correspondent chacune à des nombres différents. Si elle ne fait partie d’aucune de ces classes, l’adresse IP n’est pas privée, et donc le code est mauvais. Finalement, une fois cette sécurité passée, elle est envoyé au serveur afin qu’il soit détecté.

6.1.1.2. Détection des joueurs

Avant de décrire le fonctionnement de la détection, nous devons présenter deux fonctions de Godot pour le multijoueur qui sont très importantes pour la suite de ce chapitre:

- *ENetMultiplayerPeer*, abrégé *ENet* est la fonctionnalité principale de godot pour gérer les *peers* multijoueurs. C'est le système qui va faire la connexion entre deux appareils informatiques, des ordinateurs Windows dans notre cas, que nous nommons *peers*. Nous l'utilisons pour créer le serveur et le client simultanément.

- Les fonctions *Rpc* sont des moyens d'appeler des fonctions, mais qui, au lieu d'appeler en local uniquement, vont appeler sur tous les appareils connectés au *peer* actuel. Elles sont utilisées pour faire la synchronisation des joueurs, les recenser, et gérer la partie.

Maintenant que nous avons cité les deux fonctionnalités principales que nous avons utilisées, nous pouvons décrire la connexion des joueurs.

Une fois que le joueur "*Serveur*" a créé son *peer* côté serveur, qui est créé lorsqu'il décide d'être joueur-hôte, ses informations sont envoyées à une fonction nommée *SendPlayerInformation()*. Cette fonction prend en compte le nom du joueur, son id, et détecte s'il est un serveur. Son ID est toujours égal à 1 lorsqu'on est un hôte/"*Serveur*", le nom a été, pour ce projet, simplifié, il s'appelle donc "*Hôte*". La fonction va prendre ces informations en compte, et va le lier à un dictionnaire composé de clés et de valeurs. Les clés sont leurs ordre d'arrivée, donc 0 pour le "*Serveur*" et 1 pour le "*Client*". La valeur est les informations du joueur: son id et son nom. Ce dictionnaire est nommé *Players* et est situé dans le script *GameManager.cs*, script qui va gérer la connexion des joueurs entre eux, et est envoyé par une fonction *Rpc* à tous les joueurs. De cette façon, tous les joueurs auront une liste de qui est connectée, ordonnée par qui est "*Serveur*" ou non.

Lorsque le joueur client clique sur "*Rejoindre*", un *peer* de client est créé. Ce *peer* va retrouver le *peer* serveur, afin de relier les deux. Lorsque ce lien est fait, nous appelons aussi la fonction *SendPlayerInformation()* afin de recevoir les informations du joueur "*Serveur*" et de lui envoyer nos informations.

Ainsi, à la fin de cette étape, nous avons deux joueurs connectés entre eux, qui se reconnaissent, et plusieurs moyens de les adresser.

6.1.1.3. Lancement de la partie

Avant de passer dans les détails de ce qui lance la partie, nous devons comprendre ce qui permet au joueur "*Serveur*" de lancer la partie. Une fois sur le menu d'hébergement, le bouton "Lancer la partie" est bloqué, tant que 3 conditions ne sont pas respectées. Comme vu précédemment dans les menus, et nous verrons le fonctionnement plus en détail peu après, le menu d'hébergement est composé de deux objets: un nœud de choix de mode de jeu, et un nœud de choix de score. Il faut

obligatoirement choisir quelque chose dans ces deux nœuds. Ensuite, pour que le bouton “Lancer la partie” soit accessible, il faut que la longueur du dictionnaire *Players* de *GameManager.cs* soit égale à deux, c'est-à-dire qu’il y a deux joueurs connectés ensemble. Ensuite, nous pouvons donc déplacer nos joueurs sur la carte multijoueur.

Nous en avons déjà parlé un peu avant, nous n’avons actuellement qu’une seule carte multijoueur, nommée “*mpMap02*”. Celle-ci est reliée à un *script* dans le projet, qui va gérer l’apparition des joueurs. Premièrement, lorsque la partie se lance, la scène “*mpMap02*” est instanciée, c'est-à-dire que nous avons une copie temporaire de cette scène où nous pouvons modifier des choses dessus. Une fois instanciée, nous allons parcourir le dictionnaire *Players*, et pour chaque élément de ce dictionnaire (deux joueurs donc deux fois), nous allons instancier les joueurs sur la carte. Chaque joueur instancié voit ses données associés à son équivalent du dictionnaire *Players*, c'est-à-dire qu’ils ont les mêmes données: nom de joueur, ID multijoueur, score (initialisé à 0 par défaut).

Une fois ces joueurs instanciés, il faut les faire apparaître sur la carte. Au lieu de les instancier à des coordonnées exactes, nous employons la méthode de téléportation sur un nœud de coordonnées. C’est à dire que nous avons, dans la scène “*mpMap02*”, deux nœud nommés 0 et 1, qui appartiennent au groupe “*PlayerSpawnPoint*”. Le nœud 0 servira pour le joueur “*Serveur*”, et le 1 pour le joueur “*Client*”. Ensuite, nous utilisons une fonction nommée *Teleport*, qui va téléporter le joueur aux coordonnées du nœud auquel il est associé.

La partie peut enfin démarrer, les deux joueurs sont sur la carte en même temps. Voyons maintenant comment les joueurs sont synchronisés entre eux.

6.1.1.4. La synchronisation des joueurs

Pour que les joueurs se voient en plein jeu et que les actions se fassent sans problème, nous avons utilisé un *nœud* Godot nommé le *MultiplayerSynchronizer*. C’est un nœud à qui on va donner des propriétés, et qui va, comme son nom l’indique, les renvoyer à tous les joueurs. Comme ça, tout le monde est synchronisé sans problèmes. Voyons ainsi plus en détail comment tout cela fonctionne.

Premièrement, voyons comment tout cela fonctionne. Nous avons rattaché à la scène “*Player*” un *MultiplayerSynchronizer* que nous avons nommé “*MultiplayerSynchroniser*“. A chaque fois qu' un joueur est instancié, nous associons une autorité multijoueur à l’Id du joueur actuel. C’est à dire que désormais, le seul *peer* qui peut contrôler ce “*Player*” est le *peer* initialisé par ce joueur. Le *peer* du joueur “*Client*” sera associé au *peer* de son “*Player*” et le *peer* du joueur “*Serveur*” sera associé au *peer*

de son “*Player*”. Nous avons désormais un moyen de savoir qui contrôle qui. Cependant, le synchroniseur ne synchronise rien. Nous avons donc dû rajouter dans ses propriétés ce qu'il va synchroniser. Il synchronise donc deux choses: la position du joueur, et la rotation de ta tête (pour voir où est pointé le pistolet).

Les deux joueurs sont synchronisés au niveau du déplacement et de l'orientation, il faut désormais synchroniser les dégâts. Pour gérer ce cas, nous avons dû modifier le code du joueur dans *Player.cs*. Si le joueur tire et qu'il est en multijoueur, nous appelons la fonction *Shoot()* en Rpc. Nous détections si ce qui a été touché est un nœud “*Player*”. Si oui, on appelle la fonction *TakeDamage()* avec 20 de dégâts en Rpc. Cette fonction retire la vie au joueur concerné, et informe le joueur ayant tiré qu'il a touché son adversaire avec une petite croix noire au milieu.

Cible touchée

Cible ratée

Figure 21 : détection des dégâts

Nous avons parlé de la synchronisation du joueur et des tirs, il nous reste un dernier élément à voir. Le synchroniser s'actualise à chaque image qui s'affiche à l'écran. Cependant, pour ne pas envoyer et recevoir trop de *paquets*, nous avons rajouté un délai. Les *paquets* s'envoient toutes les 0,07 secondes, ce qui réduit grandement l'envoi de données, et donc le risque de perte de données en chemin. Le problème avec ce délai, c'est que ce n'est pas esthétiquement plaisant de voir des joueurs qui se déplacent d'une manière saccadée, voire qui se téléportent. Nous avons donc décidé d'utiliser la méthode *Lerp*, qui va donner le résultat d'une interpolation entre un *Vecteur3D* et un flottant, ici *0,1f*. Au lieu de synchroniser la position et la rotation du joueur, nous allons utiliser des variables, nommées *_syncPos* (et respectivement donc *_syncRotation*). Si le joueur qui joue est associé au “*Player*” actuel, *_syncPos* est associé à la position actuelle du “*Player*”. Sinon, un va actualiser la variable *_syncPos* en fonction de la dernière position connue du “*Player*” et de la méthode *Lerp*, qui va placer la position de l'autre

"*Player*" en fonction de où il *devrait* être par rapport au dernier appel. Cette astuce permet de grandement optimiser les données qui sont transmises entre joueurs.

6.1.2. Lancement du jeu

Maintenant que nous avons résolu la partie technique du multijoueur, nous pouvons nous amuser. Le jeu fonctionne, on peut se tuer et réapparaitre à l'infini. Il reste un problème cependant: comment définir qui est le meilleur au jeu? C'est donc avec cette partie que nous allons résoudre cette question, en explorant le fonctionnement du jeu et des scores.

Cette partie recense des codes présents dans *mpMap02.cs*, *playerHealth.cs*, *MultiplayerHUD.cs*,

6.1.2.1. Les différents modes de jeu

Comme expliqué un peu avant, nous avons deux choix à faire pour notre partie: le mode de jeu, et le score à atteindre.

Actuellement, il n'existe qu'un seul mode de jeu possible: le "*Deathmatch*", ou match à mort. Pour gagner la partie, il faut être le premier à tuer l'adversaire un certain nombre de fois.

L'autre paramètre est le score à atteindre. Nous pouvons le choisir parmi plusieurs choix: 1, 2, 5, 10, 20 et plus, c'est à vous de choisir. Une fois cette valeur choisie, elle va être envoyée au *script* de "*mpMap02*", qui va détecter qui est le serveur, puis qui va envoyer au travers d'une fonction *Rpc SetScoreToReach()*, la valeur choisie. Durant la partie, l'objectif sera affiché en haut de l'écran, au milieu.

Nous avons donc déterminé nos paramètres: le mode de jeu, le score à atteindre, et nos joueurs: la partie se lance.

6.1.2.2. La partie

Récapitulons ce que l'on a. Nous avons nos deux joueurs qui sont apparus sur la carte multijoueur, un objectif à atteindre, et un mode de jeu. La partie fonctionne

normalement, mais nous devons encore choisir que faire lorsqu'un joueur gagne, et comment afficher les scores.

Par souci de simplicité, nous avons choisi de ne pas mettre d'affichage des scores dynamiques. C'est à dire que les scores ne sont pas montrés comme "Vous" et "Adversaire", mais comme "Hôte" et "Client". Le seul impact est que l'affichage des scores sur les deux écrans sont identiques, et donc que le joueur "*Serveur*" verra son score à gauche, et que le joueur "*Client*" verra son score à droite.

Figure 22 : deux exemples de scores

Comme vu sur l'image au dessus, nous avons deux exemples de scores. Le premier représente une partie où l'objectif à atteindre est de 15, le score de l'hôte est 4, et celui du client est 7. Sur le deuxième exemple, nous avons une partie où l'objectif à atteindre est 5, l'hôte est en train de gagner avec 3 points, et le client juste derrière avec 2 points. Ces scores sont bien actualisés en direct.

Désormais que nous avons un affichage, voyons comment le mettre à jour. La gestion de la partie se fait également dans deux fichiers, *MultiplayerHUD.cs*, qui va gérer l'affichage des scores en permanence, et *playerHealth.cs* qui va gérer les cas de mort, et donc d'actualisation des scores. A chaque fois que la vie d'un joueur est égale à 0, il est téléporté au centre de la carte, en hauteur. Nous pouvons donc détecter qui meurt, et par conséquent, qui gagne un point. Prenons un exemple où le joueur "*Serveur*" tue le joueur "*Client*". Nous savons du côté "*Client*" que le joueur est mort, mais du côté "*Serveur*", on n'a aucune information. Ainsi, dans le code qui va téléporter le joueur "*Client*", nous avons rajouté un appel à une fonction nommée *UpdateServerScore()*, qui va incrémenter le score du joueur "*Serveur*" de 1. Pour que ce score soit incrémenté à la fois sur le joueur client et sur le joueur serveur, nous avons utilisé un appel Rpc à cette fonction, ainsi, le score du joueur "*Serveur*" sera incrémenté de 1 pour le joueur "*Client*" et pour le joueur "*Serveur*". Nous avons le même fonctionnement si un joueur "*Client*" tue un joueur "*Serveur*", cette fois avec une fonction nommée *UpdateClientScore()*.

Il ne manque plus qu'une seule chose à régler pour avoir un multijoueur fonctionnel, la victoire. A chaque appel de *UpdateClientScore()* ou *UpdateServerScore()*, on appelle une autre fonction, nommée *CheckWin()*. Cette fonction va vérifier deux choses: si le score du joueur "Serveur" est supérieur ou égal au score-objectif, et ensuite si le score du joueur "Client" est supérieur ou égal au score-objectif. Si une des conditions est vraie, un change la scène et les deux joueurs sont transportés sur le

menu de fin de partie. Sur ce menu, les *peers* multijoueurs seront fermés, comme ça nous pourrons relancer une partie sans problèmes si l'envie nous en prend. Finalement, le menu de fin de partie affiche le joueur qui a gagné, soit "Hôte", soit "Client".

Nous avons donc conclu cette partie sur le fonctionnement du code multijoueur. Désormais, regardons les épreuves que nous avons traversées pour atteindre tout ce qui fût décrit durant l'année.

6.2. Difficultés rencontrées

Lors de cette partie, nous allons parler de diverses difficultés que nous avons rencontrées, et comment nous avons réussi à les résoudre. Pour donner un peu de contexte à cette partie, il est utile de préciser que nous avons suivi un guide sur Youtube pour réaliser un jeu multijoueur sur Godot dont l'auteur est FinePointCGI (crédits dans l'annexe). Petit problème avec ce guide, il est pour un jeu 2D.

6.2.1. La synchronisation et transition vers Godot 4.2

Lors de la phase de recherche sur comment réaliser le multijoueur du jeu, nous nous sommes rapidement rendu compte que travailler sur Godot 3.5.3 posait plusieurs problèmes.

Tous les guides de multijoueur sur Godot sont sur Godot 4 ou Godot 4.2. Le problème de cette version, hormis certains petits problèmes d'incompatibilité de noms, où il faut trouver un autre nom pour la même fonction, est qu'entre deux, deux objets indispensables au multijoueur ont été ajoutés: le *MultiplayerSynchronizer* et le *MultiplayerSpawner*. Ces deux objets ont pour objectif, comme le nom l'indique, de synchroniser des propriétés entre les joueurs, et de gérer les cas d'apparition des personnages sur la carte.

Nous n'avons finalement pas eu besoin du deuxième, le *MultiplayerSpawner*, car comme expliqué précédemment, nous avons utilisé une autre méthode pour faire apparaître les joueurs. Cependant, le synchroniseur nous semblait indispensable. Nous avons donc opéré une transition de Godot 3.5.3 à Godot 4.2 en 3 jours.

Cette transition a permis de considérablement accélérer le développement du multijoueur, et de rendre un projet plus complet rapidement.

6.2.2. L'incident du dictionnaire

Avant de décrire cet incident, nous devons revenir sur ce qui a été dit en début de cette partie. Pour créer le multijoueur, nous avons donc utilisé une vidéo de FinePointCGI qui expliquait comment faire un jeu multijoueur sur Godot, mais en GDScript. Le problème étant que notre jeu est côté en C#. Il y a donc un moment où dans le guide, FinePointCGI utilise une variable qui se comporte comme un dictionnaire. Nous avons donc utilisé un dictionnaire pour gérer nos joueurs, que nous avons nommé *Players*, situé dans *GameManager.cs*. En communiquant sur nos avancées avec d'autres groupes, nous nous sommes rendu compte qu'il existait une version de ce tutoriel en C#, par le même auteur. Nous avons donc décidé de le reprendre au début, en accéléré vu que nous avions déjà quasiment tout. Cependant, sur cette version du dictionnaire, FinePointCGI utilise une liste de *Players*. Au lieu d'avoir la clé 0 pour l'hôte et sa valeur associée en joueur "*Serveur*" et en clé 1 le joueur "Client", il y a le joueur "*Serveur*" en première position de la liste, et le "*Client*" après. Pour suivre le tutoriel de la meilleure façon, nous décidons de suivre ce qui est fait, et de prendre une liste.

Quelle erreur! Dans les trois heures qui suivirent, nous nous confrontâmes à une suite de problèmes plus périlleux les uns que les autres. Chaque solution que nous trouvions pour un problème rendait le problème suivant encore plus compliqué à résoudre. Avant d'avoir *Players* en liste, nous pouvions lancer la partie multijoueur, et contrôler les deux joueurs en même temps. Nous ne pouvions plus nous connecter entre nous.

Nous avons donc décidé de revenir en arrière et de retrouver le dictionnaire. Cependant, juste remettre le dictionnaire n'était pas possible car chacune des fonctions étaient reliées entre elles. Il était donc trop compliqué de tout supprimer. Nous avons donc décidé de revenir en arrière grâce à GitHub Desktop. Chose plus facile à dire qu'à faire. Il ne nous restait plus beaucoup de jours pour finir Castle Of Demise, un nouveau problème n'était pas spécialement nécessaire pour notre progression. Nous avons donc décidé de retourner à un ancien *commit* sur GitHub. Une fois fait, nous nous sommes rendu compte que nous n'étions pas au bon commit, et qu'il fallait remonter encore plus. Mais en même temps, d'autres personnes travaillaient sur leur partie, et revenir en arrière allait casser leur progrès. Nous étions dans une impasse.

Après plusieurs délibérations, nous avons récupéré une clé usb qui contenait des *commits* de la veille, et nous avons *force push* ce qu'elle contenait. C'est-à-dire que toute la progression que nous avions venait de disparaître. Les deux heures suivantes servirent donc à retrouver tout le travail que nous avions perdu, sans compter le temps que nous avons perdu.

Une fois le retard rattrapé, nous avons pu continuer, jusqu'au problème suivant: les caméras.

6.2.3. Les caméras

La différence principale entre un jeu 2D et un jeu 3D, hormis les graphismes, sont les caméras. Sur un jeu 2D, les deux personnages peuvent avoir la même caméra, ça ne posera pas de problèmes. Cependant, sur un jeu 3D, chaque personnage doit avoir une caméra associée.

Dit comme ça, le problème semble logique, mais ça ne nous est pas venu à l'esprit que nous avions un problème de ce style. Il nous a donc fallu plus d'une journée entière de débug pour se rendre compte de ceci. Régler ce problème nous a juste pris une ligne de code: "*this.CameraForFov.Current = true;*". C'est tout. Vraiment.

6.2.4. Incompatibilité multijoueur et non multijoueur

Un dernier problème que nous avons vécu est l'incompatibilité entre le mode multijoueur et le mode campagne. Dès que l'on incorporait une fonctionnalité pour le multijoueur, il y avait un problème dans le mode campagne. On ne pouvait ni tirer, ni bouger, ni même tourner notre caméra.

La solution que nous avons trouvé est de rajouter un booléen *IsMultiplayer*, qui, comme son nom l'indique, va nous informer si le "*Player*" est dans une partie multijoueur. Après, chaque fonction va accéder à ce booléen pour savoir comment réagir. Par exemple, dans *Player.cs*, dès l'initialisation du joueur, en fonction de *IsMultiplayer*, l'initialisation est différente: si c'est en multijoueur, on initialise le synchroniseur et on active l'affichage des scores. Sinon, on initialise les autres ATH des joueurs.

7. Le site web

7.1. États antérieurs du site web

Pour le site web nous avons d'abord imaginé à quoi devrait ressembler le site, en faisant un fichier *html* simple qui nous a permis d'avoir une vision d'ensemble. Par la suite nous nous sommes inspirés de feuilles de styles (*css*) reconnues pour le site afin d'obtenir celui que vous aviez pu voir lors de la soutenance technique. Étant donné les modifications conséquentes sur le site et la refonte totale de ce dernier nous nous permettons de présenter son ancienne version ici.

Nous avions une page principale qui tenait dans le fichier *index.html*. Dans l'entête de la page nous avions trois boutons qui, lorsqu'on cliquait dessus, faisaient défiler la page jusqu'aux sections en question. Nous avions ainsi une section "*présentation*" composée d'un titre <*h3*> et d'une balise <*p*> pour le texte, une section "*Qui sommes nous ?*" avec un titre <*h3*> pour chaque membre accompagné d'un paragraphe <*p*>. Enfin, on avait la section "*téléchargements*" où l'on pouvait récupérer les divers documents, notamment le cahier des charges en format *pdf*.

7.2. Etat actuel du site web

Bien que ce site convenait à l'avancement antérieur du projet, nous avons constaté et jugé au cours du développement qu'il nécessitait une refonte, afin qu'il puisse être utilisé à la fois comme présentation et documentation de Castle Of Demise. Nous avons voulu, par la nature *open source* de notre projet, mettre à disposition de tout-un-chacun les explications dont de potentiels contributeurs pourraient se servir afin de collaborer ou améliorer le jeu.

Afin de pouvoir produire un site de qualité et conforme aux standards actuels de l'industrie en termes de documentation, nous avons opté pour l'utilisation d'un *Framework Javascript* intitulé *Astro*. *Astro* nous a permis, entre autre, d'établir un environnement de déploiement du site standardisé et hautement compatible avec la plateforme *Github*, et a facilité la tâche d'écriture de documentation puisqu'il permet la conversion de *Markdown* (offrant une syntaxe facile à lire et à écrire) en *HTML* (le langage de balisage classique dans le web).

8. L'intelligence artificielle

8.1. Pathfinding

Le Pathfinding consiste à calculer le chemin optimal entre les ennemis et le joueur afin qu'ils puissent se déplacer et ultimement lui faire des dégâts. Pour ce faire, nous avons utilisé le nœud *NavigationRegion3D* pour générer un *NavMesh*, puis avons, dans les scènes des ennemis, ajouté le nœud *NavigationAgent3D*. Dans le script *Enemy.cs* nous avons défini la cible qui dans notre cas est le joueur et dans un script annexe lié à la map avons fait en sorte que les ennemis puissent se servir du *NavMesh* pour se déplacer.

Grâce à cela les ennemis contournent les murs et s'adaptent au terrain, dans la plupart des cas ils peuvent contourner les obstacles et comprendre le principe d'élévation pour pouvoir atteindre le joueur qu'il soit derrière un mur ou en haut d'une rampe.

8.2. Gestion des signaux des attaques

Grâce à des nœuds *Area3D* et leurs signaux nous pouvons détecter quand le joueur est à portée d'attaque. Lorsque le joueur peut attaquer, la méthode *TakeDamage()* est appelée. Cette méthode décrémente la vie du joueur des dégâts infligés par l'ennemi et vérifie si le joueur est mort. Dans le premier cas, la vie diminue et des indications sous la forme de *camera shake*, d'effets visuels et sonores se jouent, informant le joueur qu'il a été frappé. Dans le deuxième cas, le joueur meurt et la scène change vers un menu de mort qui donne l'option au joueur de revenir au menu principal.

9. Conclusion

9.1. Crédits

Pour le tutoriel multijoueur, *FinePointCGI* sur *YouTube* a été d'une grande aide. Sans lui, ce multijoueur n'aurait pas été aussi bien.

9.2. Ressenti personnel

Vous trouverez dans cette partie tous nos ressentis sur ce projet, les négatifs et les positifs, les hauts et les bas que nous avons vécu lors de cette année 2023-2024 à l'EPITA en rapport avec notre projet Castle Of Demise.

9.2.1. Amadéo Héaulme

Personnellement, je considère ce projet comme une réussite. Au début, je ne connaissais rien, je ne savais strictement rien faire, mais désormais je suis capable de faire plusieurs choses sans aucun problème. Je m'en retrouve sincèrement grandi, et mon expérience était positive.

Je tiens à noter certains moments qui sont, pour moi, importants dans la progression du projet, et je souhaite en profiter pour remercier mes camarades, sans qui rien n'aurait été possible: Jean Herail, Paul Lacanette, Clément Grondin, et Roni Yildiz.

Nous avons eu des problèmes d'organisation et pour finir nos projets, nous travaillons peu, et le travail que nous faisions était mal. Cependant, après avoir demandé de l'aide à l'extérieur, nous avons réussi à tous retravailler ensemble, et nous avons rendu un projet de haute qualité.

Je ne pensais sincèrement pas que j'allais autant progresser lors de ce projet. Je ne connaissais rien à Godot, je ne savais même pas lancer une scène, et petit à petit, j'ai progressé. Je sais désormais l'utiliser plutôt bien, mais je continue d'apprendre de nouvelles choses, qui sont extrêmement utiles pour avancer, et même cela à une semaine de la date de rendu de projet.

Je ne peux pas non plus parler de progression sans parler du multijoueur. Sans ne rien savoir au début, et à suivre bêtement des tutoriels sur YouTube sans comprendre l'objectif de chaque fonction, avoir des problèmes de synchronisation, et des problèmes entre la 2D et la 3D m'ont été grandement utiles. Je sais désormais comment fonctionne chacune de mes fonctions, comment j'aurais pu les optimiser, comment en créer de nouvelles, sans même avoir à suivre un autre tutoriel.

La partie synchronisation m'a aussi permis de travailler en groupe, et connaître les fonctions que mes camarades ont fait, et même les comprendre. L'organisation du projet a été en effet séparatrice, chaque personne faisait ses propres fonctions, et on voyait la liaison après, mais une fois cet "après" atteint, nous avons travaillé en groupe, à deux, parfois à trois, parfois tout le groupe ensemble derrière un ordinateur.

La partie multijoueur et synchronisation s'est finie en dernier, et j'ai eu l'honneur d'être le premier à jouer à une vraie partie de Castle Of Demise contre Jean Herail. Nous avons décidé de faire un match à mort et de mettre le score à atteindre à 5. Après un combat acharné pour l'un d'entre nous, et tranquille pour l'autre, j'ai finalement pu gagner 5 à 2 contre Jean Herail, faisant officiellement de moi la première personne à gagner une partie sur Castle Of Demise.

Je me rappellerai longtemps de cette partie et de cette victoire. Nous avions déjà créé des stratégies, comme rester au centre de la carte au point d'apparition pour plus de sécurité et avoir une vision globale, ou encore de sortir de la carte pour gagner de l'accélération et ainsi avoir moins de chances d'être touchés. Le saut étant aussi très haut, sauter en même temps que tirer pour être plus dur à toucher est aussi une stratégie efficace.

En conclusion, je peux bien dire que je ne remercierai jamais assez mes camarades pour avoir réussi à créer un projet comme celui-là.

9.2.2. Paul Lacanette

Mon expérience avec Castle of Demise à l'école Epita a été très enrichissante. En tant que responsable de la jouabilité, j'ai pu plonger profondément dans la programmation en C# avec Godot 4 sur JetBrains Rider, Bien que la majorité des tutoriel soit écrit en GDScript j'ai pue traduire le tout en c#. Travailler au sein de notre équipe de cinq personnes a été gratifiant, même si l'intégration tardive de certains membres a parfois posé des défis de coordination. J'ai particulièrement apprécié la création du joueur et le processus de test continu des nouvelles fonctionnalités, qui ont renforcé non seulement mes compétences en programmation, mais aussi ma capacité à m'intégrer efficacement dans un groupe. Le retour positif que nous avons reçu des testeurs a été extrêmement motivant, validant nos efforts et notre approche. En revanche, je reconnais que nous aurions pu accorder plus d'attention à l'aspect

multijoueur du jeu. Pour l'avenir, je recommanderais vivement une planification minutieuse à chaque étape du projet, avec des délais clairement définis, afin de maintenir la dynamique et d'assurer la qualité du produit final. Cette expérience m'a non seulement permis de développer mes compétences techniques, mais aussi de mieux comprendre l'importance cruciale de la gestion du temps et de la communication au sein d'une équipe de développement de jeux vidéo et donc de n'importe quel autre projet.

9.2.3. Clément Grondin

Personnellement, ce projet a été une source de stress ainsi que de joie. En effet, les soutenances et dates de rendus ont toujours été stressantes mais malgré cela et les difficultés rencontrées, le soutien et l'esprit de corps régnant dans le groupe m'ont permis de réduire ce stress.

Au début de ce projet je ne connaissais rien de Godot mais durant l'avancée du projet, j'ai fait des recherches et me suis amélioré dans la manipulation du logiciel Godot et dans sa compréhension.

Aujourd'hui, bien que je soit moins à l'aise sur Godot que d'autres membres de mon groupe le sont, je suis content de l'aide que j'ai pu leur apporter mais regrette tout de même de ne pas m'être engagé plus tôt dans le projet ce qui aurait peut-être permis de gagner du temps et d'améliorer certains points.

Mais dans l'absolu, je suis satisfait de ce que nous avons produit et remercie mes camarades Jean Herail, Paul Lacanette, Amadéo Héaulme, et Roni Yildiz d'avoir été présents durant toute cette année.

9.2.4. Roni Yildiz

Étant passionné de jeux vidéos, notamment par les jeux de tir à la première personne, j'ai adoré travailler sur ce projet.

Cependant, je suis assez mitigé sur mon bilan personnel. J'estime que je n'ai pas réussi à investir assez de temps dans le projet. Ce qui fait que même si le jeu est fonctionnel, on aurait pu peaufiner le jeu davantage si j'avais travaillé davantage.

Notamment au niveau des cartes. J'aurais voulu faire plus de niveaux et faire du jeu un bijou de narration et de level design. Mais mon manque d'organisation personnelle a empêché celà et je me suis excusé par rapport à ça aux autres membres du groupe. En définitive je suis déçu de la qualité de mon investissement dans ce projet malgré toute la passion que j'ai éprouvé pour ce jeu. Celà dit je suis satisfait de

l'engagement de mes autres camarades qui ont comblé mes lacunes et notre dur labeur nous a quand même permis d'arriver à un produit final satisfaisant.

9.2.5. Jean Herail

À la conclusion de ce projet de développement de *Castle of Demise*, j'ai trouvé une expérience riche en défis et en découvertes.

En tant que chef d'équipe, j'ai pris plaisir à coordonner nos efforts et à résoudre les défis techniques rencontrés, notamment lors de la migration vers Godot 4. J'ai joué un rôle actif dans des aspects critiques comme la gestion de la physique du jeu et le développement du site web. La collaboration avec mes coéquipiers a été satisfaisante, bien que parfois il ait fallu rappeler à l'ordre pour maintenir le rythme.

Nous avons utilisé efficacement Github pour organiser nos tâches et Discord pour discuter des retours et des idées, facilitant ainsi une adaptation fluide au cours du projet. Consacrer environ deux heures par semaine à ce projet a été une occasion précieuse de renforcer mes compétences techniques et d'acquérir une meilleure compréhension de la dynamique de travail d'équipe.

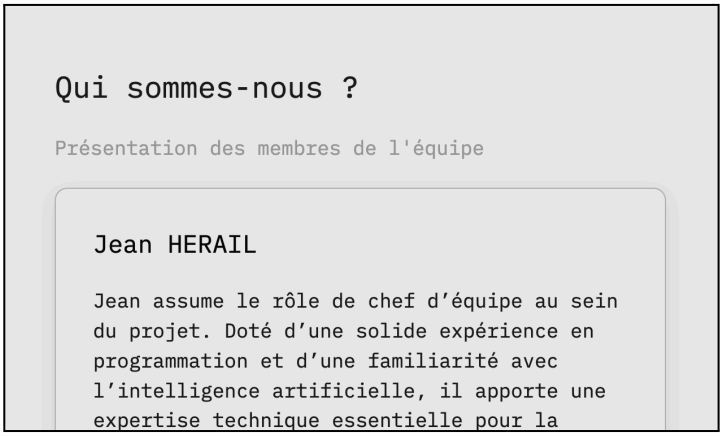
Cette expérience m'a également permis de cultiver une passion durable pour le développement de jeux vidéo, en m'offrant une perspective enrichissante sur le processus de création.

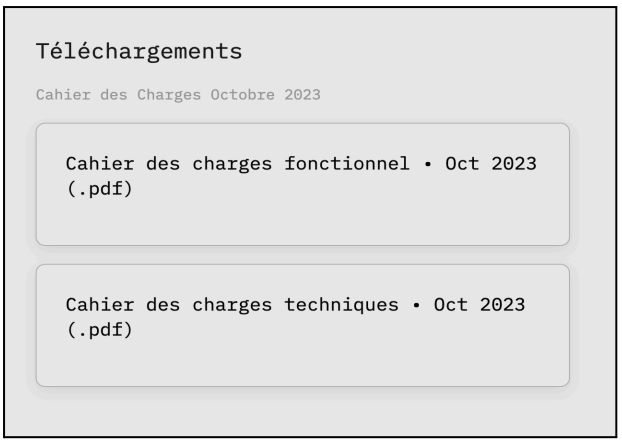
9.3. Conclusion de ce rapport

C'est donc sur ces dernières lignes que se finit notre aventure avec ce projet pour notre première année à l'EPITA. Castle Of Demise, dont l'idée fut envisagée dès Octobre 2023, est désormais un projet fini à part entière, qui a vécu des hauts et des bas, des problèmes et des solutions, mais surtout, des souvenirs.

Ce rapport signe donc l'entièreté de nos aventures avec Castle Of Demise, et nous vous remercions de nous avoir permis de réaliser ce projet, dont nous nous en sortons tous grandi.

10. Annexe





Anciennes images du site web