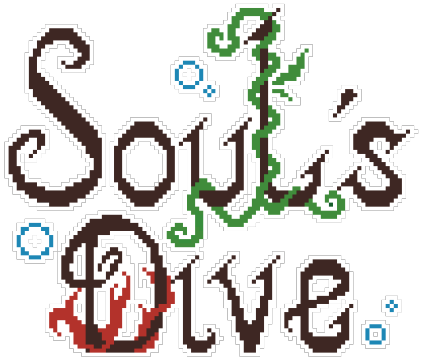
Projet SUP - Epita Strasbourg

– Rapport de Projet –



– Auteurs –
Eirys Devigne
Charlène Demontrond
Guillaume Bravetti
Lucas Herzog-Martin
Noam Raffin

Tables des matières

1 Présentation du Jeu

Soul's Dive est un jeu vidéo de plateforme de type Rogue-Like développé par BeexL Creations. Le jeu consiste en une série de parties courtes qui prennent place dans un monde généré aléatoirement. Chaque partie propose ainsi une expérience unique, avec des environnement, des ennemis et des objets disposés de façon aléatoire. Ces éléments permettent une rejouabilité importante.

Le gameplay de "Soul's Dive" se concentre sur l'exploration, la collecte de ressources, la progression et la survie. Le joueur doit utiliser ses réflexes et ses compétences stratégiques afin de naviguer au travers de niveaux remplis d'ennemis et d'obstacles.

Une partie est divisée en quatre phases principales, chacune présentant des environnements différents et une difficulté croissante. Ci-dessous, nous détaillons ces étapes :

- Le joueur commence son voyage dans la forêt, à la surface, où il peut se familiariser avec les mécaniques du jeu et collecter des ressources afin de se préparer à la descente vers les niveaux inférieurs. En effet, une difficulté accrue attend le joueur dans les parties suivants.

- Dans le deuxième niveau, le joueur plonge dans les profondeurs d'un océan rempli de mines, de projectiles et de collectibles. Malgré sa mobilité réduite sous l'eau, il doit éviter les pièges et atteindre le fond de cette mer pour accéder au troisième niveau.

- Dans cette phase, une surprise attend le joueur : plus il a éliminé d'ennemis dans le premier niveau pour amasser des ressources, plus les ennemis seront nombreux dans le troisième niveau. Les obstacles y sont également plus fréquents, et le joueur devra esquiver les pics et les projectiles pour atteindre la fin de la partie.

- Enfin, le joueur atteint la dernière phase. Il doit alors affronter un boss qui le poursuit et lui envoie des projectiles, en utilisant les éléments de l'arène à son avantage.

Une fois la partie terminée et le boss vaincu, le joueur peut choisir de quitter le jeu pour une pause ou de relancer une nouvelle partie immédiatement. Les

attributs de son personnage seront conservés, mais les points de santé et les dégâts des ennemis augmenteront également. La difficulté du jeu réside dans le fait de trouver un équilibre entre éliminer suffisamment d'ennemis dans le premier niveau pour pouvoir affronter le boss et survivre à la partie suivante, tout en évitant de rendre le troisième niveau trop difficile. Le nombre de parties consécutives du joueur est conservé, l'objectif étant d'atteindre le score le plus élevé possible.

Le jeu propose également deux styles de jeu différents : le premier est un épéiste classique avec des statistiques équilibrées en attaque, résistance et vitesse. Le second est un assassin, infligeant plus de dégâts et étant plus rapide, mais au prix d'une résistance réduite. Ce deuxième personnage est recommandé aux joueurs plus expérimentés.

Par ailleurs, le jeu propose un mode expert dans lequel le joueur bénéficie d'une résistance accrue et d'une attaque plus puissante, mais subit également le double des dégâts habituels. De plus, il est possible de jouer avec un ami grâce au mode multijoueur en ligne.

Enfin, Soul's Dive possède un style graphique rétro en pixel-art ainsi qu'une bande-son atmosphérique qui favorisent l'immersion du joueur et permettent une expérience mémorable.

2 Cahier des Charges

2.1 Rappel du Cahier des Charges

Le cahier des charges est un document que nous avons rendu en début de projet, peu après la formation des groupes. Il avait pour but de définir les objectifs, besoins et contraintes auxquels nous allions faire face lors de la production, ainsi que les détails de notre organisation.

Notre objectif principal était de produire un jeu qui nous permettrait de démontrer nos compétences techniques, tout en proposant une expérience agréable et divertissante.

Nous avons passé plusieurs idées en revue, mais avons finalement décidé qu'un Rogue Like en 2D était le type de jeu le plus à même de remplir cet objectif. En effet, la présence d'un moteur physique permettant de simuler des interactions réalistes entre les différentes entités et l'environnement paraissaient parfaites pour souligner nos compétences. La possibilité d'implémenter une intelligence artificielle pour les ennemis a également influencé notre choix. Nous avons également été séduits par l'idée d'une génération procédurale des niveaux, qui pourraient apporter un plus à notre projet en permettant une expérience unique à chaque partie.

Nous sommes donc restés sur l'idée d'un Rogue Like comprenant des niveaux générés aléatoirement ainsi qu'un système de progression du joueur entre les parties. Celles-ci devaient être courtes et se concentrer sur un gameplay rythmé et dynamique. De plus, le joueur devait avoir accès à plusieurs styles de jeu.

Notre but était donc de produire un jeu complet et divertissant. Pour ce, nous avions prévu d'utiliser les compétences acquises en cours, mais également d'acquérir nous-même de nouvelles compétences grâce à des ressources disponibles en ligne. Nous avions prévu que le projet nous permettrait de développer nos compétences en programmation, mais aussi et surtout d'apprendre à travailler en groupe, compétence essentielle pour notre future vie professionnelle.

Avant d'entamer le développement du jeu, nous avons cherché de l'inspiration du côté des nombreux Rogue Likes déjà existants afin de afin de comprendre les mécaniques populaires, d'identifier les éléments de gameplay qui retiennent l'attention des joueurs, et d'adopter les bons réflexes en matière de conception de niveaux et de progression des personnages.

Nous voulions également que le comportement des ennemis du jeu soit régi par une intelligence artificielle complexe, qui permettrait aux ennemis de poursuivre le joueur. De plus, le jeu devait comporter un boss à la palette de mouvement varié qui proposerait un challenge stimulant au joueur. Nous avions également prévu qu'il soit possible de jouer à deux en coopération en passant par un mode multijoueur en ligne.

Il était également prévu que le travail soit organisé en suivant un tableau de répartition des tâches. Chaque membre du groupe s'était ainsi vu assigner un

ensemble de domaines de développement de jeu vidéo sur lequel concentrer son travail, afin que les efforts du groupes soient organisés.

Afin de travailler sur le projet, nous avions prévu d'utiliser plusieurs outils nécessaires au développement du jeu vidéo en groupe. Ceux-ci comprenaient par exemple nos ordinateurs portables, les logiciels Unity et Rider, ou encore un repository Git.

Enfin, nous avions repéré un certain nombre de contraintes qui auraient pu se mettre en travers de la mise à bien du projet, ainsi que des solutions pour les contourner. Parmis celles-ci, nous avions pris en compte la contrainte temporelle : en effet, avec 9 mois seulement pour développer un jeu vidéo en plus des cours, il nous semblait nécessaire de nous organiser pour tirer au mieux parti de notre temps libre. Une deuxième contrainte que nous avions identifié était la coordination entre les membres du groupe : non seulement s'accorder pour ne pas provoquer de conflits dans le repository git, mais également communiquer efficacement de manière à ce que deux personnes ne travaillent pas sur le même aspect du jeu. Une troisième contrainte qui nous a semblée évidente était le dosage de la difficulté du jeu : en effet, dans un Rogue Like qui propose une progression du joueur entre les parties, il est nécessaire d'équilibrer efficacement les attributs du joueur et la difficulté des niveaux suivants afin d'empêcher que le jeu ne soit trop facile ou difficile.

2.2 Comparaison avec le produit final

Globalement, les idées énoncées en début d'année dans le cahier des charges ont été respectées.

- La notion d'aléatoire dans les niveaux, régie par un algorithme de génération procédurale des niveaux, est fonctionnelle et permet une expérience nouvelle à chaque partie.

- La palette de mouvements variée du joueur permet bien un gameplay dynamique et rythmé. Les nombreuses façon de se déplacer dans le niveau telles que le saut, double saut, ruée ou encore "wall slide" et saut mural permettent au joueur une réelle variété de mouvements et donnent lieu à différentes possibilités de style de jeu. Les parties sont

courtes et le gameplay rapide, ne laissant pas au joueur le temps de s'ennuyer. Ces éléments combinés permettent une rejouabilité élevée.

- Les ennemis sont chacun pourvus d'une intelligence artificielle différente : les slimes patrouillent, c'est-à-dire qu'ils continuent d'avancer jusqu'à un rebord avant de se retourner et de partir dans le sens opposé ; les chauve-souris sont pourvues d'un algorithme de pathfinding, ce qui signifie qu'elles empruntent constamment le chemin le plus court vers le joueur quand elle se trouvent à proximité de ce dernier. Enfin, le boss poursuit le joueur en avançant vers lui, tout en envoyant des projectiles dans sa direction. Quand le joueur lui a infligé plus de la moitié de sa barre de vie en dégâts, il passe dans sa deuxième phase : il instancie alors plus rapidement ses projectiles, pimentant ainsi le combat.

- En amassant des collectibles, le joueur a la possibilité d'augmenter les statistiques de son personnage : puissance d'attaque, résistance et capacité de soin. Ces changements sont conservés au fil des parties et permettent une progression graduelle du joueur.

Pour ce qui est des contraintes, les solutions mises en place dès le début de l'année nous on permis en grande partie de les surmonter.

- Pour la contrainte temporelle, des réunions régulières, des sessions de travail communes en présentiel et en distanciel et un travail sérieux de la part de chaque membre du groupe nous a permis de nous conformer aux délais fixés.

- Pour la contrainte temporelle, des réunions régulières, des sessions de travail communes en présentiel et en distanciel et un travail sérieux de la part de chaque membre du groupe nous a permis de nous conformer aux délais fixés.

- Pour ce qui est de la coordination entre les membres du groupe, les réunions régulières ont également aidé. De plus, la mise en place d'un repository git comprenant une branche par personne a permis d'éviter en grande partie les conflits. Des conflits sont quand même parfois survenus lorsque plusieurs personnes travaillaient ensemble sur la même fonctionnalité ; cependant, nous avons été capable de les résoudre quand cela était nécessaire.

- L'équilibrage de la difficulté a été réalisé à force de test du jeu : la solution développée a été d'augmenter le nombre d'ennemis dans le troisième niveau en fonction du nombre d'ennemis éliminé dans le premier. Cela a permis d'ajuster la difficulté de manière dynamique, offrant un challenge approprié au joueur.

En conclusion, nous avons réussi à respecter les principales idées et contraintes énoncées dans le cahier des charges grâce à une organisation rigoureuse, une bonne communication et une capacité d'adaptation continue tout au long du projet.

3 Noam

Dans le cadre de ce projet, j'ai principalement assumé la responsabilité des aspects physiques et du game design du jeu. Mon temps a été majoritairement consacré au développement des mouvements et de la physique du joueur, conformément aux indications du diagramme de Gantt. J'ai également apporté mon soutien à mes camarades, en particulier sur le développement des ennemis. De plus, j'ai pris en charge la création du magasin au sein du jeu, intégrant toutes les améliorations possibles des statistiques du joueur. Conformément au cahier des charges, j'ai implémenté un second personnage avec des statistiques distinctes par rapport au premier, ainsi que les systèmes d'attaque pour les deux personnages.
Création du Projet :
J'ai démarré le projet à partir de zéro sur Unity, en suivant des tutoriels YouTube, ce qui m'a beaucoup aidé à implémenter de nouvelles mécaniques. J'ai créé notre dépôt Git sur GitHub pour que l'équipe puisse contribuer au projet avant que le git de l'école soit ouvert.
Développement Initial :
Après la création du projet Unity 2D, j'ai configuré le GameObject du personnage, en utilisant Rigidbody2D pour gérer la gravité, Box Collider pour les collisions, et des scripts pour le code. J'ai aussi configuré la caméra pour suivre le personnage. Cette phase a été longue car j'ai dû apprendre Unity de zéro à l'aide de vidéos ou de documentation sur internet.

3.1 Mécaniques de Base pour un bon jeu de plateforme

Tout d'abord, l'implémentation du mouvement horizontal n'a pas posé de difficulté majeure : il s'agissait simplement d'appliquer un vecteur vers la gauche ou la droite en fonction de la direction souhaitée par le joueur. En revanche, le saut s'est avéré plus complexe. Initialement, nous avons créé un tag "sol" pour détecter les collisions et gérer le nombre de sauts. Cependant, cette méthode a engendré de nombreux bugs de collision lors de la génération des niveaux, rendant les mouvements du joueur moins fluides.

Pour remédier à cela, nous avons adopté une nouvelle approche consistant à utiliser un layer "Ground" et à placer un gameobject sous le joueur afin de détecter les collisions avec le sol. Cela a permis de mieux gérer les doubles sauts que j'ai implémentés.

Nous avons également ajouté un "dash" pour dynamiser le gameplay, en incluant une fonction qui annule la gravité pendant le dash, ainsi qu'une autre qui gère le cooldown de cette action. Par la suite, un "wall slide" et un "wall jump" ont été introduits. À l'origine, nous avions créé un tag "Wall" pour détecter les murs, en séparant les surfaces solides des murs. Cependant, cette approche a rencontré des problèmes similaires à ceux des sauts, nous contraignant à modifier notre méthode.

Nous avons donc supprimé le tag "Wall" et, à l'instar du saut, nous avons placé un point sur le côté du joueur pour détecter les collisions avec le layer "Ground". Lorsqu'une collision latérale est détectée, un vecteur est appliqué dans la direction souhaitée, et le nombre de sauts est réinitialisé pour permettre un wall jump, par exemple. Grâce à cette approche, avec un même layer sur les blocs, nous avons pu détecter et implémenter les sauts, le wall slide et le wall jump au personnage en utilisant deux gameobjects distincts (points). Cette solution a corrigé la majorité des bugs de collision rencontrés auparavant.

Après cette phase, la plupart des mécaniques de mouvement ont été ajoutées au jeu. Afin de rendre ces mouvements encore plus fluides et agréables à jouer, j'ai dû ajouter deux mécaniques supplémentaires : le jump buffering et le coyote time. Le principe du jump buffering est que lorsque le joueur est en l'air et qu'il appuie sur la touche de saut près du sol, il puisse sauter

immédiatement au moment du contact avec le sol, offrant ainsi des sauts plus fluides.

Le "coyote time" est un concept de game design utilisé pour rendre les sauts plus permissifs et améliorer la jouabilité. Il s'agit d'un court délai durant lequel le joueur peut encore effectuer un saut après avoir quitté le bord d'une plateforme, même s'il n'est plus techniquement en contact avec celle-ci. Pour mettre en œuvre ces deux mécaniques, j'ai simplement introduit un compteur pour gérer le temps accordé au joueur pour le jump buffering et le coyote time.

Après avoir implémenté les mouvements et les mécaniques de base de notre jeu, nous avons veillé à garantir une expérience de jeu agréable et fluide.

3.2 Les Attaques du Joueur

Pour notre jeu, j'ai mis en place un système de combat ainsi qu'un système de gestion de la vie du personnage. Le joueur dispose de trois attaques principales : une attaque latérale, une attaque vers le haut et une attaque vers le bas. Pour réaliser ces différents types d'attaques, j'ai positionné quatre points autour du joueur, correspondant à ces directions : un point au-dessus, un en dessous et un de chaque côté.

Lorsqu'une touche d'attaque est actionnée, un cercle est généré autour du point correspondant. Ce cercle détecte toutes les collisions à l'intérieur et stocke les objets touchés dans un tableau. Ensuite, je vérifie si parmi ces collisions, il y a un objet avec le tag "Ennemi". Si tel est le cas, une fonction est appelée pour réduire la vie de l'ennemi ; sinon, aucune action n'est entreprise.

Le système de vie, bien que simple à développer, est crucial pour le bon déroulement de notre jeu, en particulier pour la gestion des morts.

Bien que le système de vie et de combat soit fonctionnel, il manquait d'éléments visuels pour améliorer l'expérience de jeu. J'ai donc ajouté, à chaque préfabriqué d'ennemi, une barre de vie qui apparaît au-dessus de l'ennemi lorsque le joueur l'attaque. Cela permet une meilleure visibilité de l'état des ennemis et améliore l'immersion du joueur.

3.3 Les Ennemis

Pour mener à bien notre projet de jeu vidéo, j'ai apporté mon soutien à mes camarades dans le développement de certains ennemis. Plus particulièrement, je me suis concentré sur deux types d'ennemis.

3.4 Le Slime

Premièrement, j'ai développé un slime dont le comportement principal consiste à changer de direction lorsqu'il entre en contact avec un mur ou lorsqu'il détecte un vide à proximité. La principale difficulté résidait dans la détection du vide à côté du slime. Après plusieurs tentatives, j'ai trouvé la meilleure méthode pour y parvenir : j'ai placé un box collider en mode trigger juste devant le slime. Lorsque ce collider n'est plus en contact avec un objet du layer "Ground" (qui représente le sol du niveau), le slime change de direction. Pour détecter la collision avec un mur, j'ai utilisé une technique similaire à celle du wall slide et du wall jump, en plaçant un point de détection sur le côté du slime. Cette détection permet de changer la direction du slime en modifiant sa vélocité.

3.5 La Chauve-souris

Le deuxième ennemi que j'ai développé est une chauve-souris. Son objectif est d'attaquer le joueur dès qu'elle le repère et de le suivre pour l'attaquer. La difficulté majeure était de récupérer la position du joueur dans la partie afin d'appliquer un vecteur dirigé vers cette position. Pour ce faire, j'ai utilisé un box collider en mode trigger représentant la vision de la chauve-souris. Ce collider permet de récupérer la position du joueur. Ensuite, je calcule la direction vers la position du joueur et j'applique la vitesse de la chauve-souris dans cette direction, ce qui permet à la chauve-souris de suivre le joueur efficacement.

3.6 Amélioration de l'Expérience de Combat

En outre, après le développement des ennemis, j'ai également travaillé sur l'amélioration de l'expérience de combat en ajoutant un effet de recul lorsque le joueur frappe un ennemi. Cette tâche s'est avérée complexe car le vecteur de recul appliqué devait varier en fonction des ennemis et de leur comportement. Par exemple, le vecteur de recul général fonctionnait très bien pour la chauve-souris, mais pas du tout pour le slime. J'ai donc dû créer une fonction spécifique pour le slime afin d'appliquer un vecteur de recul adapté uniquement à ce type d'ennemi.

Ainsi, ces ajustements et développements contribuent à enrichir l'expérience de jeu et à offrir des interactions plus dynamiques et immersives.

3.7 Deuxième Personnage

L'implémentation du deuxième personnage s'est avérée relativement simple. Comparé à notre premier personnage, le second inflige moins de dégâts aux ennemis, mais en contrepartie, ses déplacements sont améliorés. Il se déplace plus rapidement, son dash dure plus longtemps et couvre une plus grande distance. Ce personnage est conçu pour les joueurs préférant une approche moins combative.

L'implémentation a été facilitée par le simple ajustement des valeurs de vitesse de déplacement et de dash. Cependant, la tâche la plus complexe a été d'intégrer le choix entre les deux personnages dans la scène solo. Nous avons donc ajouté cette option dans le menu principal du jeu, accessible via un bouton dédié au mode solo. Cette fonctionnalité permet aux joueurs de sélectionner leur personnage préféré avant de commencer la partie, enrichissant ainsi l'expérience de jeu et offrant une plus grande flexibilité dans le gameplay.

3.8 Le Shop

Dans notre jeu, le magasin permet d'acquérir diverses améliorations pour les capacités du personnage, notamment en matière d'attaque et de points de

vie. Afin de rendre le shop à la fois visuel et fonctionnel, j'ai utilisé des boutons pour les différents objets disponibles à l'achat. Ces boutons déclenchent une fonction d'achat chaque fois que le joueur les actionne.

Pour gérer les données nécessaires au bon fonctionnement du magasin, j'ai opté pour un tableau à deux dimensions, permettant de stocker principalement le prix, la quantité et l'identifiant de chaque amélioration. Le prix des améliorations augmente progressivement à mesure que le joueur en achète, afin de maintenir un équilibre avec les niveaux du jeu.

Le principal défi de l'implémentation du magasin a été de concevoir un système optimal. La récupération des pièces du joueur pour le magasin a également présenté des difficultés. J'ai dû mettre en place un box collider dédié à l'achat, permettant de détecter l'entrée du joueur et de mettre à jour le nombre de pièces disponibles dans la boutique. Appliquer les améliorations d'attaque et de points de vie sur le joueur s'est révélé assez complexe. J'ai développé des fonctions spécifiques dans les prefabs du personnage, permettant de modifier les attributs du joueur lorsqu'il entre en contact avec le box collider du shop.
Trois améliorations sont disponibles dans notre jeu :

4 Eirys

4.1 Bilan Soutenance 1

À la dernière soutenance technique j'avais travaillaient sur les jeux dans mes deux domaines principaux : le design graphique et les IA ennemies.

En effet, j'avais créé des pixelarts différents pour chaque niveau afin que les backgrounds soient le plus cohérent possible avec l'ambiance du jeu qu'on attend. J'ai alors dessiné un premier background dans une gamme couleur verte, dans la nature, assez paisible. Ensuite pour le niveau 2 j'ai desinnee un arrière plan vertical, ou la couleur bleu était predominante, et laissant apraraite des poissons et roches sous marines, pour illustrer la plongee du joueur dans les caves profondes sous terraines. Ensuite, pour le niveau 2, j'ai dessiné un arrière-plan vertical, ou la couleur bleue était prédominante,

et laissant apparaître des poissons et roches sous-marines, pour illustrer la plongée du joueur dans les caves profondes sous terrains. Enfin pour le dernier, troisième niveau, je suis partie sur une palette de couleur plus agressive avec un motif de flammes évoquant une ambiance infernale. Ainsi, de tels arrière-plans peuvent permettre au joueur une immersion totale dans l'univers lors de la progression dans le jeu.

J'ai ensuite réalisé quelques autres pixelarts tels qu'un coffre ouvert, ferme, plein et vide, ou des idées de sprite pour le personnage. Enfin, mes dernières réalisations au niveau des graphismes, étaient un logo de notre jeu et une icône. Le logo est une représentation stylisée du nom du jeu " Soul's Dive " avec des éléments rappelant chaque niveau. De même pour l'icône (mais sans texte), ou on retrouve la nature, le feu et l'eau et une arme au milieu.

Le deuxième domaine sur lequel j'ai avancé était celui des IA ennemies. En particulier, je me suis concentrée sur la création d'un ennemi capable de suivre le joueur grâce à un raycast. J'ai donc écrits un script qui permet a l'ennemi de constamment envoyer deux rayons sur la gauche et sur la droite et détecter ce que le rayon touche. Par défaut, il est en idle su place et lorsqu'il détecte un collider dans son rayon alors, s'il s'agit du joueur, alors l'ennemi se déplace vers le joueur grâce à la fonction Vector2.MoveTowards() ; sinon, c'est qu'il s'agit d'un obstacle, il reste donc en idle. De plus grâce au raycast, l'ennemi ne peut pas voir le joueur à travers un obstacle. Je faisais tourner le sprite et l'ennemi dépendamment dans la direction dans laquelle il se déplace grâce a des animations.

J'avais donc réussi à bien avancer sur mes deux domaines, même s'il me restait encore beaucoup à faire.

4.2 Graphismes

Depuis la dernière soutenance, j'ai pu finir le travail sur les graphismes et l'IA ennemie, mais j'ai aussi pu m'occuper de mon troisième domaine, c'est-à-dire le design du son donc la musique et les effets sonores.

Tout d'abord, j'ai continué le design graphique. En effet, dans un premier temps, j'ai refait un nouveau background pour le niveau 1 qui est un peu

plus simple, moins surcharge et qui est plus cohérent avec le style de jeu que nous visons. Il est aussi plus simple à diviser en layers pour plus tard l'animer.

Dans un second temps, je me suis chargé de la création du design du menu du jeu. Je voulais que celui-ci soit en pixelart aussi et qu'il aille bien avec le logo. Il rappelle lui aussi les 3 parties du jeu. J'ai ensuite dessiné des boutons pour accéder à la scène multijoueur, solo, aux paramètres, pour quitter, et le bouton infos avec les crédits son. J'avais fait plusieurs versions, plus ou moins pixelisées. Au départ, je voulais y faire apparaître le personnage principal, mais comme il est très pixelisé par rapport au logo du jeux qui est présent sur le menu, j'ai decidé de le laisser de côté.

La prochaine étape était le design d'un boss. Le but étant le lui donner deux phases d'attaque, il était nécessaire de dessiner deux pixelarts différents : Pour la première et la deuxième phase. Dans la première phase, le boss n'est pas trop agressif, j'ai donc opté pour une couleur bleue, et un dessin sympathique, presque mignon. Le jeu, devant être adapté à un jeune public, je ne voulais pas que le boss roi des démons soit trop agressif visuellement. Ainsi, pour la deuxième phase, j'ai simplement changé la couleur du sprite pour du rouge et j'ai modifié l'expression de son visage. Le roi des démons, ressemble donc un peu a un poulpe, ce qui est cohérent avec le fait que la majorité des ennemis rencontres sont des slimes (au niveau 1) et en suite leurs âmes, au niveau 3. Ainsi, en phase 1 le boss est de la même couleur que les slime du niveau 1 et en phase 2, il change et se fond dans la couleur rouge des enfers.

Enfin, j'ai fait quelques autres pixelarts plus petits tels que par exemple les projectiles de phase d'attaque un et deux du boss ou aide a Guillaume pour le dessin du grand sage. Nous avons aussi travaillé ensemble pour le design des slides du lancement du jeu.

Un autre point que j'ai abordé était l'implémentation des backgrounds avec un effet de parallax dans les trois niveaux. Dans le menu, l'arrière-plan est statique, il était donc très rapide à ajouter, mais pour le reste, j'ai dû écrire des scriptes pour le parallax que j'applique a chaque layer de chaque background, que j'ai dû séparer. Le niveau 1, 3 et la salle du boss se jouant tout autant en vertical qu'en horizontal, je fais déplacer chaque layer à une certaine vitesse de l'effet parallax entre 0 et 1 (à 1 la layer étant fixe horizontalement et a 0, la layer de déplaçant verticalement et horizontalement a la

même vitesse que la camera). En effet les layers sont des enfants de la main camera et se déplacent sur l'axe des y avec elle.

Afin de réaliser un effet de parallax uniquement sur l'axe des y, je récupère la position du joueur en y, pour la distance, je récupère la position de la camera en y et pour déplace la layer je calcule un nouveau Vecteur3 avec en x une valeur constante et en y la somme de la position de départ et la distance.

4.3 IA du Boss

La seconde tache dont j'étais chargée était la programmation d'un boss intelligent pour la fin du jeu. J'ai donc décidé de réutiliser la mécanique de l'ennemi que j'avais créé en début de l'année pour que le boss suive le joueur, mais aussi ajouter un système de tirs de projectiles dans la direction de celui-ci.

Ainsi, j'ai commencé par réimplanter la poursuite pas raycast au boss. Est alors apparu un problème technique : le joueur pouvait simplement sauter au-dessus les rayons et attaquer le boss sans se faire poursuivre. J'ai alors du entièrement changer de technique de poursuite. Je me suis donc décidée pour un boxcollider qui enveloppe toute la salle. Ensuite j'ai limité les mouvements du boss à l'axe des abscisses, pour qu'il de vol pas en essayant de suivre le joueur sur les plateformes, mais qu'il le suive du sol.

Grace a OnTriggerEnter2D() et OnTriggerExit2D(), je regarde si le joueur se trouve dans le box collider et si oui, je le fais se diriger vers le joueur (comme dit précédemment que sur l'axe des x).

Ensuite, je me suis occupée des projectiles de l'ennemi. Afin d'envoyer des projectiles à intervalles réguliers, j'initialise une variable NextFireTime que j'incrémente a chaque update en lui assignant le temps présent Time.time plus la FireRate. Son but est de décompter le moment d'envoyer le projectile suivant. Ensuite dans chaque fonction update je compare le temps présent avec NextFireTime pour savoir si le projectile doit être envoyé. Si oui alors j'appelle une autre fonction " FireProjectile() ". Dans celle-ci, j'initialise un nouveau projectile puis je le fais se déplacer vers le joueur, à partir d'un point d'apparition sur l'ennemi.

Du côté du code des projectiles en eux même, dans leur code, on utilise la méthode OnTriggerEnter2D() pour voir si les colliders des projectiles touchent un autre objet. Si oui alors on vérifie si l'objet en question est le joueur ou non. Dans les deux cas, le projectile explose, mais si l'objet touché est le joueur, on va en plus réduire ses points de vie.

Enfin, le boss a deux phases, la deuxième le rendant plus agressif. Le boss passe en deuxième phase après avoir perdu au moins la moitie de ses points de vie. Je vérifie donc dans un script a part dans quelle phase est, en regardant si la valeur du curseur de vie courante du boss est inférieur ou égale a la vie maximale de ce dernier. Lorsque le boss est en phase 2, je le fais lancer des projectiles a des intervalles plus réguliers.

Ainsi, le boss a sa scène dédiée dans laquelle il chasse le joueur qui essaye de la combattre et de finir le jeu.

4.4 Sound Design

Enfin, selon la répartition des taches du cahier des charges, je suis aussi responsable du pôle du sound design. Mes taches principales dans ce domaine étaient donc l'ajout au jeu de la musique et des sound effects.

Tout d'abord, notre jeu vidéo a besoin de musique de fond, afin de permettre au joueur une expérience plus agréable et plus immersive. Ainsi, lors du déroulement d'une partie, la musique était indispensable et il était préférable qu'elle varie, afin de ne pas être trop monotone, ennuyeuse, irritante pour le joueur et afin de mieux distinguer les divers parties du jeu. Après consultation avec les autres membres de l'équipe, nous avons décidé que la musique serait de type 8bits et qu'au premier niveau, on entendra un thème principal. Au deuxième niveau, la musique serait très différente, en guise de transition, et aussi, car la mécanique du 2e niveau est assez lointaine de celles du 1er et 3e. Cela permettrait aussi un changement d'ambiance, et un jeu moins monotone. Enfin, le troisième niveau devrait reprendre le thème principal, mais avec des variations. En effet, elle devrait être plus agressive et rythme, pour mieux aller avec les enfers.

Tout cela étant décidé, il ne resta plus qu'à créer la musique. Le problème, c'est alors posé qu'aucun membre du groupe ne sait composer ou créer de musique, ou n'en a les outils. Or, nous voulions que le jeu soit le plus que possible fait seuls. Ainsi, j'ai demandé de l'aide à mon petit frère, Yann Devigne, qui accepta avec joie de nous aider avec joie. Je lui ai expliqué les détails sur la musique qu'on aimerait et pour l'aider à se mettre dans l'ambiance de notre jeu, je lui ai montrée des pixel arts, explique le principe et montrée ce qu'on avait du jeu à ce moment-là. Il a ensuite composé 3 musiques entièrement seules sur un piano électrique et les a enregistrés. Les musiques qu'il a créées étaient parfaites pour l'ambiance visée. De plus, le thème principal reste dans la tête, ce qui causera les joueurs a pensé à notre jeu même quand ils n'y jouent plus et avoir une bonne association. On a donc eu grâce à lui des musiques pour tout le déroulement d'une partie.

La dernière étape pour Charlène et moi était donc d'importer ces musiques dans Unity, et de les implémenter a chaque scène, en ajoutant à chaque fois nouveau composant Audio Source, et faisant lire les bons enregistrements. Enfin, nous avons coché la case "loop" afin que la musique se boucle si le joueur passe plus de temps sur un niveau que le temps d'un enregistrement.

Ensuite, notre jeu ne serait pas complet sans effets spéciaux sonores. En effet, c'est grâce aux SFX que le jeu est encore davantage polis et finis. Les effets sonores permettent aussi une expérience de jeu beaucoup plus agréable. Cette fois, nous n'allions pas les créer nous-mêmes, nous avons donc décidé de les importer la librairie ouverte en ligne. J'ai alors trouve le site Pixabay qui donne accès gratuitement a une très grande quantité d'effets sonores. La recherche, c'est avéré plus complexe que prévu. En effet, il y avait vraiment beaucoup de sons au choix, et ce n'était pas facile de trouver des effets qui correspondent exactement a mes attendus et ce que je veux pour notre jeu. Après beaucoup d'écoute de beaucoup de sons, j'ai finalement trouvée tous les SFX nécessaires. Une partie importante de cette étape était de, avant de choisir les sons, de choisir pour quels événements du jeu, on veut des sons. Il me sembla alors pertinent d'inclure des sons pour les bruits des pièces ramassées, la récupération de la clef, l'ouverture des portails, les slash de l'épée, les bruits de mort les ennemis, les explosions des projectiles et bombes, etc. Il était important a ce que les sons soit fluides et présents, mais aussi qu'il n'y en ait pas trop, car cela causerait une cacophonie.

Il ne restait alors plus qu'à les implémenter. J'ai donc dû retrouver dans les scripts des différents prefabs les endroits ou mettre les effets sonores. Pour insérer un son, j'utilise AudioSource.PlayClipAtPoint(NomDuSon, positionDuSon, 1f) ;, 1f correspond au volume du son. Le son pour lequel l'implémentation était plus compliquée était les explosions des projectiles des canons. En effet, le joueur ne doit pas pouvoir entendre toutes les explosions du niveau 3, cela causerai une cacophonie et beaucoup de bruit non-nécessaire et dérangeant. J'ai donc utilisé les méthodes OnBecameVisible() et OnBecameInvisible() pour détecter si les projectiles explosent sur l'écran visible par la camera. Ainsi, on s'entend que les explosions que l'on voit.

5 Lucas

5.1 Git

Suite à la première soutenance, je me suis rendu compte qu'une grosse partie de nos problèmes venaient de conflits sur Git et de manque de connaissance sur l'utilisation de cet outil. J'ai donc pris plusieurs heures pour apprendre en profondeur cet outil, et j'ai ensuite expliqué en détail à notre équipe comment gérer plusieurs branches, merge proprement, et les bonnes pratiques utilisées dans le monde professionel, et ainsi accroître notre productivité tout en minimisant le temps perdu à cause de conflits.

5.2 Multijoueur

Après la première soutenance, j'ai continué à développer le mode multijoueur de notre jeu. Pour ce faire, j'ai dû relever un gros challenge, synchroniser la génération procédurale de la carte des niveaux 1 et 3, ainsi que tout le reste du jeu.

J'ai donc commencé une longue phase de recherche et d'apprentissage de l'outil Netcode for GameObjects de Unity. J'ai commencé par apprendre la synchronisation d'objets à travers le réseau à l'aide d'appels de procédures à distance, ou RPC. Il en existe 2 types : ClientRpc et ServerRpc.

Un ServerRpc est une fonction appelée par un client, qui s'exécute sur le serveur. Ce type de fonction est utile pour envoyer une information au serveur, comme mettre à jour la vie d'un ennemi après qu'un client lui ait infligé des dégâts, ou detruire un objet à travers le réseau.

Un ClientRpc est une fonctione appelée par le Serveur, qui s'exécute chez le client. Ce type d'appel est utile lorsque le serveur veut mettre à jour des informations chez les joueurs, comme mettre à jour une information de manière globale tel qu'un affichage ou bien une stat de point de vie.

Ces fonctions sont essentiels pour développer un jeu multijoueur. En effet, par défault le Client n'as aucune autorité de modification de valeurs synchronisés sur le réseau. Pour rendre un jeu multijoueur agréable et identique pour chque joueur, qu'importe s'il est hôte ou client, il faut utiliser une architecture ServerRpc/ClientRpc afin de faire valider les opérations que demandent les clients par le serveur, les exécuter, puis mettre à jour les informations chez les clients.

Egalement, afin d'avoir un multijoueur plus abouti que celui de la soutenance précédente, où l'on devait donner l'adresse ip de la machine à laquelle on voulait se connecter, j'ai dû apprendre les librairies Lobby et Relay de Unity, puis créer l'interface qui permet en jeu de rejoindre un lobby, puis de lancer une partie.

J'ai donc commencé par implémenter la libraire lobby, ce qui nous as permis de remplacer la necessité d'une adresse IP par un simple code pour rejoindre le Lobby d'un hôte et s'y connecter. une fois dans le lobby, il est possible de synchroniser des données à l'aide d'un attribut Data du lobby, ce qui as permis l'implémentation de la librairie Relay : lorsqu'un hôte veut lancer la partie, il attend la création d'un nouveau relay et s'y connecte, et une fois connecté, met à jour une valeur dans Data, qui servira de code pour rejoindre le relay. Lorsque le joueur détecte le changemement de valeur, il se connecte au relay, et la partie peut commencer !

Ces librairies m'ont initiés à un tout nouveau paradigme de programmation: la programmation asynchrone. En effet, les fonctions de ces libraires sont asynchrones, donc doivent être attendues grâce au mot clé "await", ce qui permet de continuer à exécuter le reste du code en attendant la réponse de

la librairie, sans quoi le jeu se bloquerait complètement à chaque intéraction avec la librairie.

Néanmoins, un nouveau problème s'est posé, à savoir la génération du terrain en lui-même. En effet, les salles générées ne contiennent pas le terrain directement, mais des points d'apparitions. J'ai donc modifié la génération procédurale afin que ce soit le joueur hôte qui crée le terrain; vu que les points d'apparition sont les mêmes chez l'hôte et le client, le terrain est désormais synchronisé.

Désormais, il fallait téléporter les joueurs au point d'apparition. J'ai alors rencontré plusieurs problèmes. Tout d'abord, les joueurs se téléportaient immédiatement après avoir trouvé le point d'apparition, mais la génération du terrain n'étant pas forcément terminée. Il était alors fréquent qu'un, voir que les 2 joueurs se retrouvent coincés dans un mur.

J'ai donc utilisé une combinaison de ServerRpc/ClientRpc que propose Netcode for GameObjects. Une fois que la génération des salles est terminée, un signal est envoyé à l'hôte. Une fois ce signal récupéré, l'hôte envoie à son tour un signal aux clients pour générer le terrain. Enfin, lorsque la génération du terrain est terminée localement, le joueur se téléporte au point d'apparition. Ce système assure le bon fonctionnement de la génération procédurale en multijoueur.

Désormais, les joueurs sont synchronisés et jouent sur la même carte. Ensuite, j'ai synchronisé les autres éléments du jeu: les pièces, les ennemis, la boutique, la clé, et le portail qui permet de passer au niveau suivant. Pour la clé, j'ai fait en sorte que lorsque la clé est récupérée, l'indicateur change pour les 2 joueurs, et que n'importe quel joueur puisse passer au niveau 2.

Pour les ennemis, j'ai fait en sorte que leurs points de vie soient synchronisés, et que la barre de vie soit mise à jour chez les 2 joueurs losque l'un des joueurs inflige des dégâts. Egalement, lorsque qu'un ennemi est tué, il est détruit chez tout les joueurs. De même, pour les pièces, lorsque qu'un joueur en récupère elle disparait chez les 2 joueurs. J'ai implémenté ces fonctionnalités grâce à plusieurs ServerRpc et ClientRpc.

Pour assurer une transition synchronisée entre les joueurs, j'ai donc dû adapter

le chargement des nouvelles scènes. Netcode for GameObject permet celà très simplement à l'aide d'un composant de gestion de scènes synchronisé sur le réseau, et de la définition d'un ServerRpc qui s'occupe de faire charger le niveau aux 2 joueurs de manière simultanée.

Dans le niveau 2, la génération procédurale est plus simple : seul la disposition des pièces, mines et des projectiles est aléatoire. Je les ai donc synchronisés de la même manière que dans le premier niveau. J'ai néanmoins dû adapter les mines du niveau au multijoueur pour qu'elles soient bien instanciées chez les 2 joueurs.

Dans le niveau 3, la génération procédural fonctionne de la même manière que dans le niveau 1. J'ai néanmoins eu beaucoup de difficultés à correctement synchroniser la carte ainsi que l'apparition des joueurs à cause du délai inhérent au multijoueur, l'hôte recevant les informations toujours plus tôt que le client. J'ai donc dû rajouter un délai chez le client afin de ne pas générer le terrain ou d'apparaître trop tôt.

Enfin, le dernier Niveau ne contient qu'une génération aléatoire pour quelques éléments de décor. Je me suis donc concentré sur l'adaptation du boss de notre jeu, de ses points de vie, et de ses animations. Il as fallu également adapter son comportement afin qu'il ne vise pas que l'hôte avec ses projectiles.

5.3 Boucle de gameplay

Dans le même temps, j'ai également cherché à finaliser la boucle de gameplay, étant donné que notre jeu est un Rogue-Like. J'ai donc implémenté un système de sauvegarde des différents attributs à l'aide de la classe PlayerPrefs de Unity, qui est un simple système de sauvegarde d'attributs dans un dictionnaire. Ainsi, on sauvegarde le nombre de parties jouées d'affilés sans mourir, le record de parties jouées sans mourir, les points de vies maximum, les dégâts d'attaque, et la puissance de soin de notre personnage lorsque le joueur termine une partie.

Vu que l'on stocke le nombre de parties jouées sans mourir, on peut donc également faire augmenter les dégâts et les points de vie des ennemis, ainsi que les dégâts des obstacles en fonction de ce nombre, pour offrir un challenge

équilibré au joueur. : à chaque partie, il peut collecter un certain nombre de pièces pour acheter des bonus, mais à chaque partie les ennemis deviennent plus fort.

Une fois la boucle de gameplay implémentée, il as fallu faire beaucoup de parties test afin d'équilibrer correctement le jeu, et de ne pas trop avantager un style de jeu agressif ou pénaliser un style de jeu passif. J'ai donc décidé de rendre la progression des attributs du personnage ainsi que des ennemis linéaires, tout comme l'augmentation des prix des bonus disponibles dans la boutique.

6 Guillaume

6.1 Génération Procédurale

D'après le tableau des tâches, ma tâche principale assignée était la génération des niveaux. Je me suis donc concentré principalement sur la génération procédurale des ces derniers au cours de la première période. Dans cette première section, je détaillerai le travail que j'ai accompli pour rendre cet aspect du jeu fonctionnel.

Pour commencer, il est important de rappeler ce qu'est la génération procédurale. Il s'agit d'une technique qui permet de générer aléatoirement certains éléments des niveaux du jeu au début de chaque partie, tels que les environnements, la disposition des ennemis et des objets. L'objectif de cette technique est d'assurer une expérience de jeu dynamique et nouvelle à chaque partie, garantissant que le joueur ne rencontre jamais deux fois le même niveau. La première étape a donc été de trouver un système de génération des niveaux qui respecte cette condition.

En premier temps, je me suis concentré sur la génération des tiles. Mon travail a commencé avec la recherche de différents algorithmes de génération procédurale utilisés dans d'autres jeux. J'ai étudié des méthodes telles que l'algorithme de Perlin Noise, les systèmes de tuiles (tile-based systems) et les générateurs de donjons basés sur des graphes (graph-based dungeon generators).

Voici une explication détaillée de l'algorithme de génération des niveaux que j'ai finalement utilisée :

Le niveau est d'abord divisé en une grille de 4x4 cellules, chaque cellule correspondant à une "room" ou une pièce du jeu. Chaque pièce est de taille fixe, dans notre cas 10x10 tuiles, ce qui permet de structurer facilement le niveau global. Pour chaque cellule de la grille, l'algorithme sélectionne une prefab de pièce parmi un ensemble de prefabs de pièces. Ces prefab peuvent être de différents types :

- room LR : ouvertures à gauche et à droite
- room LRB : ouvertures à gauche, à droite et en bas
- room LRT : ouvertures à gauche, à droite et en haut
- room LRTB : ouvertures sur chaque côté de la pièce

Ces prefabs sont elles-même stockées dans des "room templates", qui sont des gameobject contenant toutes les rooms du même type. Ceux-ci sont renseignés dans des champs du script de génération des niveaux. Il existe également plusieurs rooms spéciales, qui sont :

- Starting room : la première room de la génération, dans laquelle apparaît le personnage
- Last room : la dernière room de la génération, dans laquelle apparaît le portail permettant de passer au niveau suivant
- Key room : la room dans laquelle est instanciée la clé permettant de déverrouiller le portail de la Last Room
- Chest room : une room contenant un coffre contenant une quantité élevée de collectibles

Ces dernières sont directement renseignées dans des champs du script de génération des niveaux.

Maintenant, passons au fonctionnement du script en lui-même. Il commence par instancier la Starting room dans l'une des 4 cellules de la première ligne de la grille, en se servant des coordonnées de spawn points (ou points

d'apparitions) placés au centre de chacune d'entre elles. Ces positions sont en effet renseignées dans une liste qui existe en champ du script.

Ensuite, le script va instancier une room dans une position libre au hasard à gauche, à droite ou en bas de cette pièce actuelle. A partir de là, le script va faire apparaître un chemin de rooms qui va de la première ligne de la grille jusqu'à la dernière, en faisant en sorte que le joueur puisse traverser ce chemin de bout en bout sans jamais rencontrer d'impasses. Grâce aux différents types de rooms dont on dispose, ce processus n'engendre pas de difficultés majeures. Il n'existe qu'une situation qui pose problème : celui où le script instancie une pièce en dessous d'une autre qui ne possède pas d'ouverture vers le bas, compromettant ainsi le chemin. Pour gérer ce cas de figure, j'ai utilisé le script RoomType qui vise spécifiquement à détruire la room du dessus afin de la remplacer par une pièce possédant une ouverture vers le bas. Pour ce faire, j'utilise la fonction Physics2D.OverlapCircle qui permet de détecter la room du dessus. Ce script est également utilisé lorsque la dernière pièce est placée afin de la détruire et de la remplacer par la Last room.

Une fois le chemin placé, il restera plusieurs positions vides dans la grille. Chaque cellule étant marquée d'un point de coordonnées noté Posx dans la hiérarchie, il me suffit d'utiliser Physics2D.OverlapCircle afin de détecter la présence ou non d'une room. Tout d'abord, le script récupère l'une des positions libres et y place la Key room, puis une autre si possible afin d'y placer la Chest room. Sur les points de coordonnées sur lesquels aucune pièce n'est encore placée est appelé le script SpawnRoom, qui permet d'instancier une room au hasard afin de compléter le niveau et de la rendre plus labyrinthique.

La partie la plus complexe de la génération des niveaux a de loin été la Tilemap. En effet, Unity propose un outil appelé Tilemap, qui permet de créer des niveaux 2D à l'aide d'une grille de tuiles. Elles sont particulièrement utiles pour peindre des niveaux 2D grâce à l'usage des Ruletiles, des tuiles particulières dont l'apparence s'adapte aux tuiles environnantes. L'usage d'une Ruletile est normalement assez simple ; cependant, il se complique beaucoup avec l'introduction de la génération procédurale des niveaux. Je ne peux pas directement peindre des tuiles dans mes Prefabs de pièces, car cela reviendrait à avoir une Tilemap par pièce, ce qui empêcherait les Ruletiles d'une pièce de s'adapter à celles d'autres pièces.

La solution que j'ai retenue a été d'instancier les Ruletiles sur certains points de spawn dans le script de la génération des niveaux. Pour cela, j'ai donné le tag "SpawnRuleTile" à chaque Spawn point dans mes Prefabs de pièces sur lequel je voulais voir une tuile apparaître. Je les stocke ensuite tous dans une liste avec la méthode GameObject.FindWithTag(), avant d'instancier une tuile sur chacun d'eux dans mon script.

Cet algorithme est utilisé pour générer les premiers et troisièmes niveaux, ceux qui utilisent de la génération procédurale.

6.2 Niveaux deux et trois

En plus de la génération du premier et du troisième niveau, j'ai également travaillé sur la répartition du deuxième niveau. Dans celui-ci, le joueur apparaît à un point fixe en haut de la salle et doit se laisser chuter jusqu'en bas du niveau en évitant les projectiles et les mines.

Pour que le déroulement de ce niveau se fasse sans problèmes, j'ai commencé par désactiver la fonction qui gère le saut mural dans ce niveau afin d'empêcher le joueur de remonter le long des murs. Pour ce faire, je récupère le nom de la Scene actuelle dans le script qui gère les mouvements du joueur avec SceneManager.GetActiveScene().name.

Je me suis également occupé de programmer les mines dans le deuxième niveau. Pour leur donner un effet d'explosion dynamique et attrayant quand le personnage rentre en contact avec elles, j'ai suivi des tutoriels variés sur YouTube afin d'apprendre à me servir des Particle Effects d'Unity. Cette compétence m'a été utile par la suite, car j'ai également pu importer cet effet pour l'animation de mort des différents ennemis et celle de l'ouverture du coffre.

Toujours dans le niveau deuxième, afin d'augmenter un peu la difficulté et de rendre la chute moins monotone, j'ai décidé d'implémenter des projectiles qui apparaissent en bas de l'écran et remontent vers le joueur. J'ai utilisé mes compétences nouvellement acquises en Particle Effects pour leur donner un effet d'électricité.

C'est après avoir implémenté ces deux fonctionnalités que je me suis rendu compte d'un problème dans le deuxième niveau : les mines et les projectiles arrivaient trop rapidement depuis le bas de l'écran, rendant difficile leur évitement même en restant vigilant. Pour remédier à cela, j'ai ajusté la caméra pour qu'au début de la descente, elle soit légèrement décalée sous le joueur, se recentrant progressivement sur lui à mesure qu'il descend. Cette solution était relativement simple car elle ne nécessitait que la vérification d'une position y, ajustant ainsi la dynamique du jeu pour offrir une expérience plus équilibrée et jouable.

Ensuite, j'ai entrepris le développement du troisième niveau du jeu. Bien qu'il suive le même principe que le premier, il présente plusieurs différences notables. La difficulté y est plus élevée, avec l'introduction de canons qui remplacent les chauve-souris du niveau précédent. Pour implémenter ces canons, j'ai conçu un algorithme qui leur permet de lancer des projectiles similaires à ceux du deuxième niveau, mais avec une direction déterminée en fonction de la position du joueur par rapport à eux. J'ai utilisé la fonction GameObject.FindWithTag() pour constamment suivre la position du joueur et ainsi ajuster la trajectoire des projectiles lancés : vers la gauche s'il est à gauche du canon, et vers la droite s'il est à droite. Ce processus de développement a permis d'accentuer la complexité du jeu tout en maintenant une cohérence dans les mécaniques introduites, offrant aux joueurs une progression progressive et enrichissante à travers les différents niveaux.

De plus, pour mieux équilibrer la difficulté des niveaux, j'ai mis en place un système où la probabilité d'apparition d'ennemis dans le premier niveau est augmentée en fonction du nombre d'ennemis éliminés précédemment. Pour ce faire, j'ai récupéré le nombre d'ennemis éliminés à partir des PlayerPrefs, ce qui permet de suivre la progression du joueur tout au long de ses sessions de jeu. Ainsi, plus le joueur élimine d'ennemis dans le premier niveau, plus la fréquence d'apparition des slimes sera léevée dans le troisième niveau.

Cette approche du gameplay incite les joueurs à adopter une planification plus poussée et à ajuster constamment leur stratégie en fonction de l'évolution du défi à travers les différents niveaux du jeu.

J'ai également créé les graphismes en pixel art pour le joueur, les ennemis et les divers objets du jeu.

7 Charlène

7.1 Le site web

Concernant le site web, il avait déjà été réalisé pour la soutenance précédente mais il y avait eu un problème concernant l'hébergement du site web et donc de sa mise en ligne. Cela a pu être fixé grâce aux services d'hébergement que propose Github. J'ai pu donc me connecter à mon espace github préalablement créé pour notre projet. J'ai ensuite créé un repository avec le nom de notre jeu et j'y ai téléchargé les fichiers HTML, Javascript et CSS modifiés à l'aide de Visual Studio Code ou encore Rider ainsi que les images correspondantes à des démonstrations de notre jeu, et autre. J'ai ajouté un fichier HTML index.html qui me servira de page d'accueil et grâce à laquelle les utilisateurs pourront accéder aux différentes pages. Puis GitHub permet de mettre en ligne mon site web à l'aide d'un lien GitHub relié à mon espace utilisateur soit https://leune00.github.io/soulsdive.github.io/

Sur ce site web, j'ai actualisé les paramètres tels que les sprites de décoration car ils correspondaient encore à ceux des premiers rendus du jeux mais ils n'étaient pas très esthétiques ainsi on a amélioré le jeu, les décors et autre. J'ai donc ajouté de nouveaux décors pour compléter le site mais j'ai surtout complété les différentes informations manquantes puisque le projet était en cours. J'ai ajouté sur la page à propos de l'équipe, une frise chronologique complète qui regroupe les différents événements majeurs au cours du projet : nous avons la création de l'équipe BeexL Créations en septembre/ octobre 2023, puis le lancement du projet "Soul's Dive" ainsi que la remise du cahier des charges techniques en octobre/ novembre 2023. Puis nous avons en janvier 2024 a eu lieu la première soutenance méthodologique. En février 2024 nous avons eu la mise en ligne du site web, en mars 2024 la première soutenance technique et finalement en juin 2024 nous avons la dernière soutenance technique, celle qui nous permettra de soumettre notre projet et c'est le rendu final de tous les documents nécessaires à la réalisation de ce projet.

De plus, nous avons décidé de remplacer nos photos par des simulations de personnages avec des designs pixélisés de chacun des membres de l'équipe car

nous trouvions qu'un effet pixel aurait été plus en accord avec le thème et le design du site et ainsi du jeu en 2D et pixel. Nous les avons créés à l'aide de ce site web:

https://jazzybee.itch.io/sdvcharactercreator

Concernant la "page du jeu" j'ai ajouté les nouveaux ennemis, les différents personnages disponibles, quelques décors, qui correspondent aux différents éléments que l'on peut retrouver pour que l'utilisateur ait une idée concrète de l'aspect du jeu.
Nous pouvons aussi y retrouver rapidement une sorte de citation/ script en tant qu'avant goût du jeu, afin de nous décrire un peu le jeu.

Concernant la page de téléchargement, nous pouvons y retrouver un manuel d'installation, un manuel de désinstallation et un exécutable nous permettant de télécharger le jeu. Nous pouvons aussi télécharger le rapport de soutenance final.

Au niveau de la forme du site web, cela a aussi été modifié via les différents fichiers CSS avec différents paramètres permettant de changer la mise en forme des différents éléments apparaissant sur le site web en ligne.

Le site web a totalement été réalisé en anglais puisque nous nous étions convenus dans le cahier des charges techniques de créer un jeu entièrement en langue anglaise et ainsi pour être en accord avec le jeu, j'ai décidé de mettre le site aussi.

7.2 Les menus et interface

Je me suis chargée des menus et de l'interface du jeu.

En effet grâce aux sprites réalisés par Eirys, j'ai pu réaliser un menu principal qui regroupe différentes options nécessaires à la bonne utilisation du jeu ou encore pour le confort du joueur. Nous avons tout d'abord le choix entre jouer en single player ou en multijoueur.

Nous avons aussi un bouton "settings" qui permet de régler les différents paramètres comme le volume de la bande son du jeu, la possibilité d'être en plein écran ainsi que deux boutons permettant de retourner à la scène voulue soit single player ou multiplayer. Cela est faisable grâce à la fonction Settings() et SetVolume(). Il y a aussi la possibilité de muter le volume si l'utilisateur le souhaite. Tout cela a été réalisé à l'aide de fonctions en Csharp (C#) présentes dans le script "MainMenu" ainsi que la création de panels dans un canvas dans la scène Menus sur notre projet Unity.

Grâce à une fonction Quit(), nous pouvons en cliquant facilement sur le bouton associé, quitter le jeu.

En cliquant sur le bouton "?" en bas à droite, nous avons un panel qui apparaît avec des crédits associés tels que pour les bandes sons du jeu, réalisés par le frère d'Eirys, Yann DEVIGNE.

Si nous cliquons sur le bouton Singleplayer, une autre "vue" apparaît. Ici nous avons donc le choix entre plusieurs personnages ainsi que le mode.

Nous pouvons choisir entre deux personnages : l'homme à l'épée ou l'assassin. Il y a aussi un bouton retour qui nous permet de revenir au menu principal. Un mode expert a aussi été paramétré et nous pouvons le sélectionner en cochant la case associée à côté.

Nous pouvons apercevoir en haut à droite deux autres informations telles que le meilleur score obtenu au fil des parties effectuées ainsi qu'en dessous un autre score comme le score de parties gagnées d'affilée, on appelle cela des "streaks".

Lorsqu'on clique sur le bouton multijoueur, nous avons un nouvel affichage. Tout d'abord nous avons un bouton permettant de nous identifier grâce au bouton Authentification. Puis si nous décidons d'être le créateur d'une partie, nous cliquons sur le bouton "Host" et un code s'affiche à l'écran. Le client peut à son tour s'identifier et cliquer sur le bouton associé et taper le code pour joindre la partie avec le bouton "Join". Après avoir pressé la touche "enter", il peut donc joindre la partie et le créateur peut ainsi commencer la partie à l'aide du bouton "Start Game".

Pendant la partie, plusieurs interactions peuvent être possibles comme lorsqu'on peut acheter plusieurs éléments grâce au magasin "le shop". Il est indiqué plusieurs fois que l'on peut presser la touche "echap" ou encore "E" pour ouvrir un coffre ou accéder justement au shop.

7.3 Le menu pause

Lorsque la touche "echap" est sélectionnée, un menu va s'afficher et va donc mettre en pause le jeu. Cela est effectué à l'aide d'un script et d'un panel "Menu pause".

Nous avons donc un fond assez transparent coloré vert pour rappeler le thème du jeu et nous pouvons retrouver différents autres boutons comme un bouton "Resume" permettant de retourner et continuer le jeu. Nous avons aussi un bouton "settings" qui affiche le menu des paramètres pour régler les paramètres depuis le menu pause. Finalement nous avons un bouton Quit pour quitter finalement le jeu.

7.4 Le menu Game Over

Lorsque le joueur n'a plus assez de vie et qu'il vient à mourir, une animation est mise en place. Encore une fois, trois options sont proposées : un bouton main menu pour retourner au menu principal, un bouton settings pour les mêmes raisons déjà évoquées et un bouton Quit. Comme le menu de pause, un fond vert assez transparent apparaît par-dessus la scène de jeu.

8 Conclusion

8.1 Bilan

Le projet Soul's Dive a été une entreprise intéressante et enrichissante pour tous les membres de BeexL Creations. Tout au long du processus de développement, nous avons rencontré et surmonté divers défis, et chaque membre de l'équipe a contribué de manière importante à l'avancement du jeu. Nous avons globalement respecté les objectifs fixés dans notre cahier des charges initial et,

malgré quelques ajustements nécessaires en cours de route, nous sommes restés en phase avec notre planning initial.

Soul's Dive est donc un jeu possédant des mécaniques complexes telles qu'une palette de mouvement variée du personnage qui donne au joueur la possibilité de se déplacer librement dans les niveaux du jeu. Ces derniers, générés aléatoirement par un algorithme de génération procédurale, permettent une expérience nouvelle à chaque partie et donnent au jeu une rejouabilité importante. La difficulté du jeu est soigneusement équilibrée : par exemple, plus le joueur élimine de monstres dans le premier niveau, plus les ennemis deviennent redoutables dans le troisième niveau. Cette progression dynamique incite les joueurs à planifier stratégiquement leurs actions et à adapter leur approche tout au long du jeu. Par ailleurs, ces derniers disposent de plusieurs possibilités pour rendre leur parties plus ardues, que ce soit changer de personnage ou jouer en mode expert. Il est également possible de jouer au jeu en coopération avec un ami à l'aide du mode multijoueur en ligne.

Pour résumer, Soul's Dive est un jeu au gameplay rythmé et dynamique qui saura plaire à un public de tous âges. De plus, le jeu étant entièrement en anglais, il pourra toucher un grand nombre de personnes à l'international, renforçant ainsi son potentiel de succès sur le marché mondial.