GAP : Rapport Final

Guillaume Ambal Hugo Menet Lois Paulin Nicolas Derumigny
Pierre-Etienne Polet Pierre Mahmoud– Lamy Redouane Elghazi
Thomas Ragel Xavier Badin de Montjoye Yannis Gaziello

Mai 2017

Table des matières

1 Introduction

Le marché du jeu vidéo s'ouvre de plus en plus à un public adulte, ce qui signifie que désormais c'est l'acheteur qui est consommateur du produit, et non plus son enfant. De ce fait, l'exigence en terme de qualité est de plus en plus grande et notamment en terme de durée de vie. Une des questions est donc alors : *Comment obtenir une durée de vie infinie ?* Si avec *Minecraft* le système dit de *sandbox* ou *bac à sable* séduit tout en répondant à ces critères, une grande partie du marché souhaite une longue durée de vie sans être bornée par sa créativité ou ses capacités artistiques. Pour répondra à cela, viennent les jeux qui se génèrent aléatoirement au fur et à mesure. Malheureusement dans la plupart d'entre eux, des schémas prédéfinis sont entrés dans le programme, qui se contente de les placer dans un ordre aléatoire.

GAP est l'acronyme de Générateur Aléatoire de Plates-formes. Ici, notre but est de créer un jeu de plates-formes dans lequel le joueur peut générer en un clic autant de niveaux qu'il le souhaite tout en conservant une certaine diversité.

1.1 Gameplay

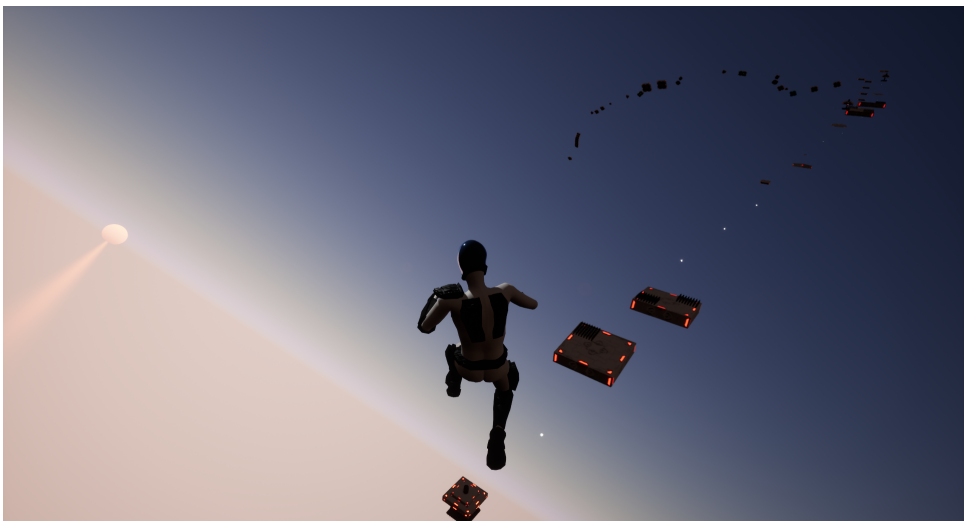

Figure 1 – Rendu en jeu

Le but du jeu est d'aller d'un bout du monde à l'autre en sautant de bloc en bloc. Pour y arriver, il faudra changer la gravité, éviter les piques, prendre garde à ne pas tomber et surtout se déplacer dans la quatrième dimension pour trouver la voie.

En plus de cela, par simple pression sur un bouton, il est possible de générer des niveaux à volonté !

1.2 Placement par rapport aux autres jeux

Lorsqu'on parle de jeux aux possibilités presque infinies, on pense immédiatement à l'échec que fut *No Man's Sky*. En effet, malgré une immense diversité visuelle, le gameplay reste répétitif, ce a qui certes ravi les fans de jeux d'exploration, mais également déçu le grand public.

Au niveau des jeux de plates-formes, l'application *Temple Run* sur téléphone portable cherche elle aussi à créer un jeu sans fin. Cependant, le jeu tombe lui aussi dans la répétition et ce de par son

fonctionnement même. En effet, dans ce jeu, une dizaine d'obstacles arrivent dans un ordre aléatoire jusqu'à ce que le joueur meure. Ses possibilités ainsi limitées le placent au rang de jeu passe-temps plutôt que de jeu intéressant.

Finalement, un des jeux se rapprochant le plus de ce qui nous intéresse est sans doute *Super Mario Maker*. Mais là, au lieu d'utiliser un algorithme, *Nintendo* décide de profiter de sa communauté pour produire suffisamment de niveaux pour que ses joueurs ne tombent jamais à court. Cependant lorsqu'on joue au jeu, on se rend compte que les niveaux ainsi créés sont pour la plupart peu esthétiques. En effet, dans le but de se démarquer, de faire un niveau très difficile ou par goût personnel, les niveaux créés par les amateurs n'ont pas l'équilibre d'une création professionnelle.

Ainsi, en créant un générateur de plates-formes aléatoire non basé sur la répétition d'un motif, nous espèrons pouvoir créer un jeu stimulant qui conserve toutefois une identité visuelle.

Pour cela, il nous faut une singularité propre à notre jeu pouvant apporter des mécaniques différentes. Nous avons décidé que notre jeu n'utiliserait pas trois mais quatre dimensions.

Lorsque nous parlons de jeu en 4 dimensions, nous ne parlons pas d'immersion physique, mais de l'espace dans lequel notre personnage peut se déplacer. Même si ce concept existe déjà dans le monde vidéo-ludique, à notre connaissance aucune grande franchise ne semble l'utiliser de la même façon que nous.

2 La quatrième dimension

Comment représenter cette quatrième dimension ?

Il n'est déjà pas toujours aisé de produire un rendu de jeu en 3 dimensions sur nos écrans plats. Une quatrième semble donc assez difficile à appréhender. De plus, nous n'avons pas de compréhension intuitive de ce qui se passe en 4 dimensions.

L'astuce est donc de ne pas la représenter, et de se contenter de montrer une projection en trois dimensions. Pour mieux le comprendre, on peut imaginer un scanner médical. Ce que nous avons sous les yeux correspond à une coupe en 2 dimensions d'un être vivant lui même en 3 dimension. Ainsi, on peut visionner une vidéo correspondant à une succession de coupes, qui commencerait par nous montrer une petite tâche grossissante au fur et à mesure qu'on avance et qui formerait la tête, avant de se séparer en trois avec notre tronc et nos deux bras, pour redevenir un et finir par deux tâches s'éclipsant lorsqu'on aura atteint la plante des pieds.

Ainsi, tout comme nous, notre personnage ne possède des sens que pour observer en trois dimension mais possède un appareil pouvant l'aider à se déplacer dans une quatrième. Les objets sur lesquels notre personnage évolue étant continus, lorsque l'appareil est utilisé, ils se déforment, se déplacent, et à un moment disparaissent. En appelant k la quatrième dimension et x une dimension parallèle au sol, on peut considérer une plate-forme représentée par la Figure 2 dans (x, k).

On remarque qu'en se déplaçant sur k, d'abord notre plate-forme apparaît, puis elle s'agrandit, avant de se stabiliser et enfin de rétrécir pour disparaître.

Ce modèle mathématique, cette vision de la 4D, a l'avantage de ne pas avoir besoin d'être compris par le joueur. En effet, il peut juste considérer qu'il possède une machine pouvant rétrécir, agrandir ou déplacer certains objets.

Par souci de commodité on indiquera à l'aide d'un petit scintillement où se trouvent les objets non visibles à l'écran pour encourager le joueur à se déplacer dans la quatrième dimension.

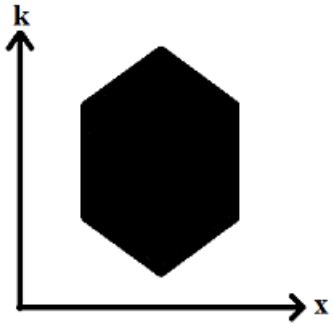

FIGURE 2 – Plate-forme ponctuelle sur (x,k)

3 État Actuel : Partie Algorithme

3.1 Principe de génération aléatoire

La création d'un jeu de plates-formes entièrement généré aléatoirement pose directement une question : *qu'entendons nous par « entièrement » ?* La première vision qui nous vient à l'esprit est sans doute celle des applications de téléphone à durée de vie illimitée, où la même succession d'obstacles se répète infiniment avec une fréquence d'apparition de plus en plus élevée.

Pourtant, on ne retrouve pas ici un défi particulièrement intéressant à relever. En effet, de telles applications sont basées sur la répétition d'un même schéma. Nous ne portons aucun jugement de valeur et comprenons que ce type de divertissement ait ses intérêts. Il nous semble cependant important de préciser que cela n'est pas l'objectif visé par ce projet.

Cette première idée écartée, notre esprit s'égare et nous nous mettons à rêver d'un programme à qui on demanderait en entrée un certain niveau de difficulté, et qui nous ressortirait un jeu nouveau, avec des personnages intéressants, une histoire ouverte, un visuel unique et créé rien que pour nous. Mais même le plus optimiste des idéalistes se rend compte de l'envergure de cette tâche. Le premier problème qui saute aux yeux est d'ordre très pragmatique. Qu'en est-il des graphismes ? Comment peut-on créer une plate-forme à partir de rien pour ensuite lui assigner une texture suffisamment explicite quant à son fonctionnement pour le joueur ? Et puis d'abord qu'est-ce qu'une plate-forme ?

Cette question d'apparence simple nous laisse bien embêté. Il est possible de lister des propriétés qu'on pourrait lui associer comme sa position, la force de friction qu'elle exerce, sa zone d'influence, son déplacement etc... Et définir algorithmiquement toutes ses propriétés serait possible mais bien fastidieux. De plus cela augmenterait considérablement la complexité des calculs à venir.

De là, le plus simple est alors de considérer chaque plate-forme comme une boîte noire, prenant un personnage à un point d'entrée et l'emmenant à un point de sortie. Par la suite, on ajoutera un changement d'état à notre personnage, mais pour l'instant cela importe peu.

Nous voilà donc avec une définition de nos plates-formes, mais toujours aucune idée de ce qu'on veut réellement dire par génération aléatoire. Pourtant, nous savons que nous voulons créer un jeu où le but premier est d'aller d'un point A à un point B, en utilisant des objets qui possèdent exactement la même définition. À partir de là, une idée de possible récursion se forme clairement. On peut donc voir dès à présent comment obtenir cette génération : en utilisant des objets élémentaires, on peut créer une entité plus complexe qu'on utilise par la suite comme objet élémentaire pour en créer une autre etc...

Dans notre cas, nous travaillerons sur deux échelles que nous nommerons Niveaux et Sous-Niveaux, le premier utilisant le second qui lui même se sert de nos briques élémentaires. On décrit ces deux

algorithmes de façon plus détaillée.

3.2 Partie Niveau

Le module Niveau a pour objectif de créer la silhouette du niveau. Cette silhouette est ensuite remplie de plates-formes par le module Sous-Niveau. L'algorithme choisi doit répondre à certains critères :

— le niveau ne doit pas être trop ramassé, il doit prendre de la place ;
— il ne doit pas entrer en collision avec lui-même ;
— le niveau généré ne doit pas toujours être le même.

L'algorithme choisi est un algorithme itératif. À chaque itération, l'espace pour un sous niveau est alloué. Pour ce faire, les paramètres du sous-niveau sont choisis, à savoir une direction et les dimensions dans l'espace d'un pavé dans lequel s'inscrira le sous-niveau. Le sous-niveau est ensuite créé dans une sandbox par le module Sous-Niveau. Enfin, ce sous-niveau est déplacé afin de se placer dans l'espace lui ayant été alloué.

Afin de choisir ces paramètres, un jet aléatoire pondéré est fait. Cette pondération dépend de la distance disponible avant la première collision, afin d'éviter au niveau d'être trop ramassé.

En utilisant cette méthode, les collisions peuvent survenir mais sont très rares. En effet, il se peut que toutes les directions possibles engendrent une collision. Dans une telle situation, il suffit de reculer de quelques itérations et de reprendre l'algorithme.

Enfin, le choix d'un modèle probabiliste nous permet de générer des silhouettes différentes les unes des autres.

La communication avec le module sous-niveau et la gestion du niveau en cours a grandement été facilité par les outils proposés par le module interface Algorithme/Moteur.

3.3 Partie Sous-Niveau

Le programme Sous-Niveau prend en entrée deux points, un espace de fonctionnement ainsi qu'une bibliothèque de types de plates-formes et renvoie une liste de plates-formes dont les positions forment un chemin pour le joueur entre nos deux points.

Cet algorithme se sert des briques élémentaires, les types de plates-formes, et la forme du chemin qu'il dessine est donc apparente aux yeux du joueurs. Il faut donc se concentrer sur ce qui est attendu comme expérience pour ce type de jeu.

On choisit de créer une courbe répondant à certaines conditions pour ensuite y disposer nos plates-formes élémentaires.

3.3.1 Le choix de la courbe

Tout d'abord, on remarque que les demi-tours brusques sont assez rébarbatifs et qu'il est préférable de minimiser leur nombre au maximum. Ainsi, on choisit de n'en effectuer aucun avec notre courbe. Au final, les seuls présents seront ceux demandés explicitement dans une plate-forme élémentaire ou par la fonction Niveau. On veut donc pouvoir prendre une courbe douce allant de l'entrée à la sortie. On appelle cette courbe la courbe directrice.

Il est bien évident qu'un tel choix n'apporte qu'un divertissement des plus ténus à un potentiel utilisateur. Il est donc nécessaire d'y ajouter des perturbations sous la forme d'une courbe de bruitage. La somme des deux nous donne ce que nous appelons la courbe royale.

On commence par s'interroger sur le choix de fonctions pour la courbe directrice. Notre idée est d'utiliser une courbe de Bézier. Pour rappel, une courbe de Bézier de degré n prend en paramètre $n+1$ points $(P_i)_{0\leq i\leq n}$ et renvoie l'ensemble des points définis par la combinaison affine pour $t \in [0,1]$:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} P_i$$

Cette courbe, en plus d'avoir le bon goût de partir de P_0 pour s'arrêter sur $P_n + 1$, est définissable dans plusieurs dimensions, est dérivable et est suffisamment lisse pour jouer le rôle de guide. Malheureusement, si les deux points sont trop éloignés l'un de l'autre, la courbe est alors très plate. Cependant, pour l'utilisation prévue par Niveau, elle nous convient parfaitement.

Ainsi, chaque point de notre courbe est défini par un réel t entre 0 et 1. Par simplicité, on voudrait qu'il en soit de même pour notre bruitage. On choisira donc une courbe sinusoïdale avec une forte pulsation. De plus, on sélectionne des fréquences différentes sur chacune de nos quatre coordonnées pour créer une impression chaotique.

De ce fait, après avoir placé n points aléatoirement, on peut construire une fonction déterministe qui à un réel t entre 0 et 1 associe un point sur la courbe royale.

FIGURE 3 – Aperçu de la courbe d'un sous-niveau

3.3.2 Implémentation

On utilise C++ pour implémenter la fonction Sous Niveau. De plus, pour la géométrie de l'espace et les calculs polynomiaux, on utilise nos propre classes.

Le fonctionnement de l'algorithme est le suivant : on part d'une position Pos_i et on souhaite se rendre à la position Pos_{i+1}. On connaît également $\tilde{t}_i$ tel que $B(\tilde{t}_i)$ approxime notre position sur la courbe directrice. En effet, il se peut qu'on s'en retrouve décalé, ne serait-ce qu'à cause de la courbe de bruitage.

De là, on va chercher sur la courbe un point assez proche pour être atteignable, mais suffisamment éloigné pour laisser une part de divertissement. On note qu'on est obligé d'utiliser une dichotomie entre $\tilde{t}_i$ et 1 pour trouver un t^* acceptable. On obtient alors le point $CourbeRoyale(t^*)$ qui, dans l'idéal, correspond à notre point Pos_{i+1}. On calcule donc l'équation du saut en partant de Pos_i pour

aller à $CourbeRoyale(t^*)$. Si le point est atteignable, on accepte, sinon on prend le point le plus proche correspondant. Il nous faut donc au besoin trouver un $\tilde{t}^*$ qui approxime notre position sur notre courbe directrice. Pour cela on utilise l'algorithme de Durand-Kerner.

Enfin, il nous reste à prendre une décision : *quel type de plate-forme élémentaire choisir ?* Pour cela, on se laisse deux choix, soit prendre un type ponctuel, soit prendre un type long. Le premier ayant son entrée et sa sortie au même endroit tandis qu'elles sont distinctes dans le second.

Si on la veut ponctuelle, on en choisit une au hasard et on recommence. Dans le cas d'une plate-forme longue, il nous faut en sélectionner une qui nous convient. Pour ce faire, on décide de s'intéresser à une plate-forme formant un angle semblable à la dérivé de notre courbe et de taille raisonnable. On en choisit une alors qui convient et on utilise de nouveau Durand-Kerner pour obtenir Pos_{i+2} et $\tilde{t}_{i+2}$ et recommencer.

Ainsi, peu à peu on se rapproche de notre sortie, et lorsqu'on est assez proche, on renvoie le chemin ainsi créé au module Niveau.

4 État Actuel : Transmissions Algorithme-Moteur

4.1 Bibliothèque pour l'algorithme

Le fonctionnement de l'algorithme de sous-niveau repose sur la sélection de plates-formes possédant des caractéristique précises. Afin de rendre cette sélection souple et simple à utiliser, nous avons créé une classe gérant le parsing des fichiers de description de plates-formes ainsi que la sélection d'ensembles de plates-formes en utilisant un prédicat booléen sur les plates-formes pouvant être écrit par l'utilisateur.

L'implémentation a volontairement été faite sous forme de liste, car la mise en place d'une structure de données utilisant de l'indexation sur les caractéristiques des plates-formes aurait était trop coûteuse par rapport à l'ampleur finale de la base de données (une dizaine de plates-formes).

4.2 Description des plates-formes

La bibliothèque de plates-formes lit les informations nécessaires aux sélections à partir de fichiers au format suivant

ID = "Identifiant de la plateforme"
formes = "Liste d'entiers correspondant aux formes que peut prendre la plateforme"
additiveAcceleration = "Vecteur4 représentant l'accélération additive de la plateforme"
multiplicativeAcceleration = "Vecteur4 représentant
l'accélération multiplicative de la plateforme"
distSorties = "Liste de Vecteur3 décrivant les différentes direction selon lesquelles la plateforme peut être traversée"
size = "Vecteur3 décrivant un pavé englobant la plateforme"
apparitionWeight= "Entier définissant la probabilité d'apparition de la plateforme"
angleSorties = "Liste d'entiers définissant la variation de hauteur des traversées de la plateforme"
reshapeAble = "Booléen à vrai si la plateforme est déformable"

FIGURE 4 – Format d'un fichier de description de plate-forme

4.3 Description des niveaux

Les niveaux doivent être décris selon un format contenant de nombreuses informations sous une forme peu instinctive. Il est donc nécessaire de fournir une interface facile d'utilisation permettant à la partie niveau de générer le fichier de description à partir d'une représentation abstraite des plates-formes du niveau. Cette interface doit aussi permettre de transformer le niveau (rotations, déplacements, redimensionnement) au cours de sa construction.

Ces objectifs sont atteints par l'utilisation de classes permettant de représenter les plates-formes et les niveaux et de les manipuler. Il y a cependant des problèmes sur la composition de rotations qui impose des conversions complexes et dépendantes de l'ordre dans lequel les angles sont considérés dans le modèle Roll / Pitch / Yawn. Ce problème aurait pu être anticipé de façon à choisir un modèle des rotations plus facilement composable.

5 État Actuel : Partie Moteur de Jeu

5.1 Unreal engine

Le niveau est lu et chargé par un programme réalisé à l'aide de l'Unreal Engine 4.15 (abrégé U.E.). En effet, la gratuité de l'Unreal Engine et sa simplicité d'utilisation (certains membres de l'équipe ayant déjà touché à l'IDE auparavant) en ont fait le candidat idéal pour GAP.

La plupart des objets définis dans les parties ci-après ont été implémentés via des *blueprints*, représentation graphique intuitive des classes héritant du code de l'Unreal Engine. La facilité d'utilisation des blueprints, autant du point de vue de l'implémentation (pas d'erreurs de syntaxe, documentation présente en infobulles) que du débogage (suivi graphique du chemin d'exécution).

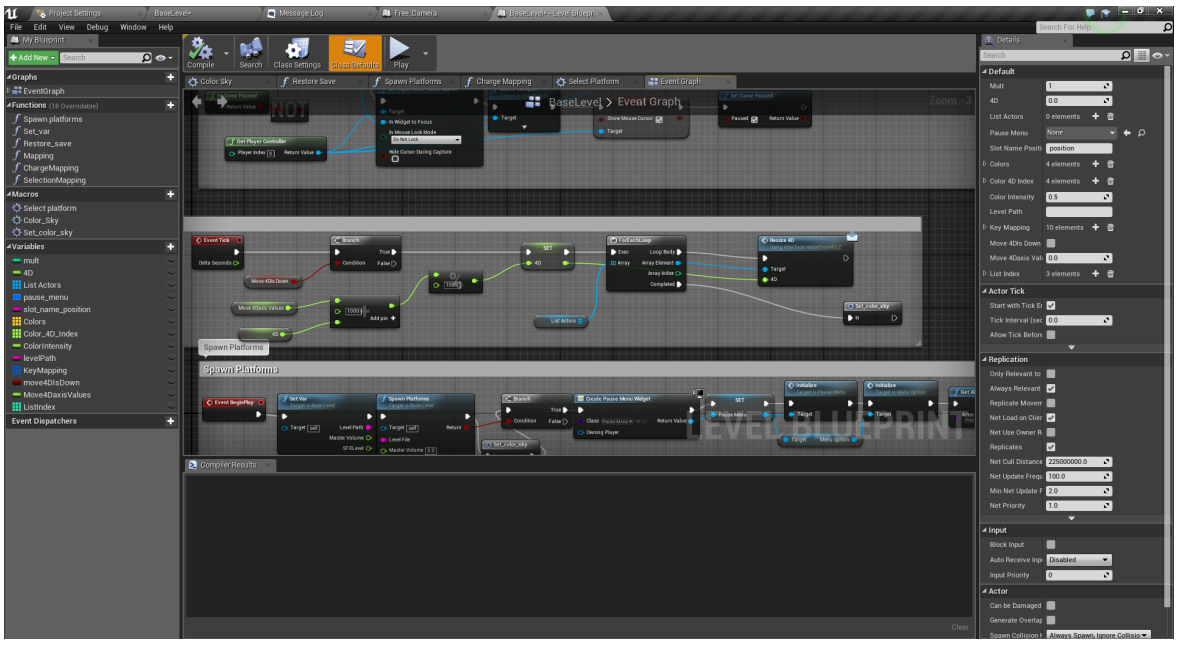

FIGURE 5 – L'interface de modification des blueprints, ici sur le niveau `BaseLevel`

Bien que l'Unreal Engine soit un outil puissant, certaines portions de code ont dû être écrites et compilées comme blueprints, notamment le parsing du fichier de niveau ainsi que certains utilitaires

de gestion de position des plates-formes. Le support de la compilation C++ vers blueprint est encore partiel, c'est pourquoi une version linux, web ou android n'est pas envisageable pour le moment.

5.2 Structure Générale

Seulement deux niveaux sont actuellement implémentés : `BaseLevel` et `MainMenu`, responsables respectivement du chargement du niveau généré par l'algorithme décrit précédemment, et le menu principal. Un tutoriel devra être implémenté prochainement sous la forme d'un troisième niveau codé sans génération aléatoire introduisant le joueur à une trame narrative simple.

Certaines classes on dues être réimplémentées, notamment le `PlayerCharacter`, à la fois pour inclure les différents contrôles et les diverses animations qui se sont ajoutées tout au long du développement.

Les menus, c'est-à-dire l'emplacement, le visuel et l'interactivité des boutons sont codés de manière modulaire, permettant l'ajout simple d'une autre section via le simple ajout d'un pointeur vers la structure fille dans la structure parent.

5.3 Gestion de la 4D

La 4D est implémentée comme une variable propre au niveau : les futurs projets d'implémentation d'un mode multijoueur devront donc conserver une 4D propre à chaque joueur au sein du niveau.

Toutes les classes nécessitant la valeur de `4D` utilisent un *Event Dispatcher* afin d'appeler une fonction du `BaseLevel` correspondant l'effet voulu.

Les plates-formes utilisent une interface, `ResizeFrom4D()`, leur permettant d'être modifiées par une commande de l'utilisateur sans qu'il y ait besoin de pointeur reliant les deux structures.

5.4 Gestion des changements de gravité

Les changements de gravité sont implémentés dans la classe C++ `GravityPlayerCharacter`, déjà implémentée par la communauté [1]. La gravité peut alors être modifiée, comme parallèle à un axe ou provenant d'un point. Ces caractéristiques sont spécifiques aux plates-formes de changement de sous-niveau, et leur utilisation dépend de l'effet voulu dans la plate-forme de changement.

5.5 Gestion des sauvegardes

Les sauvegardes sont gérées par des `SaveObjects`, répartis en deux classes : `save_options` qui prend en charge le son, le nom du dernier niveau chargé et les option graphiques ; et `save_position`, qui prend en charge la position du personnage dans les quatre dimensions.

5.6 Gestion des réglages

Les réglages se font via le menu option qui comporte trois sous-menus :

— Vidéo : Permet de passer en mode plein écran et de sélectionner sa résolution.

— Son : Permet de régler le volume des sons.(voir section 5.7).

1. `https://forums.unrealengine.com/showthread.php?57376-Dynamic-gravity-for-characters`

— Contrôles : Permet de régler les touches clavier pour les commandes suivantes : avancer/reculer, aller à droite/gauche, bouger dans la 4ème dimension, mettre le jeu en pause.

Un fichier de sauvegarde est enregistré localement, de sorte que les paramètres sont rechargés à leurs dernières valeurs au redémarrage de l'application.

5.7 Gestion des sons

Les sons utilisés dans le projet ont été récupérés sur `http://www.universal-soundbank.com/` et modifiés à l'aide du logiciel Audacity. Les sons utilisés sont des bruits de pas et de saut qui ont été accélérés et filtrés avec un filtre passe bas.

Le jeu contient deux types de sources de son :

— Une source unique au niveau des pieds du joueur qui génère le son des pas et des sauts.

— Les autres sources sont placées sur les plates-formes de type "crusher" pour générer le bruit de la partie mécanique mouvante.

Une troisième source devra être ajoutée pour les sons d'ambiance au niveau de la tête (principalement pour avoir une gestion séparée des volumes.

Le volume des sources sonores est géré via le menu son qui permet de régler le volume des effets en jeu et le volume de la musique séparément. Une gestion du volume générale est aussi proposée.

5.8 Implémentation des plates-formes

Les plates-formes sont toutes des classes héritant de la classe abstraite `Platform`, possédant les attributs de base : une sphère lumineuse annonçant sa position, une source sonore, et une fonction générique permettant la modification de sa taille selon une interpolation linéaire entre des valeurs fournies.

5.9 Personnage

Le personnage principal est le blueprint comportant le plus de fonctions. En effet, la classe `PlayerCharacter` comporte les animations de caméra, dont un changement de caméra lors des morts ou de la fin du niveau afin d'avoir un aperçu du personnage immobile, tout en tournant autour de lui. Cette structure permettra également une implémentation simple de choix entre première et troisième personne comme mode de jeu (une seule variable devra être modifiée). Les contrôles permettant à la caméra de se détacher de la direction vers laquelle le personnage se dirige, ainsi que, plus simplement, la navigation dans la 4D et les double sauts y sont également implémentés.

5.10 Modèles 3D et animations

Les différents modèles 3D ont été réalisé sur Maya Autodesk, un logiciel de modélisation qui permet la création de modèles polygonaux, l'application de textures, la création de squelettes 3D et la création d'animations.

Les diverses plates-formes du jeu ont créées selon le modèle suivant :

— création du modèle polygonal de la plate-forme, via des formes simples (pavés, cylindres, pyramides pour les piques, ...)

— UV mapping du modèle : étant donné une texture, l'UV mapping consiste à définir pour chaque face (polygone) du modèle quelle partie de la texture sera affichée sur la face. Ce travail doit prendre en compte le fait que des faces adjacentes doivent "coller" d'un point de vue texture. De plus la texture ne doit pas être trop déformée/étirée sur une face

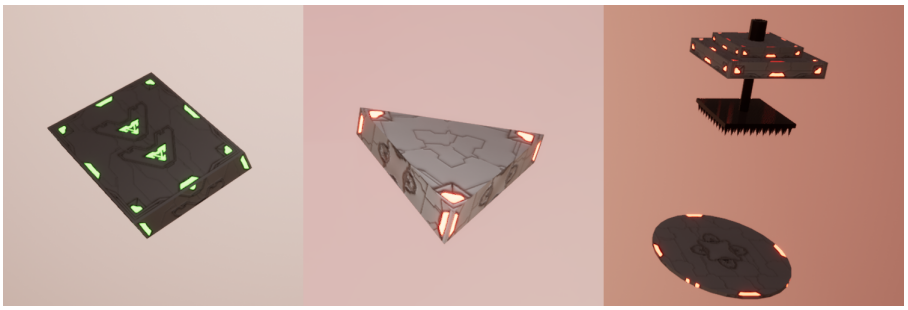

FIGURE 6 – Différents types de plates-formes

Les différentes textures ont été conçues à l'aide d'une tablette graphique sur Corel Painter, un logiciel de création d'images. Concrètement en jeu, les textures sont rendues réalistes par l'application d'un Normal sur la texture, i.e. une image qui définit la façon dont les différentes parties de la texture gèrent une lumière incidente pour donner l'impression de relief. Le logiciel utilisé pour concevoir les Normals est CrazyBump. Une autre image est utilisée pour l'émission de lumière par la plate-forme.

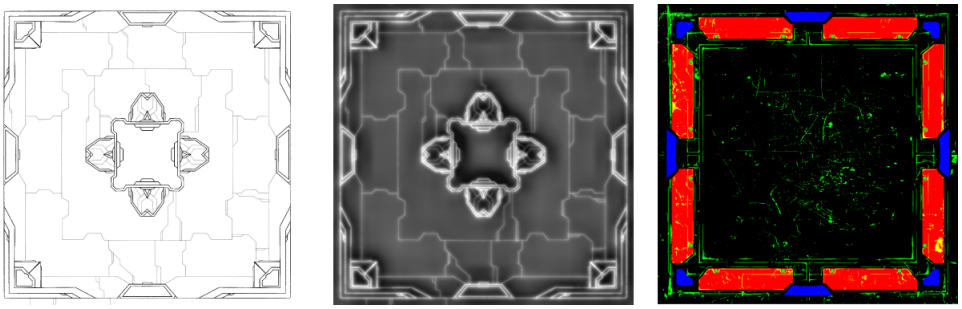

FIGURE 7 – Texture d'une plate-forme, Normal et image d'émission

La création du personnage a demandé beaucoup plus de travail de modélisation que les plates-formes, voici les différentes étapes de modélisation et d'animation :

— création du modèle : le personnage est plus détaillé qu'une plate-forme, avec beaucoup plus de polygones et de sommets. Il a été réalisé en modifiant des formes simples de base (cylindre pour les membres, pour le torse, sphère pour la tête, ...) en déplaçant les sommets pour obtenir une forme humaine. Après quoi le modèle est lissé, des points sont ajoutés pour donner une impression moins "carrée" au personnage. Une armure a été ajoutée pour apporter plus de détails au personnage

— l'UV mapping a été fait, même si la texture finale est uniforme. L'aspect métallique de la texture de l'armure est obtenue au sein d'U.E. en gérant les différents paramètres de création d'une texture

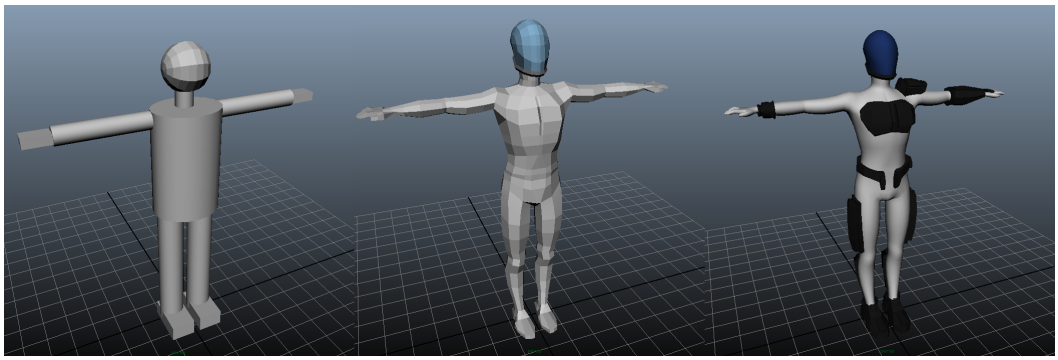

FIGURE 8 – Modélisation du personnage

— L'animation nécessite une structure à laquelle attacher le personnage : c'est le squelette. Il est constitué de "joints", qui suivent les articulations des différents membres. Les joints sont liés et disposent d'une structure d'hérédité, si bien qu'en appliquant une rotation sur le joint de l'épaule, tous les autres joints du bras suivent

— Le weight mapping est l'étape la plus importante de l'animation : pour chaque sommet on décide la distribution du poids des joints. Concrètement, on décide à quels joints un sommet est lié et en quelles proportions. Par exemple le joint de l'épaule a un poids de 1 pour les sommets situés sur le haut de l'épaule, mais a aussi un poids partiel, de l'ordre de 0.4, sur les sommets sous le bras au niveau des côtes, qui se déplacent quand on lève le bras. Le modèle du personnage compte plusieurs milliers de sommets

— Pour réaliser une animation sur 60 frames, on doit fixer une rotation pour les différents joints sur certaines frames. Si les positions/orientations des joints sont fixées aux frames 20 et 40, un algorithme détermine une approximation de la position des joints pour les frames intermédiaires. En fixant une dizaine de frames on peut ainsi obtenir une animation complète sur 60 frames.

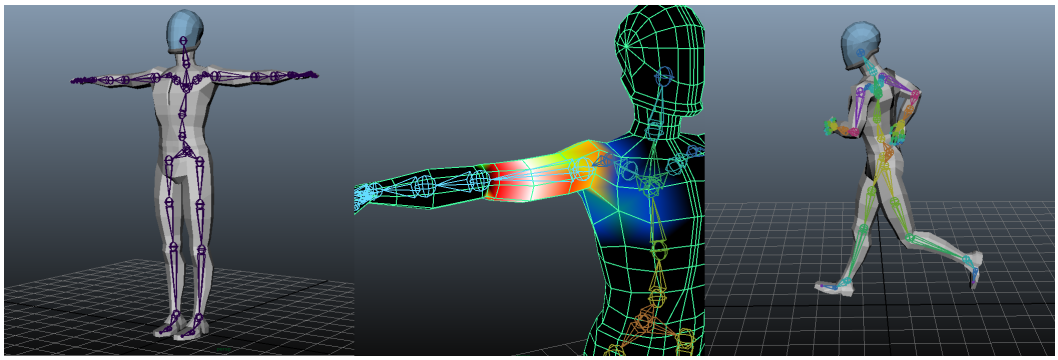

FIGURE 9 – Etapes de l'animation : squelette, weight mapping, animation des frames

FIGURE 10 – Rendu final en jeu

6 Feuille de route

La quasi-totalité de l'équipe souhaite maintenir ce projet à jour. Outre certaines améliorations déjà évoquées du moteur de jeu (Passage de la première à la troisième personne, ajout de nouvelles plates-formes, de nouvelles animations), deux axes sont principalement à développer :

— Un mode multijoueur, autant du point de vue théorique (Qu'est-ce qu'un niveau amusant à deux ?) que pratique.

— La possibilité de générer des niveaux possédant plusieurs chemins, et donc plusieurs sorties ; ainsi que la présence d'autres objectifs qu' "aller du point A au point B". Pour cela, un système de score devra être choisi et implémenté.

A noter également que le site sera amené à migrer, probablement de `perso.ens-lyon.fr/nicolas.derumigny` à `peros.ens-lyon.fr/nicolas.derumigny/GAP`, voire une autre adresse si l'achat d'un nom de domaine se justifie.