EPITA

Projet de Sup

25.05.2018

Rapport de la troisième présentation

THE FRAME WORKSHOP

Programme d'Animation 3D

Halwafel

Brian Bang, Etienne Benrey, Léane Duchet, Julie Hazan

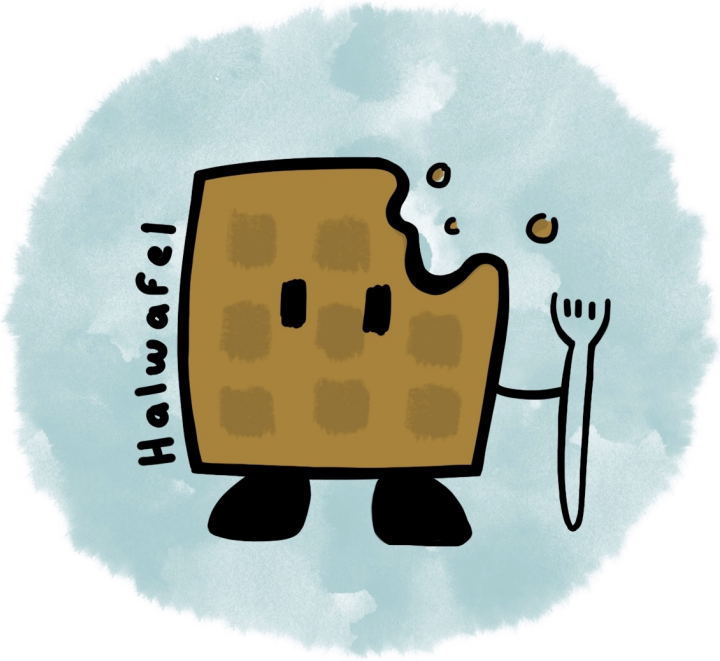


Table des matières

Introduction

Halwafel est un groupe composé de quatre membres, Brian Bang, Etienne Benrey, Léane Duchet et Julie Hazan. Il a été fondé le 16 Octobre 2017. Le nom du groupe vient d'une contraction stylisée de deux mots half et waffle.

Ce rapport présentera en détail notre logiciel The Frame Workshop, de son modèle initial aux détails de sa construction, en passant par les difficultés et les succès que nous avons rencontrés. Nous allons tout d'abord présenter le groupe Halwafel, puis expliquer le concept de notre projet, et après avoir détaillé ce que nous avons accompli nous allons approfondir les détails des fonctionnalités et leurs implémentations.

1 Présentation du groupe et du projet

1.1 Le groupe

Halwafel est, comme mentionné auparavant, un groupe constitué de quatre membres. Nous avons tous des parcours différents, avec des expériences différentes ce qui a permis à chacun d'apporter sa pierre à l'édifice.

Brian Bang

Brian était un étudiant du Lycée Albert-de-Mun. Il avait déjà une bonne base dans l'écriture de code, ayant déjà fait un projet de programmation pendant son année de terminale en classe ISN, qui était un Mastermind en Python. Bien qu'il n'ait pas d'expérience dans le montage ou l'utilisation de programmes comme celui que nous avons implémenté, ses compétences en création et design de site lui ont donné une expérience que personne d'autre dans le groupe n'avait acquise.

Etienne Benrey

Etienne était étudiant au Lycée International de Saint-Germain-en-Laye. Avant son entrée à EPITA, il avait déjà codé en C++ en ISN pendant son année de terminale. Le projet sur lequel il a travaillé cette année-là était un jeu de carte appelé Président. Il a aussi suivi un cours optionnel, l'AP informatique, lui donnant des bases en code en plus de son projet d'ISN. Son expérience en montage lui a permis d'avoir une bonne vision de notre projet même s'il n'avait aucune expérience en animation.

Léane Duchet (Chef de projet)

Léane vient du Lycée Sainte Thérèse. Avec une expérience limitée mais non négligeable de programmation en Python grâce au stage GirlsCanCode de l'association Prologin et une bonne compréhension des logiciels d'animation simple, elle a été le membre garde-fou de notre groupe. Sa rigueur a été un avantage pendant le projet.

Julie Hazan

Julie était au Lycée International de Saint-Germain-en-Laye avec Etienne. Sa première expérience de code était pour son TPE en première qui consistait au développement d'algorithmes d'encryptage et décryptage d'information en Visual Basic (Caesar, Vigenere, RSA). Elle a aussi utilisé Blender, ce qui a apporté une touche artistique au groupe.

1.2 Le projet

La première idée que nous avons eue était un jeu de RTS (pour Real Time Strategy). Nous avons par la suite décidé que nous ne voulions pas faire de jeu notamment puisque certains d'entre nous ne jouent pas régulièrement aux jeux vidéo impliquant un manque de motivation pour ce type de projet.

La seconde idée était de faire un réseau neuronal. Le problème le plus évident qu'un tel projet puisse résoudre serait un labyrinthe. Nous avons abandonné cette idée après avoir appris qu'un tel projet serait entrepris en S3 et que cela ne serait pas intéressant de le faire en doublon.

La troisième idée consistait à avoir un système GPS de recherche de chemin utilisant la méthode A*. Cette idée est venue suite au visionnage d'une série de documentaires vidéo expliquant diverses méthodes de recherche de chemin. Il était également intéressant de constater que cela rejoint l'idée de résolution de labyrinthe, nous ramenant ainsi à l'idée précédente. Cette idée a été abandonnée car en utilisant des méthodes préexistantes le projet aurait pu être terminé facilement.

L'idée suivante était la génération spontanée de cartes en utilisant les fractales, incluant la génération de montagnes et de forêts. Le principal inconvénient de cette idée est la puissance de calcul requise pour générer ces cartes et nous craignions que nos ordinateurs ne soient pas assez puissants.

Ensuite, nous avons pensé à faire quelque chose concernant la domotique. L'idée était de traiter des informations à partir de capteurs placés dans une mai-

son qui pourraient permettre d'identifier certains problèmes (comme des fuites) et de les afficher sur une carte de la maison dans le but de pouvoir agir au plus vite. L'inconvénient est que si nous devions en faire une démonstration en situation réelle nous aurions dû acheter des capteurs, ce qui est interdit.

Nous avons eu entre temps une idée spontanée de jeu similaire au casse-tête Tetris. Ce jeu aurait impliqué la création d'un jeu Tetris en 3D dans lequel les joueurs pourraient affecter le plateau des joueurs adjacents. Nous nous sommes aperçus que l'on revenait à notre première idée de jeu. De plus, cette idée était trop simpliste et aurait eu plus d'obstacles créatifs que d'obstacles de programmation.

Enfin, nous en sommes arrivés à notre idée finale, inspirée d'un des projets du père d'un des membres du groupe qui nécessitait la création d'animation 2D ou 3D. De là nous avons eu l'idée de développer un programme de montage d'animation. Cela semblait être une bonne option pour notre projet, puisque, si cela pouvait être utile à une entreprise, cela pouvait être utile dans un contexte plus général. Par exemple, notre programme pourrait être utilisé pour des bandes-annonces, des vidéos commerciales, des films d'animation, des publicités ou encore des cinématiques de jeu vidéo. Avec la possibilité d'utiliser Unity, l'aspect 3D de ce projet semblait faisable.

The Frame Workshop est un logiciel d'animation qui permet à l'utilisateur faire ses propres animations, il peut importer ses propres objets depuis Blender ou d'autres programmes de modélisations similaires, ou bien utiliser l'Unity Asset Store. L'utilisateur peut créer un squelette pour différents meshs. De plus, de nombreux outils lui sont fournis dans une interface graphique d'utilisateur (GUI) afin de manipuler tous les objets le plus efficacement possible : les timelines pour les objets, les caméras, les effets visuels ainsi que la synchronisation du son.

Cf. Annexe 1

2 Suivi du cahier des charges

Les pourcentages entre parenthèses sont ceux qui devaient être atteints pour la date indiquée, les autres sont ceux qui ont été atteints.

Parties du projet	**15 Mars 2018 (S1)**	**30 Avril 2018 (S2)**	**11 Juin 2018 (S3)**
Import du son et des objets	100% (100%)	100% (100%)	100% (100%)
Création du squelette	50% (100%)	50% (100%)	50% (100%)
Interface utilisateur (GUI)	30% (30%)	85% (80%)	100% (100%)
Création du mouvement	0% (0%)	55% (55%)	100% (100%)
Images d'arrière-plan et sons	0% (0%)	0% (0%)	100% (100%)
Animation de particules	0% (0%)	0% (0%)	100% (100%)
Site internet	50% (40%)	90% (70%)	100% (100%)

La description détaillée des différentes fonctionnalités sera faite dans la troisième partie.

Parties du projet	**15 Mars 2018 (S1)**	**30 Avril 2018 (S2)**	**11 Juin 2018 (S3)**
Import du son et des objets	Julie Brian		
Création manuelle du squelette	Julie		
Création automatique du squelette	Léane Etienne		
Interface utilisateur (GUI) **Mouvement dans l'espace et premier jet de l'interface**	Julie		
Interface utilisateur (GUI) **Timelines, bouton option, propriétés et inventaire**		Etienne	
Interface utilisateur (GUI) **Boutons de la Toolbar**		Léane	
Création du mouvement **Skinning manuel et mouvement**		Brian	
Création du mouvement **Skinning automatique et mouvement**		Julie	
Images d'arrière-plan et sons			Etienne Léane
Animation de particules			Léane
Site internet	Brian		

Parties du projet	Faite	Modifiée	Abandonnée
Import du son et des objets	✓		
Création manuelle du squelette	✓	✓	
Création automatique du squelette			✓
Interface utilisateur (GUI)	✓	✓	
Création du mouvement	✓		
Images d'arrière-plan et sons	✓		
Animation de particules	✓		
Site internet	✓		

Toutes les explications détaillées sur les parties modifiées et abandonnées seront faites dans la troisième partie : description des tâches. Il y a aussi des parties ajoutées qui seront détaillées dans cette troisième partie

3 Description des tâches

3.1 Interface utilisateur (GUI)

Nous nous sommes tout d'abord occupés de l'aspect général de l'interface, en organisant les différentes fenêtres et en décidant de l'emplacement des différents boutons et autres éléments de l'interface. Ensuite, une autre partie importante du travail était de faciliter l'interaction de l'utilisateur avec la scène et ses objets. Ceci comprend la modification de la vue dans la fenêtre 3D, le mouvement des objets et l'interaction avec ces objets.

Cf. Annexe 2

3.1.1 Création de l'interface grâce à l'utilisation des UI elements

L'aspect de l'interface a été fait en utilisant les UI panels de Unity, dont la position dépend de la caméra. Pour chaque mouvement de caméra, l'interface bouge en conséquence. Ces panneaux ont aussi une propriété utile qui est l'adaptation à la taille de l'écran. La taille des panneaux sera augmentée si le programme est ouvert dans un écran plus grand ou, au contraire, sera diminuée si la fenêtre est réduite.

Notre interface est formée de quatre panneaux : une barre d'outils, deux panneaux alignés sur la droite, un pour l'inventaire, l'autre pour les propriétés des objets et un dernier panneau pour les timelines. Le reste de l'écran est occupé par la vue 3D. L'interface est accessible en tant que panneaux indépendants qui peuvent être déplacés et dont la taille peut être modifiée.

L'interface avait besoin d'éléments permettant l'interaction avec le programme, par conséquent de nombreux objets UI ont été implémentés : des boutons, des boîtes de texte, ou encore des curseurs. Ces éléments n'ont pas les mêmes propriétés de distorsion que les panneaux, ils sont de taille fixe. Ceci signifie que leur taille ne s'adapte pas à la taille de l'écran. Les boutons de la barre d'outils sont différents comme ils ne sont pas fixés verticalement ; leur hauteur change en fonction de celle du panneau auxquels ils sont ancrés. Pour ce faire, l'outil ancre de Unity a été très utile : il détermine le niveau de distorsion d'un objet en fonction d'un axe et du parent de l'objet.

La barre d'outils est composée de cinq boutons : File, Edit, Insert, View et Option.

3.1.1.1 Barre d'outils (Toolbar)

Pour chacun des boutons suivants, un panneau apparaît lorsque l'utilisateur clique dessus. Sur ces panneaux se trouvent différents boutons dont les fonctions vont être expliquées.

Insert

Ce bouton donne accès à cinq autres : l'import, les fonctions de chargements des objets et du son, la suppression d'objet et le bouton pour les particules.

Les fonctionnalités et l'implémentation de l'import seront expliquées dans la partie "Import".

Le bouton "Load Object" permet à l'utilisateur de charger un objet importé sur la scène en donnant le nom de l'objet (sans son extension) que l'utilisateur

veut ajouter à la scène dans le champs de saisie. Deux apparaissent, l'un pour les objets qui bougent durant l'animation, l'autre pour les objets qui restent en place quand la scène est jouée.

La fonction Resouce.Load de Unity a été utilisée pour charger ces objets. Pour rendre ces objets visibles, ils ont dus être copiés dans un GameObject. Ce gameobject va être stocké dans une liste et pour le premier type d'objet, une timeline va lui être associée.

Le bouton Remove permet à l'utilisateur de détruire un objet de la scène. L'utilisateur donne son nom, comme lorsqu'il a chargé l'objet sur la scène. L'objet et toutes ses informations (squelette et timeline) vont être supprimés avec. Il restera disponible dans le fichier si l'utilisateur veut à nouveau le charger sur la scène.

Le bouton Load Sound est utilisé pour charger un son sur la scène. Son fonctionnement est similaire au bouton qui charge un objet, à la différence que le son n'est joué que quand la scène est jouée au moment choisi par l'utilisateur dans la timeline du son.

Le bouton Particles donne accès à cinq types de particules. L'utilisateur peut donc ajouter à sa scène des feux d'artifices, de la fumée, de la vapeur, une tempête de poussière et un signal lumineux. Tant que le bouton play des timeline n'est pas pressé, les particules restent sur la scène, une fois play cliqué, les particules apparaîtront et disparaîtront au moment donné par l'utilisateur.

View

Cette section contient six différentes vues : de front, de dos, de haut, du bas, de droite et de gauche. Quand un nouveau projet est commencé, la vue de front est celle qui est utilisée. Ces vues changent la position de la caméra en fonction d'un plan invisible.

Edit

Les boutons utilisés pour la création manuelle du squelette sont placés sous l'Edit. Les fonctionnalités de ces boutons sont expliquées dans la partie 3.3.1 : Création manuelle.

File

Le bouton File contient quatre autres boutons : New project, Save project, Load project, Render.

Le bouton **New Project** charge une scène vide avec seulement l'interface utilisateur visible.

Le **Save** permet à l'utilisateur de sauvegarder le travail effectué sur la scène : les objets qui ont été importés et toutes leurs informations (comme leur squelette). Pour ce faire, l'utilisateur a juste à indiquer le nom de la sauvegarde qu'il veut réaliser.

Pour ce bouton, différentes méthodes de sérialisation ont été explorées. La première méthode était d'utiliser la fonction de sérialisation de Unity qui permettait de sauvegarder les GameObjects et les autres classes spécifiques à Unity. Après un peu d'expérimentation, il a semblé que les sauvegardes de GameObject n'étaient pas bien faites. Comme Unity utilisait un binary formatter, il était difficile de savoir d'où venaient les erreurs, nous avons donc opté pour une autre méthode. Il y avait deux choix majeurs : le Xml ou le Json. Ayant très peu de connaissances dans les deux, le premier choix a été retenu, comme il était simple à comprendre et à apprendre.

La première chose qu'il a fallu faire est un parser pour les différentes classes que nous avons créées. Tout d'abord, le squelette devait être sauvegardé. Ce squelette étant un arbre, il a fallu le sauvegarder en tant que liste, qui contenait tous les nœuds de l'arbre, chaque nœud ayant son nombre d'enfants. Nous nous sommes rendus compte que cet arbre n'était pas utile après que le squelette automatique ait été abandonné, un nouveau parser a donc dû être fait pour le squelette. Ce squelette est maintenant fait grâce à des relations parents/enfants entre des sphères qui représentent les anciens nœuds. Pour le sauvegarder, les informations des sphères sont enregistrées ainsi que tous ses enfants (un objet ne pouvant pas être sauvegardés par du Xml). Ensuite, les os liant les différentes sphères doivent être sauvegardés car ils contiennent les informations de skinning. Le parser a été fait pour les timelines qui contiennent le mouvement et les objets sur la scène. Pour sauvegarder chaque objet, une classe le contenant avec toutes ses informations dans leur forme utilisable par Unity a été faite, quand l'utilisateur veut sauvegarder, les informations contenues dans cette classe sont transformées en structures qui peuvent être sauvegardées en Xml.

Le bouton **Load** est utilisé pour charger une sauvegarde. Pour ce faire l'utilisateur doit juste écrire le nom de la sauvegarde dans l'emplacement correspondant. Le procédé pour écrire cette fonction était très similaire à celui de la

fonction save. Des types sérialisés, le Load recrée la scène d'où elle a été laissée. Tout d'abord l'objet est chargé du dossier Resource comme pour la fonction Load Object puis son squelette est créé des listes qui ont été sauvegardées et ses timelines sont importées. Ceci est fait pour chaque objet qui avait été chargé sur la scène. La caméra à travers laquelle l'utilisateur voit la scène est elle aussi sauvegardée afin que la scène puisse être recréée exactement comment elle a été laissée.

Le dernier bouton dans cette partie est le bouton **Render**, qui est utilisé pour transformer l'animation créée par l'utilisateur en vidéo. Ceci est fait par le lancement de l'animation sur une caméra secondaire et l'enregistrement en temps réel de l'animation vue par la caméra. Pour ce faire, nous avons décidé d'utiliser un asset de Unity appelé FramRecorder, qui est une base générique pour différents scripts d'enregistrement. Il accorde une grande flexibilité dans son utilisation, en offrant une solution simple pour enregistrer l'écran de l'utilisateur pendant que le programme tourne. Il est divisé en trois grandes parties : l'entrée média, l'enregistrement et les paramètres ou la structure de base. Ceci suit la structure générale qui est visible dans de nombreux programmes où les informations sont entrées puis traitées et enfin apportées à l'utilisateur sous forme d'interface utilisateur. Dans notre cas, notre paramètre d'entrée est la caméra et le traitement est la transformation en vidéo. L'interface est cachée de notre utilisateur mais représente les paramètres qui sont donnés à l'enregistreur. Chaque partie est composée d'une ou plusieurs classes, mais nous allons considérer chaque partie comme une classe regroupant les autres, bien que ce soit une simplification.

Le media input est responsable de la récupération des informations qui doivent être enregistrées à une certaine frame, et transforme cette information en data utilisable par l'enregistreur. Ceci est lancé à chaque frame, et récupère soit l'audio soit la bitmap (image) représentant tous les pixels de la scène ou les deux. Certains paramètres agissent sur les paramètres de l'enregistreur comme la résolution de l'écran. La fréquence d'images ou les autres paramètres pour la vidéo n'influent pas sur cette fonction car elle est appelée une fois par frame, au début de la frame, ignorant tout de la séquence de frame que l'on veut récupérer. Elle se focalise sur une seule frame. Les paramètres pour cette fonction restent les mêmes pendant toute la durée de l'enregistrement, ce qui signifie qu'ils vont garder les paramètres donnés au début de l'enregistrement. Une classe Editor est implémentée avec cette fonction pour pouvoir modifier ces paramètres initiaux.

La classe Recorder est la plus compliquée des trois, tout étant une partie essentielle. Son but est d'avoir une structure générique pour tous les types d'enregistreur qui prennent leur data du media input et le transforme en un format

différent. C'est dans ce but que le formater pour la data est récupéré. Pour ce faire, l'enregistreur est d'abord notifié du début de l'enregistrement. Ceci crée tous les éléments nécessaires pour que la data soit transformée. Similairement à l'input, il est appelé à chaque frame pour traiter les nouvelles entrées de cette frame. Un enregistreur peut prendre autant de paramètres qu'il y en a d'implémentés mais, à moins que ce soit implémenter, ne peut pas prendre un nombre différents d'entrées que celui qui est attendu. Comme pour l'input, puisque la classe n'est appelée qu'une fois par frame, elle n'est pas influencée par la séquence de frames qui est appelée. Elle n'est pas non plus influencée par les paramètres des inputs, puisque les deux classes sont complètement séparées. De plus, le fichier de sortie que la classe crée n'est pas sauvegardé dans la mémoire, mais seulement dans un fichier temporaire. Elle a donc besoin d'être gérée par une structure générale, qui comprend à la fois l'input et le Recorder.

La partie finale est la structure de base, qui est une généralisation du "Support" et les "Editors", qui peuvent être combinés pour faire fonctionner les deux premières classes. Elle prend le début et la fin de l'enregistrement et tous les paramètres restant, notamment ceux qui sont liés à la fréquence d'images. L'asset permet de manipuler beaucoup plus de paramètres, mais qui ne sont pas nécessaires pour notre utilisation. Lorsque l'enregistrement est activé, cette structure de base prend tous les arguments, lance l'input et le Recorder, puis s'occupe de chaque frame individuellement, appelant successivement l'input puis le Recorder. Lorsque l'enregistrement se termine, cette même structure transforme le fichier temporaire créé par le Recorder pour le sauvegarder dans l'ordinateur de l'utilisateur.

L'asset FrameRecorder est ainsi une grande aide pour l'enregistrement de l'écran, et pour la transformation de l'information en un fichier vidéo. Cependant, cet asset n'est pas un outil d'enregistrement, mais simplement une structure générale des classes nécessaires pour créer son propre asset d'enregistrement. Nous avons donc utilisé FrameRecorder pour nous aider a faire un outil d'enregistrement. Ceci a été fait avec un l'aide d'un deuxième asset : FrameCapture. Cet asset est ce qui nous a permis d'utiliser FrameRecorder pour transformer les bitmaps de l'écran de l'utilisateur en fichier mp4, sortant ainsi un fichier vidéo. Cet asset est fait pour fonctionner avec FraneRecorer, nous permettant d'implémenter facilement tous les éléments nécessaires. Nous avons décidé de modifier les paramètres pour avoir une fréquence d'image constante à 30 fps, ce qui est suffisant pour le type de rendement qui serait réalisable avec notre programme. Avec l'aide de ces deux outils, le rendement peut être fait facilement avec un enregistrement en temps réel de l'animation.

Option

Le bouton Option permet à l'utilisateur de bouger les panneaux dans l'écran, voire de les mettre de côté. Quand le bouton est cliqué, les panneaux peuvent alors être bougés. Cliquer le bouton une deuxième fois désactive la mobilité des panneaux et les verrouillent à l'emplacement actuel. Cette opération peut être répétée autant de fois que l'utilisateur le veut. Pour ce faire, le programme détecte quel panneau est visé par l'utilisateur en trouvant l'endroit où est sa souris quand il clique ; puis les coordonnées du panneau sont ramenées à celles de la souris, qui sont translatées des coordonnées par rapport à l'écran aux données en coordonnés du monde. Finalement, le panneau arrête de suivre la souris quand le bouton est relâché.

Avec cette fonctionnalité, nous avons choisi d'implémenter un système de redimensionnement des fenêtres quand elles sont dans leur position originelle. Ceci se fait en déplaçant les bords des panneaux qui sont marqués comme déplaçables par une ligne rouge. Cette fonctionnalité est désactivée quand le bouton option est cliqué mais les panneaux gardent la taille donnée par l'utilisateur, conférant à l'utilisateur le contrôle complet sur la hauteur des panneaux dans un premier temps et leur position dans un second. Ceci est vrai pour tous les panneaux à l'exception du panneau de l'inventaire qui a une taille définie comme il est possible de le faire coulisser.

3.1.1.2 Inventaire

Le panneau Inventaire est assez simple dans sa conception, puisque son but est simplement de montrer à l'utilisateur tous les objets qui sont présents sur la scène. Ceci est possible par la manipulation d'une chaîne de caractères qui est représentée dans un texte Unity UI element. Insérer un élément sur la scène ajoute son nom à la suite de la chaîne de caractères, suivi d'un retour à la ligne. Supprimer un élément de la scène retire son nom de l'inventaire et replace les autres éléments jusqu'à revenir au format d'origine, sans retour à la ligne. Dans le but d'éviter que la liste d'objets soit scindée dû à un très grand nombre d'objets dans la scène, le bas du panneau peut être étiré pour montrer une plus grande étendue de la liste.

3.1.1.3 Propriétés

Le panneau affichant les propriétés est lié au timeline. En effet, il montre les propriétés de n'importe quelle clé sur laquelle on clique pendant quelques secondes (il va donc montrer les propriétés d'une clé en déplacement). Relâcher le clic gauche vide le panneau. Le panneau donne les informations suivantes : l'objet auquel la clé est liée, le type de la clé (position, rotation ou échelle), le

type de mouvement (soit linéaire soit courbe), la courbe de vitesse, et le moment d'exécution. Ce dernier change en temps réel quand la clé est déplacée.

3.1.1.4 Timelines

Implémentation

Ce panneau de ce GUI est ce qui relie le mouvement des objets à l'interface que nous voyons. Son rôle est de permettre à l'utilisateur de librement choisir ce que le programme fait avec les objets lorsqu'ils sont en mouvement. Pour cela, un système de timeline a été implémenté, ce qui permet aux utilisateurs de placer des clés correspondant aux diverses positions, rotations ou échelles d'un objet donné. L'implémentation de ces clés sera expliquée dans une future partie. Nous avons trois timelines fondamentales, Camera, Visuelle et Audio, qui seront respectivement pour le déplacement de la caméra, les effets visuels, et enfin l'implémentation du son. Pour chaque objet ajouté sur la scène, une nouvelle timeline est automatique créée sous les précédentes timelines. Cela s'ajoute à ce qui a été dit précédemment, les mouvements de tous les objets étant placés sur la timeline visuelle. Nous avons décidé d'aller plus loin pour faciliter l'utilisation du programme. Pour s'assurer que les timelines restent faciles à utiliser, lorsque les timelines dépassent la taille du panneau, ce dernier peut défiler par un clic gauche de l'utilisateur. Cette fonctionnalité est désactivée lorsque le panneau est assez large pour que toutes les timelines soient visibles en même temps.

Lors de la création d'un nouvel objet, la nouvelle timeline doit être placée sous les autre timelines, ce qui est fait en allongeant le panneau des timelines, puis en dupliquant une timeline cachée, qui est ensuite déplacée vers la bonne position, en fonction de la taille de l'écran. Cette timeline de base a toutes les fonctionnalités pré-implémentées, sauf pour le lien vers un objet de la scène. Toutes les nouvelles timelines sont des simples copies de cette timeline de base.

Lors de la suppression d'un objet, la timeline correspondante doit également être supprimée. Ce faisant, toutes les timelines situées en dessous de celle qui est supprimée doivent être remontées pour remplir l'espace libéré. La translation des timelines est calculée en fonction de la taille de l'écran. Une fois toutes les timelines déplacées, le panneau est rétréci pour compenser le plus petit nombre de timelines. Ceci est essentiel pour éviter que le panneau reste défilable alors que ce n'est pas nécessaire. Une petite marge d'erreur est laissée pour s'assurer que l'utilisateur puisse toujours avoir accès à ses timelines même en cas de problème.

L'utilisateur a l'option d'ajouter à toute timeline les quatre types de clés par le biais de deux boutons placés dans la partie gauche de la timeline correspon-

dante. Ces clés sont placées au centre de la timeline, et peuvent être déplacées le long de cette dernière, ce qui correspond à des changements dans la valeur temporelle de la clé. Le déplacement d'une clé désactive de façon temporaire le défilement des timelines, ce qui facilite le positionnement des clés à des moments précis. Les quatre types de clés peuvent être divisés en deux catégories : les clés de translation et les clés de mouvement.

Toute clé doit être liée à un objet dans la scène, à l'exception des clés liées au son et aux particules. Une façon de relier des objets ensemble est de rendre l'un parent de l'autre. Cependant, ceci relie les coordonnées de l'un à celles de l'autre, faisant en sorte que l'enfant se déplace avec le parent. Pour éviter ceci, nous avons décidé de rendre tout objet de la scène enfant d'un objet vide situé à l'origine des coordonnées de Unity. L'ordre des enfants de cet objet vide est le même que l'ordre des timelines, reliant ainsi la timeline à son objet par leur ordre.

Chaque modèle d'une clé de translation a un objet vide qui lui est relié, qui stocke toutes les informations nécessaires pour la clé, à part la valeur temporelle de la clé, et l'objet auquel elle est liée. Son type (position, rotation ou échelle), sa linéarité et sa courbe de vitesse sont stockés dans le nom de l'objet vide, en suivant le format suivant : "R012". Le premier caractère correspond au type de la clé, avec 'L' correspondant à la position (pour Location), R étant la Rotation, et S pour l'échelle (pour Scale). Le deuxième caractère représente un booléen qui indique si le mouvement est linéaire ou courbe. S'il est courbe, le caractère sera '1', et s'il est linéaire le caractère sera '0'. Les deux derniers caractères sont les digits correspondant aux nombres de 0 à 13. Chacun correspond à une courbe de vitesse pré-enregistrée. Leur implémentation est expliquée dans la section suivante. La position, rotation ou échelle de la clé est donnée par la position, rotation ou échelle de cet objet vide.

Les clés de mouvement sont implémentées de façon similaire, puisqu'une majorité des informations nécessaires sont les mêmes. Chaque clé est reliée à un objet, qui est encore une fois donnée par l'ordre de l'objet et de la timeline. Chaque clé de mouvement a également une liste qui équivaut à la valeur de la position, rotation ou échelle pour les clés de translation, mais qui permet de manipuler plusieurs articulations à la fois. Chaque élément de la liste a une articulation qui lui est associée, la rotation que l'utilisateur veut que l'articulation ait lorsqu'on arrive à la clé en question, une courbe de vitesse et un booléen de linéarité. La valeur de la rotation est de nouveau donnée par la rotation d'un objet vide qui est enfant de la clé. La linéarité et la courbe de vitesse sont également codées dans le nom de cet objet vide, à cela s'ajoute l'information de sérialisation de l'articulation.

Enfin, il y a trois types de clé spéciales, une pour chaque timeline pré-implémentée : les clés de caméra, les clés de particules et les clés de son. Les clefs de caméra sont implémentées de façon similaire à celles des clés de translation, mais sont limitées à la position et rotation, avec ajouté à cela deux booléens qui indiquent si la caméra se focalise sur un objet ou suit cet objet. L'objet en question est donné par sa position parmi ses frères, et est stockée de la même façon que les clés de mouvement. Les clés de particules et les clés de son ont un attribut unique : la valeur temporelle.

Comme dit précédemment, chaque clé a une valeur temporelle qui est donnée par sa position sur la timeline. Le valeur minimale est toujours 0, mais la valeur varie. Cette dernière est donnée par un curseur placé au dessus des timelines. Pour récupérer cette valeur, deux objets invisibles sont placés aux extrémités de la timeline. On peut calculer la coordonnée X en tant que pourcentage de distance entre le premier et le deuxième objet invisible. En le multipliant pas le le temps maximal, nous pouvons récupérer la valeur temporelle de la clé.

Application

Avec tous les outils de stockage de données mis en place, l'utilisateur a besoin de pouvoir utiliser les timelines comme il le veut. Toutes les timelines ont la même structure du point de vue de l'utilisateur. Sur la partie gauche, une boite permet à l'utilisateur de créer des clés.

Lors de la création d'une clé de translation, sa valeur, donc sa position, rotation ou échelle, est immédiatement stockée. Si l'utilisateur veut modifier cette valeur, il devra créer une nouvelle clé. Ceci est fait pour évité d'avoir une interface graphique trop chargée. Pour créer une telle clé, l'utilisateur doit cliquer sur le bouton du haut sur la timeline. Pour choisir le type de clé, l'utilisateur utilise un pointeur qui a trois positions, une position par type. Le type est écrit sous le pointeur, et est indiqué par un code couleur. La modification des autres attributs est fait en faisant un clique-droit pour changer la linéarité, ou un Alt-clique-droit pour sélectionner les différentes courbes de vitesse. Tout changement est marqué en bas à droite de l'écran, et peut aussi être vu dans le panneau des propriétés qui, comme dit précédemment, est accessible avec un simple clique-gauche sur une clé.

Les clés de mouvement sont légèrement plus compliquées à utiliser, puisqu'elles peuvent contenir un nombre d'articulations arbitrairement grand. L'insertion de ses articulations ne se fait donc pas lors de la création de la clé, mais plus tard dans sa manipulation. Créer une clé de mouvement est fait en cliquant sur le bouton du bas sur la timeline. Pour commencer à sélectionner et modifier les articulations de la clé, l'utilisateur doit réaliser un Crtl-clique-

droit sur la clé en question pour entrer dans le mode modification. Une fois le mode modification lancé, l'objet de la scène devient invisible, laissant apparaître son squelette. Cliquer sur une articulation la sélectionne, la mettant automatiquement en tant qu'articulation de la clé. L'utilisateur peut ensuite modifier la rotation de cette articulation par le biais d'un panneau qui apparaît au dessus du panneau de la timeline, ce qui lui permet d'effectuer une rotation dans les trois dimensions spatiales. Déplacer une coordonnée complètement vers un coté tourne l'articulation de 90 degrés dans l'axe donné. Une fois toutes les modifications effectuées, l'utilisateur peut sortir du mode modifications on effectuant un second Crtl-clique-droit sur la clé.

Le cas spécial de la timeline de caméra de mouvement est implémentée de la même façon que les clés de translation pour la sélection du type et de sa création, mais reprend les éléments des clés de mouvement pour sélectionner un objet à suivre, si l'utilisateur le souhaite.

Les clés de son et de particules sont différentes de toutes les autres, puisque l'utilisateur ne peut pas en créer. Elles sont générées automatiquement lorsqu'un son ou qu'une particule est importée. L'utilisateur a juste simplement à déplacer la clé pour modifier sa valeur temporelle.

Toutes les clés ont une valeur temporelle qui doit pouvoir être modifiée. Lorsqu'une clé est créée, elle est placée au centre de la timeline, c'est-à-dire entre le temps 0 et le temps maximal. Changer la valeur temporelle est fait simplement en déplacent la clé le long de la timeline. Mettre la clé tout à gauche permet d'activer la clé instantanément, alors que la placer tout à droite l'activera à la fin du mouvement. Nous pouvons remarquer que déplacer une clé demande un clique-gauche sur la clé, activant ainsi le panneau de propriétés, pour que l'utilisateur puisse savoir exactement quelle clé il déplace.

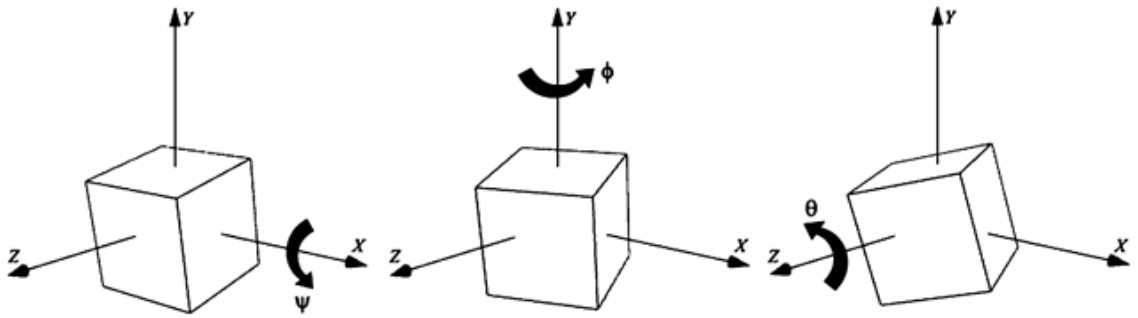
Les timelines est ce qui relie l'utilisateur au mouvement, de façon similaire au reste du GUI, qui relie les objets de la scène à l'utilisateur.

3.1.2 Mouvement de la caméra

Pour permettre à l'utilisateur de créer sa scène et bouger dans l'espace 3D, nous avons dû implémenter des mouvements de caméra. Un premier essai utilisait les mouvements de la souris en les synchronisant à la caméra. Cette méthode était très intuitive mais ne permettait pas un large éventail de mouvements : l'axe Y de Unity n'était pas pris en compte. La seconde idée était de reprendre les déplacements de Unity. Cela signifiait que les mouvements de la caméra devaient être décomposés en trois sections : la rotation (clic droit et tirer), les mouvements de côté (appuyer sur la molette et tirer) et le zoom (faire

défiler avec la molette).

La rotation dans Unity peut être identifiée par les angles d'Euler. Cette méthode compare les axes locaux d'un corps rigide avec des espaces fixés. Toute orientation peut être décrite par trois éléments de rotation : les trois angles d'Euler, ceci nous permet d'agir sur l'orientation de la camera dans l'espace en lui appliquant un vecteur 3D.



Rotation du cube avec la représentation des angles d'Euler

Ici, nous pouvons voir les trois composants qui déterminent l'orientation du cube. Les trois angles d'Euler sont ψ, ϕ and θ, respectivement sur les axes x, y et z.

Dans notre cas, la rotation affecte seulement les axes x et y de la caméra. Le programme se lance quand la molette est cliquée. A ce moment, la position de la souris est enregistrée en tant que position initiale. Si le curseur se déplace, la différence entre la position initiale et la position actuelle est calculée, ceci donne le vecteur 2D de rotation (sa dernière coordonnée z est mise à 0). Le vecteur est alors ajouté aux angles d'Euler de la caméra, multiplié par la variable de temps et la vitesse choisie.

Les mouvements sur le côté ont été la partie la plus simple à implémenter. Le programme doit détecter quand l'utilisateur appuie sur la molette et soustraire la position de la souris (multipliée par les variables de temps et de vitesse) à la position de la caméra. Seulement les axes x et y ont besoin d'être mis à jour comme les mouvements sur le côté traitent seulement des déplacements parallèles à l'écran.

Finalement, le zoom consiste en un déplacement linéaire sur l'axe z (dans Unity l'axe z correspond à la profondeur de la scène). Cependant, nous avons dû prendre en compte l'orientation de la caméra : l'axe z local de la caméra n'est peut-être pas aligné avec l'axe z absolu. Une solution à ce problème était de lier la caméra à un objet vide, un parent, garderait toujours la même orientation et qui suivrait la caméra dans ses mouvements. Pour obtenir l'axe z de la caméra, il suffit de récupérer les localEulerAngles de la caméra et prendre seulement le

composant z. Ainsi, chaque changement de position doit être appliqué au parent de la caméra (de ce fait, la position locale de la caméra est toujours le vecteur zéro) et toute rotation doit être appliquée sur les angles d'Euler locaux de la caméra, sans affecter son parent.

3.1.2.1 Déplacement des objets

Un autre élément important qui permet à l'utilisateur d'interagir avec les objets de la scène est d'avoir la possibilité de les déplacer dans l'espace 3D. Cela est rendu possible grâce à l'outil Raycast de Unity qui dessine une ligne perpendiculaire à l'écran et qui a pour origine le curseur de la souris. Le rayon généré va croiser le collisionneur de l'objet et puis retourner l'objet en question.

Lorsque la souris se déplace, sa position est ajoutée à celle de l'objet ciblé. De plus, nous voulions que le déplacement de l'objet soit plus précis et qu'il soit possible de déplacer l'objet le long de l'axe z (ce qui ne pouvait être réalisé qu'avec la souris). L'idée est donc de décomposer le mouvement de l'objet cible sur les trois axes. Suivant ce principe, si l'on appuie sur la touche "X", l'objet doit se déplacer le long de l'axe x (respectivement avec les axes y et z).

3.1.3 Éléments d'interface graphique supplémentaires

En même temps que le GUI prenait forme, nous nous sommes aperçus que certains éléments devraient être ajoutés. Premièrement, le système de drag and drop utilisé pour déplacer des objets dans la scène devait être étendu à la rotation et à l'échelle. Leurs implémentations ont alors été faites dans la même optique. Afin de les activer, l'utilisateur doit simplement maintenir la touche "R" pour la rotation, "S" (pour scale) pour l'échelle, en même temps que "X", "Y" ou "Z". Nous avons également ajouté un système pour élargir les panneaux grâce à un système de curseurs. Ce dernier a été expliqué en détails lors de l'explication du bouton "Option".

3.2 Import des meshs et du son

Cette partie était compliquée, au début notre but était d'ouvrir une fenêtre de dialogue et de laisser l'utilisateur choisir son fichier, le programme devait vérifier si le fichier était un fichier audio ou un modèle 3D. Quand nous avons commencé, le premier problème que nous avons rencontrer était que nous ne savions pas où les fichiers choisis allaient, nous avons donc créé deux dossiers Models et Sounds et avons changé le chemin dépendant de l'extension du fichier choisi. Ensuite, quand nous faisions tourner le programme, la copie prenait une à deux minutes. De plus, quand le programme tourne, nous n'avions pas accès

à la classe UnityEditor qui est nécessaire pour ouvrir la fenêtre de dialogue pour que l'utilisateur puisse choisir son fichier, nous avons donc opté pour une autre méthode.

L'autre méthode n'est pas la plus pratique sachant que l'utilisateur doit aller chercher ses fichiers lui-même, comme le chemin vers le fichier est demandé, puis ce fichier est copié dans le fichier Ressource (nous avons remarqué qu'un seul dossier est nécessaire pour les deux types de fichier). Comme cette méthode est plus rapide que la première, nous avons décidé de garder celle-ci, malgré sa difficulté d'utilisation.

3.3 Squelette

3.3.1 Création manuelle

L'utilisateur a la possibilité de générer et de placer lui-même les points du squelette spécifique à un objet. Le processus commence par l'activation du script par le bouton "Manual skeleton". Un objet de la scène doit ensuite être sélectionné. Cette étape fait appel à la fonction Raycast de Unity qui permet au programme d'identifier un objet de la scène qui se trouve sous le curseur, lors d'un clic gauche. Une sphère rouge qui représente la racine du squelette est générée, puis attachée à l'objet principal en tant qu'enfant de celui-ci. Ainsi, tous les mouvements du parent affecteront la position de la racine, donc du squelette.

Le reste du squelette se constitue d'un arbre général, chaque articulation représentant un nœud de cet arbre. Il y a deux manières de compléter le squelette. En pressant la touche 'E' du clavier, il est possible d'ajouter un enfant (une sphère) au dernier nœud ajouté. L'utilisateur peut également simplement sélectionner une sphère du squelette, déplacer cette sphère à l'aide du curseur puis ajouter un enfant.

La relation parent-enfant entre les sphères du squelette permet d'obtenir une hiérarchie dans le mouvement du squelette. Ainsi, la translation d'une sphère entraînera le même mouvement pour tous les enfants de cette sphère mais n'affectera pas la position du parent. La hiérarchie entre les points du squelette est très utile pour implémenter l'animation du squelette, nous l'expliquerons par la suite.

3.3.2 Création automatique

Le squelette automatique n'a pas été finie car des problèmes lors de la programmation se sont posés.

Pour la première présentation, nous avions des idées assez claires sur la façon dont nous allions procéder, mais nous avons eu des problèmes avec la fonction qui récupérait les collisions avec le mesh. En effet, nous ne comprenions pas correctement comment le mesh était construit. Ainsi pour la seconde présentation nous avons tout d'abord essayé de comprendre le concept de mesh dans Unity et son fonctionnement.

Nous allons dans cette partie expliquer tout d'abord les fonctions que nous avons implémenté puis détailler pourquoi elles ne marchent pas.

Pour le squelette automatique, deux classes étaient nécessaires, la classe Point et la classe Tree, utilisées pour faire la hiérarchie de l'arbre. Ces classes ont été supprimées avec l'abandon du squelette automatique.

Classe Point

Cette classe est utilisée pour faire les noeuds du squelette. Elle prend une chaîne de caractères et une sphère en paramètres. Le premier paramètre permet de donner un nom au point et le deuxième le place en fonction des coordonnées de la sphère.

La classe Point a différentes méthodes pour que la position et la rotation puissent être modifiées. Chaque point doit être ajouté à un arbre et a donc sa position dans l'arbre modifié à la création de l'arbre.

Classe Tree

Cette classe est utilisée pour créer les liens entre les différents points. C'est une implémentation d'un arbre général, avec un point qui est la racine de l'arbre et une liste qui contient les sous-arbres. Il est possible d'ajouter un arbre ou juste un point (un arbre est créé à partir du point dans le deuxième cas).
Il a ensuite fallu implémenter les fonctions pour créer le squelette.

Trouver les collisions avec le mesh

Une des plus importantes fonctionnalités que nous devions implémenter pour pouvoir créer le squelette automatiquement était la recherche des collisions d'une ligne avec le mesh. Ceci nous permet de savoir si les points dans un espace 3D sont à l'intérieur ou à l'extérieur du mesh ou si un segment est entièrement compris dans le mesh. Nous avons dû essayer d'implémenter cette fonction comme aucune fonction de Unity nous donnait les résultats que nous voulions. La fonction marche ainsi : le programme détermine si le segment intersecte un objet, pour chaque triangle du mesh. S'il y a un point d'intersection

alors il est ajouté à une liste en tant qu'objet de la classe Point. Ceci est fait en quatre étapes :

1. Tout d'abord la fonction calcule l'intersection entre la droite et le plan défini par les trois points du triangle. Cette intersection peut soit être vide, dans ce cas on ignore le triangle, un point ou une infinité de point. Le dernier cas est extrêmement rare, car même si la droite appartient théoriquement au plan ce résultat ne sera probablement pas trouvé à cause de la précision des points flottants. Si cela arrive tout de même, il peut être ignoré, comme un autre triangle connecté à celui-ci aura aussi un point d'intersection avec cette droite.

   Trouver le point d'intersection entre le plan et la ligne requiert l'équation cartésienne du plan, donc un vecteur normal au plan et un système d'équations paramétriques de la droite. Cela nous amène à résoudre le système suivant :

$$S = \begin{cases} a * x + b * y + c * z + d = 0 \\ x = \alpha * t + x_M \\ y = \beta * t + y_M \\ z = \gamma * t + z_M \end{cases}$$

   avec $M \begin{pmatrix} x_M \\ y_M \\ z_M \end{pmatrix}$ un point de la droite $\vec{n} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ un vecteur normal au plan défini par les points des triangles.

   Avec quelques calculs nous trouvons la solution pour $I \begin{pmatrix} x_I \\ y_I \\ z_I \end{pmatrix}$, l'intersection entre le plan et la droite :

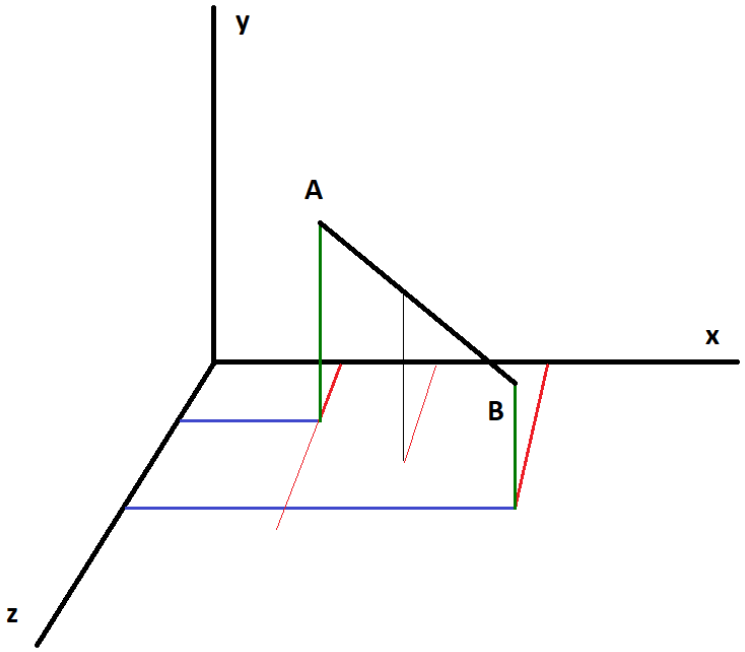
$$S = \begin{cases} t = \frac{-(a*x_A+b*y_A+c*y_A+d)}{a(x_B-x_A)+b(a(y_B-y_A)+c(a(z_B-z_A)} \\ x_I = (x_B - x_A) * t + x_M \\ y_I = (y_B - y_A) * t + y_M \\ z_I = (z_B - z_A) * t + z_M \end{cases}$$

   avec $A \begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix}$ et $B \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix}$ les deux points définissant la droite.

   Si le dénominateur de t est nul, nous sommes dans le cas où l'intersection n'est pas un seul point, nous pouvons simplement renvoyer une exception quand c'est le cas. Cela permet d'avoir une seule formule pour chaque coordonnée de I.

2. Deuxièmement, le programme regarde si l'intersection du point est incluse dans le triangle. La méthode utilisée pour chaque côté du triangle est la suivante : on vérifie si le point d'intersection est du même côté de la droite formée par deux points du triangle que le dernier point du triangle. Ceci est fait en multipliant les coordonnées du produit vectoriel des vecteurs connectant les deux points n'appartenant pas à la droite avec un des points de la droite. Si les deux points sont du même côté alors le produit vectoriel aura la même orientation et la multiplication scalaire sera positive. S'ils sont sur des côtés différents alors la multiplication sera négative. Tester ceci pour les trois côtés du triangle nous permet de savoir si le point d'intersection est compris dans le triangle ou non.

3. Nous avons à présent une liste avec toutes les intersections entre la droite et le mesh. Cependant ceux qui sont demandés sont ceux entre les deux points que l'on a donné en entrée. La prochaine étape est donc de retirer les points qui ne font pas partie du segment. Ceci est fait en regardant coordonnée par coordonnée. En supposant que la différence entre les coordonnées x des deux points ne soit pas nulle, n'importe quel point entre les deux extrémités aura ses coordonnée x comprise entre les coordonnées x des deux extrémités. Pour éviter les problèmes de précision des points flottants, le programme vérifie pour les coordonnées (x, y ou z) pour lesquels la différence entre les deux point est la plus grande.



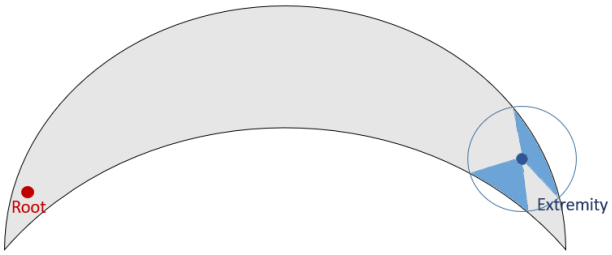
Identification du point dans le segment

4. La dernière étape est de trier les points pour qu'ils soient en ordre. Cela signifie les organiser par ordre croissant de distance à un des deux points. Pour ce faire, le programme génère des couples (implémentés en tant que liste de longueur 2) avec la distance et l'indexe du point dans la liste

originale. Un simple tri à bulle est effectué sur cette liste de distances, une nouvelle liste est ainsi créée en utilisant les nouveaux indexes. La fonction retourne alors la liste des intersections.

Trouver les extrémités du squelette

Les outils pour commencer à trouver les points du squelette sont en place, le programme peut commencer par trouver les extrémités. Pour ce faire le programme traverse l'ensemble de l'intérieur du mesh, en regardant des points à intervalle donnée. Chacun de ces points est analysé pour savoir si c'est une extrémité. Dans un rayon déterminé par la taille de l'objet et la densité par l'utilisateur, le programme regarde à quel point le point est entouré de mesh. Ceci est fait par une approximation, en regardant dans vingt-six directions, à un angle de $\frac{\pi}{2}$ les unes des autres. Si le point est considéré comme une extrémité alors il est ajouté à la liste des extrémités. Comme les positions qui sont regardées sont à une distance proportionnelle à la taille du cercle, des points superposés sont évités.



recherche des extrémités

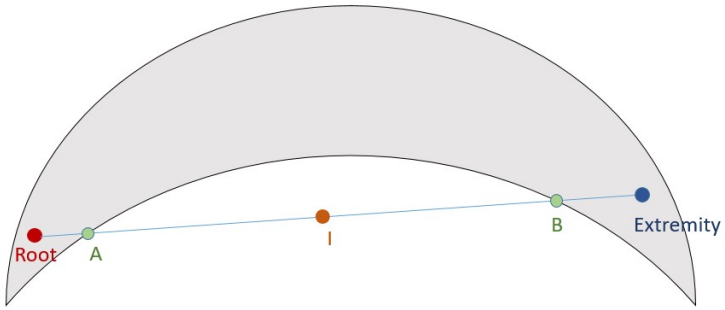
Création de points et liaison entre la racine du squelette et ses extrémités

La prochaine chose qu'il fallait faire était des liens entre la racine choisie par l'utilisateur et chacune des extrémités du mesh. Pour chacun des bouts, le but était de soit lier directement la racine à l'extrémité soit de trouver des points intermédiaires qui permettaient ce lien.

S'il est possible de les lier directement, c'est-à-dire que la fonction déterminant les collisions avec avec le mesh n'en a pas trouvé, alors l'extrémité est ajoutée à l'arbre en tant qu'enfant de la racine.

Autrement, nous commençons à chercher des points intermédiaires qui permettraient de faire des liaisons. Ceci se fait en quatre étapes.

1. Si le point ne peut pas être lié à l'extrémité, cela signifie qu'il doit y avoir au moins deux points de collisions avec le mesh entre les deux, nous prenons donc ces deux points que nous appellerons A et B et nous cherchons leur milieu I.

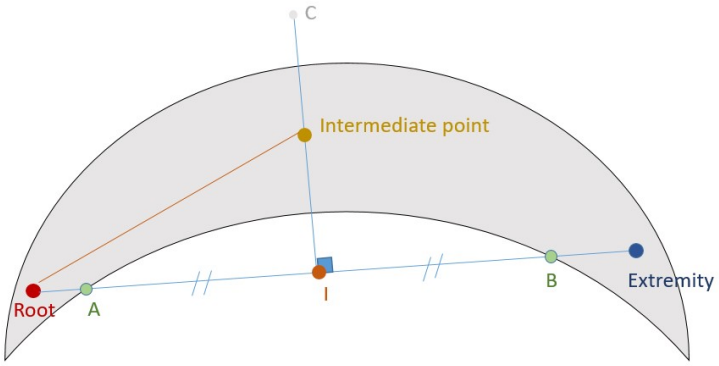


Recherche du centre de AB

2. Après avoir trouvé I, il faut trouver les points du cercle de centre I et de rayon donné (assez grand pour pour qu'il y ait potentiellement des collisions avec le mesh) Ce cercle est l'intersection de la sphère de centre I et rayon R avec le plan contenant I et orthogonal à (AB). Les points de ce cercle sont donnés par :

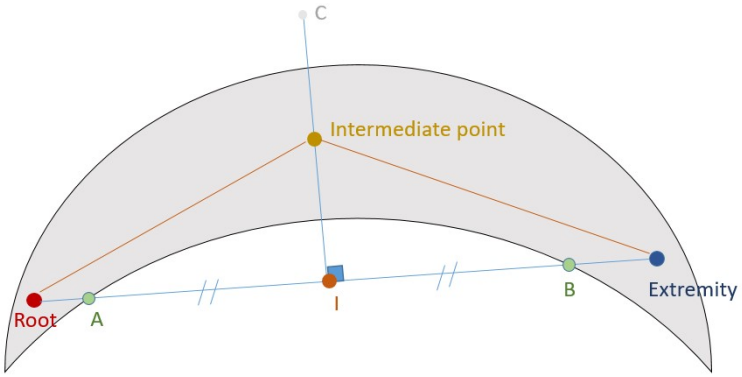
$$C : \begin{cases} x = \frac{-(2*y_I+R*sin(\Phi)*sin(\theta))*(y_B-y_A)-2*z_I+R*cos(\Phi))*(z_B-z_A)}{x_B-x_A} \\ y = y_I + R * sin(\Phi) * sin(\theta) \\ z = z_I + R * cos(\Phi) \end{cases}$$

avec $0 \leq \Phi \leq 2\pi$ et $0 \leq \theta \leq \pi$.
Si $x_B - x_A = 0$ alors l'intersection est calculée différemment et donnera des formules similaires à la précédente avec $y_B - y_A$ ou $z_B - z_A$ en tant que diviseurs. Comme A et B ne peuvent pas être le même point (c'est-à-dire superposés), il y aura au moins un de ces diviseurs différents de 0.
Ensuite, il faut regarder pour différents angles de θ et Φ s'il y a des collisions avec le mesh entre le point I et le point calculé auparavant. S'il n'y en a pas on continue juste jusqu'à ce qu'on trouve un point pour lequel on trouve au moins deux intersections ou jusqu'à ce qu'on ait effectué un cercle complet. Dans ce dernier cas, il n'est pas possible de créer un point intermédiaire en utilisant le point I, ainsi la première étape est refaite à droite et à gauche de I en utilisant I en tant que A ou B.
3. Si un point pour lequel le mesh est traversé au moins deux fois est trouvé, alors un point dans le mesh est pris (entre la première et la deuxième collision avec le mesh). Ensuite, si ce point peut être lié à la racine, un point de la classe du même nom est créé et ce point devient un enfant de la racine. Sinon des essais sont faits sur d'autres points du segment entre les deux collisions, un nombre d'essais maximum est donné dans le code. Si aucune possibilité n'a été trouvée, l'étape 1 est recommencée.



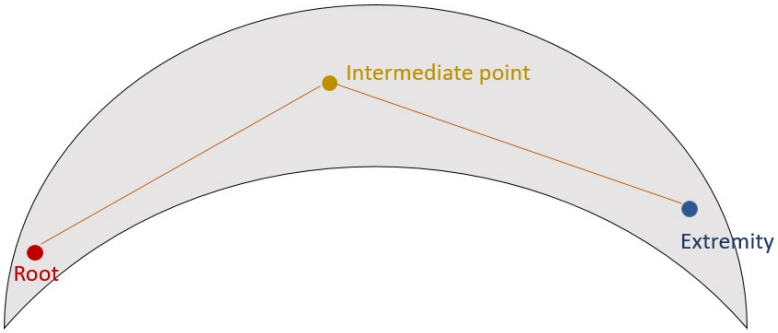
Recherche d'un point du mesh et liaison à la racine

4. Si la racine a été liée à un point intermédiaire, l'extrémité est liée à ce point si possible, sinon la première étape est répétée, en utilisant le point créé en tant que racine. Les autres extrémités sont testées pour voir si elles ne peuvent pas être liées à ce point intermédiaire. Si c'est possible pour un extrémité alors elle est liée sinon les quatre étapes sont répétées sur la prochaine extrémité si elle n'est pas liée à la racine.



Liaison du point intermédiaire à l'extrémité

Le squelette obtenu par l'utilisation des trois fonctions présentées aurait été composé d'une racine liée aux extrémités par des points intermédiaires.



Squelette final

Nous allons à présent aborder les raisons pour lesquelles nous avons abandonné le squelette automatique

Au commencement, après avoir lu la documentation à propos de la liste de vertices qui peut être récupérée d'un GameObject de Unity, nous avons pensé qu'elles pouvaient être facilement manipulées en les convertissant en listes de triangles dans l'espace tridimensionnel. Nous étions assez surpris de découvrir que nous obtenions des résultats très éloignés de ce que nous espérions, que ce soit en utilisant les fonctions convertissant les coordonnées locales en coordonnées absolues de Unity ou bien en faisant nos propres coordonnées. Nous avons plus tardivement découvert que la liste de vertices n'était pas construite de la manière dont nous l'avions initialement comprise en lisant la documentation, puisque chaque vertex est triplé et leur position dans la liste était visiblement aléatoire. Nous n'avons pas réussi à trouver d'information concernant la construction de cette liste par Unity, rendant la tâche plus compliquée que ce à quoi nous nous attendions.

Suite à quelques recherches à ce sujet, nous avons découvert la fonction RayCast, cependant en apprenant qu'elle ne renvoyait que le premier objet touché nous l'avons rapidement mise de côté. A la première présentation, nous nous étions déjà intéressés cette fonction, mais nous l'avons rejetée. Nous nous sommes rendus compte que le problème venait du fait que nous ne récupérions pas la position à laquelle il fallait s'intéresser, par conséquent nous nous retrouvions avec de mauvais paramètres d'entrée pour nos fonctions. Après quelques tests, nous savons que les fonctions auraient fonctionné si nous avions les bons paramètres d'entrée, mais cette voie n'a pas fonctionné pour nous.

Peu de temps après la première présentation, nous avons redécouvert la fonction RayCast, plus précisément la fonction RayCastAll, qui retourne de multiples intersections. Nous espérions qu'utiliser une telle fonction nous aurait permis de ne pas avoir à utiliser les vertices, et simplement en récupérant les intersections entre le mesh et les rayons de la fonctions RayCast qui nous intéressaient. Cela était presque parfait, mais nous avons rapidement compris que RayCastAll ne retournait pas de multiples points pour de multiples collisions avec avec le même objet. De ce fait, nous sommes revenus à la case départ à la recherche de nouvelles solutions, l'une de ces solutions était d'utiliser le simple RayCast plusieurs fois, chaque rayon ayant pour point de départ l'endroit où le précédent s'est arrêté. Cela, une fois de plus, a eu des défauts majeurs, notamment lorsqu'aucune collision n'est détectée avec l'objet depuis lequel le rayon est diffusé. Ce qui signifie, que peu importe ce que nous faisons, nous manquions des collisions. Premièrement, nous manquions la première collision puisque nous commencions à l'intérieur de l'objet que nous souhaitions détecter, puis la moitié des collisions restantes puisqu'elles venaient de l'intérieur de

l'objet également et non de l'extérieur.

Une potentielle autre solution est d'avoir deux boucles de RayCast, toutes les deux dans des directions opposées et commençant à l'extérieur du mesh de l'objet. Cela signifie que nous devrions calculer la distance totale couverte jusqu'à ce que nous arrivions au premier point que nous voulons en ignorant toutes les autres collisions. Nous aurons ensuite besoin de conserver toutes les collisions jusqu'à ce que nous arrivions au second point voulu, enfin le programme s'arrêtera ou ignorera les collisions restantes. Cela doit être fait dans les deux directions, ensuite les listes résultantes devront être fusionnées nous donnant ainsi une liste fonctionnelle. Cependant, cela serait très compliqué à gérer, sans même tenir compte du fait que calculer la distance parcourue par le rayon manquerait de précision, du fait que nous ajouteront continuellement une marge d'erreur à chaque collision notamment due à la nature des flottants. Cette méthode pose un dernier problème, qui est de trouver les bords extérieurs de l'objet dans le but de pouvoir commencer à diffuser le rayon depuis l'extérieur du mesh de l'objet. Nous expliquerons plus tard pourquoi cela a été problématique. Pour éviter cela, nous pourrions simplement choisir une distance arbitraire à partir du point initial, mais cela pourrait amener à des erreurs lorsque nous utilisons des objets larges, ou ils devront être assez larges pour que dans n'importe quelle scène relativement complexe, le programme puisse avoir une complexité telle qui le ralentirait drastiquement, ce qui rendrait la fonctionnalité inutilisable, et totalement inutile.

La fonction gérant les collisions n'étant pas opérationnelle, le squelette ne peut pas être fait, car les deux fonctions qui construisent le squelette ont besoin des collisions avec les mesh.

En plus de la fonction utilisée pour la collision, nous avons une fonction qui permet de récupérer les extrémités du mesh. Pour cela, nous avons besoin des dimensions du mesh qui, de nouveau, n'étaient pas aussi simples que ce à quoi nous nous attendions. Notre première idée était de prendre une liste complète de vertices du mesh et faire un simple calcul du minimum et du maximum pour chacune des dimensions. Comme dit plus tôt, cela pourrait ne pas fonctionner dû au fait que nous ne récupérions pas correctement la position des vertices. Les tests ont confirmés cela, puisque nous avions des résultats inattendus. Par conséquent, nous nous sommes tournés vers une autre méthode qui consistait à récupérer les dimensions de l'objet que nous étudions. En même temps que le centre de l'objet, nous aurions pu récupérer les bords du mesh de l'objet. Cela semblait fonctionner au début, mais nous avons découvert que récupérer le centre de l'objet est impossible puisque cette information n'est pas fournie par Unity. Utiliser la position absolue de l'objet ne résout pas le problème, puisque la position ne correspond pas à la position du centre de l'objet. De ce fait nous

n'avions aucune solution pour déterminer les dimensions d'un mesh.

Dépourvus d'autres possibilités, nous avons finalement décidé d'abandonner l'implémentation du squelette automatique dans le but de pouvoir se concentrer sur les autres parties de ce projet, du fait que la construction du squelette automatique ne constitue pas une fonctionnalité importante de notre programme.

Même si le squelette automatique n'est pas opérationnelle, il nous reste toujours le squelette manuel qui, lui, est complètement opérationnel et sert de base à la création du mouvement.

3.4 Création du mouvement

Pour cette partie, nous devions différencier deux types de mouvement : le déplacement des objets et les articulations du modèle. Ce dernier implique la déformation du mesh ainsi que l'utilisation du squelette pour créer l'illusion que le modèle est animé.

3.4.1 Déplacement des objets

3.4.1.1 Classe Key

L'animation d'un objet se décompose en trois composantes de mouvement : la position d'un objet, sa rotation et son échelle. L'utilisateur doit être apte à décider des valeurs de ces composantes à n'importe quel moment de l'animation dans l'espace 3D. Notre système gérant ces informations est inspiré du système de gestion des clés du logiciel Blender.

Une clé est un objet placé sur la timeline par l'utilisateur qui contient des informations relatives à un objet : soit sa position, sa rotation ou son échelle (selon le type de la clé). De cette manière, l'utilisateur donne des valeurs fixes à certains moments de l'animation que le programme va compléter avec les étapes intermédiaires entre ces valeurs. Pour un mouvement linéaire d'un point A à un point B en 10 secondes, l'utilisateur devra créer une clé de position au point A au temps 0, puis une autre clé, à 10 secondes d'intervalle sur la timeline, qui contient les coordonnées du point B. Le même principe a été utilisé pour la rotation et l'échelle : le programme calculera les changements d'état de rotation ou d'échelle à un autre, en utilisant les clés.

Ainsi, nous devions implémenter la classe Key. Les objets de cette classe sont caractérisés par six attributs : un type, le GameObject auquel elle appartient, une courbe de vitesse, une valeur, une information de temps ainsi qu'un booléen qui indique si l'objet suivra une trajectoire linéaire jusqu'à la clé suivante. Le GameObject est l'objet sur la scène qui sera animé. Chaque objet possède sa propre timeline avec ses propres clés. Il y a trois types de clé, une

pour la position que nous appelons Loc, une pour la rotation appelée Rot et enfin une pour l'échelle que nous appelons Scale. Ces types nous permettent de déterminer quelle donnée de l'objet doit être modifiée parmi la position, la rotation et l'échelle. Le temps donné en secondes correspond au moment auquel l'état de l'objet doit atteindre la valeur donnée par la clé. La valeur est un Vector3 puisque nous devons exprimer le mouvement des différentes composantes au sein de l'espace 3D. L'utilisation des courbes de vitesse sera prochainement expliqué.

3.4.1.2 Calcul de la trajectoire

Maintenant que nous avons nos trois listes par objet, nous devons traiter les informations et calculer une trajectoire. En premier lieu, le but était de permettre à l'utilisateur de dessiner des courbes de Bézier dans l'espace qui représenteraient le parcours de l'objet. Cependant les courbes sont difficiles à coder et ne sont pas nécessairement intuitives pour l'utilisateur.

Nous nous sommes par la suite aperçus qu'il existait de nombreux assets dans l'Unity Asset Store qui facilitent la création du mouvement. L'un d'entre eux, nommé iTween Editor, est une bibliothèque nous permettant de gérer les changements de position, rotation et échelle aisément. Notre première idée était d'utiliser iTween afin de déplacer les objets puisque cette bibliothèque contient des fonctions qui, à partir de deux valeurs et un intervalle de temps, modifie l'état de l'objet au cours du temps. Cependant, puisqu'il existe une fonction pour chacune des composantes (position, rotation, échelle), il était impossible de toutes les modifier en même temps.

Notre seconde idée était d'utiliser la fonction Update() qui est appelée à chaque image de l'animation. Cela nous a permis de mettre à jour l'état des objets à tout instant pour chacun des paramètres.

L'état de l'objet est décrit à tout instant par une fonction de trajectoire qui prend en paramètres la valeur et le temps de deux clés. Il existe deux options de trajectoire. Premièrement la trajectoire linéaire, les clés y sont reliées et l'équation du mouvement est une fonction affine de la forme $f(x) = ax + b$ avec a la différence de coordonnées entre deux clés divisée par la différence de temps, donnant ainsi la pente, et b la coordonnée à l'origine. Ensuite, il y a la trajectoire courbe qui est partiellement gérée par iTween. Nous avons utilisé la fonction PutOnPath() qui reçoit trois paramètres : un GameObject (celui que nous voulons déplacer), un tableau de coordonnées ainsi qu'un nombre entre 0 et 1 représentant le pourcentage du chemin auquel l'objet doit être placé. Ce nombre est égal au temps actuel divisé par le temps total de l'animation. A chaque image, le pourcentage augmente et donne l'illusion du mouvement.

La difficulté était de gérer trois types de donnée simultanément. Les paramètres de rotation et d'échelle ont également les options de progression linéaire et courbe. La même fonction affine est implémentée pour la progression linéaire. Cependant, la fonction PutOnPath de iTween fonctionne seulement pour une

trajectoire de position. Une des solutions est d'appeler la fonction avec une échelle et une rotation avec deux GameObjects vides. Ces deux GameObjects vides seront placés sur le chemin et le GameObject principal prendra la position des GameObjects vides en tant que valeur de rotation ou d'échelle, les valeurs de position, rotation et échelle étant tous les trois des Vector3s.

Cf. Annexe 3

3.4.1.3 Courbes de vitesse

Après avoir obtenu la trajectoire de l'objet, l'utilisateur doit pouvoir régler un autre paramètre : la vitesse le long de cette trajectoire. Les courbes de vitesse permettent de définir la variation de vitesse au cours de l'animation, entre deux clés. De la même manière que pour la trajectoire, notre première idée fut de coder une courbe de Bézier modifiable qui représenterait cette variation de vitesse. Ainsi, il serait possible de donner une accélération au début du mouvement ainsi qu'une décélération.

iTween possède un ensemble de de courbes prédéfinies donc plutôt que d'utiliser des courbes de Bézier, nous avons repris cet ensemble de courbes. L'utilisateur aura ainsi à choisir celle qui correspond à l'accélération et la décélération attendues. Cependant, les courbes utilisées par iTween ne sont accessibles que depuis les fonctions de mouvement qu'il nous était impossible d'utiliser à cause du problème expliqué précédemment. Il était pourtant possible pour nous de récupérer les fonctions mathématiques de ces courbes puis de les recoder afin de les intégrer à la fonction Update().

Les courbes de vitesse d'iTween sont inspirées des fonctions d'atténuation de Robert Penner. Il existe trois types d'atténuation : "ease-in" (accélération progressive au départ), "ease-out" (ralentissement à la fin du mouvement) et "ease-in-out" (combinaison des deux précédents : accélération puis ralentissement). Il reste à définir le degré d'atténuation. Cela est déterminé par le degré du polynôme de la fonction ou du type de la fonction (sinus, exponentielle...). iTween donne accès à 32 courbes d'atténuation. Nous avons décidé d'en retenir 10 qui offrent malgré tout, un large choix.

Cf. Annexe 4

La courbe linéaire présente la variation la plus simple. La vitesse est constante le long de la trajectoire. Dans ce cas-là, il n'y a pas besoin d'implémenter d'"easing-in" ou "out", la vitesse étant constante.

Cf. Annexe 5

La deuxième courbe de mouvement correspond à une atténuation quadratique. Cette fonction se compose d'un polynôme du second degré. La courbe représente donc une demi-parabole. L'accélération ou "ease-in" est exprimé $f(t) = c * (t/d) * (t/d) + b$ avec t le temps, b la coordonnée initiale, c la différence de

coordonnées pendant l'intervalle de temps ainsi que d l'intervalle de temps. La fonction de décélération ou "ease-out" est alors : $f(t) = -c * (t/d) * ((t/d) - 2) + b$ avec des paramètres identiques. La fonction "ease in-out" prend la première ou la seconde équation selon l'avancement de l'animation.

Cf. Annexe 6

La troisième courbe d'atténuation est une fonction sinus. En ease-in, elle s'exprime : $f(t) = c * (t/d)^5 + b$ et en ease-out : $f(t) = c * ((t/d)^5 + 1) + b$.

Cf. Annexe 7

Finalement, l'atténuation quintique possède la plus forte accélération, elle se compose d'un polynôme du cinquième degré. Elle s'exprime $f(t) = c * (t/d)^5 + b$ pour l'ease-in et $f(t) = c * ((t/d)^5 + 1) + b$ pour l'ease-out.

Ainsi, pour la variation de vitesse de la position, rotation et taille d'un objet, l'utilisateur aura le choix entre une progression linéaire ou courbe, sachant que l'option courbe est disponible en en trois versions. Cela peut paraître contraignant pour l'utilisateur d'avoir à choisir parmi des courbes préexistantes mais notre logiciel est ainsi plus simple d'utilisation tout en offrant un large panel de possibilités.

3.4.2 Caméra

Pour l'animation, il est nécessaire d'avoir des caméras. Le rendu final utilisera les caméras ajoutées par l'utilisateur. De ce fait, son mouvement durant l'animation est un point important. La caméra aura le même système de déplacement que les autres objets de la scène. Elles auront un ensemble de clés puisqu'il s'agit également de GameObjects, elles ont aussi une position, une rotation mais pas d'échelle. En plus de cela, nous pensions que nous pouvions ajouter d'autres fonctionnalités qui pourraient être utiles pour les mouvements de la caméra. Les méthodes implémentées permettent à la caméra de suivre son propre chemin et d'adapter sa focalisation sur un GameObject qui pourrait être invisible puisque l'utilisateur pourrait ne pas vouloir que la focalisation soit exactement sur l'objet qu'il anime. Une autre fonctionnalité est le déplacement de la caméra avec un objet animé, ainsi le GameObject animé restera dans le champ de la caméra.

3.4.3 Mouvement interne

Le mouvement interne d'un modèle peut être décomposé en deux étapes de création. Premièrement le skinning du modèle, c'est-à-dire que tous les vertices du mesh seront associés à un os du squelette. Ainsi, chaque fois qu'un os est déplacé, les vertices associés suivront le mouvement. La seconde étape du mouvement interne est l'articulation du modèle, en utilisant une version modifiée

des clés de mouvement de la classe Key, l'utilisateur sera apte à déplacer les articulations et à déterminer leurs mouvements sur la timeline.

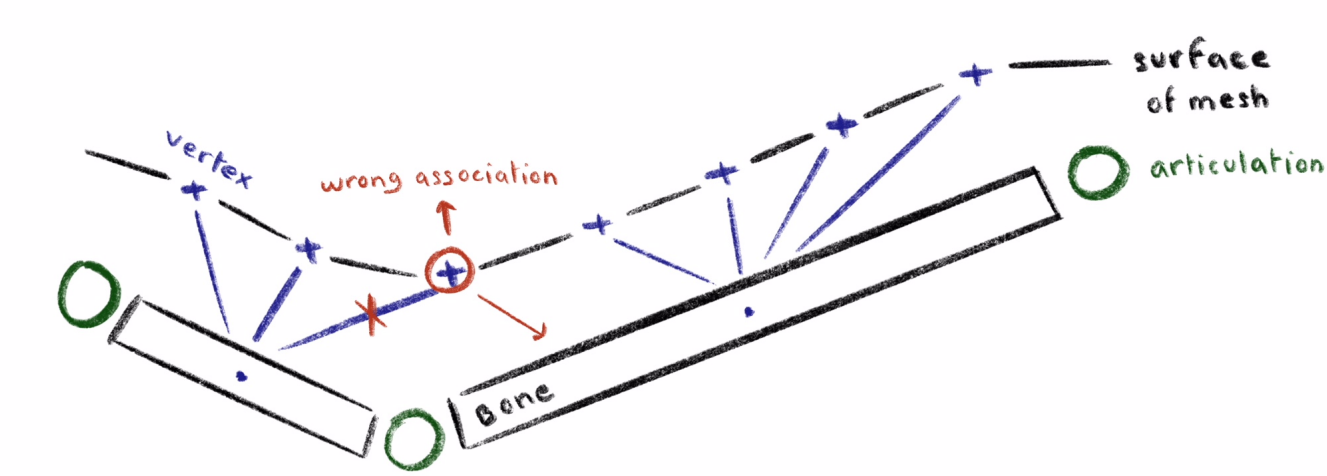
Skinning

Dans beaucoup de logiciel d'animation, le processus de skinning est accessible en deux versions : manuel ou automatique. Nous avons choisi d'implémenter ces deux types de skinning. Ainsi, l'utilisateur aura le choix entre une sélection manuelle de groupes de vertices puis association de ces groupes aux os du squelette ou de lancer le skinning automatique.

Dans un premier temps, afin de pouvoir modifier l'emplacement des vertices, il était nécessaire de comprendre le fonctionnement des vertices sous Unity. Nous avons rencontré le même problème lors de la création automatique du squelette. Les vertices d'un mesh sont accessibles grâce à la méthode GetVertices qui retourne une liste de triplets qui représentent les coordonnées des vertices. Pour le rendu de l'objet, Unity duplique tous les vertices en trois exemplaires. Cela implique qu'il existera toujours trois vertices au même emplacement. Par exemple, un cube aurait 8 vertices en temps normal mais la fonction GetVertices retourne une liste de 24 vertices, en ordre aléatoire. Si l'on déplace une de ces vertices, les deux autres aux positions identiques devraient également se mettre en mouvement. Ensuite, si nous voulons déplacer des vertices nous devrions être capable de changer leurs valeurs de position mais GetVertices ne renvoie qu'une liste de triplets de flottants, pas les vertices en tant que tel. Leurs coordonnées ne peuvent donc pas être modifiées de cette manière. En effet, un vertex n'est pas un objet que l'on peut déplacer ou assigner à un autre objet et ses coordonnées sont en lecture seule. Si nous voulions modifier un vertex il faudrait modifier l'intégralité du mesh. Cela représentait une contrainte importante puisque la déformation d'un mesh implique la modification des coordonnées des vertices. Une solution était d'écrire un script qui s'occuperait de ces deux problèmes : il synchroniserait le mouvement des vertices possédant les même coordonnées (si l'un de vertex bougent, les deux autre suivent) et représenterait chaque triplet par un gameObject sphère. Le fait que les vertices soit représentés par des gameObject facilite leur manipulation et leur localisation. Une sphère aura ainsi en attribut trois indexes qui permettent d'accéder aux vertices dans la liste de la fonction GetVertices. A chaque frame, le programme créé une nouvelle liste de vertices selon la position des sphères et établit ensuite cette liste en tant que nouvelles vertices du mesh.

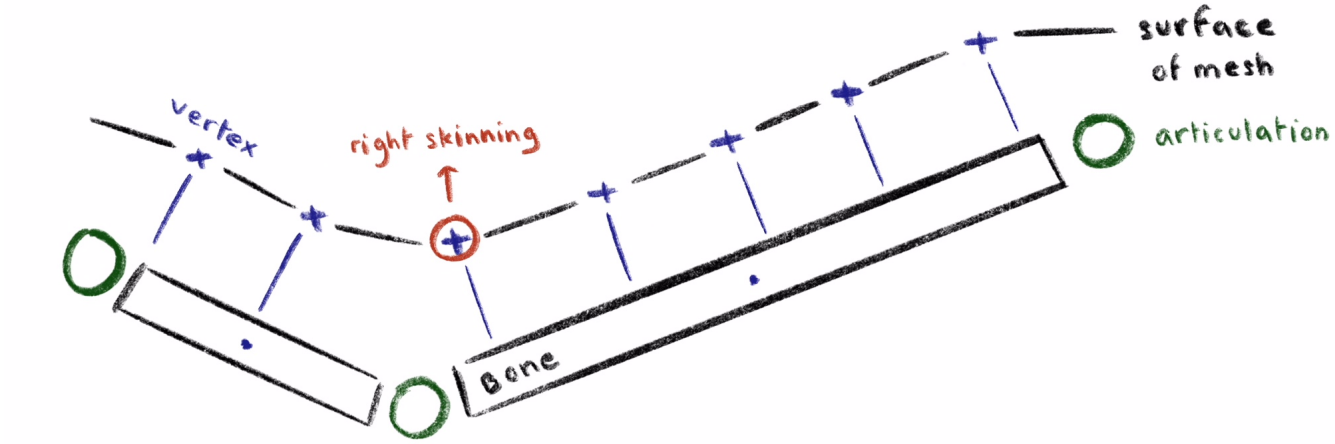
Dans la plupart des skinning, un vertex sera associé à l'os le plus proche. Le skinning automatique est basé sur ce principe : tous les vertices seront des enfants de l'os le plus proche. Il n'y a pas de fonction prédéfinie dans Unity pour identifier le gameObject le plus proche d'une certaine position. Notre première

tentative était de parcourir la liste d'os et de calculer la distance entre les os et les vertices puis de choisir l'os correspondant à la plus courte distance. Cependant, cette méthode ne fonctionne pas dans tous les cas. La position d'un os est déterminée par son centre mais dans certaines situations, un vertex sera considéré plus proche d'un os simplement parce que le centre de cet os est plus proche.



Dans cet exemple, le vertex encerclé est légèrement plus proche du centre de l'os gauche. Pourtant, il devrait être associé à l'os droit. Cette méthode peut donc, dans certains cas, entraîner de mauvaises associations.

Une méthode plus adaptée serait de considérer le point, sur la surface de l'os le plus proche du vertex pour calculer la distance. Nous avons utilisé les collisionneurs de Unity, qui permettent de détecter une collision avec un objet aux alentours. Une sphère est générée à la position du vertex. Tant que le collider de la sphère n'entre pas en contact avec le collisionneur d'un os, le rayon de la sphère augmente. De cette manière la distance par rapport à l'os est calculée en fonction du point à la surface de l'os le plus proche du vertex. Contrairement à notre première tentative, cette méthode fonctionne dans tous les cas.



Si nous reprenons l'exemple précédent, la seconde méthode a résolu le problème de mauvaise association : le vertex est maintenant attaché à l'os droit.

Articulations

L'implémentation des articulations nous aide à déterminer les os auxquels les vertices d'un mesh seront reliés, ce qui permettra au mesh de se déformer lors de la création du mouvement interne.

Nous avons alors implémenté deux classes : ArticulationKey et Articulation. Ces deux classes combines les attributs de la classe Key qui sont un GameObject, un type, une valeur (correspondant à la nouvelle position, rotation ou échelle), une courbe de vitesse, une information de temps ainsi qu'un booléen permettant de savoir si le mouvement sera linéaire. Cependant, créer un clé pour chacune des articulations signifierait avoir des sous-timelines pour chacune d'entre elles ce qui aboutirait à un écran surchargé. C'est pour cette raison que nous avons décidé de séparer les attributs de la classe Key entre ces deux nouvelles classes. Les ArticulationKeys auront pour attribut le GameObject principal, dont nous voulons bouger les articulations, l'information de temps et enfin une liste d'articulations. Quant à elles, les Articulations auront pour attributs un GameObject qui représente l'articulation-même qui sera une sphère, une valeur qui sera une valeur de rotation, une courbe de vitesse et finalement le booléen qui indique la linéarité du mouvement.

Pour le mouvement, une articulation sera seulement capable de se déplacer de son parent et de le suivre. Pour chaque articulation, un GameObject vide qui aura la même position que son parent sera assigné en tant que nouveau parent de l'articulation et ce GameObject vide se verra assigner désormais l'ancien parent de l'articulation ce qui nous permettra de gérer la rotation. Lorsque cela est fait, toutes les articulations sont traitées comme n'importe quel autre

GameObject.

3.5 Images d'arrière-plan et sons

Cette fonctionnalité permet à l'utilisateur d'ajouter des sons et des objets qui ne sont pas attachés à une timeline. Les objets sont juste ajoutés à la scène et sont mobiles sur la scène mais ne le sont pas quand la scène est jouée contrairement aux objets reliés à une timeline. Le son est lié à une timeline pour qu'il puisse être joué au bon moment.

3.6 Animation de particules

L'utilisateur peut charger des particules sur la scène. Ces particules sont prédéfinies et viennent du pack Particle System de Unity. Il en existe cinq types, qui sont actives pendant toute la durée de la scène.

3.7 Site internet

Le site du projet a été commencé pour la première soutenance, nous avons utilisé les outils standards pour sa création : de l'HTML (Hypertext Markup Language) qui est un langage de balises permettant de gérer le contenu de nos pages ainsi que du CSS (Cascading Style Sheet) qui sert à la mise en page. Nous n'avons pas beaucoup utilisé de CSS puisque nous étions autorisés à utiliser des templates, téléchargeables sur Internet, qui sont des modèles prédéfinis de CSS servant ainsi de base à la construction du site. De ce fait, le modèle que nous avons choisi a reçu peu de modifications.

Pour la première soutenance, la structure générale de notre site a été mise en place avec les différentes pages dont nous avions besoin, chacune d'entre elles correspondant à une section : Page d'accueil, Cahier des Charges, A propos, FAQ et Contacts. A ce stade toutes les pages possédaient une version française et anglaise exceptée la section Cahier des Charges. Nous voulions pour cette soutenance que le site soit déjà hébergé, pour cela nous avons utilisé un des services proposés par la plateforme Github qui est Github Pages permettant d'héberger le site ainsi que d'y apporter des mises à jour facilement.

Ensuite, pour la seconde soutenance, la section Cahier des Charges s'est vue remplacée par une section Téléchargement. Dans cette section, notre cahier des charges, les différents rapports de soutenance que nous avons rédigés au cours du semestre ainsi que le logiciel sont téléchargeables. Même si ces pages du site ont été traduites, ce n'est pas le cas pour les différents documents, tous les documents, excepté ce rapport, sont uniquement disponibles en anglais. En

plus de cela nous avons décidé d'utiliser un carnet de bord pour avoir une meilleure idée de l'avancement du projet, pour cela nous nous sommes servis de Google Sheets. Une autre des modifications que nous avons apportées au site est l'ajout d'une musique de fond ayant pour but de créer une atmosphère relaxante.

Cf. Annexe 8

Enfin, ayant conscience que cette musique de fond puisse ne pas avoir l'effet escompté, une version silencieuse du site a été créée pour la dernière soutenance. De plus, le logiciel et le manuel ont été ajoutés dans la section dédiée aux téléchargements. Vous pouvez visiter notre site à l'adresse suivante : https ://kairew.github.io/a

4 Avis personnel sur le projet

Brian

Selon moi, ce projet est une première étape vers le futur où nous serons amenés à travailler avec des gens que nous ne connaissons pas. En effet, au début de l'année nous ne nous connaissions pas vraiment voire pas du tout mais au cours du temps, de complets étrangers nous sommes devenus coéquipiers, et même plus, amis.

Je trouve que nous avons bien fonctionné en tant qu'équipe, il arrivait que les idées divergent à propos de l'implémentation de certaines choses mais cela ne s'est jamais terminé sur une dispute et nous étions toujours capables de trouver un terrain d'entente au bout du compte. Ceci est aussi quelque chose que j'ai appris au cours de ce projet, des idées divergentes ne signifient pas que quelqu'un a tort ou raison mais qu'il n'y pas une unique façon de penser.

D'autre part, ce projet était assez inhabituel comparé aux autres projets auxquels j'ai pu prendre part. La majorité d'entre eux nécessitaient seulement un travail de recherche comme les TPE en Première où le but est seulement de rassembler des informations puis de les trier pour enfin les présenter alors que pour ce projet, même si cette démarche de recherche est également nécessaire, il était important d'être créatif dans sa construction. Sûrement pas aussi créatif que dans le cadre d'un jeu, mais au moins créatif dans le cadre de la résolution de problème, à chaque problème rencontré nous devions trouver un moyen de le résoudre même si cela signifiait restreindre nos perspectives.

Le type de projet étant libre, le choix de quelque chose d'autre qu'un jeu est selon moi audacieux, et nous nous en sommes rapidement rendu compte

lorsque nous nous demandions plus que souvent pourquoi nous n'avions pas plutôt choisi le jeu. D'autant plus que ce projet constitue l'un des premiers, si ce n'est pas le tout premier projet de programmation jamais entrepris pour chacun d'entre nous. Tout cela pris en compte, je pense que nous pouvons dire que nous nous en sommes relativement bien sortis et je me réjouis du fait que nous soyons encore tous vivants.

Etienne

En commençant ce projet, je m'attendais, avec raison, à ce qu'il soit une vraie opportunité d'apprentissage pour moi. J'ai commencé très motivé, et légèrement trop ambitieux. Ayant de nombreuses idées au début du projet, et de nombreuses réponses partielles par rapport à comment implémenter ces idées. J'avais le sentiment de pouvoir faire quelque chose de grand, qui demandait de nombreuses heures de travail, et dont je pouvais être fier. Je savais que ces rêves d'un programme au niveau d'un programme commercial étaient irréalisables, donc j'ai dû limiter mes attentes.

Une fois cette difficulté de départ dépassée, le projet était une expérience agréable. J'ai appris beaucoup sur mes propres méthodes de travail lorsque je travaillais sur du code, malgré le fait que ce projet est loin d'être mon projet à long terme. Celui-ci était différent de toute autre expérience que j'ai pu avoir. Je pense que j'ai surtout appris à naviguer entre avoir des attentes trop élevées et trop faibles pour le travail de ceux qui m'entourent, et comment mettre en vigueur ces attentes. J'ai eu peu, ou pas de problèmes par rapport à la quantité ou qualité de travail des autres dans le groupe. Cependant, je pense que j'aurais pu mieux m'assurer que tout le monte était tout le temps sur la même page, que tout le monde comprenait bien chaque partie du projet de la même façon. Ces petites difficultés m'ont permis de continuer à m'améliorer dans mon travail en groupe, surtout lorsqu'il doit être très bien organisé.

Le résultat du projet final est similaire à ce que je m'attendais à avoir après avoir compris les limites de l'interface graphique dans Unity, qui est moins net et pratique que ce que j'aurais pu espérer. J'ai appris qu'il faut bien rechercher les outils que l'on utilise avant de s'attendre à des résultats spécifiques. Le reste du projet est fonctionnel, ce qui est un accomplissement dont je suis content. Je suis donc fier de ce projet de code car c'est le premier de cette envergure que je réalise.

Léane

Ce projet nous a fait apprendre beaucoup de choses, de l'utilisation de Unity au travail en équipe. Ce projet était très différents de ceux que j'avais fait jusqu'à présent, nous avions plus de libertés sur le sujet et sur notre organisa-

tion, mais cela nous a aussi donné plus de responsabilités. Nous avons dû faire des recherches sur comment Unity fonctionne et comment le faire fonctionner comme nous le souhaitions car notre utilité pour Unity était très différente de la création d'un jeu. Il y avait des parties qui s'apparentaient à la création d'un jeu mais d'autres parties étaient très différentes, nous avons donc dû nous inspirer de fonctions faites pour des jeux en les transformant pour obtenir les résultats que nous attendions. Nous avons dû faire beaucoup de recherche sur les fonctions propres à Unity, dont certaines n'ont pas donné les résultats attendus, majoritairement parce que nous n'avions pas totalement compris les explications sur la doc de Unity.

Pendant le projet, je passais souvent de l'enthousiasme à l'angoisse. Je suis souvent passée d'un état où je pensais que notre projet était un très bonne idée à l'idée totalement inverse. Plus d'une fois nous avons remis en question notre choix de sujet pour ce projet, pensant que nous aurions dû faire un jeu. Malgré ces questionnements, nous sommes tous arrivés au bout. Bien qu'un jeu soit un projet attrayant, je suis tout de même contente d'avoir fait un logiciel d'animation parce que nous avons dû réfléchir à comment détourner les fonctions de Unity pour qu'elles soient à notre avantage et nous avons donc dû être créatif.

La seule chose qui m'a déçue dans ce projet est le fait que nous n'ayons pas réussi à faire tout ce que nous voulions. Ceci est en partie dû au fait que nous avons été trop ambitieux et nous avons un peu surestimé nos capacités par rapport au temps que nous avions.

Notre projet était un défi intéressant pour nous tous, d'autant plus que certains d'entre nous ne codaient pas régulièrement avant de venir à EPITA, et aucun de nous n'avait fait de C# ou n'avait essayé d'utiliser Unity avant cette année. A aucun moment nous ne nous sommes disputés sérieusement. Même lorsque nos idées concernant le projet divergeaient, tout s'est bien passé, nous avons eu quelques problèmes de communication mais rien qui ait entravé l'avancement du projet. Je pense que nous nous en sommes bien sorti, nous n'avons pas réussi à faire tout ce que nous voulions mais nous avons fait ce que nous pouvions pour réaliser le logiciel que nous imaginions, et nous avons tous survécu jusqu'à la fin.

Julie

Le projet The Frame Workshop fut une toute nouvelle expérience. A l'exception du TPE, qui ne demandait pas de travail de création, je n'avais jamais participé à un projet de code avant cette année. Je possède quelques bases en animation 3D, c'est pourquoi je trouvais cela intéressant de comprendre le fonctionnement d'un logiciel d'animation. Dans un premier temps, j'appréhendais

ce projet. C'est en effet un travail complètement différent de ce que l'on nous demande en temps normal. Nous avions beaucoup plus de temps et de liberté que pour les autres travaux de l'école. Nous devions gérer la quantité de travail tout en s'organisant en équipe.

Dès le départ, ce projet était un challenge et nous nous demandions parfois s'il était réalisable. En première année, les étudiant choisissent généralement de développer un jeu vidéo. C'est pourquoi nous voulions trouver une idée originale de projet et avons finalement opté pour un logiciel d'animation. D'une part, aucun des membres du groupe n'avais contribué à un projet de programmation aussi important. D'autre part, la plus grande difficulté de notre travail venait du fait que nous n'avions pas choisi un jeu. En effet, Unity est spécifiquement adapté à la création de jeu vidéo et il était parfois complexe de rendre nos idées réalisables. De plus, la plupart des tutoriels et vidéo explicatives, en ligne, sont dédiés aux jeux vidéos, ce qui offrait une aide limitée. Il nous fallait parfois être créatifs afin de réaliser le logiciel que nous avions imaginé. Nous nous demandions souvent ce qui nous a poussé à ne pas développer un jeu vidéo, après tout.

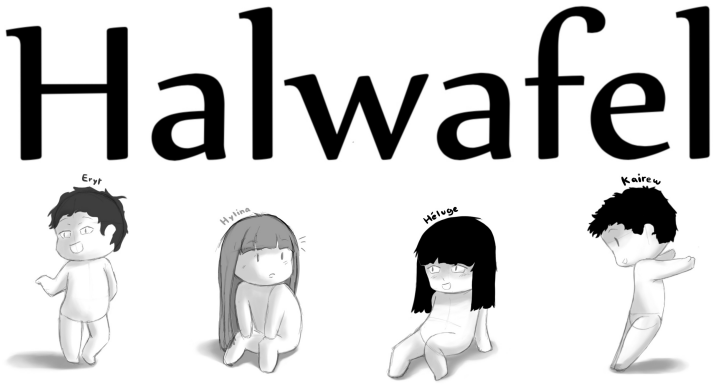
A propos du travail d'équipe, en revanche, nous n'avons pas rencontré beaucoup de difficultés. Tous les membres du groupe étaient très investis dans leur travail. Je suis ainsi très contente de faire partie de ce groupe. Nous avions divisé le travail en plusieurs parties qui correspondaient aux différentes fonctionnalités du logiciel. Certaines personnes s'occupaient de l'interface pendant que d'autres implémentaient le mouvement, par exemple. Ainsi, nous travaillions tous sur des parties différentes ce qui a permis d'éviter toute contradiction sur le code.

Ainsi, le choix d'un logiciel d'animation n'était pas une si mauvaise idée. Cela était possiblement plus difficile lors de l'implémentation mais cela nous a poussé à apprendre nous-même le fonctionnement de Unity. Nous essayions différentes manières d'implémenter nos idées. Nous avons finalement réussi à créer un logiciel qui correspondait à notre idée de départ.

Conclusion

Le but que nous nous étions fixé pour ce projet était de créer un programme d'animation 3D en utilisant le moteur de Unity pour nous aider. Au cours de ces trois derniers mois, nous avons eu de grand succès, comme nous avons rencontré des moments d'adversité. Malgré ces hauts et ces bas, nous pouvons enfin conclure que nous avons accompli notre but. Nous avons créé un programme d'animation 3D.

Le chemin semé d'embûches qui nous a amené jusqu'ici nous a fait traverser des épreuves notamment avec la génération automatique du squelette, que nous avons malheureusement dû abandonner car il causait trop de problèmes à notre projet. Ce même chemin nous a mené vers de nombreux succès : un mouvement fonctionnel et une interface utilisateur dénuée de bugs. Notre détermination et notre volonté de bien faire nous ont permis de surmonter les obstacles.



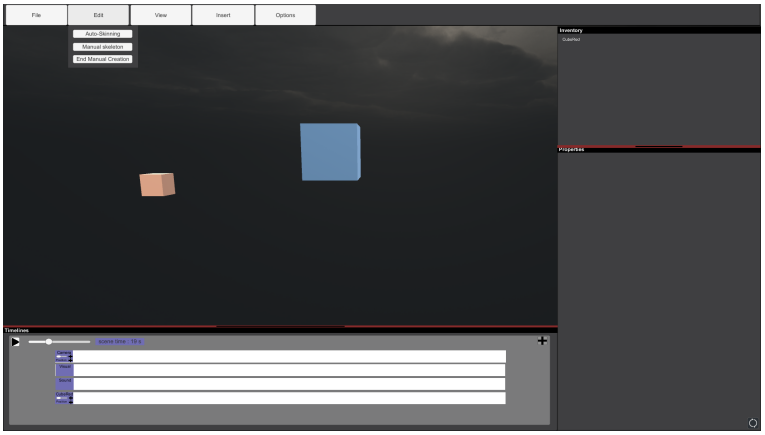
Halwafel n'est pas un groupe, Halwafel n'est pas un projet ;
Halwafel est un art de vivre

Merci à chaque parent, ami et animal de compagnie pour leur soutien moral.

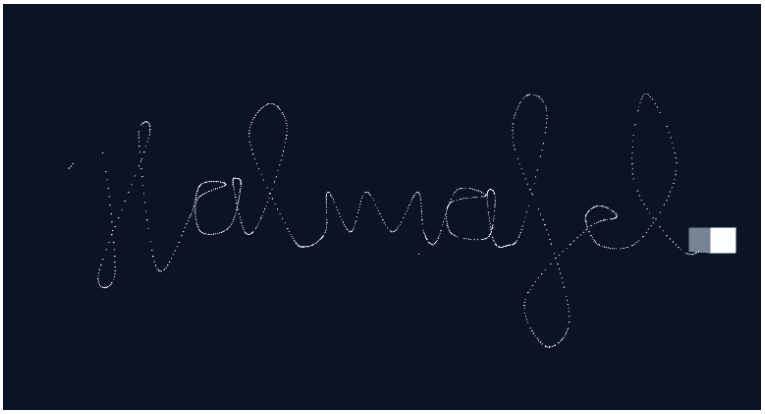
Annexes

The Frame Workshop

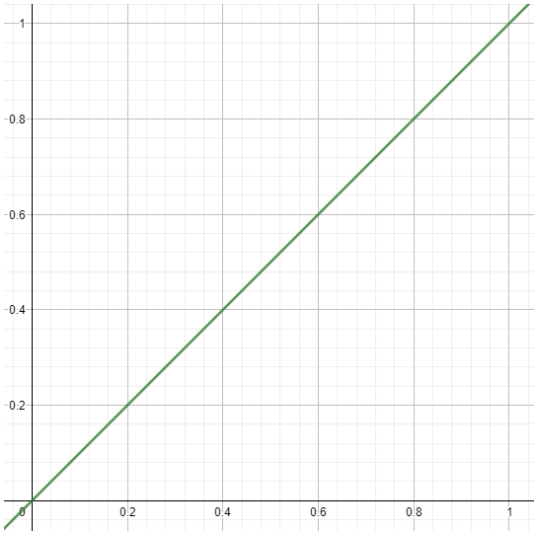
Annexe 1



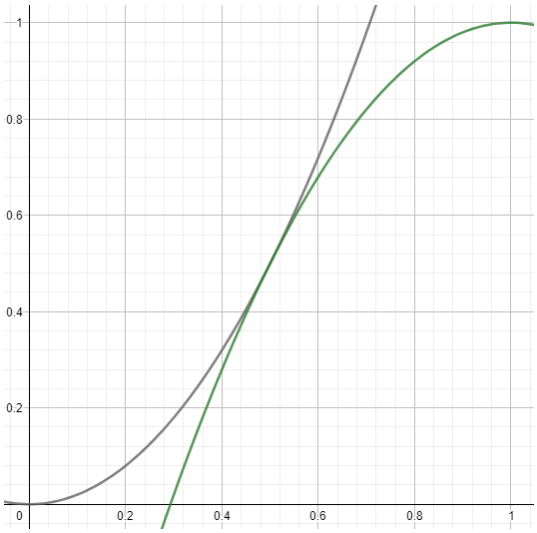
Annexe 2



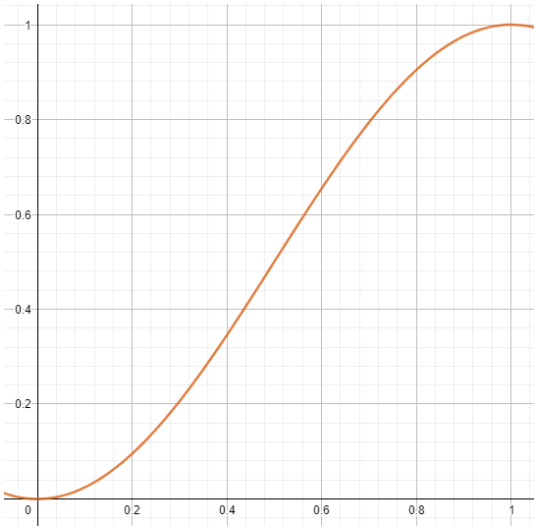
Annexe 3



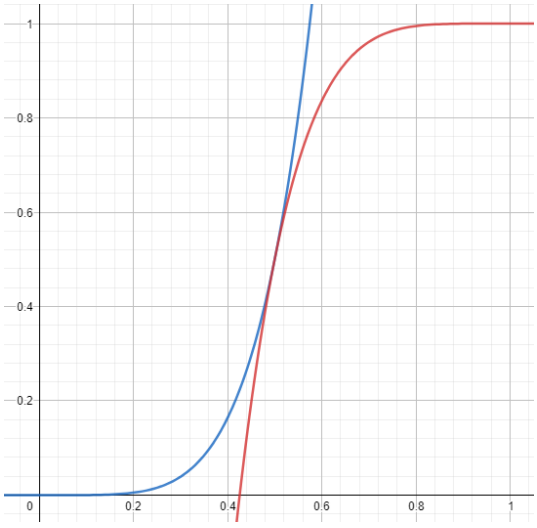
Annexe 4



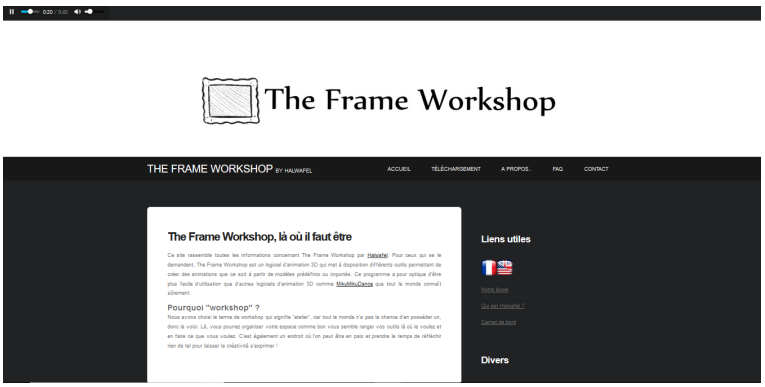
Annexe 5



Annexe 6



Annexe 7



Annexe 8

Webographie :

— answers.unity.com

— autodesk.eu

— blender.org

— docs.unity3d.com

— ffmpeg.org

— forum.unity.org

— github.com

— learnmmd.com

— msdn.microsoft.com

— pixelplacement.com

— stackoverflow.com

— unity3d.com

— wikipedia.com

— youtube.com