Rapport de projet TER 2023-2024

Techniques avancées de moteur de jeux

Luna Bossu, Arthur Chateauneuf, Luca Chaudillon, Valentin Noyé

Encadrante : Noura Faraj

Master 1 Imagine

UNIVERSITÉ DE MONTPELLIER

Résumé

Lorsque les premiers jeux-vidéos sont arrivés sur le marché, les limitations techniques de places et de performances des machines obligeaient les développeurs à reprendre la programmation des divers modules à partir de zéro à chaque nouveau projet. Chaque ligne de code devait être placée dans une structure logique avec le reste du projet.

Avec l'évolution de la complexité des machines, les jeux-vidéos sont devenus de plus en plus complexes et lourds. Lors de l'arrivée de la 3D, un tout nouveau paradigme de rendu et de logique a dû être inventé, et les projets nécessitaient désormais des équipes de plusieurs dizaines de personnes. Le champ des possibles a continué sa rapide expansion en englobant toujours de plus en plus de nouvelles méthodes et évolutions.

C'est ainsi que le concept de moteur de jeu a émergé. Un moteur de jeu est un logiciel permettant de créer un jeu vidéo interactif à partir de bases prédéfinies. Il incorpore également une multitude d'outils permettant de simplifier le développement et de réutiliser un maximum de fonctionnalités redondantes. Il permet ainsi de gérer l'affichage, la physique, la logique de jeu, la logique d'agent, les entrées utilisateurs, les sorties, ou même la mise en réseau de l'application avec d'autres machines.

Ce projet vise ainsi a mettre en place des améliorations ainsi que des fonctionnalités supplémentaires à un moteur de jeu préexistant.

Remerciements

Nous tenons à exprimer notre sincère gratitude à Mme. Faraj pour son encadrement et les enseignements qu'elle nous a apporté tout au long de cette année. Nous remercions également M. Lafourcade pour sa gestion et son organisation des projets d'étude et de recherche de cette année.

Table des matières

Table des figures

1 Introduction

1.1 Catégories de moteurs de jeu

En 2024, il existe deux catégories de moteurs : ceux dits grand public comme Unity, Unreal ou Godot et ceux plus spécialisés. Les premiers sont destinés à couvrir un maximum de fonctionnalités, afin que n'importe quel type de jeu vidéo puisse être développés dessus. Les derniers, quant à deux, sont maintenus le plus souvent en interne par un studio de développement afin de créer un type de jeu souvent précis (example RE engine développé par Capcom qui sert à faire des jeux 3D photoréalistes 3D à grosse échelle de production).

1.2 Base de code et organisation

Nous travaillons sur Vulpine Engine, un moteur préexistant créé et maintenu par Arthur Chateauneuf. Ce logiciel est entièrement écrit en C++ pour la partie CPU et en GLSL pour la partie GPU. L'API d'OpenGL est utilisée pour le système de rendus. Toutes les librairies utilisées sont trouvables sur le page GitHub du projet.

Voici une liste non exhaustive des fonctionnalités qui étaient présentes sur le moteur lorsque nous avons commencé notre projet :

- Rendu de scène 3D
- Hiérarchie de scène
- Système de rendu Forward
- Support multi-éclairage
- Chargement de modèles 3D, de textures et sons
- Intégration OpenAL (moteur audio)
- Rendu en mode profondeur uniquement
- Ombres et ombres douces
- Passe de rendu
- Divers effets post-traitement : correction de couleur en plage dynamique élevé, occlusion ambiante en espace écran, flou lumineux...
- Gestion des tampons de rendu
- Gestion de prétraitement et de compilation des shaders (programmes GPU écrits en GLSL)
- PBR et gestion des réflexions basées images
- Interface simple de débogage

Afin d'organiser le développement et le test en parallèle de plusieurs composants différents du moteur, nous utilisons plusieurs répertoires GitHub. La base de code ayant été créée pour être utilisée en tant que sous module, il nous est plus pratique de pouvoir utiliser des démos de tests différents, tout en travaillant sur la même branche du moteur. Cette organisation s'est avérée grandement avantageuse lors de notre travail.

Pour finir, nous construisons notre projet à l'aide d'un MakeFile. Tous les modules sont compilés en `C++23` avec l'option d'optimisation `-Ofast` de `gdb`. Ce dernier active l'intégralité des optimisations du compilateur, même celles ayant un impact sur la précision des calculs flottants. Pour un moteur de jeu ce compromis est important, car la rapidité prime sur tout. Il n'y a également pas besoin d'une précision scientifique dans les divers calculs effectués. L'important est que le rendu et la prise en main soit agréable pour un observateur humain.

1.3 Introduction de nos objectifs

Notre objectif avec ce projet est d'améliorer le moteur en implémentant dedans une série de techniques avancées. Nous souhaitons couvrir un bon ensemble des pans de la gestion d'un moteur de jeux en testant pour chaque module son efficacité.

Nous souhaitions ainsi implémenter :

- Un système de test et de mesure de métrique
- Un système de rendu dit "Clustered" afin d'accélérer les calculs d'éclairages, section attribuée à Arthur Chateauneuf
- Un système d'animation squelettique complet travaillé par Luna Bossu

- Un système de recherche de chemin et de génération de graphe de monde, partie affectée à Luca Chaudillon
- Un moteur physique complet et optimisé attribué à Valentin Noyé

Chacun de ces modules améliore une partie différente du moteur. Nous souhaitions également créer un court jeu de démonstration de ces fonctionnalités. Cependant, nous n'avons pas eu le temps de le finir à temps pour la période de rendu.

1.4 Lien de nos travaux et exécutables

Comme précédemment expliqué, nous travaillons sur plusieurs répertoires différents afin de tester les fonctionnalités du moteur. L'intégralité des modules développés pour le moteur sont présents sur le répertoire GitHub de ce dernier. Chaque répertoire de test possède une "release" contenant un exécutable ainsi que l'ensemble des fichiers nécessaires. Voici tous les liens importants :

- Vulpine Engine
- Répertoire de test du module Graphique
- Répertoire de test du module Animation
- Répertoire de test du module Physique

Note importante : Le moteur Vulpine Requiert la compatibilité avec OpenGL 4.6. Nous recommandons fortement de lancer les exécutables sur des machines puissantes possédant un GPU non intégré au CPU.

2 Clustered Rendering

2.1 Concept

Afin d'afficher sur un écran 2D une scène 3D en temps réel, le moteur Vulpine utilise l'API OpenGL. Le processus de rendu utilisé dans OpenGL commence par envoyé du CPU vers le GPU toutes les informations de la scène 3D comme les géométries, les textures, la liste des lumières, etc. Dans le passe de rendu principal, les géométries envoyés au GPU passent par une étape de vertex shader, où les modèles sont projetés à l'écran à l'aide de matrices de transformations. Pour finir, chaque primitive va être remplie de pixels en exécutant le fragment shader. Dans un tel système, le nombre de fragments généré est une partie très importante du volume de calcul et du temps nécessaire à l'affichage d'une scène 3D.

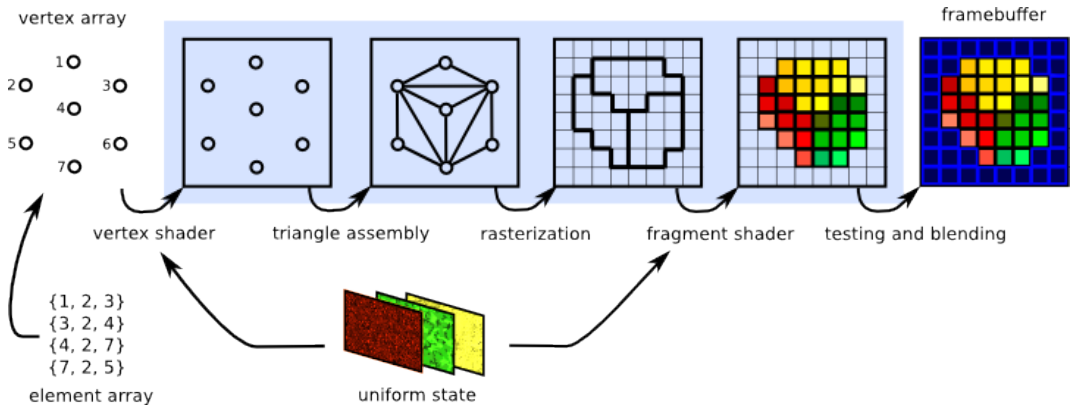


FIGURE 1 – Illustration simplifiée de le pipeline graphique d'OpenGL[1]

Le rendu 3D en temps réel utilisant la rastérisation est très puissant, mais pose certaines problématiques quant à l'expansion et la complexification des scènes que l'on souhaite rendre. L'un des éléments les plus couteux en performance graphique reste les calculs complexes de lumières et d'ombrages. Dans un système basique comme le Forward Rending, le nombre de calculs augmente de façon proportionnelle au nombre de lumières dans la scène et monte vite. Dans ce système de rendu, pour chaque pixel calculé, l'intégralité des lumières de la scène sont testés afin d'obtenir le résultat final.

De nombreuses améliorations ont été proposées au fil du temps, comme le Deffered Rendering ou le Tile-Based Rendering, mais nous souhaitons nous intéresser au système proposant les meilleurs résultats et le moins de limitations, tel que le Forward Clustered Rendering[2] , souvent appelé Forward +.

	Forward	**Deffered**	**Forward Tiled**	**Deffered Tiled**	**Deffered Clustered**	**Forward Clustered**
Apparition	1996	2001	2005	2005	2010	2010
Objets transparents	oui	non	oui	non	non	oui
MSAA	oui	non	oui	non	non	oui
Multi-matériaux	oui	non	oui	non	non	oui
Rapidité	-	+	+	++	++++	+++

FIGURE 2 – Comparaison des algorithmes de rendus 3D[3]

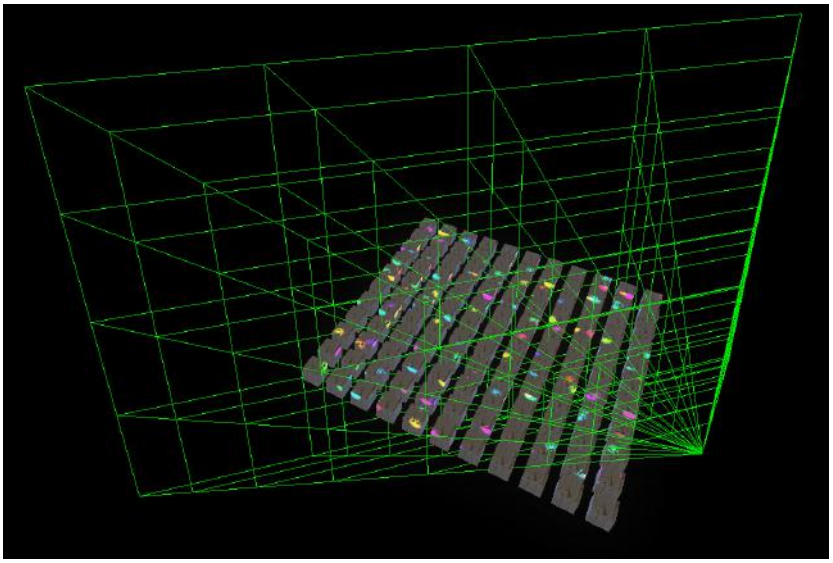


FIGURE 3 – Visualisation d'une segmentation de l'espace caméra

Le Clustered Rendering consiste à segmenter l'espace de la caméra, appelé frustum, en une grille pyramidale 3D (cf. figure 3). Un pré-passe est alors calculé avant chaque rendu d'image afin de tester l'intégralité des intersections entre lumières et cellules de frustum. Lors du rendu des fragments, le shader n'itère ainsi que sur la liste des lumières dont l'intersection avec sa grille a été confirmée. Cette approche, de plus en plus utilisé dans les productions vidéoludiques, permet de réduire significativement le nombre de calculs d'éclairages réalisés.

Notre objectif pour optimiser le moteur de rendu de Vulpine, qui est basé sur un système Forward simple, est d'y ajouter une option d'activation pour le Clustered Forward.

2.2 Implémentation

Afin d'implémenter le Clustered Rendering, nous devons découper le frustum en trouvant des équations de changement de coordonées entre l'espace de vue et l'espace de grille de frustum. L'article sur lequel nous nous sommes basés[2] utilise une caméra à projection simple, utilisant un plan proche ainsi qu'un plan lointain pour déterminer l'espace totale de vue. Cependant, le moteur Vulpine utilise une projection alternative à cela : la profondeur inversée. Afin de créer une matrice de projection à profondeur inversée, nous faisons :

$$f = \frac{1}{tan(\frac{1}{2}focal_{angle})}$$

$$a = \frac{Res_y}{Res_x}$$

$$projection = \begin{bmatrix} fa & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & Z_n & 0 \end{bmatrix}$$

Cette approche a de nombreux avantages, elle permet d'obtenir une bien meilleure précision proche d'une profondeur classique, car le fonctionnement des chiffres flottants binaires y est bien mieux utilisé. De plus, aucune profondeur maximale n'est utilisée, rendant la distance maximale d'affichage théoriquement infinie.

Ne pouvant pas utiliser les équations données dans notre article de référence[2], nous avons créé les nôtres. Cependant, nous n'avons pas réussi à trouver d'équations à découpage de la profondeur exponentielle, qui est le meilleur type de découpage que le papier de référence a trouvé. De plus, le découpage de la grille ne pouvant être infini, nous proposons le concept de plan lointain virtuel. Bien

qu'une géométrie puisse être rendue à une distance quelconque de la caméra, aucune lumière ne sera rendue au-delà du plan lointain virtuel.

Pour connaître la position d'une cellule dans l'espace de vue, nous utilisons l'équation suivante, qui contient la distance du plan proche Z_n, la distance du plan lointain virtuel Z_f, la dimension 3D de la grille de frustum $Cdim_{xyz}$ ainsi que la coordonnée dans la grille de la cellule C_{xyz} :

$$screen_{xyz} = \left[2\frac{C_x}{Cdim_x} - 1 \qquad 2\frac{C_y}{Cdim_y}] - 1 \qquad \frac{Z_n}{Z_f}\frac{1}{2}\frac{Cdim_z}{C_z}\right]$$

$$view_{xyz} = \left[screen_x \frac{Z_n}{fa} \qquad 2\frac{Z_n}{f} \qquad -Z_n\right] \frac{1}{screen_z}$$

Nous passons ainsi de l'espace de la grille à l'espace NDC (normalized device coordinates), puis de l'espace NDC, nous effectuons la transformée inverse de la projection à l'écran. Nous pouvons ainsi, pour chaque cellule, générer deux arêtes opposées afin d'obtenir un AABB (axis aligned bonding box). Effectuer le tri des lumières dans l'espace de vue permet de prendre avantage de la rapidité des calculs de collisions AABB-sphères (nos lumières étant représentées par des sphères).

Une fois les AABB de chaque cellule générée, les positions des lumières sont converties en espace vue à l'aide de la matrice de vue. Le programme dispose ensuite de toutes les informations nécessaires au tri des lumières. Un tableau 3D contigu est alors rempli par le CPU et sera envoyé au GPU sous la forme d'un SSBO (Shader Storage Buffer Object).

Lors du calcul de lumière, chaque fragment retrouvera sa position sur la grille à partir de sa position NDC et itérera sur la liste prédéfinie des lumières intersectant sa cellule. Aucun autre aspect du calcul d'éclairage n'est changé, rendant le Clustered Rendering simple à adapter pour le système de rendu que le moteur utilise déjà.

2.3 Optimisation

L'une des étapes la plus coûteuse dans le Clustered Rendering est le tri des lumières, car il faut effectuer des calculs de collisions entre toutes les cellules et les sources lumineuses. Notre implémentation, par exemple, utilise une grille de frustum de dimension 16x9x24, ce qui représente 3456 itérations, multiplié par le nombre total de lumières. Le papier que nous avons utilisé en référence s'aide d'un compute shader afin d'utiliser la puissance de parallélisation du GPU afin d'accélérer ce processus.

Nous proposons une autre approche, axée CPU, qui a pour objectif de réduire au maximum le nombre d'itérations. L'idée est d'utiliser les formules de projections et de conversions que nous avons construit afin de déterminer un intervalle de collision potentielle pour chaque dimension. Pour ce faire, nous considérons les points de la sphère les plus proches et lointains de la caméra, du plan haut ainsi que du plan droit. Ce processus correspond à six projections peu coûteuses qui permettent de réduire significativement le nombre d'itérations de collisions. Le nombre d'itérations n'étant pas connu à l'avance, cet algorithme est moins parallélisable, mais réduit significativement le volume opérationnel du tri.

Notre moteur possède ainsi deux versions du tri de Clustered Rendering, toutes les deux sur CPU : une version dite naïve où toutes les itérations sont faites ainsi qu'une version optimisée utilisant des projections pour réduire le nombre d'opérations.

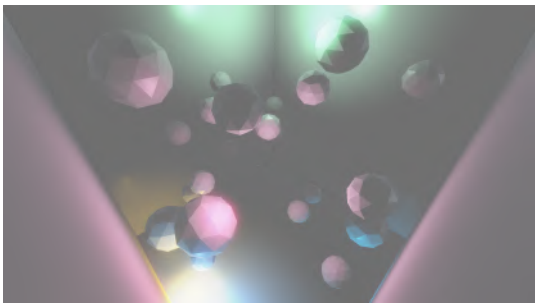
2.4 Résultats

Afin de tester notre implémentation, nous avons préparé une scène de test contenue en un seul objet 3D, sans corps physique ni logique de jeu. De ce fait, la quasi-intégralité du temps de génération d'une image est pris par la génération des fragments. Nous avons préparé une liste de plusieurs ensembles de lumières de différentes tailles.

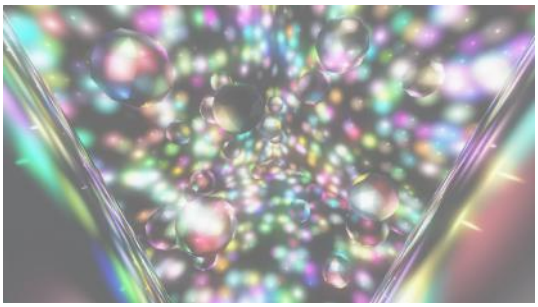
Notre implémentation n'entraine aucune différence de temps de rendu graphique entre des lumières statiques et dynamiques, seule la logique de déplacement est affectée. De ce fait, l'intégralité des résultats qui seront présentés utilise des lumières statiques, afin que le temps de génération des positions ne soit pas pris en compte, car non pertinent dans la mesure.

Pour finir, nous avons décidé de commencer nos mesures approximativement deux secondes après le lancement de l'application. Cette décision a été prise, car lorsqu'un logiciel coûteux en ressource GPU est lancé, le système d'exploitation va enclencher une série de routines afin d'augmenter la cadence du

CPU et du GPU. Les performances juste après le lancement de l'application sont ainsi aléatoires et non représentatives de la suite de l'exécution. Un temps d'attente de deux secondes est suffisant pour que ce processus soit entièrement terminé.

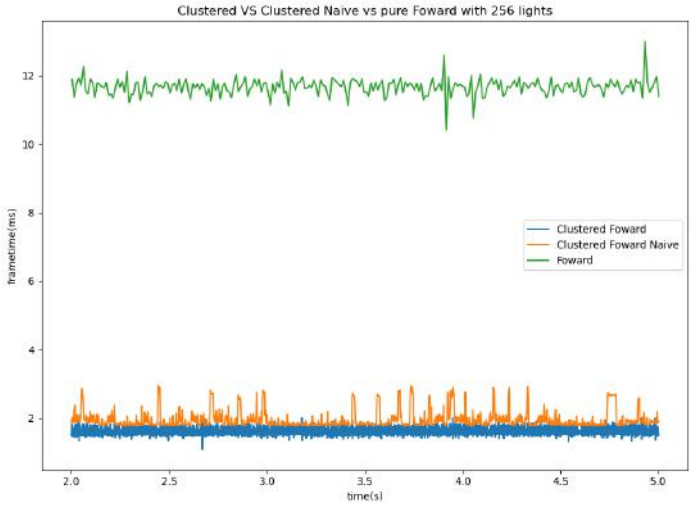


(a) 5 lumières

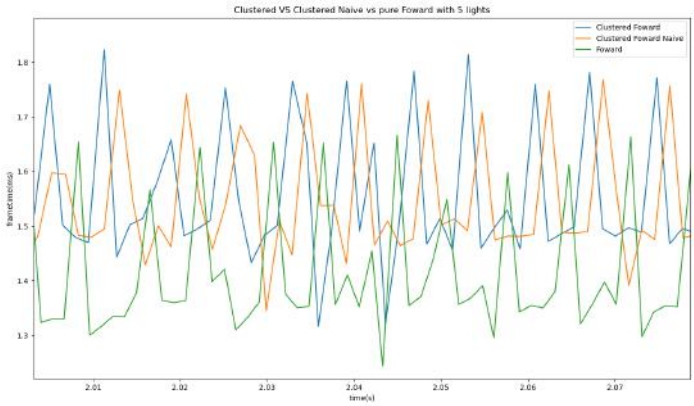


(b) 8192 lumières

FIGURE 4 – Captures d'écran de la scène utilisée pour tester les algorithmes de rendu



(a) 256 lumières



(b) 5 lumières

FIGURE 5 – Temps de génération du rendu entre nos algorithmes

La figure 5 montre une très forte amélioration des performances pour la scène a. Un rendu basé Cluster ne prend qu'une fraction du temps nécessaire à un système plus classique pour une scène contenant un grand nombre de lumières. Lorsque le nombre de sources diminue, les algorithmes deviennent

équivalents. Lorsque nous chargeons des scènes contenant plus de 2048 lumières en utilisant le Forward simple, le temps de rendu devient trop grand pour une utilisation temps réel (moins de 15 images par secondes).

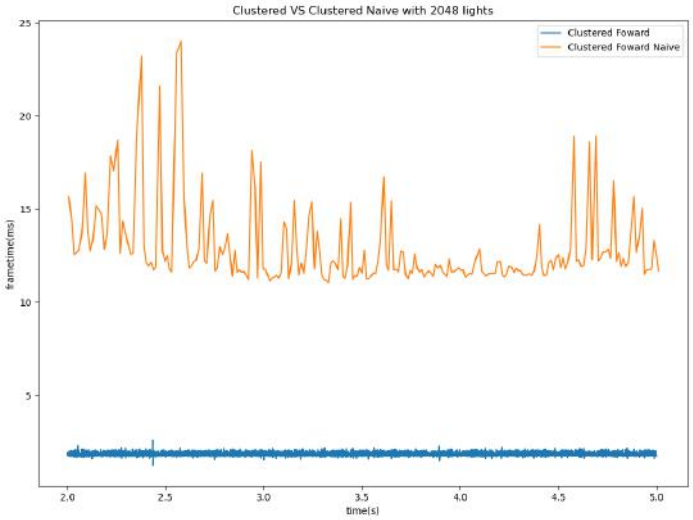


FIGURE 6 – Temps de génération du rendu entre nos algorithmes Clustered

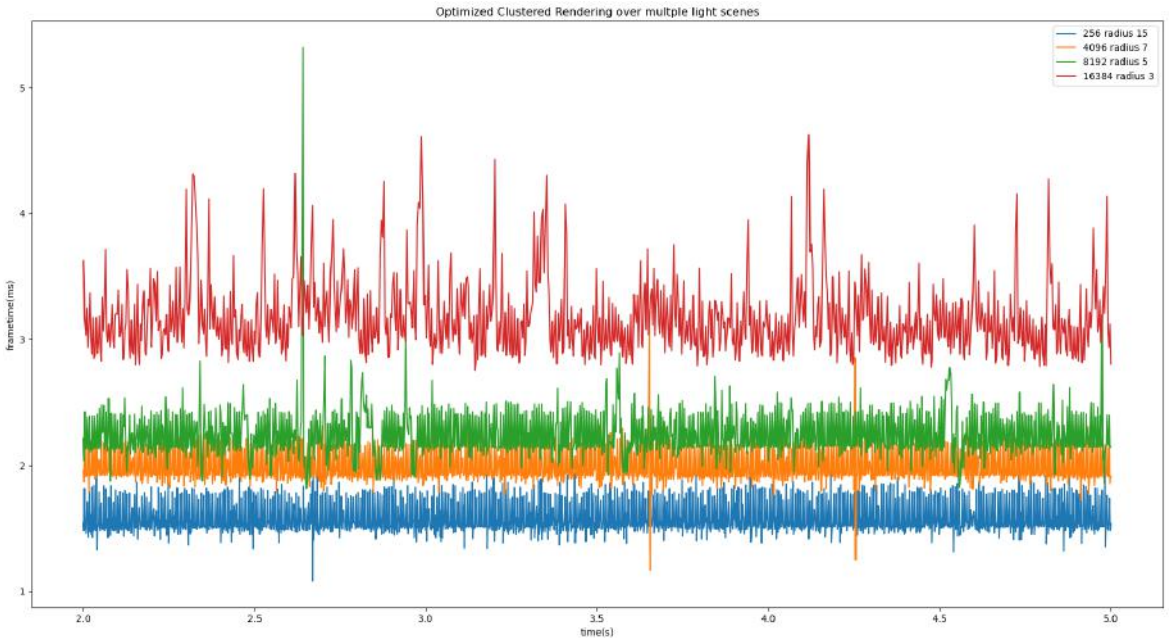


FIGURE 7 – Temps de génération du rendu avec le Clustered Forward sur différentes scènes

La figure 6 montre l'efficacité de l'algorithme optimisé. La réduction du temps de tri réduit significativement le temps global de rendu d'une image. Nous pensons que notre algorithme CPU est suffisant pour n'importe quelle application utilisant quelques milliers de lumières ou moins.

La figure 7 montre que même avec notre scène la plus dense (8192 lumières de 5 mètres de rayon) le moteur est capable de générer approximativement 290 images par secondes.

2.5 Documentation de la partie Clustered Rendering

2.5.1 CPU (C++)

Classe ClusteredLightBuffer

- constructeur `ClusteredLightBuffer::ClusteredLightBuffer()` constructeur vide.
- destructeur `ClusteredLightBuffer::~ClusteredLightBuffer()` libère la mémoire GPU en supprimant le SSBO lié s'il existe. Le buffer côté CPU est libéré automatiquement.
- méthode `ClusteredLightBuffer::allocate(`dim`)` génère le SSBO et alloue l'espace suffisant dans un buffer intelligent.
- méthode `ClusteredLightBuffer::send()` envoie l'intégralité du buffer au GPU par le biais du SSBO tout en allouant l'espace suffisant côté GPU.
- méthode `ClusteredLightBuffer::update()` envoie l'intégralité du buffer au GPU, en supposant que l'espace a déjà été alloué.
- méthode `ClusteredLightBuffer::activate(`$location$`)` active le SSBO à la position donnée. La position correspond à une position de liaison avec les shaders. Dans Vulpine la position de liaison de la grille de frustum est 1.

Classe Scene Contient l'ensemble des éléments permettant de gérer et d'afficher un graphe de scène contenant des groupes d'éléments, des géométries, des lumières ou des matrices de modèles (utilisées par exemple pour l'instantiation). L'utilisation conseillée de cette classe est d'avoir deux scènes, une 3D et une 2D. De ce fait, elles peuvent chacune avoir leur propre paramétrage (les options avancées d'éclairages ne seront par exemple jamais activé dans la scène 2D). Cette classe étant très large, nous ne montrerons que les méthodes utiles à l'activation et à l'utilisation du Clustered Rendering.

- méthode `Scene::activateClusteredLighting(`$dimention, vfar$`)` alloue le ClusteredLightBuffer et initialise ses valeurs avec les paramètres de la dimension et de la distance du plan lointain virtuel.
- méthode `Scene::deactivateClusteredLighting()` désactive l'utilisation du Clustered Rendering. L'intégralité des matériaux 3D devront être recompilés, et les options de pré-traitement des shaders devront également être changés afin de refléter ce changement. Dans l'organisation par défaut des matériaux de Vulpine, l'activation et la désactivation globale du Clustered Rendering est définie par l'ajout ou la suppression de `"#define USE_CLUSTERED_RENDERING\n"` dans la chaîne de caractère globale `Shadinclude::shaderDefines`.
- méthode `Scene::genLightBuffer()` lie les cartes d'ombres de toutes les lumières en possédant, met à jour la liste des lumières de la scène et l'envoie au GPU. Si le Clustered Rendering est activé pour la scène, appel generateLightClusters() et met à jour le buffer de la grille sur le GPU.
- méthode `Scene::generateLightClusters()` génère la liste des AABB de grilles si elle n'existe pas déjà. Effectue ensuite le trie de toutes les lumières de la scène. L'utilisation du tri optimisé est activée par l'existence du composant de pré-traitement
  `#define PER_LIGHT_OPTIMIZED_CLUSTERED_RENDERING`

Classe App Définie un vaste ensemble de fonctionnalités permettant de gérer une fenêtre d'application graphique interactive. Elle contient la méthode `App::activateMainSceneClusteredLighting(`$dimention, vfar$`)` permettant d'activer le Clustered Rendering sur la scène 3D de l'application et de définir automatiquement l'option de pré-traitement globale des shaders activant cette fonctionnalité.

2.5.2 GPU (GLSL)

Positions de liaisons shaders définies et utilisées par le moteur Vulpine. Ces positions permettent de pouvoir organiser la liaison CPU-GPU des informations telles que les données uniformes (définissable avant chaque lancement d'un passe de rendu), de l'agencement des attributs de sommets pour les géométries, de l'agencement des buffers SSBO ainsi que de l'agencement des diverses textures utilisables dans le rendu.

Position	Type	Name	TAGS
0	ivec2		
1	float	_iTime	
2	mat4	_cameraMatrix	
3	mat4	_cameraViewMatrix	
4	mat4	_cameraProjectionMatrix	
5	vec3	_cameraPosition	
6	vec3	_cameraDirection	
7	mat4	_cameraInverseViewMatrix	
8			
9			
10			
11	vec4	lodHeightDispFactors	USING_LOD_TESSELATION
12	vec4	lodTessLevelDistance	USING_LOD_TESSELATION
13	float	vFarLighting	USE_CLUSTERED_RENDERING
14	ivec3	frustumClusterDim	USE_CLUSTERED_RENDERING
15	vec3	ambientLight	
16	mat4	_modelMatrix	
17			
18			
19			
20	vec3	bColor	used in basic.frag
20-...	sampler2D	bColor/bMaterial	same as models textures, but bindless
...			
24/32	-	safe user defined uniforms	

FIGURE 8 – Positions de liaisons des données uniformes

Position	Type	Name	TAGS
0	vec3	_positionInModel	
1	vec3	_normal	
2	vec3	_color	!USING_VERTEX_TEXTURE_UV
2	vec2	_uv	USING_VERTEX_TEXTURE_UV
3-4	mat4	_instanceMatrix	USING_INSTANCING
5	ivec4	_weightsID	USE_SKINNING
6	vec4	_weights	USE_SKINNING

FIGURE 9 – Positions de liaisons des attributs de sommets

Position	Name	TAGS
0	lightsBuffer	
1	lightsClustersBuffer	USE_CLUSTERED_RENDERING
2	animationStateBuffer	USE_SKINNING

FIGURE 10 – Positions de liaisons des SSBO

Position	Name	TAGS
1	bColor	
2	bMaterial	
3	bHeight	USING_LOD_TESSELATION
3	bDisp	USING_LOD_TESSELATION
4	bSkyTexture	
16-32	bShadowMaps[]	

FIGURE 11 – Positions de liaisons des textures

Le module functions/MultiLight.glsl

- La fonction `getMultiLight()` renvoie le résultat d'éclairage du fragment en itérant sur toutes les lumières disponibles. Cette fonction possède deux définitions, une utilisant le Clustered Rendering et une ne l'utilisant pas.
- La fonction `getClusterId(`$ivFar, steps$`)` renvoie automatiquement la position 3D du fragment actuel dans la grille de frustum définie.

3 Animations

3.1 Animations Squelettiques

L'animation squelettique est une méthode d'animation qui consiste à déplacer certaines parties d'un modèle 3D en fonction de la position d'os qui sont arrangés comme un graphe appelé un squelette. Chaque os possède une transformation tridimensionnelle (position, rotation et homothétie).

Par un processus appelé "skinning" chaque sommet d'un modèle 3D se voit attribué une liste d'os qui affecteront sa position[4]. La relation de la transformation entre un os et un sommet dépend de la position de ce dernier dans la pose originale du modèle, appelée "bind-pose".

Chaque os peut éventuellement avoir un os parent qui affecte sa position et rotation (par exemple, la position et rotation de l'os du tibia dépend de la position et rotation de l'os du fémur)[4].

Il est important de noter que ces os ne représentent pas exactement un squelette, mais plutôt une approximation qui permet à l'animateur de facilement contrôler les mouvements du modèle 3D.

Cette technique est particulièrement appropriée pour représenter l'animation de personnages humains ou animaux et est utilisée dans virtuellement tous les systèmes d'animation 3D.

En pratique, un animateur créant une animation squelettique définie les mouvements du squelette à l'aide de "poses clés". Ces poses clés représentent l'état d'un squelette à un instant T. Une interpolation linéaire entre deux poses clés permet ensuite de fluidifier le mouvement[5].

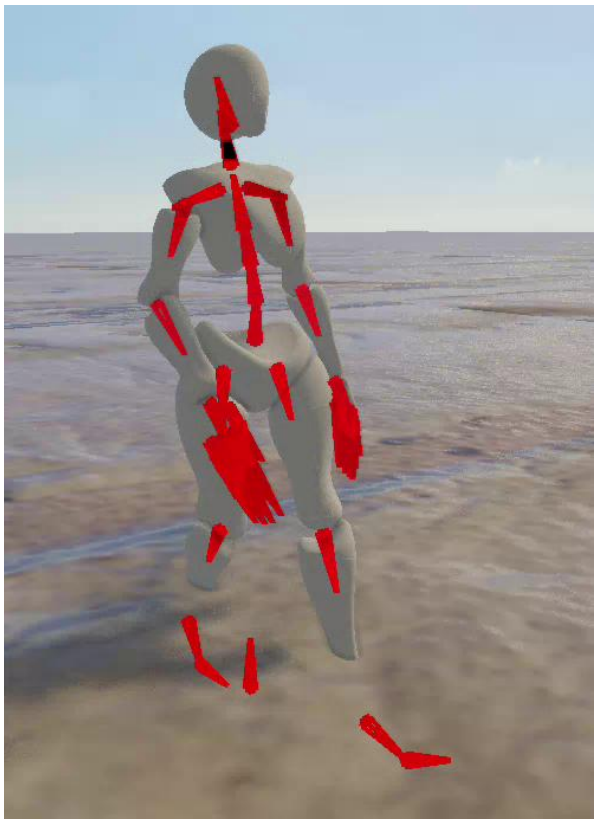
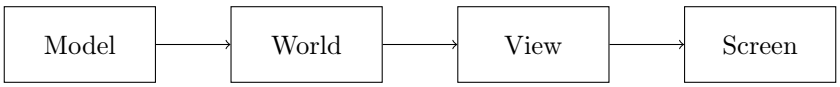


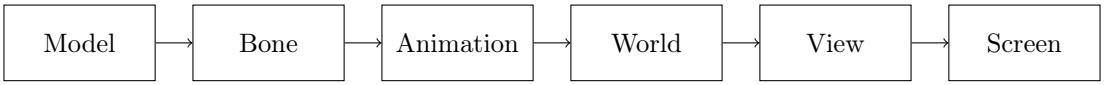
FIGURE 12 – Squelette (en rouge) sur un modèle de personnage

3.2 Implémentation et format de conversion

Dans le moteur Vulpine, les géométries sont envoyées du CPU au GPU sous la forme de maillages dans l'espace modèle. Un vertex shader sera alors exécuté sur chaque sommet et renverra leur position à l'écran sous forme NDC. Voici la série de transformations réalisée durant cette étape, chacune est représentée par une matrice 4x4 :



Pour un maillage utilisant l'animation squelettique, le processus change et de nouvelles étapes sont ajoutées :



La matrice de transformation d'os est générée en additionnant la transformation de la même position modèle pour chaque os affectant le sommet. Dans notre implémentation, le nombre d'os maximal sur un seul sommet est de 4.

La matrice d'animation est générée en calculant l'interpolation entre deux poses clés de l'animation actuelle. Notre contrôleur d'animation permet d'ajouter une étape supplémentaire en interpolant entre deux animations. Ce système est représenté sous la forme d'un automate déterministe à état fini que l'utilisateur peut configurer comme il le souhaite.

Il est ainsi possible d'utiliser des fonctions de rappel afin de configurer les conditions de transition entre deux animations, la vitesse et les routines de chaque animation ou de paramétrer l'animation en activant ou désactivant la répétition par exemple.

Afin de charger les animations dans notre moteur, nous avons créé notre propre format. Ce dernier contient un chiffre magique permettant de facilement repérer si le type est valide ainsi que la liste des poses clés pour chaque os. Nous avons également notre propre format de squelette, qui contient une liste contiguë de tous les os avec leur transformation et la liste de leurs enfants.

Pour finir, nous avons créé un logiciel de conversion utilisant la librairie ASSIMP afin de transformer des animations contenues dans n'importe quel type de fichiers en fichiers compatible avec le moteur. Cette approche est largement utilisée dans les applications vidéoludiques afin de pouvoir contrôler précisément le rapport poids / vitesse de chargement de chaque type de fichier.

Offset	0	1	2	3
Octet				
	En-tête du fichier			
0	Nombre magique			
4	Nom de l'animation			
⋮				
128				
132	durée de l'animation (en secondes) : f32			
136	nombre total d'os : u32			
140	nombre total d'os animés : u32			
	En-tête de chaque os (répété pour chaque os)			
0	id de l'os : u32			
4	nombre de poses clés : u32			
	données des poses clés			
0	temps de la pose clé (en secondes) : f32			
4	position de la pose clé (vecteur de f32)			
8				
12				
16	rotation de la pose clé (quaternion de f32)			
20				
24				
28				
32	homothétie de la pose clé (vecteur de f32)			
36				
40				

FIGURE 13 – Structure d'un fichier d'animation squelettique

3.3 Documentation de la partie Animation

3.3.1 Animation

Le fichier *Animation.hpp* implémente et définie les classes, structures et fonctions qui servent a réaliser une animation squelettique simple.

Structures et fonctions diverses

- structure `AnimationKeyframeData(`*time*`,` *translation*`,` *rotation*`,` *scale*`)` stocke les informations d'une pose clé pour un os a un temps spécifié.
- fonction `interpolateKeyframes(`$Animation_A$`,` $Animation_B$`,` t_1`,` t_2`,` α`)` calcule et retourne une liste de poses clés pour tout les os en interpolant l'animation A au temps t_1 avec l'animation B au temps t_2 avec le facteur d'interpolation α.

Classe Animation

- constructeur `Animation::Animation(`*name*`,` *length*`,` $keyframes$`,` *onEnterAnimationCallback*`,` *OnExitAnimationCallback*`,` *getSpeedCallback*`)` définit l'animation avec le nom *name* de longueur *length* secondes avec comme poses clés $keyframes$. On définie aussi les fonctions de rappel qui sont appelées lors qu'on entre dans l'animation et lors qu'on en sort. Enfin, on définit une fonction de rappel qui définit la vitesse de l'animation (cela nous permet d'avoir une vitesse qui varie en fonction du temps).
- méthode statique `Animation::load(`$filename$`)` charge l'animation contenue dans le fichier $filename$. Affiche une erreur si l'animation n'est pas valide et retourne un pointeur nul.
- méthode `Animation::getName()` retourne le nom de l'animation.
- méthode `Animation::getLength()` retourne la longueur de l'animation.
- méthode `Animation::getKeyframes()` retourne une liste de toutes les poses clés pour chaque os de l'animation.
- méthode `Animation::getKeyframe(`$boneID$`)` retourne les poses clés pour l'os avec l'identifiant $boneID$
- méthode `Animation::getCurrentFrames(`*time*`)` calcule les poses clés au temps *time* pour chaque os en effectuant une interpolation linéaire des poses clés qui bordent *time*. Cette méthode est optimisée de telle manière a ce que la recherche des poses clés qui bordent *time* n'est effectuée que si nécessaire.
- méthode `Animation::isFinished(`*time*`)` retourne vrai si *time* est supérieur ou égal a la longueur de l'animation, sinon on retourne faux.

3.3.2 Contrôleur d'animation

Le fichier *AnimationController.hpp* implémente et définie les classes et structures nécessaires au système du contrôleur d'animation.

Structures et fonctions diverses

- énum `AnimationControllerTransitionCondition` cette énumération représente une condition de transition, c'est a dire comment une transition se réalise. Il peut prendre une de deux valeurs $COND\_ANIMATION\_FINISHED$ qui représente une transition lorsque l'animation se termine et $COND\_CUSTOM$ qui signifie une condition définie par l'utilisateur grace a une fonction de rapel.
- énum `TransitionType` cette énumération représente le type de transition, c'est a dire la fonction d'interpolation des deux animations. $TRANSITION\_LINEAR$ est une interpolation linéraire et $TRANSITION\_SMOOTH$ est une interpolation d'Hermite cubique.
- structure `AnimationControllerTransition(`$from$`,` to`,` *condition*`,` *transitionLength*`,` *type*`,` *conditionFunction*`)` représente une transition entre l'animation $from$ et to ou encore une arête dans le graphe du contrôleur d'animation. *condition* est une *AnimationControllerTransitionCondition* et *type* est une *TransitionType*. La longueur de la transition est contrôlée par *transitionLength* et si *condition* vaut $COND\_CUSTOM$, alors on peut définir une fonction de rappel *conditionFunction* qui retourne un booléen, vrai si on doit effectuer la transition, faux sinon.

Classe AnimationController

- constructeur `AnimationController::AnimationController(`*initialState*`,` *transitions*`,` *animations*`)` construit un *AnimationController* avec les transitions *transitions*, les animations *animations* et dans l'état initial *initialState*.
- méthode `AnimationController::getTransitionsFromCurrentState()` construit une liste des transitions que l'automate peut prendre depuis l'état actuel.
- méthode `AnimationController::update(`*deltaTime*`)` est la méthode de "tick" du contrôleur, cette méthode est appelée chaque frame et elle contrôle la génération de poses clés et de la logique automate a état finis.
- méthode `AnimationController::applyKeyframes(`*skeleton*`)` applique les poses clés générés lors de l'appel de la méthode *update* au squelette *skeleton*.

4 Recherche de Chemin

4.1 Introduction

La recherche de chemin (aussi appelée pathfinding) consiste a trouver un chemin entre un point de départ et un point d'arrivée. Cette recherche est souvent effectuée sur un graphe où les points d'arrivée et de départ sont des noeuds sur le graphe.

Dans le domaine du jeu vidéo, la recherche de chemin est principalement utilisée pour permettre a des agents autonomes de ce déplacer dans un monde virtuel tout en suivant certaines contraintes (tel que de ne pas passer a travers les murs par exemple)[6].

Il existe plusieurs algorithmes de recherche de chemin mais, dans dans le domaine du jeu vidéo, le plus populaire est l'algorithme A* pour son efficacité en terme de calcul et le fait qu'il trouve un chemin optimal dans certaines conditions[7].

4.2 Représentation du monde

Comme mentionné dans la section précédente, le monde est représenté sous la forme d'un graphe. Chaque noeud de ce graphe possède des information sur sa position dans le monde, ainsi qu'une liste de ces voisins (limité a 8 pour des raisons de performance).

Chaque sommet de ce graphe est appelé un "waypoint"[6] et chaque arrête représente un chemin traversable par l'agent sans obstacle.

4.3 A* et implémentation

L'algorithme A* (prononcé A étoile) est un algorithme de recherche de chemin dans un graphe[8]. Créé en 1968 par Nils Nilsson, il combine des aspects de l'algorithme de Dijkstra et de la recherche gloutonne. A chaque noeud on associe une fonction de coût *f(n)* définie comme :

$$f(n) = g(n) + h(n)$$

Où :

- $g(n)$ est le coût depuis le noeud de départ jusqu'au noeud n
- $h(n)$ est la fonction d'heuristique qui estime le coût entre le noeud n et le noeud but[9].

Afin que l'algorithme trouve une solution optimale, la fonction heuristique $h(n)$ doit être[9] :

- admissible, cela signifie que la fonction ne surestime jamais le vrai coût entre le noeud n et le noeud but.
- monotone, on s'assure que $h(n) \leq c(n, n') + h(n')$ pour toute arête (n, n'), où $c(n, n')$ est le coût de n a n'.

Algorithme 1 A*

Entrée : un noeud de départ D, un noeud but B, un graphe G

Initialiser $openSet$ = File Prioritaire triée par $f(n)$ contenant D
Initialiser $closedSet = \emptyset$
Initialiser $gScore$ = tableau associatif initialisé a ∞
$gScore[D] \leftarrow 0$
Initialiser $fScore$ = tableau associatif initialisé a ∞
$fScore[D] \leftarrow h(D)$
tant que $openSet$ n'est pas vide **faire**
  $noeud \leftarrow$ défiler $openSet$
  si $noeud$ est B **alors**
    On a trouvé le chemin
    On reconstruit le chemin et on termine
  fin si
  $openSet \leftarrow openSet \setminus \{noeud\}$
  $closedSet \leftarrow closedSet \cup \{noeud\}$
  pour tout $v \in$ voisins de $noeud$ dans G **faire**
    si $v \notin closedSet$ ET $gScore[noeud] + distance(noeud, v) < gScore[v]$ **alors**
      $gScore[v] \leftarrow gScore[noeud] + distance(noeud, v)$
      $fScore[v] \leftarrow gScore[noeud] + distance(noeud, v) + h(v)$
      $openSet \leftarrow openSet \cup \{v\}$
    fin si
  fin pour
fin tant que
On n'a pas trouvé de chemin
On termine avec une erreur

4.4 Documentation de la partie recherche de chemin

Le fichier *NavGraph.hpp* implémente et définie les classes, structures et fonctions qui servent à la recherche de chemin

4.4.1 Classe Node

La classe Node représente un sommet du graphe de navigation

- structure `Link(`*id*`,` *cost*`)` est possédée par une *Node* et représente un lien (arête) entre le noeud actuel et un noeud voisin avec l'identifiant *id* avec le coût égal a *cost*.
- constructeur `Node::Node(`*id*`,` *position*`)` Construit un noeud avec l'identifiant *id* a la position dans le monde *position*.
- méthode `Node::getPosition()` retourne la position du noeud dans le monde.
- méthode `Node::getLink(`*n*`)` retourne l'arête numéro *n* dans la liste de voisins.
- méthode `Node::getID()` retourne l'identifiant du noeud.
- méthode `Node::getNeighborsN()` retourne le nombre de voisins du noeud.
- méthode `Node::connectNode(`*id*`,` *cost*`)` connecte le noeud a un voisin avec l'identifiant *id* pour un coût *cost*.
- méthode `Node::deconnectNode(`*id*`)` déconnecte le noeud d'un noeud voisin avec l'identifiant *id*.
- méthode `Node::rearrangeNeighbors()` s'assure que la liste des voisins reste correctement triée (contiguë) et recalcule le nombre de voisin. Cette méthode est appelée lorsqu'on modifie le nombre de voisins du noeud avec $Node :: deconnectNode$.
- méthode `Node::print()` affiche les détails sur le noeud dans la console. La méthode affiche : l'identifiant du noeud, la position dans le monde, les voisins et le coût du lien entre le noeud actuel et le voisin.

4.4.2 Classe NavGraph

Cette classe représente un graphe de navigation, les noeuds sont des *Node*.

- constructeur `NavGraph::NavGraph(`id`)` construit un graphe de navigation avec l'identifiant id.
- méthode `NavGraph::addNode(`$position$`)` crée et ajoute un noeud a la position $position$
- méthode `NavGraph::connectNodes(`id_A`,` id_B`)` crée une arête entre le noeud avec l'identifiant id_A et le noeud avec l'identifiant id_B.
- méthode `NavGraph::deconnectNodes(`id_A`,` id_B`)` enlève l'arête entre le noeud avec l'identifiant id_A et le noeud avec l'identifiant id_B.
- méthode `NavGraph::shortestPath(`id_{start}`,` id_{end}`,` $Path$`)` Utilise l'algorithme A* pour trouver le meilleur chemin entre le noeud id_{start} et le noeud id_{end}. Le chemin est stocké dans $Path$.
- méthode `NavGraph::reconstructPath(`$cameFromList$`,` $id_{current}$`,` id_{start}`,` $Path$`)` reconstruit récursivement le chemin généré par $shortestPath$ a l'aide de la liste $cameFromList$. Stocke le chemin dans $Path$.
- méthode `NavGraph::nearestNode(`pos`)` retourne le noeud dans le graphe le plus proche en distance euclidienne de la position pos.
- méthode `NavGraph::getNodes()` retourne les noeuds du graph.
- méthode `NavGraph::print()` affiche les détails du graphe dans la console. l'identifiant du graphe et ensuite appelle `Node::print` sur tout les noeuds du graphe.

4.4.3 Classe Path

Cette classe représente un chemin entre deux points, elle utilise le graphe de navigation pour trouver un chemin entre deux points du monde.

- constructeur `Path::Path(`pos_{start}`,` pos_{dest}`)` définie un chemin entre pos_{start} et pos_{dest}.
- méthode `Path::update(`$navGraph$`)` met a jour le chemin grace au graphe de navigation $navGraph$. La méthode appelle `NavGraph::shortestPath` pour trouver le plus cours chemin jusqu'à la destination.

5 Moteur physique

5.1 Modèle physique

Le moteur physique joue un rôle central dans de nombreuses applications, notamment dans les logiciels de modélisation et d'animation qui privilégient une simulation réaliste et précise. En particulier, lorsqu'il est crucial d'optimiser la vitesse d'exécution selon les spécifications du projet, l'utilisation d'un modèle physique en temps réel est largement préférée. Cela en devient ainsi une nécessité pour assurer une expérience de jeu fluide et immersive pour les joueurs lorsqu'il doit fait partie intégrante d'un jeu vidéo.

Généralement, ces moteurs reposent sur les concepts de base de la physique et de la mécanique classique. Ces principes sont au cœur des moteurs physiques les plus répandus comme PhysX, Havok, Euphoria ou encore Box2D et Bullet Physics en 2D, qui exploitent les lois de la dynamique et de l'inertie de Newton afin de générer des mouvements.

Cette partie s'alimente largement des diverses approches développées dans les moteurs existants, et notamment des travaux d'Erin Catto, le développeur du moteur Box2D. Diverses sources proposent différentes approches quant à la modélisation de la dynamique dans un contexte informatique. Par exemple, Müller et al. proposent une approche basée sur la dynamique de position[10], tandis que D. Baraff, comme détaillé dans la feuille de route[11], privilégie un modèle basé sur les forces. En outre, B. Mirtich présente dans sa thèse un modèle basé sur l'impulsion[12], qui se voit finalement très populaire dans le développement de moteurs physiques en raison de sa simplicité et de son efficacité à maintenir des dynamiques réalistes en temps réel, pour des corps rigides. C'est donc cette dernière approche qui a été explorée au long de ce TER.

Soit l'équation de l'impulsion $\vec{J}$ résultante de forces de contact $\vec{F}(t)$ entre deux corps pendant un temps t

$$\vec{J} = \int \vec{F}(t)\, dt$$

Le principe derrière ce modèle s'explique par le choix de contraindre la mécanique sur des corps totalement rigides (à l'inverse de corps déformables). De ce fait, car aucune déformation ne prend place durant la collision, la durée de contact entre deux corps se retrouve extrêmement proche de 0. Ainsi, l'impulsion se réduit seulement au calcul de la force exercée à l'instant précis où le contact prend place, et donc $\vec{J} = \vec{F}$.

5.2 Le mouvement

Le moteur physique présente un ensemble de corps munis des vecteurs décrivant les états du mouvement des objets dans l'espace, dont la position $\vec{p}$, la vélocité $\vec{v}$, l'accélération $\vec{a}$, la rotation $\vec{\theta}$ et la vélocité angulaire $\vec{\omega}$. Un pas de temps Δt sépare deux instants durant lesquels les calculs du mouvement sont effectués. En théorie, ce mouvement entre un instant t et $t + \Delta t$ se décrit par les équations fondamentales de la cinétique :

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \int_t^{t+\Delta t} \vec{a}(t')\, dt'$$
$$\vec{p}(t + \Delta t) = \vec{p}(t) + \int_t^{t+\Delta t} \vec{v}(t')\, dt'$$
$$\vec{\theta}(t + \Delta t) = \vec{\theta}(t) + \int_t^{t+\Delta t} \vec{\omega}(t')\, dt'$$

Or, en pratique, l'intégration d'un mouvement en fonction du temps ne peut pas être aisément calculée. La méthode d'intégration d'Euler explicite est la plus classique, car il s'agit seulement de considérer les composantes comme constantes entre t et $t + \Delta t$. Or, la précision devient problématique lorsque la vitesse où Δt sont relativement grandes, ou lorsque l'accélération varie fortement. La méthode d'intégration de Runge-Kutta ainsi que celle de Verlet[13] démontrent de meilleurs résultats lorsque l'objet est soumis à des fluctuations de l'accélération, mais l'intégration de Verlet a été retenue, car non seulement elle conserve l'énergie cinétique entre deux moments, elle donne d'aussi bons résultats qu'une intégration de Runge-Kutta d'ordre 3 ou 4 pour des $\vec{v}$ ou Δt de petite magnitude, et minimise

le temps de calcul. Par ailleurs, une approche hybride peut encore être envisageable, en considérant l'amplitude des précédents paramètres.

L'accélération est calculée à chaque instant t selon la position de l'objet. Pour toute fonction de champs d'accélération $\vec{f}(\vec{p})$, nous calculons l'accélération du corps $\vec{a}$ en fonction de sa position $\vec{p}$,

$$\vec{a} = \sum_{\vec{f}} \vec{f}(\vec{p})$$

Elle correspond à la somme des champs d'accélération (gravitationnel, magnétique, etc.) auxquels le corps est soumis, pour des champs aussi bien uniformes que non uniformes.

Par ailleurs, on définit l'amortissement linéaire ξ_v et angulaire ξ_ω agissant directement sur la vélocité linéaire et angulaire du corps. Une approximation de cette action sur $\vec{v}$ et $\vec{\omega}$ est donnée,

$$\vec{v}'(t + \Delta t) = \vec{v}(t + \Delta t)(1 - \xi_v \Delta t)$$
$$\vec{\omega}'(t + \Delta t) = \vec{\omega}(t + \Delta t)(1 - \xi_\omega \Delta t)$$

5.3 Propriétés de l'objet

En plus des différentes composantes caractérisant la cinétique d'un objet, ce dernier est muni de davantage de paramètres permettant au développeur une manipulation plus poussée du modèle physique. À un objet est attribué un collisionneur, forme géométrique permettant de résoudre plus tard l'intersection et la collision avec cet objet, sa masse m, et donc sa densité d dépendante de son volume V qui est calculé selon la forme du collisionneur et enfin un matériau définissant les propriétés physiques de la surface de l'objet, c'est-à-dire la restitution ϵ ainsi que la friction μ. L'objet possède également des contraintes sur son mouvement que le développeur peut utiliser afin de fixer soit la position, soit la rotation de l'objet sur la scène. Finalement, l'objet appartient à une couche dont les règles de contact sont définies entre les différentes couches présentes sur la scène. En l'occurrence, les objets appartenant à la même couche peuvent se heurter tandis que les objets de couche d'identifiant zéro peuvent passer au travers d'objets de la couche une selon les spécifications du développeur.

5.4 Détection de la collision

5.4.1 Méthodes de détection

La détection de la collision se base sur un modèle discret, couramment utilisé dans les moteurs de jeu vidéo. Elle oppose la collision continue qui vise à déterminer l'instant précis entre deux pas de temps durant laquelle une ou plusieurs collisions prennent place[14]. La vérification de la collision dans un tel modèle discret s'effectue alors simplement à chaque pas de temps en omettant intégralement ce qui aurait pu se passer entre ce pas de temps et le précédent. Le choix d'un tel modèle relève de ses performances, compromettant tout de même la précision de la détection et de la réponse à la collision qui, dans la majorité des cas, restent totalement suffisantes. Or, cette approche ne permet pas de résoudre l'ensemble des contraintes du modèle, tels que les contraintes de pénétration, qui doivent, dans des moteurs plus avancés, bénéficier d'un système plus poussé de résolution de contraintes au contact.

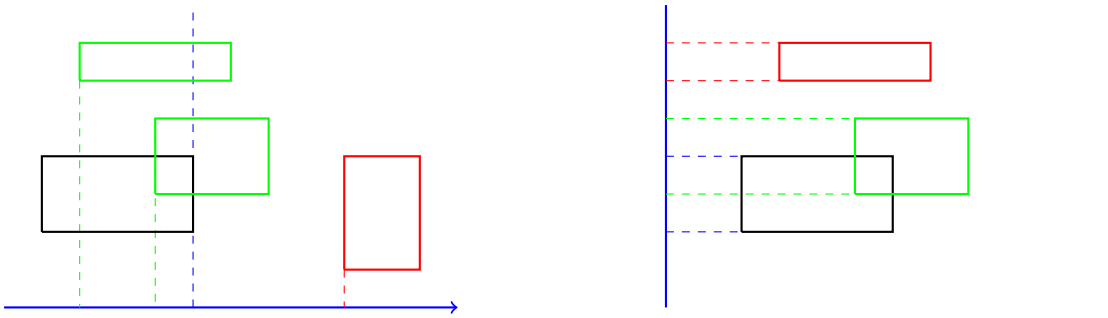
Une collision prend effet lorsque deux corps rentrent en contact, c'est-à-dire lorsque l'intersection de leurs formes géométriques est non-nulle. Afin de détecter cette collision, l'intersection de chaque paire de corps doit être vérifiée, ce qui constitue $O(n^2)$ tests d'intersection à effectuer. Or, il est clair que l'ensemble des objets n'entre pas constamment en collision, il s'agit donc d'éliminer d'abord les paires d'objets dont nous pouvons déterminer avec certitude qu'ils ne se percuteront pas.

Cette élimination fait référence à la "broad-phase"[15], ou phase large. Elle élimine d'abord les paires d'objets statiques qui n'auront pas d'effet sur leur collision, les paires d'objets appartenant à des calques dont les règles de pénétration ne sont pas définies, celles où les corps s'éloignent entre eux, ou simplement les paires dont les objets se situent à des positions plus distantes, ce qu'il est possible de déterminer à l'aide d'un arbre de partitionnement spatial binaire ou d'un arbre kd. Par ailleurs, l'algorithme de phase large "Sweep and Prune"[16] a permis le remplacement de l'usage d'un arbre kd grâce à son efficacité.

La "narrow-phase"[15] ou la phase étroite permet de déterminer avec évidence si les paires d'objets sélectionnées rentrent en contact. Il s'agit ainsi de vérifier l'intersection entre les deux polyèdres formant

l'enveloppe des objets. Le cas échéant, elle indique la pénétration δ entre les deux objets, la normale de contact $\vec{n}$ et le point de contact $\vec{r}$.

5.4.2 "Sweep and Prune"



(a) Étape 1 : Balayage des corps (b) Étape 2 : Résolution des potentielles intersections

FIGURE 14 – Résolution des paires de contact

"Sweep and Prune" est le nom donné par J. D. Cohen[17] pour la méthode d'élagage des paires de potentiels objets en collisions, et développé en premier lieu par D. Baraff dans sa thèse[16]. Il désigne l'algorithme qui balaie l'ensemble des corps rigides dans une direction et détermine s'ils sont de potentiels candidats pour la collision sur l'axe de balayage. Son efficacité repose sur l'omission des vérifications au-delà des limites définies par la boîte englobante de chaque objet dans l'axe de la direction, ce qui laisse place à une simple vérification de l'intersection des boîtes englobantes dans le plan normal à cette direction (voir figure 14). Le choix de l'axe de balayage n'est pas important, mais l'algorithme implémenté emploie l'axe x du monde, ce qui facilite également les calculs des boîtes englobantes. Le plan normal est donc défini par les axes y et z, ce qui rend les calculs de l'intersection sur ces axes triviaux.

Algorithme 2 Sweep and Prune

Entrée : Un ensemble de corps rigides $S = B_1, ..., B_n$

Trier S sur l'axe x
Initialiser $P = \emptyset$
pour $i = 1$ **à** n **faire**
  $j \leftarrow i + 1$
  tant que $j \leq n$ **et** $S[i]_{\mathbf{max}} \leq S[j]_{\mathbf{min}}$ **faire**
    si les boîtes englobantes de $S[i]$ **et** $S[j]$ **intersectent sur les axes** y **et** z **alors**
      $P \leftarrow P \cup \{(S[i], S[j])\}$
    fin si
    $j \leftarrow j + 1$
  fin tant que
fin pour
retourner P

L'algorithme 2 réduit donc le nombre de tests d'une complexité de $O(n^2)$ à $O(n \log n)$. Il est également possible pour cet algorithme de négliger la phase de tri si ce dernier est effectué dynamiquement à chaque mise à jour de la position des corps, réduisant donc la complexité en un temps linéaire.

5.4.3 Intersection de polyèdres

Les corps rigides sont ici représentés par quatre classes de polyèdres. Il s'agit ici des boîtes orientées (OBB), pouvant également être optimisées en boîtes d'axes alignés (AABB), des sphères, des capsules, ainsi que des formes convexes plus générales. Le choix de ces polyèdres n'est pas arbitraire, car il représente l'ensemble des formes suffisantes à l'élaboration d'un jeu vidéo donc la physique est relativement simplifiée, mais également, car elles possèdent la propriété de convexité facilitant l'implémentation

d'algorithmes de résolution de l'intersection. Parmi ces algorithmes, nous notons l'algorithme reposant sur le théorème des axes séparateurs (SAT)[15] ainsi que l'algorithme GJK[18,19].

	OBB	**Sphère**	**Capsule**	**Convexe**
OBB	SAT ou (1)	(2)	(4)	GJK
Sphère		(3)	(3)	(4)
Capsule			(3)	(4)
Convexe				GJK

FIGURE 15 – Méthodes de résolution des intersections

La figure 15 démontre les différentes méthodes[15,18] utilisées afin de déterminer si l'intersection des polyèdres existe :

1. Pour des AABBs, une version simplifiée de SAT peut être utilisée en ne vérifiant que l'intersection sur les trois axes principaux.
2. L'algorithme détermine le point le plus proche dans l'OBB avec la sphère en projetant les coordonnées de la sphère sur les trois axes de l'OBB et en contraignant les projections obtenues à l'intérieur de l'OBB. La distance entre ce point et le centroïde de la sphère est ensuite comparée au rayon de la sphère.
3. L'algorithme détermine la paire de centroïdes ou de points plus proches sur l'axe de la capsule et compare la distance entre ces points avec la somme des rayons.
4. L'algorithme triangulise ou utilise la forme triangulisée du polyèdre et détermine le point le plus proche parmi l'ensemble des triangles sur la surface du convexe. Il compare la distance entre ce point et le centroïde de la sphère, ou le point sur l'axe le plus proche de la capsule, avec son rayon.

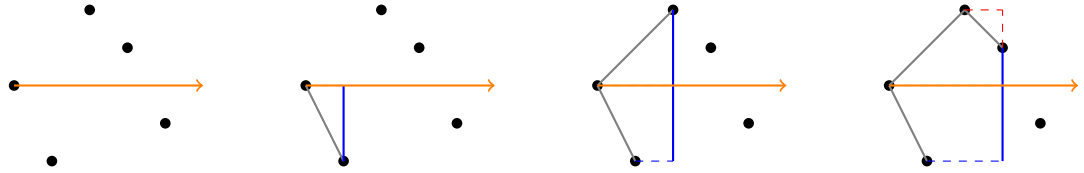
Pour une capsule, un tel point sur l'axe se situe notamment sur le segment reliant les deux hémisphères. Il peut être déterminé à partir d'un système linéaire pour trouver la paire de points qui réduit la distance entre deux segments, ou entre un segment et un point.

Dans certains cas, l'information obtenue ne permet pas d'en déduire la surface de contact résultant de la collision de deux corps, et dans d'autres cas, cela avait tendance à générer une réponse insatisfaisante à cette collision. SAT et GJK possèdent ce problème, et même si ce dernier est capable de générer un point de contact au moyen de l'algorithme d'expansion des polytopes (EPA)[20], il ne permet pas d'obtenir le point sur lequel se concentre la pression exercée par la collision. De ce fait, il suffit de déterminer le polygone convexe de contact des deux objets, puis d'appliquer un algorithme d'intersection tel que Bentley–Ottmann[21], ici utilisé, puis de calculer le barycentre de ce polygone, correspondant au point de contact.

Cet algorithme se base sur une approche par balayage. Un axe orthogonal à la normale des polygones est choisi, puis les sommets sont triés selon cette direction. La phase de balayage permet de vérifier s'il existe une intersection entre deux segments, puis l'ajouter au polygone final. Dans le cas d'un convexe, l'algorithme a été amélioré afin de découper chacun des polygones en deux, permettant simplement de tester deux segments à la fois, ce qui permet également de déterminer les points à l'intérieur de chacun de ces polygones.

Le barycentre d'un polygone convexe peut se calculer de manière naïve en prenant la moyenne de chaque point, ce qui suffit dans la plupart des cas où les points sont régulièrement répartis dans le convexe.

5.4.4 Triangulation et tétraédralisation d'un ensemble convexe

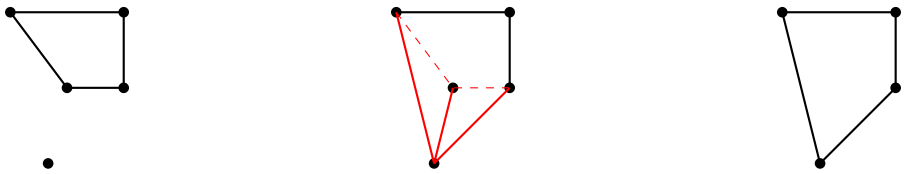


(a) Initialisation (b) Itération 1 (c) Itération 2 (d) Itération 3

FIGURE 16 – Premières itérations de la triangulation de la surface d'un polytope convexe

Diverses méthodes proposent des approches de triangulation de polytopes, se basant notamment sur une triangulation de Delaunay. Ces approches se voient relativement disproportionnées quant aux conditions de départ d'un tel algorithme, puisque le polyèdre est convexe et ses points forment sa propre enveloppe. Car la recherche d'une méthode de triangulation se basant sur ces conditions spécifiques semblait difficile, un algorithme a été conçu spécialement pour résoudre ce problème en se basant sur les propriétés relatives aux convexes, et en particulier sur les propriétés et approches concernant le maintien d'un polygone convexe 2-dimensionnel.

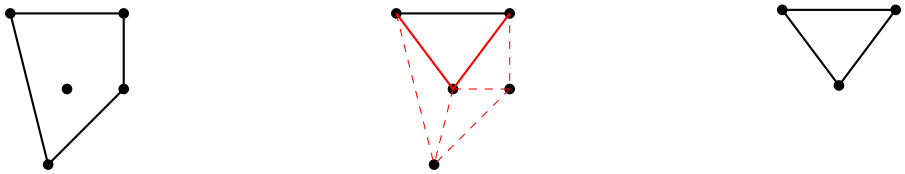
Le fonctionnement, tel que décrit dans l'algorithme 3 repose sur notre perspective d'un polyèdre. Selon un axe de vue, la silhouette du polyèdre convexe est un polygone convexe défini par un ensemble de points $S_s \subseteq S$. On définit itérativement les points à l'extérieur de la projection de l'enveloppe, qui est un plan normal à l'axe de vue, comme ceux qui ne sont pas masqués par la silhouette. Par exemple, dans la figure 16, il s'agit des trois premiers points de gauche à droite, ce que nous pouvons en déduire visuellement à partir de la seconde itération. Les points à l'intérieur sont naturellement tous les points qui ne sont pas à l'extérieur.



(a) Ajout d'un point extérieur (b) Construction des triangles (c) Maintien de l'enveloppe

FIGURE 17 – Triangulation à l'ajout d'un point extérieur

En partant du point le plus proche de notre point de vue, l'algorithme parcourt tous les points, les projette sur le plan normal à l'axe de vue et vérifie si ces derniers sont présents à l'extérieur de l'enveloppe. Si c'est le cas, le point le plus proche sur ce plan forme l'arête d'un triangle. Pour tous ses voisins sur l'enveloppe dans les deux directions, nous formons un triangle jusqu'à maintien de la convexité de ce polygone, tout en supprimant les points à l'intérieur afin de maintenir la notion d'enveloppe. Cela indique qu'il n'est plus possible de tracer de triangles à partir des points retirés (voir figure 17).



(a) Ajout d'un point intérieur (b) Construction des triangles (c) Maintien de l'enveloppe

FIGURE 18 – Triangulation à l'ajout d'un point intérieur

À l'inverse, si un point se retrouve à l'intérieur de la projection des points de l'enveloppe, alors celui-ci se situe derrière le polyèdre selon notre axe de vue. Il s'agit donc de supprimer tous les points

les plus proches dans l'ordre de proximité sur la projection, jusqu'à ce que le polygone forme une enveloppe convexe, puis tracer les triangles entre ce point et deux points adjacents de l'ancienne enveloppe parcourus. C'est donc l'opération inverse (voir figure 18).

Lorsque l'algorithme est complété, les derniers points projetés sur l'enveloppe peuvent à leur tour être triangulisés, ce qui est une opération plus simple, car le dernier point relie tous les autres points restants sur la surface du polyèdre, et sont nécessairement adjacents.

Algorithme 3 Triangulation

Entrée : Un ensemble de sommets $S = P_1, ..., P_n$

Trier S sur l'axe x
$p \leftarrow$ plan orthogonal à x
Initialiser $E = \emptyset$
Initialiser $T = \{P_1\}$
pour tout $P_i = P_2$ **à** P_n **dans** S **faire**
  $C \leftarrow$ **le point le plus proche de** P_i **dans** $proj(E, p)$
  $C_1, C_2 \leftarrow$ **les voisins de** C **dans la direction** d_1 **et** d_2 **respectivement**
  si $proj(P_i, p)$ **est à l'intérieur de** E **alors**
    $T \leftarrow T \cup \{(P_i, C_1, C), (P_i, C_2, C)\}$
    $E \leftarrow E \setminus \{C\}$
    tant que $proj(P_i, p)$ **est à l'intérieur de** $proj(E, p)$ **faire**
      $C_j \leftarrow$ **le point** C_1 **ou** C_2 **le plus proche de** P_i **sur** p **dans** E
      (Procédure intermédiaire 4)
    fin tant que
  sinon
    pour tout $j \in \{1, 2\}$ **faire**
      tant que la convexité de $proj(E, p)$ **n'est pas satisfaite dans la direction** j **faire**
        (Procédure intermédiaire 4)
      fin tant que
    fin pour
  fin si
  $E \leftarrow E \cup \{P_i\}$
fin pour
pour tout (P_i, P_{i+1}) **dans** $E \setminus E[0]$ **faire**
  $T \leftarrow T \cup \{(E[0], P_i, P_{i+1})\}$
fin pour
retourner P

Algorithme 4 Étape intermédiaire de la triangulation

$V \leftarrow$ voisin de C_j dans la direction d_j
$T \leftarrow T \cup \{(P_i, C_j, V)\}$
$C_j \leftarrow V$
$E \leftarrow E \setminus \{C\}$

En revanche, cet algorithme ne possède pas la capacité d'une triangulation de Delaunay à maintenir une stabilité numérique lors d'ultérieurs calculs avec les triangles. Or, cela ne se voit pas problématique en pratique.

La tétraédralisation quant à elle consiste en décomposer le polyèdre en un nombre fini de tétraèdres, ce qui se fait par l'intermédiaire de la précédente fonction. N. J. Lennes a prouvé en 1910 que ce procédé n'est pas toujours possible sans l'ajout d'au moins un point[22]. De ce fait, la moyenne des sommets du polyèdre est un point appartenant à ce tétraèdre et dont chaque triangle sert de base au tétraèdre lorsque relié à ce dernier.

5.4.5 Théorème des axes séparateurs (SAT)

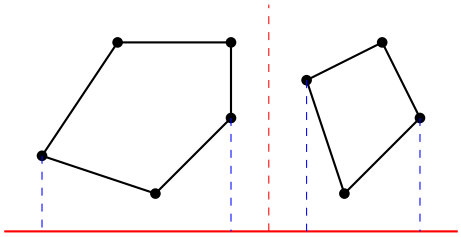


FIGURE 19 – Un axe séparateur entre deux ensembles convexes indiquant une disjonction

Le théorème des axes séparateurs ou SAT (pour "Separating Axis Theorem")[15,18] stipule que deux ensembles sont disjoints si et seulement s'il existe une projection disjointe des points de l'ensemble sur un axe séparateur (voir figure 19).

Pour des polyèdres convexes discrets tels que dans notre cas, l'ensemble des axes séparateurs correspond non seulement à la normale de toutes les faces de chaque polyèdre, mais aussi au produit vectoriel de ces normales afin de traiter les cas où l'intersection s'effectue au niveau des arêtes de chaque ensemble. Pour deux parallélépipèdes rectangles, nous avons ainsi 3 axes par polyèdre, puis 9 axes correspondant au produit vectoriel des 3 axes de chaque polyèdre, voire moins si deux axes sont orthogonaux, puisque leur produit vectoriel forme le vecteur nul.

On définit le support d'un ensemble convexe S dans la direction d comme le point s_d de S tel que :

$$s_d = \arg\max_{p \in S} \{d \cdot p\}$$

Algorithme 5 Algorithme du théorème des axes séparateurs

Entrée : Deux ensembles de sommets S_1 et S_2, un ensemble d'axes A

Trier S sur l'axe x
Initialiser $P = \emptyset$
pour tout axe a **dans** A **faire**
  $s_{(1,a)} \leftarrow support(S_1, a)$
  $s_{(1,-a)} \leftarrow support(S_1, -a)$
  $s_{(2,a)} \leftarrow support(S_2, a)$
  $s_{(2,-a)} \leftarrow support(S_2, -a)$
  si $[s_{(1,-a)}; s_{(1,a)}] \cap [s_{(2,-a)}; s_{(2,a)}] = \emptyset$ **alors**
    retourner faux
  fin si
fin pour
retourner vrai

L'algorithme 5 retourne `vrai` si une intersection existe, et `faux` sinon en une complexité temporelle de $O(mn)$ où n correspond au nombre de sommets des ensembles et m au nombre d'axes séparateurs. Notre implémentation retourne également la pénétration

$$\delta = \min_a \{|s_{(1,a)} - s_{(2,-a)}|, |s_{(1,-a)} - s_{(2,a)}|\}$$

On calcule également la normale $\vec{n} = \frac{a'}{|a'|}$ tel que a' minimise la pénétration δ dans la direction $d_2\vec{d}_1$ où

$$d_i = \frac{s_{(i,-a)} + s_{(i,-a)}}{2}$$

Or cet algorithme ne permet pas de retourner la surface de contact, faisant donc appel à la méthode indiquée dans la section 5.4.3.

5.4.6 Algorithme GJK

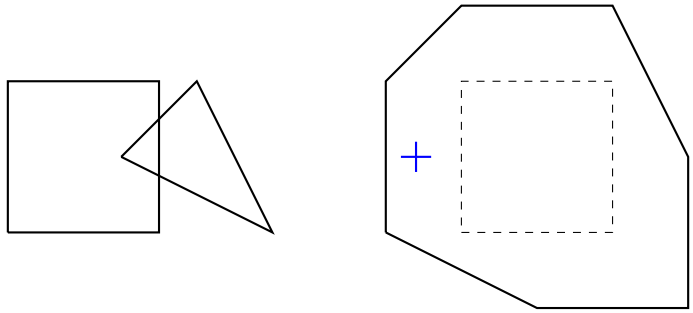
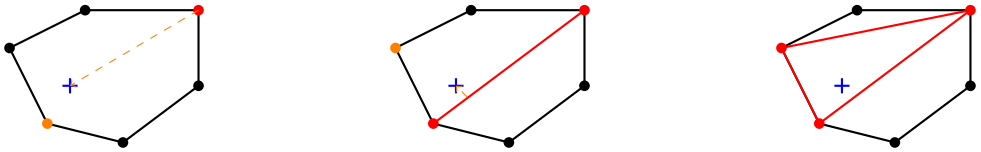


FIGURE 20 – Différence de Minkowski (CSO)

L'algorithme de GJK[19], portant le nom de ses chercheurs E. G. Gilbert, D. W. Johnson, et S. S. Keerthi est un algorithme permettant de déterminer la distance minimale entre deux ensembles convexes. Cet algorithme se base sur un objet dans l'espace de configuration (CSO), autrement dit, sur la différence de Minkowski $S_1 \ominus S_2 = \{p_1 - p_2 \mid p_1 \in S_1, p_2 \in S_2\}$ des ensembles convexes S_1 et S_2 formant à son tour un ensemble convexe. Si l'origine est contenue dans cette différence, l'intersection des ensembles convexes est non-nulle (voir figure 20).



(a) Initialisation du 1-simplexe s et calcul du support du CSO

(b) Expansion du 1-simplexe et répétition de (a)

(c) Expansion du 2-simplexe et fin de l'algorithme

FIGURE 21 – Déroulement de l'algorithme GJK

À chaque itération, cet algorithme détermine le simplexe (point, segment, triangle ou tétraèdre) le plus proche du point en calculant le support entre l'origine du CSO et s puis en reliant ce point afin de reconstruire ce simplexe. Si s englobe l'origine, alors une intersection existe (voir figure 21), sinon si le point de support appartient déjà à s, alors aucune intersection n'existe.

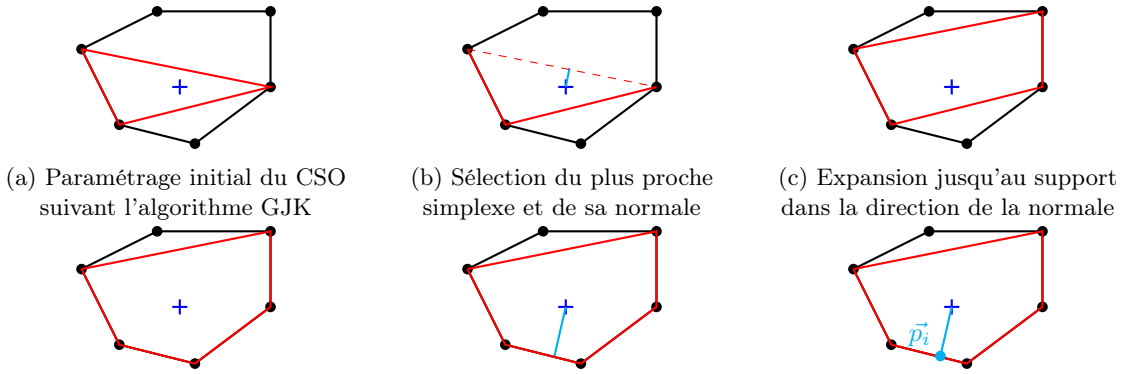
Algorithme 6 Algorithme GJK

Entrée : Deux ensembles de sommets S_1 et S_2, un axe de départ d

$A \leftarrow support(S_1, d) - support(S_2, -d)$
$s = \{A\}$
$D = -A$
tant que vrai **faire**
  $A \leftarrow support(S_1, d) - support(S_2, -d)$
  si $A \cdot D < 0$ **alors**
    retourner faux
  fin si
  $s, D \leftarrow simplexePlusProche(s)$
  si s englobe $\vec{0}$ **alors**
    retourner vrai
  fin si
fin tant que
retourner vrai

L'axe de départ d peut être choisi de manière arbitraire, mais la vélocité relative des deux objets forme une heuristique valable.

5.4.7 Algorithme d'expansion des polytopes (EPA)



(a) Paramétrage initial du CSO suivant l'algorithme GJK

(b) Sélection du plus proche simplexe et de sa normale

(c) Expansion jusqu'au support dans la direction de la normale

(d) Répétition des précédentes étapes (b) et (c)

(e) Sélection du plus proche simplexe

(f) Sur la surface du CSO, on détermine le point d'intersection

FIGURE 22 – Déroulement de l'algorithme EPA

Car l'algorithme de GJK ne permet pas de retourner directement la pénétration δ ou la normale de collision $\vec{n}$, il est impératif de recourir à l'algorithme d'expansion des polytopes (EPA)[20] qui fonctionne sur la base du simplexe s comprenant l'origine du repère $\vec{0}$ ainsi que des coordonnées des ensembles convexes de départ.

L'algorithme 7[23] consiste en étendre le simplexe sur le CSO afin de trouver, dans le cas d'un polyèdre, le triangle de l'une des faces de ce CSO par lequel l'intersection prend place et minimise δ. C'est-à-dire que l'algorithme sélectionne itérativement le triangle du polyèdre le plus proche puis l'étend en ajoutant le point de support dans la direction de la normale de ce triangle (voir figure 22).

Il se termine jusqu'à ce que ce que le point obtenu se situe sur la surface du CSO. On note $\Gamma(s)$ la bordure de Minkowski du polytope convexe s, c'est-à-dire sa surface.

Algorithme 7 Algorithme EPA

Entrée : Le CSO obtenu depuis l'algorithme GJK C, un simplexe s

tant que vrai **faire**
  $\vec{p_i} \leftarrow \arg\min_{p \in \Gamma(s)} |p|$
  $\delta \leftarrow |\vec{p_i}|$
  $c \leftarrow support(C, \vec{p_i})$
  si $c \in s$ **alors**
    retourner $\delta \leftarrow \vec{p_i}, \vec{n} \leftarrow \frac{\vec{p_i}}{\delta}, \vec{p_i}$
  fin si
  $s \leftarrow s \cup \{c\}$
fin tant que

En revanche, le point de contact obtenu $\vec{p_i}$ se situe sur le CSO. Il est possible de déterminer le point de contact sur les polyèdres en contact en effectuant une projection des coordonnées barycentriques de p_i du triangle du CSO sur celui de S_1 et de S_2 associés à ce triangle du CSO construit lors du déroulement de l'algorithme GJK[24]. Par ailleurs, la méthode présentée dans la section 5.4.3 permet également de récupérer un point de contact plus précis dans certains cas.

5.4.8 Choix de SAT ou de GJK

Ces deux algorithmes ainsi que leurs extensions génèrent les informations nécessaires afin de résoudre la collision. L'algorithme SAT se voit notamment très efficace lorsque le nombre d'axes est réduit ou lorsque les ensembles convexes se situent à une grande distance de l'autre, puisqu'il possède une complexité asymptotique de $\theta(n)$, avec n le nombre de sommets total, dans l'éventualité où les premiers axes permettent d'en déduire l'absence d'intersection, et et une complexité en $O(mn)$ dans le pire cas, avec m le nombre d'axes séparateurs. En revanche, l'algorithme GJK couplé avec EPA permet

également d'obtenir un résultat asymptotique en $\theta(n)$, puisque le nombre d'itérations ne dépasse que rarement 10. Il possède également une complexité de $O(n^2)$ dans le pire cas. Selon le nombre de points des convexes, nous obtenons généralement $m > n$. Ainsi, l'usage de SAT se retrouve réservé aux formes géométriques plus simples telles que les boîtes, laissant GJK traiter les formes plus complexes.

5.5 Résolution de la collision

5.5.1 Calculs de l'inertie

Avant de procéder à la résolution de la collision, le calcul des tenseurs d'inertie se voit prioritaire afin d'en déterminer l'impulsion selon la rotation de l'objet. La figure 23 permet de faire correspondre à un polyèdre son tenseur d'inertie[25]. Notons également au préalable que toute rotation d'un tenseur d'inertie par une matrice de rotation R s'effectue de cette manière :

$$I' = R^T I R$$

ce qui se vera utile par la suite.

Polyèdre	**Inertie** I
OBB de masse m et de dimensions (x, y, z)	$\frac{m}{12}\begin{pmatrix} y^2+z^2 & 0 & 0 \\ 0 & x^2+z^2 & 0 \\ 0 & 0 & x^2+y^2 \end{pmatrix}$
Sphère de masse m et de rayon r	$\frac{2m}{5}\begin{pmatrix} r^2 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 \end{pmatrix}$
Cylindre de masse m, de rayon r et de hauteur h	$\frac{m}{2}\begin{pmatrix} a & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & a \end{pmatrix}$ avec $a = \frac{1}{6}(3r^2 + h^2)$
Capsule de masse m_C+m_H, de rayon r et hauteur h[26]	$I_{Cyl}(m_C) + 2m_H \begin{pmatrix} b & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & b \end{pmatrix}$ avec $a = \frac{4r^2}{5}$ et $b = a + \frac{h^2}{2} + \frac{3hr^2}{4}$
Tétraèdre de masse m, et de points (A, B, C, D)	(1)
Convexe de masse m, et de points $(P_0, ...P_n)$	(2)

FIGURE 23 – Tenseurs d'inertie des formes convexes

Les tenseurs d'inertie d'un tétraèdre et d'un polyèdre convexe sont plus compliqués à calculer que ceux des formes convexes plus basiques :

1. Le tenseur d'inertie d'un tétraèdre correspond à la matrice suivante autour de l'origine du repère.

$$m\begin{pmatrix} a & -b' & -c' \\ -b' & b & -a' \\ -c' & -a' & c \end{pmatrix}$$

   où chaque composante correspond est décrite par une combinaison linéaire des coordonnées du tétraèdre dans l'article de F. Tonon[27].

2. Le tenseur d'inertie d'un polyèdre convexe est défini par la somme des tenseurs d'inertie de chaque tétraèdre qui le compose[25] (voir 5.4.4 pour la tétraédralisation). Selon la position du centre de la masse du convexe, ces derniers sont soumis à une transformation du centre d'inertie selon le théorème des axes parallèles :

$$I_C = \sum_{i=1}^{n} I_{T_i} + V_{T_i}((r_{T_i} - r_C)^2 I_3 - (r_{T_i} - r_C)(r_{T_i} - r_C)^T)$$

   où I_{T_i} correspond à l'inertie du tétraèdre i, r_{T_i} à son centre de masse et r_C à celui du convexe. $(r_{T_i} - r_C)(r_{T_i} - r_C)^T$ dénote le produit dyadique de la différence des centres de masse des polyèdres. I_3 correspond à la matrice identité dans $\mathbb{R}^3$.

5.5.2 Méthode de résolution

La méthode de résolution de la collision se développe en plusieurs étapes. La première consiste en appliquer la perturbation de la position des objets en fonction de la pénétration des deux corps physiques. Les précédentes méthodes de détection de collision génèrent un coefficient de pénétration δ ainsi qu'une normale $\vec{n}$.

$$m^{-1} = \frac{1}{m_1 + m_2}$$

$$\vec{p_1} = \vec{p_1'} - m^{-1} m_1 \delta \vec{n}$$
$$\vec{p_2} = \vec{p_2'} + m^{-1} m_2 \delta \vec{n}$$

La seconde étape consiste en appliquer une impulsion le long de la normale[28]. On définit $\Delta\vec{v} = \vec{v}_2 + \vec{\omega}_2 \times \vec{r}_2 - \vec{v}_1 - \vec{\omega}_1 \times \vec{r}_1$ la vélocité relative entre les deux corps, où $\vec{r}_1$ et $\vec{r}_2$ sont les points de collision relatifs à la position de ces derniers. Cette impulsion se calcule à partir du tenseur d'inertie I_1 et I_2 de ces derniers. Un coefficient de restitution $\epsilon \in [0, 1]$ détermine la fraction d'énergie cinétique conservée lors du contact.

$$j_n = \frac{-(1 + \epsilon_1 + \epsilon_2)(\Delta\vec{v} \cdot \vec{n})}{m_1^{-1} + m_2^{-1} + (I_1^{-1}(\vec{r}_1 \times \vec{n}) \times \vec{r}_1 + I_2^{-1}(\vec{r}_2 \times \vec{n}) \times \vec{r}_2) \cdot \vec{n}}$$

permettant donc d'appliquer l'impulsion selon la normale $\vec{J}_n = j_n \vec{n}$ au point de contact, résultant en une mise à jour du mouvement,

$$\vec{v}_1(t + \Delta t) = \vec{v}_1(t) - \vec{J}_n$$
$$\vec{\omega}_1(t + \Delta t) = \vec{\omega}_1(t) - I_1^{-1}(\vec{r_1} \times \vec{J}_n)$$
$$\vec{v}_2(t + \Delta t) = \vec{v}_2(t) + \vec{J}_n$$
$$\vec{\omega}_2(t + \Delta t) = \vec{\omega}_2(t) + I_2^{-1}(\vec{r_2} \times \vec{J}_n)$$

Les différents tenseurs d'inertie utilisés ici sont référencés dans le tableau 23.

Troisièmement, et finalement, l'impulsion est appliquée sur la tangente du corps solide. Il s'agit ici de l'impulsion relative à la force de friction dont seul le coefficient de friction dynamique μ du modèle de friction de Coulomb est utilisé, puisqu'il permet également de rendre les corps statiques selon la vélocité et donc l'impulsion. Il s'applique de la même manière que l'impulsion le long de la normale, mais sur la tangente de collision

$$\vec{t} = \frac{\vec{\Delta v} - (\vec{\Delta v} \cdot \vec{n})\vec{n}}{|\vec{\Delta v} - (\vec{\Delta v} \cdot \vec{n})\vec{n}|}$$

$$j_t = \frac{-\Delta\vec{v} \cdot \vec{t}}{m_1^{-1} + m_2^{-1} + (I_1^{-1}(\vec{r}_1 \times \vec{t}) \times \vec{r}_1 + I_2^{-1}(\vec{r}_2 \times \vec{t}) \times \vec{r}_2) \cdot \vec{t}}$$

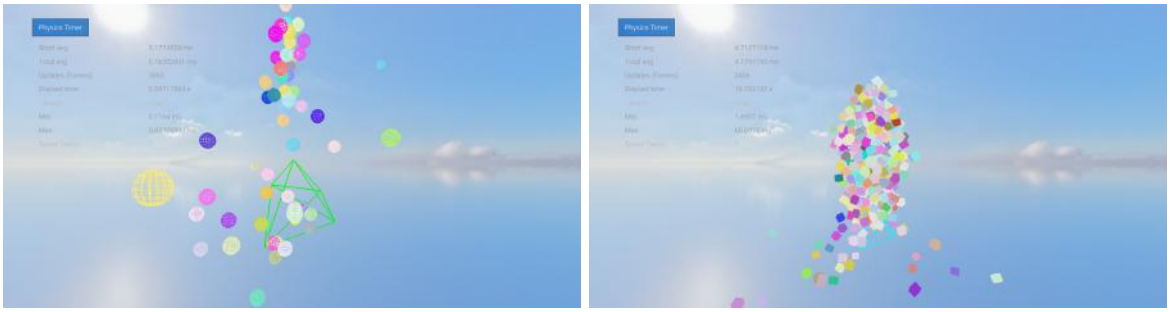
Nous contraignons j_t à une fraction $\mu = \mu_1\mu_2$ de l'impulsion normale, résultant en une friction nulle si $\mu = 0$ et plus prononcée lorsque μ croît :

$$-\mu j_n \leq j_t \leq \mu j_n$$

L'impulsion le long de la tangente est donc appliquée de la même manière sur l'objet que pour l'impulsion le long de la normale.

5.6 Résultats

5.6.1 Stabilité



(a) 100 sphères (b) 500 parallélépipèdes rectangles

FIGURE 24 – Démonstration de la collision sur un convexe

Nous obtenons une physique relativement stable pour des corps à rotation fixe, et l'impulsion sur la normale ainsi que sur la tangente forment une réponse tout à fait valide. Malgré cela, l'impulsion sur des corps en rotation a tendance à générer des réponses peu précises, possiblement dus à des erreurs numériques. En effet, dans le moteur de physique de test, certains objets positionnés au-delà d'un autre auront tendance à générer un faible mouvement au lieu de maintenir le corps au repos. De plus, un problème semble tout de même persister dans l'ajout du module dans le moteur Vulpine puisque certains objets en rotation ont tendance à causer des rotations incorrectes, mais il semblerait que dans la majorité des cas, cela soit seulement visuel, puisque dans d'autres cas, les rotations, et notamment celles d'objets convexes génèrent des rotations "NaN" invalides.

En revanche, les méthodes de détection de la collision implémentées font voir des résultats très satisfaisants, tels que démontrés par les algorithmes SAT et GJK/EPA dans la figure 24b. Néanmoins, seul un problème de pénétration se fait remarquer, notamment dans la figure 24a et même si la collision prend place et sa réponse est fonctionnelle, les sphères avec une vélocité plus élevée se retrouvent coincées dans la forme convexe.

5.6.2 Performances

FIGURE 25 – Démonstration de la collision de 2500 sphères sur un parallélépipède rectangle

Sur un processeur Intel Core i5-12450H, le modèle est capable de gérer la collision de 4500 sphères sur un sol et entre elles au-delà d'une image par seconde en l'état actuel. Pour une expérience fluide, le modèle peut atteindre 60 images par seconde entre 2500 et 3000 sphères sur la scène (voir le rendu dans la figure 25).

Le graphe 26 met en relation les performances du modèle en fonction du nombre de sphères sur la scène. Il permet également de percevoir la différence entre l'approche naïve en $O(n^2)$ de l'approche optimisée à l'aide de Sweep and Prune et des différentes vérifications supplémentaires indiquées dans 5.4.1. Le recensement du temps inter-image final est effectué suite à la convergence du temps inter-image moyen depuis l'instant initial t_0.

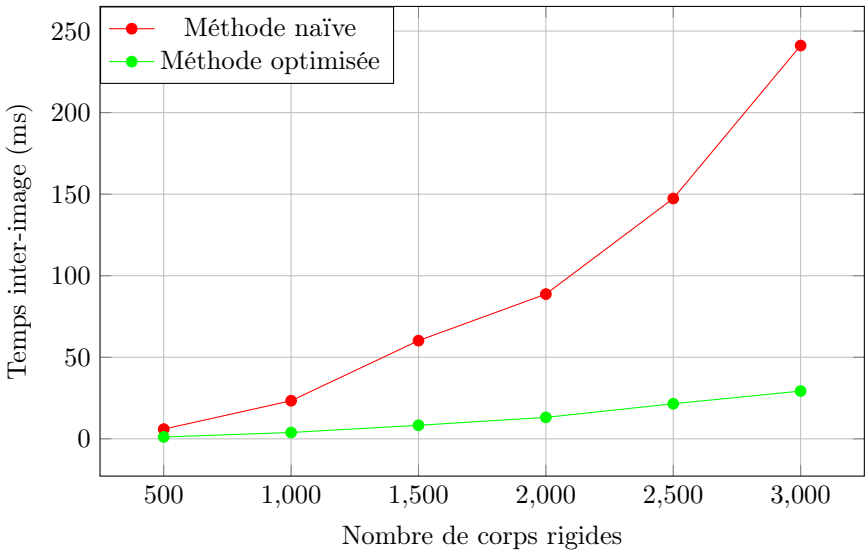


FIGURE 26 – Comparaison des performances du modèle

5.6.3 Discussion sur les résultats

Le modèle, malgré les différentes contraintes parcourues dans cette analyse, reste très performant, notamment depuis l'ajout d'optimisations de la phase large. Concernant les erreurs dans la phase étroite, nous estimons qu'il est possible de résoudre les problèmes de pénétration de convexes en rendant l'algorithme plus robuste afin de traiter les cas où la sphère ou le cylindre se situe à l'intérieur de ce dernier.

Or, quant aux rotations, une révision du modèle et de son affichage est nécessaire afin de s'assurer de son bon fonctionnement, car le résultat obtenu, quoique légèrement instable sur le moteur de test, se voit plus fortement sur le moteur Vulpine. Lors des recherches du problème, il était possible de constater que la plupart des angles de rotation n'étaient pas conservés suite à la conversion entre les quaternions et les angles d'Euler durant l'affichage, ce qui pouvait s'avérer donner des résultats visuellement surprenants, mais des physiques plus ou moins correctes. Par ailleurs, la section 5.7 discute des différentes méthodes en ce qui concerne l'amélioration de la réponse à la collision et donc à la mise à jour des rotations au contact.

5.7 Pistes d'approfondissement

5.7.1 Impulsions séquentielles

Durant la conférence GDC09, E. Catto renseignait la présence d'une méthode de résolution de collisions par impulsions séquentielles dans son moteur[29]. Cette méthode consiste en appliquer durant le même pas de temps, plusieurs résolutions supplémentaires de la collision générées se basant sur les réponses antérieures. L'avantage de cette méthode est de résoudre l'un des problèmes qui a été observé durant nos essais, et concernant le maintien d'un corps au repos, puisque le principe derrière cette idée est d'accumuler les impulsions sur chacun des points en contact plutôt que d'avoir un unique point de contact. Cette accumulation devient alors une approximation de l'impulsion réelle.

5.7.2 Résolution des contraintes

Un système de résolution de contraintes se voit un grand atout pour les moteurs physiques où de nombreuses contraintes jouent sur le résultat final. Nous avions effectivement parcouru diverses

contraintes tels que les contraintes de non-pénétration ou de friction qui ont été résolues en une seule itération. L'idée est d'étendre le principe d'impulsions séquentielles afin de corriger les différents paramètres du système de contraintes en appliquant diverses impulsions. En théorie, cette résolution se base sur le calcul du gradient de ce système au moyen de la matrice Jacobienne de l'ensemble des paramètres sous format matriciel ; gradient qu'il s'agit donc de diminuer à chaque impulsion afin d'approximer la résolution de ces contraintes[29,30]. On retrouve ce fonctionnement à la fois sur des contraintes d'égalité, telles que pour des joints de position, ou d'inégalité, tels que pour la friction ou encore la non-pénétration.

5.7.3 Joints et tissus

Une implémentation subséquente de celle du réseau de contraintes reste inévitablement, dans la plupart des moteurs qui en possède, celle des joints[29,30]. Il s'agit de contraintes posées entre deux ou plusieurs corps, que nous retrouvons sous différentes formes[15]

- Les joints rigides, associant les caractéristiques de deux corps en ceux d'un seul corps.
- Les joints de distance, assurant la conservation d'une distance fixée entre le point d'un corps et un autre point.
- Les joints de révolution, permettant la rotation d'un corps autour d'un autre point.
- Les joints prismatiques, ou joints de translation, bloquant le déplacement sur un axe ou un plan.
- Les joints pivot, dont la rotation sur un corps est transmise sur un autre corps autour d'un axe.
- Les joints poulie, correspondant à un joint de distance entre deux points passant par un ou plusieurs autres points.
- Les joints universels ou les joints de Cardan, permettant la transmission d'une rotation angulaire entre deux corps.

Certains de ces joints peuvent ainsi modéliser beaucoup de systèmes tels que l'ensemble des structures osseuses du corps humain, lorsque leurs diverses caractéristiques sont également contraintes de par la mise en place d'un angle de translation ou de rotation maximum. Par extension à cela, nous pouvons être en mesure de modéliser une corde ou bien un tissu, correspondant à une matrice de particules reliées par des joints de distance entre voisines[15].

D'autres usages de ces joints peuvent être retrouvés dans plusieurs autres applications, tels que pour modéliser la destruction ou la rupture d'un système du à des forces de tension (ponts), ou d'autres forces aussi bien internes qu'externes.

5.7.4 Physique continue

Un modèle de physique continu décrit certaines propriétés de mouvement ou encore des corps eux-mêmes comme continues[14]. L'ajout de méthodes continues a été réfléchie en amont de l'implémentation et devant faire suite à celle du modèle discret qui restait la priorité de ce TER. Le principe est d'ici d'accommoder le système de détection et de réponse à la collision sur des intervalles de temps entre deux instants t et $t+\Delta t$ pouvant en théorie générer un grand nombre de collisions et stabiliser fortement le modèle. En l'occurrence, nous y voyons notamment l'utilisation de méthodes de détection continue de la collision telle que celle des volumes balayés ("Swept volumes") ou bien des approches basées sur des spéculations ou prédictions.

5.8 Documentation de la partie physique du moteur

5.8.1 Moteur physique

Le fichier *Physics.hpp* contient l'ensemble des structures et fonctions principales du moteur physique permettant la simulation. L'ensemble des fonctions publiques est ainsi documenté, mis à part les "setters" (`set*()`) et "getters" (`get*()`) qui restent majoritairement accessibles au développeur.

Structure principale :

- structure `PhysicsMaterial(`μ`, `ϵ`, `ξ_v`, `ξ_ω`)` : définit les caractéristiques du matériau physique d'un corps
- structure `OBBCollider(`(h_x, h_y, h_z)`)` : construit un collisionneur "OBB" de dimensions $(2h_x, 2h_y, 2h_z)$.
- structure `SphereCollider(`r`)` : construit un collisionneur "Sphère" de rayon r.

- structure `CapsuleCollider(`h`, `r`)` : construit un collisionneur "Capsule" de hauteur $2h$ et de rayon r.
- structure `ConvexCollider(`S`)` : construit un collisionneur "Convexe" de sommets S.
- méthode `Collider::getColliderType()` : renvoie le type de collisionneur en tant que valeur de l'énumération `Collider::ColliderType` (`OBB`, `SPHERE`, `CAPSULE`, `CONVEX`)
- méthode `Collider::getInertiaTensor()` : renvoie le tenseur d'inertie de masse 1 associé à ce collisionneur et ses dimensions (voir 5.8.2).
- méthode `Collider::getVolume()` : renvoie le volume associé à ce collisionneur et ses dimensions (voir 5.8.2).
- méthode `Collider::getBoundingBox(`$\vec{p}$`, `$\vec{\theta}$`)` : renvoie la boîte englobante associée à ce collisionneur, ses dimensions, sa position $\vec{p}$ et sa rotation $\vec{\theta}$ (voir 5.8.2).
- méthode `Collider::getCollisionInfo(`$\vec{p_1}$`, `$\vec{\theta_1}$`, `$\vec{p_2}$`, `$\vec{\theta_2}$`, `C_2`)` : renvoie les résultats du test d'intersection entre le collisionneur à la position $\vec{p_1}$ et rotation $\vec{\theta_1}$ et un autre collisionneur C_2 de position $\vec{p_2}$ et rotation $\vec{\theta_2}$ (voir 5.8.2).

Champs de force (classe ForceField) :

- méthode statique `ForceField::make(`C`, `$\vec{p}$`, `$\vec{\theta}$`, `$\vec{f}$`)` : construit un champ de force de collisioneur C, de position $\vec{p}$, rotation $\vec{\theta}$ et fonction $\vec{f} : \vec{p} \to \vec{a}$.
- fonction `ForceField::uniformForceField(`$\vec{a}$`)` : retourne une fonction constante $\vec{f}(\vec{p}) = \vec{a}$.
- fonction `ForceField::pointForceField(`$\vec{p}$`, `P`, `r`)` : retourne une fonction de champ de force ponctuel, de rayon r et de puissance P autour de $\vec{p}$.

Corps rigide (classe RigidBody) :

- constructeur `RigidBody::make(`C`, `m`, `M`, `L`, `$b_1?$`, `$\vec{p}$`, `$\vec{v}$`, `$b_2?$`, `$\vec{\theta}$`, `$\vec{\omega}$`)` : construit un corps solide de collisionneur C, masse m, matériau M, couche L (valeur entière), position $\vec{p}$, vélocité initiale $\vec{v}$, rotation $\vec{\theta}$ et vélocité angulaire initiale $\vec{\omega}$. $b_1 = vrai$ si l'objet est statique et $b_2 = vrai$ si la rotation de l'objet est fixe.
- méthode `RigidBody::tick(`$\{F_1, ..., F_n\}$`, `Δt`)` : mise à jour du corps en fonction d'un ensemble de champs d'accélération $\{F_1, ..., F_n\}$ entre l'instant courant t et $t + \Delta t$.
- méthode `RigidBody::applyImpulse(`$\vec{r}$`, `I^{-1}`, `$\vec{J}$`)` : applique l'impulsion $\vec{J}$ au point $\vec{r}$ relatif à la position du corps. Le tenseur d'inertie I^{-1} est donné par son collisionneur ou les fonctions d'inertie dans 5.8.2.

Zones de déclenchement (classe Trigger) :

- méthode statique `Trigger::make(`C`, `$\vec{p}$`, `$\vec{\theta}$`, `E_e`, `E_s`, `E_t`)` : construit une zone de déclenchement d'évenèment de collisioneur C, de position $\vec{p}$, rotation $\vec{\theta}$ et de fonction de rappel d'entrée $E_e : B, \Delta t$, de sortie $E_s : B, \Delta t$ et de mise à jour $E_t : B, \Delta t$ d'un corps B entre deux pas de temps Δt.
- méthode `Trigger::registerOn[Enter, Exit, Tick]Event(`E`)` : met à jour la fonction de rappel E pour l'un des évènements possibles.
- méthode `Trigger::triggerOn[Enter, Exit, Tick]Event()` : appelle la fonction de rappel associée à l'un des évènements possibles.
- méthode statique `Trigger::nullEvent()` : renvoie un rappel d'évènement E n'effectuant aucune opération lors d'un évènement (rappel par défaut).

Simulation (classe PhysicsEngine)

- méthode `PhysicsEngine::tick(`Δt`)` : mise à jour des évènements physiques de la simulation entre l'instant courant t et $t + \Delta t$.
- méthode `PhysicsEngine::addRigidBody(`B`)` : ajoute un corps solide B dans la simulation.
- méthode `PhysicsEngine::addForceField(`F`)` : ajoute un champ d'accélération F dans la simulation.
- méthode `PhysicsEngine::addTrigger(`T`)` : ajoute une zone de déclenchement T dans la simulation.
- méthode `PhysicsEngine::setCollisionRule(`$\{L_1, ..., L_n\}$`)` : active la collision entre les couches $L_1, ..., L_n$.

- méthode `PhysicsEngine::setNoCollisionRule(`$\{L_1, ..., L_n\}$`)` : désactive la collision entre les couches $L_1, ..., L_n$.
- méthode `PhysicsEngine::doLayersCollide(`L_1`,` L_2`)` $\rightarrow b$: retourne vrai si L_1 et L_2 ont une règle de collision, faux sinon.

5.8.2 Utilitaires géométriques

L'ensemble des fonctions géométriques utilisées dans ce projet sont définies dans *GeometryUtils.hpp*. Les paramètres de cette section ne seront pas explicités afin de maintenir la compacité de l'entièreté de cette section. Ces fonctions sont définies dans un espace de nom `geometry` afin de réduire toute pollution de l'espace de nommage global du moteur entier. Elles totalisent 1 600 lignes d'implémentation sans usage externe de librairies autres que GLM (OpenGL Mathematics).

Points intérieurs

- fonction `isPointInsideTetrahedron` : vérifie la présence d'un point à l'intérieur d'un tétraèdre.
- fonction `isPointInsideConvex` : vérifie la présence d'un point à l'intérieur d'un ensemble convexe.

Plus proche(s) point(s)

- fonction `closestPointOnLine` : détermine le point le plus proche d'un autre sur une ligne ou un segment.
- fonction `closestPointsOnLines` : détermine la paire de points les plus proches entre deux lignes ou segments.
- fonction `closestPointOnTriangle` : détermine le point le plus proche d'un autre sur un triangle.
- fonction `closestPointsLineTriangle` : détermine la paire de points les plus proches entre une ligne ou segment et un triangle.
- fonction `closestPointOnPolygon` : détermine le point le plus proche d'un autre sur un polygone.
- fonction `closestPointOnPlane` : détermine le point le plus proche d'un autre sur un plan.
- fonction `closestPointOnTetrahedron` : détermine le point le plus proche d'un autre sur un tétraèdre.
- fonction `closestPointsLineTetrahedron` : détermine la paire de points les plus proches entre une ligne ou segment et un tétraèdre.
- fonction `closestPointOnConvexSet` : détermine le point le plus proche d'un autre sur un ensemble convexe.
- fonction `closestPointsLineConvexSet` : détermine la paire de points les plus proches entre une ligne ou segment et un ensemble convexe.

Lancers de rayon ("Raycasting")

- structure `RaycastInfo(`$\vec{p}$`,` $\vec{n}$`,` t`)` : définit la présence d'une intersection lors du lancer de rayon et ses caractéristiques (point de contact $\vec{p}$, normale $\vec{n}$ et paramètre t).
- méthode statique `RaycastInfo::miss` : indique l'absence d'intersection au lancer de rayon.
- fonction `raycastTriangle` : retourne les résultats du lancer d'un rayon sur un triangle.
- fonction `raycastSphere` : retourne les résultats du lancer d'un rayon sur une sphère.
- fonction `raycastBox` : retourne les résultats du lancer d'un rayon sur un parallélépipède rectangle.
- fonction `raycastCapsule` : retourne les résultats du lancer d'un rayon sur une capsule.
- fonction `raycastTetrahedron` : retourne les résultats du lancer d'un rayon sur un tétraèdre.
- fonction `raycastConvexSet` : retourne les résultats du lancer d'un rayon sur un ensemble convexe.

Boîtes englobantes

- fonction `sphereBoundingBox` : retourne la boîte englobante d'une sphère.
- fonction `capsuleBoundingBox` : retourne la boîte englobante d'une capsule.
- fonction `convexSetBoundingBox` : retourne la boîte englobante d'un ensemble convexe.

Tests d'intersection (voir 5.4.3)

- structure `DiscreteIntersectionInfo(`$\vec{p}$`, `$\vec{n}$`, `δ`)` : définit la présence d'une intersection et ses caractéristiques (point de contact $\vec{p}$, normale $\vec{n}$ et pénétration δ tel que $\vec{p} = \vec{o} + t\vec{d}$, avec $\vec{o}$ l'origine du lancer de rayon et $\vec{d}$ sa direction).
- méthode statique `DiscreteIntersectionInfo::miss` : indique l'absence d'intersection.
- fonction `AABBAABBIntersection` : retourne les résultats de l'intersection de deux parallélépipèdes rectangles d'axes alignés.
- fonction `boxBoxIntersection` : retourne les résultats de l'intersection de deux parallélépipèdes rectangles.
- fonction `boxSphereIntersection` : retourne les résultats de l'intersection d'un parallélépipède rectangle et d'une sphère.
- fonction `boxCapsuleIntersection` : retourne les résultats de l'intersection d'un parallélépipède rectangle et d'une capsule.
- fonction `pointSphereIntersection` : retourne les résultats de l'intersection d'un point et d'une sphère.
- fonction `sphereSphereIntersection` : retourne les résultats de l'intersection de deux sphères.
- fonction `sphereCapsuleIntersection` : retourne les résultats de l'intersection d'une sphère et d'une capsule.
- fonction `sphereConvexIntersection` : retourne les résultats de l'intersection d'une sphère et d'un convexe.
- fonction `capsuleCapsuleIntersection` : retourne les résultats de l'intersection de deux capsules.
- fonction `capsuleConvexIntersection` : retourne les résultats de l'intersection d'une capsule et d'un convexe.
- fonction `capsuleConvexIntersection` : retourne les résultats de l'intersection de deux convexes.
- fonction `supportsIntersection` : retourne le polygone d'intersection du support de deux ensembles convexes, soit la surface de contact.

Volumes

- fonction `tetrahedronVolume` : retourne le volume d'un tétraèdre.
- fonction `sphereVolume` : retourne le volume d'une sphère.
- fonction `boxVolume` : retourne le volume d'un parallélépipède rectangle.
- fonction `cylinderVolume` : retourne le volume d'un cylindre.
- fonction `capsuleVolume` : retourne le volume d'une capsule.
- fonction `convexSetVolume` : retourne le volume d'un ensemble convexe.

Inerties (voir 5.5.1)

- fonction `shiftInertiaTensor` : application du théorème des axes parallèles afin de transformer le tenseur d'inertie.
- fonction `rotateInertiaTensor` : applique une rotation du tenseur d'inertie.
- fonction `tetrahedronInertiaTensor` : retourne le tenseur d'inertie d'un tétraèdre de masse 1.
- fonction `sphereInertiaTensor` : retourne le tenseur d'inertie d'une sphère de masse 1.
- fonction `boxInertiaTensor` : retourne le tenseur d'inertie d'un parallélépipède rectangle de masse 1.
- fonction `cylinderInertiaTensor` : retourne le tenseur d'inertie d'un cylindre de masse 1.
- fonction `capsuleInertiaTensor` : retourne le tenseur d'inertie d'une capsule de masse 1.
- fonction `convexSetInertiaTensor` : retourne le tenseur d'inertie d'un ensemble convexe de masse 1.

Transformations

- fonction `translatePoints` : applique une translation à un ensemble de points.
- fonction `transformPoints` : applique une translation et une rotation à un ensemble de points.

Autres fonctions

- fonction `sweepSortingFunction` : retourne une fonction de tri selon une direction.
- fonction `convexSetSupportPoint` : retourne un support d'un ensemble convexe (voir 5.4.5).

- fonction `convexSetSupportTriangle` : retourne les trois derniers points dans une direction .
- fonction `convexSetSupportPolygon` : retourne le polygone composé de l'ensemble des supports.
- fonction `polygonTriangulation` : retourne l'ensemble des triangles formant un polygone.
- fonction `convexSetTriangulation` : retourne l'ensemble des triangles formant la surface d'un ensemble convexe (voir 5.4.4).
- fonction `convexSetTetrahedralization` : retourne l'ensemble des tétraèdres formant un ensemble convexe (voir 5.4.4).
- fonction `sat` : algorithme du théorème des axes séparateurs (voir 5.4.5).
- fonction `gjk` : algorithme GJK et son extension EPA (voir 5.4.6 et 5.4.7).
- fonction `arbitraryOrthogonal` : retourne un vecteur orthogonal arbitraire à un autre vecteur.
- fonction `convexSetAverage` : retourne la moyenne des points d'un ensemble convexe.
- fonction `weightedConvexSetAverage` : retourne la moyenne pondérée des points d'un ensemble convexe.
- fonction `barycentricCoordinates` : retourne les coordonnées barycentriques d'un point sur un triangle.
- fonction `getBoxVertices` : retourne les sommets d'un parallélépipède rectangle en fonction de ses dimensions.
- fonction `capsuleHemispheres` : retourne les centroïdes des hémisphères d'une capsule en fonction de sa hauteur et de son rayon.
- fonction `arePointsOnSameSide` : retourne vrai si deux points sont du même côté d'un plan.

6 Conclusion

6.1 Complications et pistes d'améliorations

Nous avons eu quelques complications lors de l'implémentation de nos divers modules.

Pour le Clustered Rendering, les équations de division de la grille ont été difficiles à tester. De plus, le développement d'algorithmes avancés sur GPU, et plus précisément dans le cadre d'un système de rendu graphique, est difficile à déboguer. Nous avons cependant réussi à remplir nos objectifs pour ce module. Ce dernier peut être amélioré en passant le tri des lumières sur GPU et en utilisant des équations de division exponentielle de la profondeur.

Pour le système d'animation, il nous a été difficile de trouver des ressources expliquant précisément les processus nécessaires à l'implémentation d'animations squelettiques. La librairie ASSIMP, bien que vaste et très utilisée, a montré des problèmes de compatibilité importants pour le chargement de certains formats comme le FBX. Cependant, nous avons réussi à créer un système complet et fonctionnel de graphe d'animation, remplissant ainsi les objectifs de ce module.

Pour la recherche de chemin, nous n'avons pas réussi à générer le graphe de scène à partir d'un monde physique uniquement. De plus, nous n'avons pas eu le temps de correctement lié ce système à un agent capable de ce déplacement de façon naturelle entre ses destinations. L'abandon de notre membre travaillant principalement sur ce module nous a empêché de l'avancer davantage.

Finalement, le moteur physique met en évidence des résultats satisfaisants quant à ses performances et sa stabilité sur des corps à rotation fixe. Tandis que les objectifs concernant la détection et réponse à une collision ont été atteints, la stabilité complète du modèle n'a pas pu être maintenue intégralement, ce qui est discuté dans la section 5.6.3. Tout de même, ce moteur excède certains des attendus, puisque le problème ne relève que d'une faible portion de l'ensemble de l'implémentation, et il reste tout à fait possible de l'utiliser pour certaines applications plus limitées.

6.2 État du moteur après le projet

Bien que nous n'ayons pas eu le temps de finir le jeu de démonstration, nous considérons que notre objectif principal de rendre le moteur Vulpine plus avancé est réussi. De nombreuses pistes d'améliorations et d'évolutions sont encore possibles, mais l'état actuel du moteur en fait un logiciel moderne, léger et capable de grandement aidé à la création d'œuvres vidéo ludiques.

Références

[1] Ian Dunn et Zoë Wood. « Chapter 2 : The Graphics Pipeline ». In : *Graphics Programming Compendium* (2021). URL : https://graphicscompendium.com/intro/01-graphics-pipeline.

[2] Ola Olsson, Markus Billeter et Ulf Assarsson. « Clustered Deferred and Forward Shading ». In : *High Performance Graphics* (2012). URL : https://www.cse.chalmers.se/~uffe/clustered_shading_preprint.pdf.

[3] Ángel Ortiz. « A Primer On Efficient Rendering Algorithms & Clustered Shading ». In : *High Performance Graphics* (2018). URL : http://www.aortiz.me/2018/12/21/CG.html.

[4] Eike Anderson. *Real-Time Character Animation for Computer Games*. Jan. 2001. URL : https://www.researchgate.net/publication/250362465_Real-Time_Character_Animation_for_Computer_Games.

[5] « Skeletal Animation ». In : *Beginning XNA 3.0 Game Programming : From Novice to Professional*. Berkeley, CA : Apress, 2009, p. 299-336. ISBN : 978-1-4302-1818-0. DOI : 10.1007/978-1-4302-1818-0_12. URL : https://doi.org/10.1007/978-1-4302-1818-0_12.

[6] et Sheridan Stephen Graham Ross McCabe Hugh. « Pathfinding in Computer Games ». In : (2003). DOI : 10.21427/D7ZQ9J. URL : https://arrow.tudublin.ie/itbj/vol4/iss2/6.

[7] Xiao Cui et Hao Shi. « A*-based pathfinding in modern computer games ». In : *International Journal of Computer Science and Network Security* 11.1 (2011), p. 125-130.

[8] Peter E Hart, Nils J Nilsson et Bertram Raphael. « A formal basis for the heuristic determination of minimum cost paths ». In : *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), p. 100-107.

[9] Nils J Nilsson. *The quest for artificial intelligence*. Cambridge University Press, 2009, p. 220-221.

[10] Matthias Müller et al. « Position Based Dynamics ». In : *3rd Workshop in Virtual Reality Interactions and Physical Simulation (VRIPHYS)*. 2006. URL : https://matthias-research.github.io/pages/publications/posBasedDyn.pdf.

[11] David Baraff. « Physically Based Modeling Rigid Body Simulation ». In : 2003. URL : https://graphics.pixar.com/pbm2001/pdf/notesg.pdf.

[12] Brian Mirtich. « Impulse-based Dynamic Simulation of Rigid Body Systems ». Thèse de doct. University of California at Berkeley, 1996. URL : https://people.eecs.berkeley.edu/~jfc/mirtich/thesis/mirtichThesis.pdf.

[13] Game Technology Group. « Physics Tutorial 2 : Numerical Integration Methods ». 2017. URL : https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/physics2numericalintegrationmethods/2017%20Tutorial%202%20-%20Numerical%20Integration%20Methods.pdf.

[14] Nelson Souto. *Video Game Physics Tutorial - Part II : Collision Detection for Solid Objects*. 2015. URL : https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects.

[15] Gabor Szauer. *Game Physics Cookbook*. Packt Publishing, 2017.

[16] David Baraff. « Dynamic Simulation of Non-Penetrating Rigid Bodies ». Thèse de doct. Computer Science Department, Cornell University, 1992. URL : https://www.cs.cmu.edu/~baraff/papers/index.html.

[17] Daniel J. Tracy, Samuel R. Buss et Bryan M. Woods. « Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal ». In : *IEEE Virtual Reality Conference*. 2009. URL : https://mathweb.ucsd.edu/~sbuss/ResearchWeb/EnhancedSweepPrune/SAP_paper_online.pdf.

[18] Gino van der Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers, 2004.

[19] Elmer G. Gilbert, Daniel W. Johnson et S. Sathiya Keerthi. « A fast procedure for computing the distance between complex objects in three-dimensional space ». In : *IEEE Journal on Robotics and Automation* (1988). URL : https://graphics.stanford.edu/courses/cs448b-00-winter/papers/gilbert.pdf.

[20] Gino van den Bergen. « Proximity Queries and Penetration Depth Computation on 3D Game Objects ». 2001. URL : https://graphics.stanford.edu/courses/cs468-01-fall/Papers/van-den-bergen.pdf.

[21] J. L. Bentley et T. A. Ottmann. « Algorithms for reporting and counting geometric intersections ». In : *IEEE Transactions on Computers* (1979). URL : https://www.semanticscholar.org/paper/Algorithms-for-Reporting-and-Counting-Geometric-Bentley-Ottmann/62278010df9000e5496ea2523d46e98bb2206c56.

[22] Braxton Carrigan. « Triangulations and Simplex Tilings of Polyhedra ». Thèse de doct. Graduate Faculty of Auburn University, 2012. URL : https://etd.auburn.edu/bitstream/handle/10415/3199/TriangulationsandTilings.pdf?sequence=2.

[23] Winter.dev. *EPA : Collision response algorithm for 2D/3D*. 2020. URL : https://winter.dev/articles/epa-algorithm.

[24] Ming-Lun 'Allen' Chou. *Game Physics : Contact Generation – EPA*. 2013. URL : https://allenchou.net/2013/12/game-physics-contact-generation-epa/.

[25] A. R. Abdulghany. « Generalization of parallel axis theorem for rotational inertia ». In : *American Journal of Physics* (2017). URL : https://pubs.aip.org/aapt/ajp/article/85/10/791/1041336/Generalization-of-parallel-axis-theorem-for.

[26] Bojan Lovrovic. *Capsule Inertia Tensor*. 2015. URL : https://www.gamedev.net/tutorials/programming/math-and-physics/capsule-inertia-tensor-r3856/.

[27] F. Tonon. « Explicit Exact Formulas for the 3-D Tetrahedron Inertia Tensor in Terms of its Vertex Coordinates ». In : *Journal of Mathematics and Statistics* (2005). URL : https://docsdrive.com/pdfs/sciencepublications/jmssp/2005/8-11.pdf.

[28] Game Technology Group. « Physics - Collision Response ». 2017. URL : https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/5collisionresponse/Physics%20-%20Collision%20Response.pdf.

[29] Erin Catto. « Modeling and Solving Constraints ». 2009. URL : https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc09/slides/04-GDC09_Catto_Erin_Solver.pdf.

[30] Nelson Souto. *Video Game Physics Tutorial - Part III : Constrained Rigid Body Simulation*. 2015. URL : https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation.