



TER HAI823I

**Rapport Simulation de Fumée**

Master 1 IMAGINE  
Université de Montpellier

20 mai 2024

**Auteurs :**

Benjamin SERVA  
Brian DELVIGNE  
Ibrahim HARCHA

**Encadrant :**

Noura Faraj

## **Remerciements**

*Nous tenons à exprimer notre gratitude :*

*Aux professeurs d'université qui nous ont accompagnés tout au long de notre première année de Master, pour leur suivi, leur soutien et leurs conseils précieux.*

*À madame Noura Faraj, pour avoir accepté le sujet que nous avons proposé, pour ses conseils sur les sujets à aborder et les idées à explorer.*

*Aux chercheurs et autres internautes qui partagent leurs travaux sur internet. Grâce à leur générosité, nous avons pu accéder à des informations précieuses et fiables.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Nos motivations . . . . .	5
1.2	Glossaire . . . . .	5
<b>2</b>	<b>Organisation</b>	<b>6</b>
2.1	Diagramme de Gantt . . . . .	6
2.2	Outils utilisé . . . . .	6
<b>3</b>	<b>Articles</b>	<b>7</b>
3.1	Article 1 . . . . .	7
3.2	Article 2 . . . . .	8
3.2.1	Résumée . . . . .	8
3.2.2	Point Clés . . . . .	9
3.3	Article 3 . . . . .	10
<b>4</b>	<b>Explication sur la programmation graphique</b>	<b>12</b>
<b>5</b>	<b>Structure du Programme</b>	<b>15</b>
<b>6</b>	<b>Les particules</b>	<b>17</b>
6.1	Structure . . . . .	17
6.2	Méthode d'envoie des particules . . . . .	18
6.2.1	Bind du buffer . . . . .	18
6.2.2	Envoie du buffer au shader . . . . .	19
<b>7</b>	<b>Effets visuels</b>	<b>20</b>
7.1	Texture . . . . .	20
7.2	Transparence . . . . .	20
7.2.1	Comment est-elle représenté . . . . .	20
7.2.2	Intérêt . . . . .	20
7.2.3	Amélioration : Discard . . . . .	21
7.2.4	Problématique importante . . . . .	21
7.3	Fonction de load de texture . . . . .	23
7.4	Mélange de texture . . . . .	24
7.5	Bruit de perlin . . . . .	25

<b>8</b>	<b>Interaction avec un cube</b>	<b>29</b>
<b>9</b>	<b>Mécanique des fluides</b>	<b>31</b>
9.1	Navier-Stokes . . . . .	31
9.2	Gausse-Seidel . . . . .	31
<b>10</b>	<b>Mesh</b>	<b>32</b>
10.1	Méthode Utilisé . . . . .	32
10.2	Flotabilité . . . . .	34
<b>11</b>	<b>Scène et contrôle utilisateur</b>	<b>35</b>
11.1	Scene . . . . .	35
11.1.1	Modèle d'illumination de Phong . . . . .	35
11.1.2	Calcul des normales pour une particules . . . . .	36
11.2	Interaction utilisateur avec ImGui . . . . .	37
<b>12</b>	<b>Résultats</b>	<b>40</b>
12.1	Démonstration des forces . . . . .	40
12.2	Analyse des résultats . . . . .	42
<b>13</b>	<b>Problèmes rencontrés</b>	<b>43</b>
13.1	Base de code . . . . .	43
13.2	Particules parasites . . . . .	43
13.3	Chargement de la texture . . . . .	44
<b>14</b>	<b>Conclusion</b>	<b>45</b>
14.1	Objectifs Atteint ? . . . . .	45
14.2	Ce que nous avons appris . . . . .	45
14.3	Amélioration futures . . . . .	45
<b>15</b>	<b>Références</b>	<b>47</b>
15.1	Recherches . . . . .	47
15.2	Algorithmes . . . . .	47
15.3	Articles . . . . .	47

## List of Figures

1	diagramme de Gantt . . . . .	6
2	architecture pipeline graphique . . . . .	12
3	représentation simpliste du programme . . . . .	16
4	structure utilisé pour stocker les particules . . . . .	17
5	fonction pseudo aléatoire qui génère la position de départ .	18
6	fonction pseudo aléatoire qui génère le déplacement . . . . .	18
7	bout de code permettant le bind du buffer . . . . .	19
8	section de code permettant l'envoie des particules au shaders	19
9	Schéma illustratif. . . . .	20
11	texture avec utilisation de la transparence et du discard avec un seuil de 0.1 . . . . .	21
12	transparence traité dans le mauvais ordre . . . . .	22
13	transparence traité dans le bon ordren . . . . .	22
14	fonction modifié permettant de charger la transparence .	23
15	texture utilisée . . . . .	24
17	fonction rand pour le bruit de Perlin . . . . .	25
18	fumée sans utilisation du bruit de Perlin . . . . .	26
19	fumée avec utilisation du bruit de Perlin d'amplitude 0.15	27
20	utilisation abusive du bruit de Perlin pour une amplitude de 1.5 . . . . .	28
21	Collision avec le cube. . . . .	29
22	Autre exemple de collision avec le cube. . . . .	30
26	mesh de suzanne utilisant la deuxième méthode . . . . .	33
27	menu 1 . . . . .	38
28	menu 2 . . . . .	38
29	menu 3 . . . . .	38
30	menu 4 . . . . .	39
31	Il y'a un tas de résultat différents que l'on peut obtenir, puis ce qu'il y'a beaucoup de mouvement il est bien plus intéressant d'observer la simulation en démonstration qu'avec des images . . . . .	41

# 1 Introduction

## 1.1 Nos motivations

Nous avons engagé ce projet en raison de notre profond intérêt pour les simulations 3D. Notre objectif principal était de développer une simulation réaliste de la fumée en utilisant des techniques avancées. Ce sujet nécessite une bonne compréhension des dynamiques des fluides et des techniques de rendu. C'était aussi un bon moyen pour nous de renforcer et améliorer nos compétences 3d acquise dans le premier semestre.

## 1.2 Glossaire

Tout au long du rapport nous utilisons certains mots particuliers qui sont définis ici :

- vec3 : vecteur sur 3 dimension
- vec4 : vecteur sur 4 dimension
- GPU : (Graphical Processing Unit) processeur spécialisé dans le rendu graphique.
- pixel : Un pixel est la plus petite unité de base composant une image numérique affichée sur un écran
- shader : programme exécuté sur le GPU qui détermine comment les pixels et les vertices sont dessinés à l'écran. Durant ce projet on utilise 3 types de shader différents :
  1. vertex shader : traite chaque sommet individuellement pour les transformations géométriques et la manipulation des propriétés des vertices.
  2. geometry shader : transforme, divise, ou génère de nouvelles primitives (points, lignes, triangles) pour des effets complexes.
  3. fragment shader : calcule la couleur et d'autres attributs de chaque pixel pour déterminer l'apparence finale des surfaces.
- buffer : zone de mémoire temporaire pour stocker des données comme les vertices, indices, couleurs, textures.
- mesh : en français maillage, permet de stocker une forme 3D.

## 2 Organisation

Durant le projet nous nous sommes réunis avec notre encadrant de façon bi-hebdomadaire que ce soit pour montrer l'avancement de notre travail, mais aussi pour demander des conseils sur les difficultés rencontrées ainsi que pour fixer les objectifs des prochaines réunions.

### 2.1 Diagramme de Gantt

Voici le diagramme de Gantt réalisé au tout début du projet présentant les différentes étapes de conception.

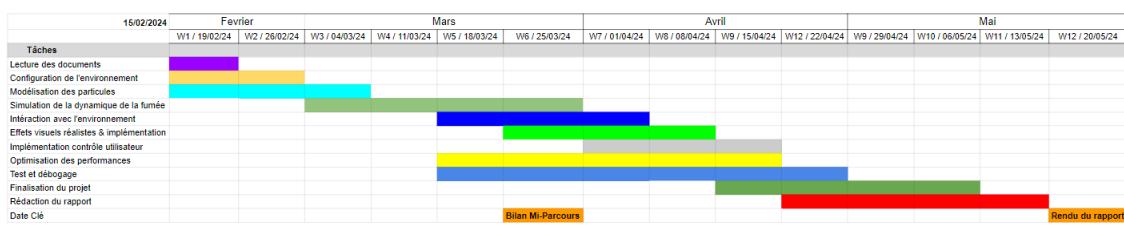


Figure 1: diagramme de Gantt

### 2.2 Outils utilisé

- Discord : pour communiquer entre nous mais aussi avec notre encadrant
- Github : pour stocker et combiner le travail [lien du git ici](#)
- Bibliothèques C++ utilisées :
  1. ImGui
  2. GLM
  3. GLFW

## 3 Articles

Chaque membre du groupe à du choisir et lire un article scientifique à propos de notre thème, permettant de récolter des informations et d'avoir une bonne base de connaissance sur celui-ci. Nous allons vous résumer ce que nous avons compris de ces articles et ce qui nous a servi dans la réalisation de ce projet.

### 3.1 Article 1

#### Target Particle Control of Smoke Simulation

##### Introduction

Le principal défi de la modélisation des phénomènes naturels pour les effets spéciaux réside dans le contrôle. Sans un système de contrôle avancé, les artistes doivent ajuster de nombreux paramètres physiques et ajouter des forces pour obtenir l'effet désiré. Idéalement, les systèmes de contrôle doivent être intuitifs et efficaces, facilitant la tâche des artistes pour atteindre les résultats souhaités. L'objectif est de développer un système de contrôle suffisamment rapide pour une utilisation en temps réel, privilégiant la vitesse à la précision physique, ce qui est crucial pour les jeux et les médias interactifs.

##### Approche Proposée

L'approche divise le champ de vitesse en deux composants :

Vitesse Principale : Un champ de vitesse basse fréquence qui oriente le volume de fumée vers une forme cible. Vitesse de Détail : Un modèle de turbulence utilisant des particules de vortex pour générer un mouvement fluide à fine échelle.

##### Champ de Vitesse Principal

Pour générer ce champ, des particules de contrôle sont placées uniformément dans le volume de fumée, tandis que des particules cibles sont générées dans la forme cible. Un algorithme d'optimisation fait correspondre les particules de contrôle aux particules cibles, créant une force d'attraction qui guide la fumée vers la forme souhaitée. Cette méthode interpole les vitesses des particules de contrôle dans l'espace pour produire un champ de vitesse fluide.

## **Champ de Vitesse de Détail**

Pour ajouter de la turbulence, une méthode de particules de vortex est utilisée. Ce modèle induit des mouvements de rotation caractéristiques de la fumée turbulente, combinant ce champ avec le champ de vitesse principal pour obtenir un mouvement fluide réaliste. **Implémentation et Résultats**

L'algorithme est décrit en plusieurs étapes : l'initialisation des particules, le calcul des champs de vitesse, la mise à jour des particules et le rendu. Les résultats montrent que cette approche permet de contrôler efficacement la forme de la fumée tout en maintenant une performance interactive, essentielle pour les applications en temps réel.

## **Conclusion**

Les contributions spécifiques de cette recherche incluent :

Un algorithme unique de correspondance de forme basé sur des points. Une méthode d'évaluation directe efficace et parallèle pour les particules de vortex. Une méthode combinant des particules de vortex avec une vitesse de correspondance de forme pour le contrôle ciblé de la fumée.

Cette approche permet de diriger la fumée vers une forme cible tout en ajoutant des détails turbulents réalistes, avec une efficacité computationnelle permettant des simulations interactives.

## **3.2 Article 2**

### **Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Based Methods :**

#### **3.2.1 Résumée**

Ce tutoriel vise à simplifier les concepts complexes et les détails de mise en œuvre de la simulation de fluides pour les graphiques par ordinateur, les rendant accessibles aux étudiants de premier cycle et aux praticiens sans une formation mathématique approfondie. Le document s'inspire fortement du livre complet de Robert Bridson "Fluid Simulation for Computer Graphics" tout en offrant un guide plus abordable et en étendant la discussion aux méthodes basées sur les particules telles que la Smoothed Particle Hydrodynamics (SPH).

### 3.2.2 Point Clés

1. Types de Fluides : L'article distingue entre les fluides incompressibles (liquides comme l'eau) et compressibles (gaz comme l'air). Les fluides incompressibles sont souvent traités comme tels pour simplifier les simulations, bien que tous les fluides soient quelque peu compressibles.
2. Techniques de Simulation : Deux principales techniques de simulation de fluides sont abordées : Basée sur la Grille (Eulerienne) : Utilise une grille fixe pour calculer les propriétés des fluides à des points discrets. Elle est précise mais coûteuse en termes de calcul et peut souffrir de pertes de masse. Basée sur les Particules (Lagrangienne) : Simule les fluides comme des particules discrètes, ce qui est plus rapide et conserve mieux la masse mais peut avoir des difficultés à représenter des surfaces lisses.
3. Équations de Navier-Stokes : Les équations gouvernant le mouvement des fluides sont présentées, avec un focus sur les équations de Navier-Stokes incompressibles. L'article simplifie ces équations en négligeant la viscosité pour une mise en œuvre plus facile, résultant en les équations d'Euler.
4. Aperçu de l'Algorithme pour la Simulation Basée sur la Grille : Initialisation : Mettre en place la grille de simulation avec les conditions initiales du fluide. Étapes de Temps : Itérer à travers chaque image de l'animation, en mettant à jour les propriétés du fluide à travers les étapes d'advection, de projection de pression, et d'advection de surface. Grille Échelonnée : La technique de la grille MAC (Marker-and-Cell) est utilisée pour stocker différentes quantités (par exemple, la vitesse, la pression) à différents endroits dans chaque cellule de la grille pour plus de précision.
5. Condition CFL : La condition de Courant-Friedrichs-Lowy (CFL) est utilisée pour déterminer le pas de temps maximal autorisé afin d'assurer la stabilité et la précision de la simulation.
6. Advection : L'advection semi-lagrangienne est utilisée pour mettre à jour les propriétés du fluide, ce qui implique de tracer les particules en arrière dans le temps pour déterminer leurs nouvelles positions. Un intégrateur d'ordre supérieur est recommandé pour une meilleure précision.

7. Détails de Mise en Œuvre : Les aspects pratiques de la mise en œuvre de la simulation, y compris les structures de données, le calcul des pas de temps, et la gestion des conditions aux limites, sont discutés pour guider le lecteur dans le développement d'un simulateur de fluides fonctionnel.

Le document sert de guide pratique pour ceux qui s'intéressent à la mise en œuvre de simulations de fluides en graphiques par ordinateur, équilibrant les bases théoriques avec des étapes de mise en œuvre concrètes.

### 3.3 Article 3

#### **Visualization of smoke using particle systems**

##### **Introduction**

L'article traite de l'utilisation courante des techniques de visualisation dans divers contextes tels que les films, les jeux vidéo et les publicités. Il met en avant la capacité des technologies modernes à produire des phénomènes visuels réalistes et se concentre sur les systèmes de particules comme méthode pour atteindre ce réalisme. L'étude compare des systèmes de particules simples et avancés en termes de leur efficacité à créer des visualisations réalistes.

##### **Contexte**

Le rendu en temps réel, comme dans les jeux vidéo, impose des contraintes de performance plus strictes comparé aux effets pré-rendus utilisés dans les films. La complexité des systèmes de particules varie en conséquence. Par ailleurs, la visualisation de la fumée dépend de la source et de l'échelle de la fumée. La fumée à grande échelle, comme celle des explosions, se comporte différemment de la fumée à petite échelle, comme celle des bougies par exemple.

##### **Travaux connexes**

La fumée se comporte comme des gaz légers et est influencée par les forces environnantes plutôt que par des dynamiques internes. Les techniques de dynamique des fluides computationnelle (CFD), telles que les équations de Navier-Stokes, sont souvent utilisées pour modéliser ce comportement. D'autre part, les méthodes avancées impliquent l'utilisation de

particules pour créer des lignes de courant et des surfaces NURBS pour une visualisation réaliste de la fumée à petite échelle.

## Approche

L'implémentation utilise C++ et OpenGL, et est composé de deux composants principaux : la particule et le système de particules. Chaque particule possède des propriétés telles que la couleur, la taille, la durée de vie, la localisation et la vitesse. Le système de particules gère ces particules, en ajoutant de nouvelles à chaque cycle pour créer un flux continu. La classe principale lie tous ces éléments ensemble, ce qui permet de gérer le dessin et la mise à jour des particules.

## Résultats

L'étude présente diverses visualisations de fumée avec différentes tailles de particules et taux d'émission. Bien que les résultats ne soient pas entièrement réalistes, ils montrent un potentiel, certaines configurations semblent plus réalistes que d'autres. Cet article note également que l'augmentation du nombre de particules peut améliorer le réalisme, mais cela impose également des contraintes de performance.

## Conclusion

Les résultats visuels sont quelque peu réalistes mais peuvent être améliorés. Plus précisément l'apparence et le comportement des particules de fumée peuvent être améliorés en appliquant la dynamique des fluides et en utilisant des formes de particules plus complexes. L'intégration de la dynamique des fluides permet une meilleure interaction des particules et l'expérimentation avec différentes formes et couleurs de particules pour améliorer le réalisme. Par ailleurs, l'article souligne l'importance de trouver un équilibre entre la fidélité visuelle et l'efficacité computationnelle, surtout dans les applications en temps réel comme les jeux vidéo.

## 4 Explication sur la programmation graphique

Avant toutes choses il est bien important de comprendre comment fonctionne la programmation graphique utilisant OpenGL, et comment on utilise les shaders.

Le schéma présenté ci-dessous illustre le pipeline graphique moderne d'OpenGL 4.3, qui est une série de phases ou d'étapes que les données graphiques traversent pour être transformées en pixels visibles à l'écran.

### Pipeline Moderne

OpenGL 4.3

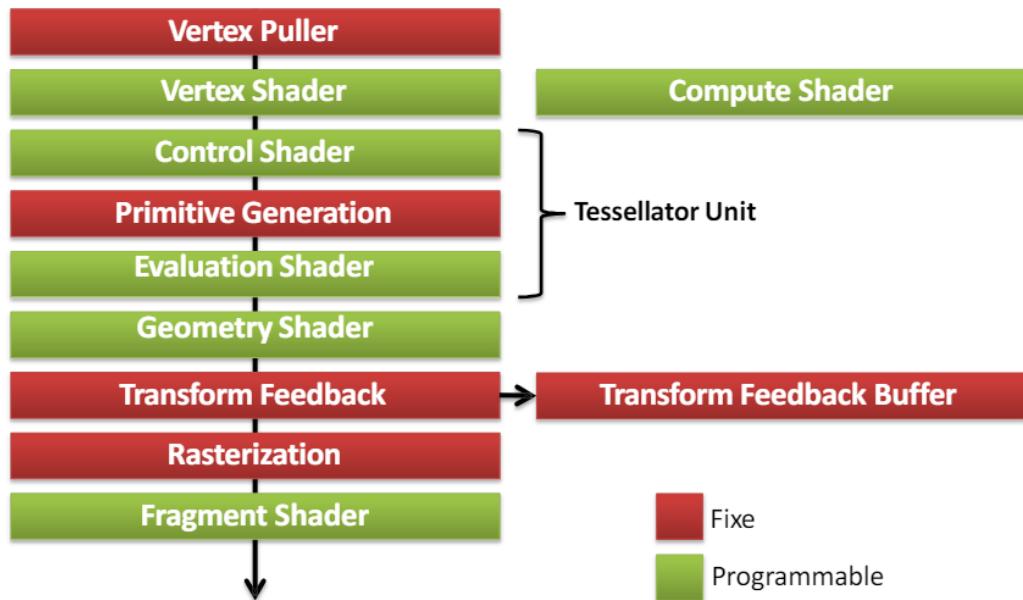


Figure 2: architecture pipeline graphique

#### 1. Vertex Puller (Fixe) :

- Récupère les données des sommets (vertices) depuis les buffers mémoire vers le pipeline pour traitement.

#### 2. Vertex Shader (Programmable) :

- Transforme les coordonnées des sommets en coordonnées de clip space. C'est ici que les transformations géométriques (comme la transformation de modèle, de vue, et de projection) sont appliquées.

#### 3. Control Shader (Programmable) :

- Partie du tessellateur qui contrôle le niveau de tessellation. Il détermine comment une primitive sera subdivisée.

#### 4. Primitive Generation (Fixe) :

- Génère de nouvelles primitives en subdivisant les primitives d'entrée selon les niveaux de tessellation spécifiés.

#### 5. Evaluation Shader (Programmable) :

- Évalue les nouvelles positions des sommets créés par la tessellation. Applique des transformations supplémentaires si nécessaire.

#### 6. Geometry Shader (Programmable) :

- Prend des primitives (points, lignes, triangles) en entrée et peut générer de nouvelles primitives ou modifier celles existantes. Utile pour les effets comme les ombrages volumétriques ou la génération de géométrie.

#### 7. Transform Feedback (Programmable) :

- Capture les résultats intermédiaires des shaders de vertex, de tessellation ou de géométrie, et les enregistre dans un buffer pour un usage ultérieur.

#### 8. Rasterization (Fixe) :

- Convertit les primitives en fragments (potentiels pixels) pour être traités par les shaders de fragment. Applique les opérations de découpage et de perspective.

#### 9. Fragment Shader (Programmable) :

- Calcule la couleur et les autres attributs de chaque fragment. C'est ici que les textures, l'éclairage et les autres effets visuels sont appliqués.

#### 10. Compute Shader (Programmable) :

- Shader généraliste qui peut être exécuté indépendamment du pipeline graphique traditionnel pour effectuer des calculs de traitement massivement parallèles. Utilisé pour des tâches de calcul complexes qui ne sont pas nécessairement liées au rendu graphique.

## Résumé

Le pipeline graphique moderne d'OpenGL 4.3 permet une grande flexibilité grâce à ses nombreuses étapes programmables (en vert), permettant aux développeurs de contrôler et de personnaliser chaque étape du traitement graphique. Les étapes fixes (en rouge) assurent des opérations essentielles et standardisées. Ainsi, cette architecture permet des rendus graphiques optimisés pour diverses applications graphiques en temps réel.

De plus il est important de noter que l'on peut transmettre n'importe quel variable au shader, et c'est pour cela que l'on peut grâce à notre interface utilisateur modifié en temps réel les paramètres de la fumée et autres.

## 5 Structure du Programme

Notre Programme va posséder une game loop. Pour que vous compreniez bien voici une rapide définition :

**Définition de la Game Loop** Une game loop est une boucle itérative centrale dans un jeu vidéo qui gère le cycle de vie de l'application en réalisant les actions suivantes de manière continue :

Gestion des Entrées : Récupère et traite les entrées des utilisateurs (clavier, souris, contrôleurs, etc.).

Mise à jour de l'état du Jeu : Met à jour la logique du jeu, y compris les mouvements des personnages, les règles de jeu, la physique, et autres mécanismes en fonction des entrées et du temps écoulé.

Rendu Graphique : Dessine les graphismes à l'écran, en actualisant l'affichage pour refléter les changements de l'état du jeu.

Vu que la fumée est quelque chose qui se déplace et a un aspect qui évolue au cours du temps, l'utilisation de la game loop était le plus adapté.

```

#include
.
.
.

#define

int main(){
    déclaration
    pré calcul possible

    while(){
        fenêtre imgui
        les particules
        tous les envois au shaders
    }

    nettoyage de tous les buffers et autres

    return 0;
}

```

Figure 3: représentation simpliste du programme

1. Avant le main : on aura tous les inclusions des bibliothèques externes utiles, ainsi que les déclarations des différents variables globales possibles. Mais aussi les déclarations de structures et de fonctions.
2. Dans le main avant la boucle while qui représente la game loop : on va pouvoir déclarer et créer tous notre environnement externe qui n'évoluera pas. On peut aussi faire des pré calcul important pour réduire le temps de calcul du programme. Vous le verrez plus tard mais pour le mesh on fait tous les chargements et calcul de position dans cette section ce qui fait que quand on veut afficher un mesh en fumée ça se fait instantanément.
3. dans cette boucle se trouve tous ce qui doit être fait à chaque frame.

## 6 Les particules

Les particules actuelles sont gérées d'une manière différentes à laquelle nous pensions au début de ce projet. Nous parlerons de cela plus en détail dans la partie consacrée aux problèmes rencontrés. Le programme est basé sur le fait que l'information d'une particule est détenu par son indice. Cette structure contient tous les vecteurs détenant les informations sur les particules. Par exemple la position et la vitesse initiale de la première particule est détenu dans Position[0] et baseVelocity[0].

Il est important de noter que le traitement des particules se fait dans un shader exclusif à celle-ci. Ensuite l'utilisation des vector de données est très pratique dans notre cas car ils nous permettent de supprimer une particule sans avoir à recréer le tableau (méthode `erase()` de la classe `vector`)

### 6.1 Structure

```
struct Particle{
    std::vector<glm::vec3> position;
    std::vector<float> life;
    std::vector<glm::vec3> deplacement;
    std::vector<float> baseVelocity;
    std::vector<float> densite;
};
```

Figure 4: structure utilisé pour stocker les particules

Au début du programme on crée donc en dehors de la game loop notre structure.

Ensuite à chaque frame on va générer un certain nombre de nouvelles particules, qui à leur création vont avoir une position aléatoire, un déplacement aléatoire, une durée vie fixe, une vitesse fixe et une densité fixe.

Quand on dit fixe, ce n'est pas la même valeur pour toutes les particules, c'est simplement la valeur actuelle de la variable concerné (elle peut évoluer si on la change en temps réel dans le menu).

Pour la position et le déplacement ce n'est pas vraiment aléatoire, c'est juste une valeur aléatoire dans un intervalle défini par nos soins.

```

vec3 generate_position(){
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<float> dis(-10.0f, 10.0f);

    float y=-2.;
    float x=dis(gen)*0.01;
    float z=dis(gen)*0.01;
    return vec3(x,y,z);
}

```

Figure 5: fonction pseudo aléatoire qui génère la position de départ

```

vec3 generate_deplacement(){
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<float> dis(-2.0f, 2.0f);
    std::uniform_real_distribution<float> dis2(1.0f, 10.0f);
    float y=0.0005f*dis2(gen);
    float z=0.f;
    float x=dis(gen)*0.001;
    while(x==0){
        x=dis(gen)*0.001;
    }
    return vec3(x,y,z);
}

```

Figure 6: fonction pseudo aléatoire qui génère le déplacement

Enfin à chaque frame, on va regarder réduire la vie de chaque particules de un et si elles ont encore de la vie alors on leur applique leurs déplacement et différentes forces externes.

Il est important de noter que l'on vient de vous présenter un générateur de particules qui pourrait aussi fonctionner pour générer du feu de façon réaliste. Mais c'est surtout les différentes méthodes que l'on va vous présenter par la suite qui sont propre à la simulation de fumée réaliste.

## 6.2 Méthode d'envoie des particules

### 6.2.1 Bind du buffer

La méthode de bind du buffer est légèrement différentes que pour envooyer un objet qui ne va pas être actualisé. Lors du remplacement de l'ensemble de la mémoire, envisager d'utiliser glBufferSubData plutôt que complètement recréer la mémoire de données avec glBufferData. Cela permet d'éviter le coût de réaffecter la mémoire de données.

```

glGenBuffers(2, &particleBuffer);
glBindBuffer(GL_ARRAY_BUFFER, particleBuffer);
glBufferData(GL_ARRAY_BUFFER, particles.position.size() * sizeof(glm::vec3), nullptr, GL_DYNAMIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, particleBuffer);
glBufferSubData(GL_ARRAY_BUFFER, 0, particles.position.size() * sizeof(glm::vec3), &particles.position[0]);

```

Figure 7: bout de code permettant le bind du buffer

### 6.2.2 Envoie du buffer au shader

L’envoie reste globalement la même à la différence que dans la fonction **glDrawArrays** le premier argument est **GL\_POINTS**. En temps normal quand on veut dessiner une forme géométrique on va lui passer tous les sommets mais aussi les indices des sommets de chaque triangles de la forme, pour finir on utilise l’option **GL\_TRIANGLES**, ça va donc dessiner l’objet. Mais dans le cas des particules on ne passe que les positions et on dessine chaque particules comme un point.

```

glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, particleBuffer);
glVertexAttribPointer(
    0,           // attribute
    3,           // size (vec3)
    GL_FLOAT,    // type
    GL_FALSE,    // normalized?
    0,           // stride
    (void*)0     // array buffer offset
);
glPointSize(particleSize);

glDrawArrays(GL_POINTS, 0, particles.position.size());

```

Figure 8: section de code permettant l’envoie des particules au shaders

Une fonction utilise le **glPointSize**, elle va permettre de gérer la taille des particules que l’on dessine.

## 7 Effets visuels

### 7.1 Texture

Pour les textures nous avons utilisé un geometry shader, ce dernier permet de générer un quad pour chaque particule ainsi que des coordonnées de texture. Dans le fragment shader, la texture est chargée et appliquée à chaque quad généré précédemment.

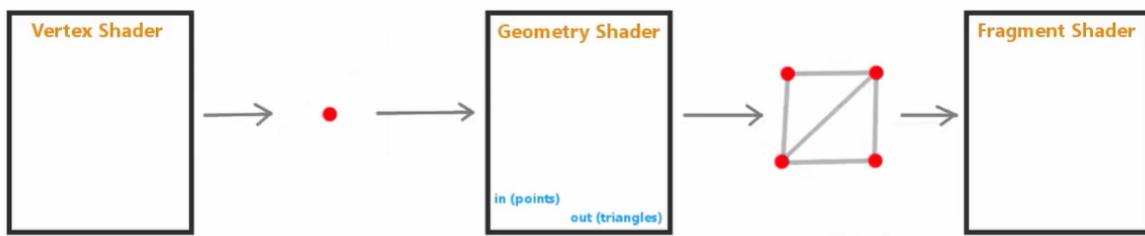


Figure 9: Schéma illustratif.

Comment la texture est-elle appliquée ?

Grâce au coordonnées de textures (`vec2`), qui doivent toujours être normalisés, c'est à dire compris entre 0 et 1. Ensuite pour déterminer la couleur du fragment on va utiliser la fonction `texture` qui en prend en entrée la texture transmise depuis la main et les coordonnées de textures. Cette fonction nous donnera en sortie un `vec4`.

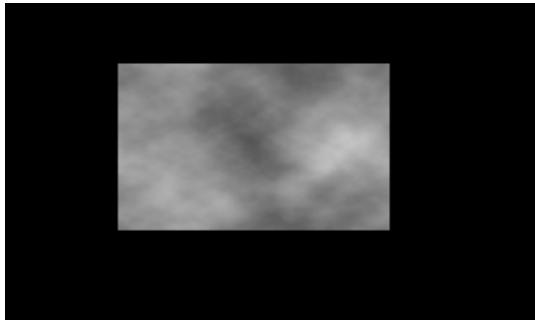
### 7.2 Transparence

#### 7.2.1 Comment est-elle représenté

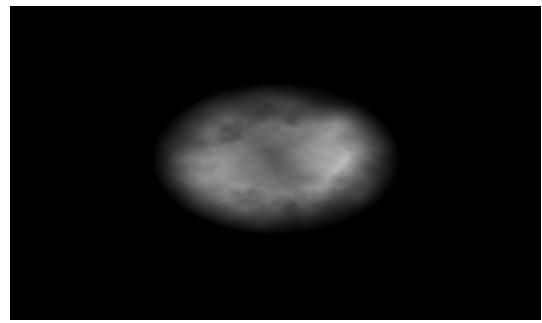
Dans le fragment shader on doit sortir une couleur qui sera la couleur affiché pour le pixel traité. Stockage de la couleur : sous forme de `vec3` où le premier élément correspond à la constante R (Red), le deuxième élément correspond à la constante G (Green) et enfin la troisième qui correspond au B (Blue). Si on veut on la possibilité de sortir un `vec4 = vec3 + 1` dimension, dimension supplémentaire qui correspond à la Transparence.

#### 7.2.2 Intérêt

Notre texture à un canal de transparence on plaque donc la texture sur un quad mais à l'image on n'a pas un quad.



(a) texture sans utilisation de la transparence



(b) texture avec utilisation de la transparence

### 7.2.3 Amélioration : Discard

Quand on appelle la fonction `discard()` dans le fragment shader on va tous simplement ignorer le fragment que l'on traite. On peut donc en fonction d'un seuil que l'on compare avec la transparence appliquer un discard. Utile car on voit encore une sorte d'auréole autour de notre particule de fumée.

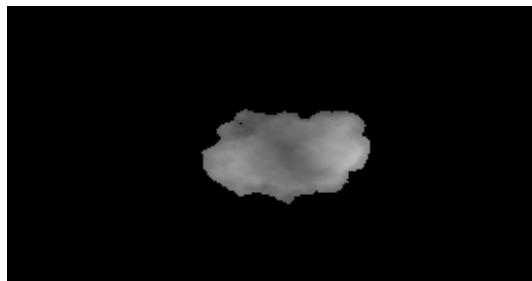


Figure 11: texture avec utilisation de la transparence et du discard avec un seuil de 0.1

### 7.2.4 Problématique importante

La transparence est traité par le shader dans un ordre, donc on doit pour chaque frame trié les particules en fonctions de leurs profondeurs, c'est à dire mettre les plus lointaines en premières.

Il y a un exemple très parlant dans openGl pour comprendre l'intérêt de trier nos particules.

On voit très clairement sur la première image que les objets derrière la première vitre ne sont pas dessiné ce qui indique que cette vitre a été traité par le shader avant celle de derrière. Quand l'ordre de traitement est bon on obtient l'image suivante.

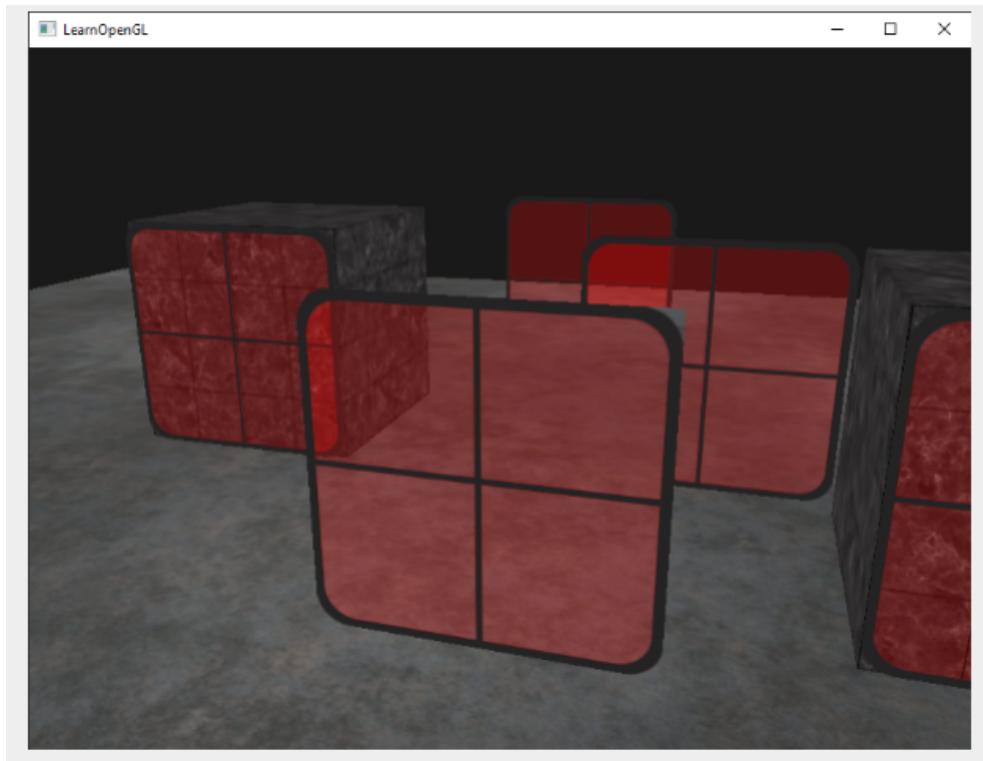


Figure 12: transparence traité dans le mauvais ordre

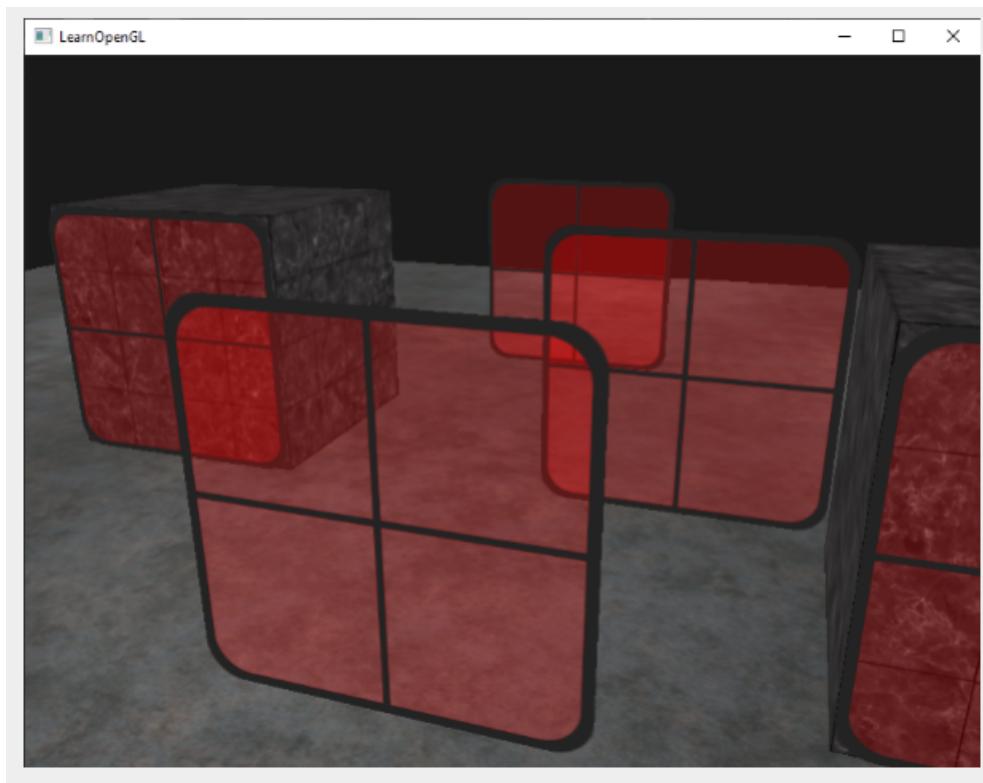


Figure 13: transparence traité dans le bon ordre

Ce calcul prend énormément de temps de calcul et ne peut pas être fait dans le shader; donc c'est une problématique importante.

### 7.3 Fonction de load de texture

```
GLuint loadTexture2DFromFilePath(const std::string &path)
{
    GLuint texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    int width, height, nrChannels;
    unsigned char *data = stbi_load(path.c_str(), &width, &height, &nrChannels, 4);
    if (!data)
    {
        stbi_image_free(data);
        throw std::runtime_error("Failed to load texture: " + path);
    }

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
    stbi_image_free(data);
    setDefaultTexture2DParameters(texture);
    return texture;
}
```

Figure 14: fonction modifiée permettant de charger la transparence

Pour éviter que la particule ne disparaisse d'un coup quand elle meurt, on applique une transparence en fonction de la vie. On envoie la vie de chaque particule de la même manière que pour la position. Ensuite dans le fragment shader on fait un calcul, et donc plus la particule est jeune et donc plus sa transparence avoisine 1 alors que plus elle est vieille et plus elle s'approche de 0.

## 7.4 Mélange de texture

Dans le cas présent on utilise cette texture :



Figure 15: texture utilisée

Mais on peut aussi en utiliser d'autres ou bien faire un mélanage des textures avec la fonction mix. La fonction mix prend en entrée deux textures ainsi qu'un coefficient de mélange : s'il vaut 0 alors il n'y aura que la deuxième texture prise en compte, inversement si ce coefficient vaut 1. Dans le cas il vaudrait 0.5 on aura un mélange équitable entre les deux textures.

Voici deux exemples de textures supplémentaire qu'on a mélangé mais qui n'ont pas donné de résultat convaincant.



(a) deuxième texture

(b) troisième texture

Une explication du problème de ces textures serait leurs grosse densité comparé à celle utilisé.

## 7.5 Bruit de perlin

### Définition du Bruit de Perlin

Le bruit de Perlin est une fonction mathématique qui génère des valeurs de bruit continues et lisses, souvent utilisées pour simuler des textures naturelles telles que le bois, la pierre, les nuages, et les terrains. Contrairement au bruit blanc, qui est complètement aléatoire, le bruit de Perlin produit des variations plus douces et continues, ce qui le rend idéal pour créer des motifs organiques.

Dans le cas de notre fumée ca va nous permettent de lier les particules entre elles pour un rendu plus réaliste.

Comme on procède ?

$$\text{noiseUV} = uv * 5.0 + \text{vec2}(\text{global\_time} * 0.1)$$

Pour rappel uv correspond à la coordonnées de texture et global\_time au temps total écoulé depuis le début. Ensuite on passe cette valeur à la fonction rand ci-dessous :

```
float rand(vec2 co) {
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}
```

Figure 17: fonction rand pour le bruit de Perlin

Enfin on ajoute cette valeur à la coordonnées de texture :

```
vec4coloracc = texture(particuleTexture, uv + noise * amplitude);
```

Amplitude permet de gérer la quantité de bruit de Perlin utilisé, c'est une variable que l'on peut gérer dans notre menu.



Figure 18: fumée sans utilisation du bruit de Perlin

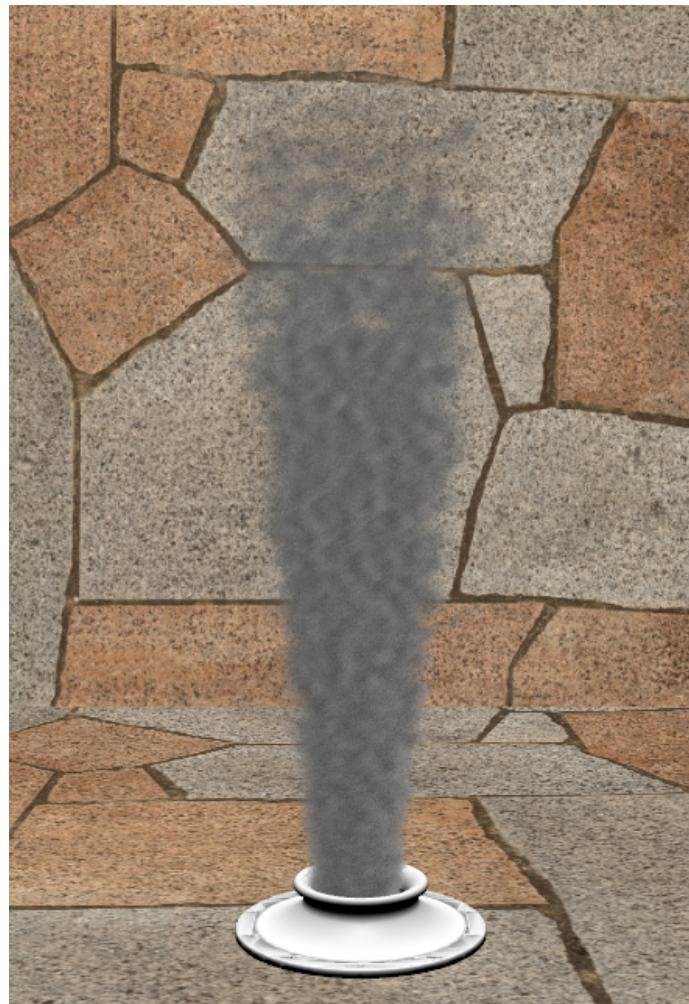


Figure 19: fumée avec utilisation du bruit de Perlin d'amplitude 0.15

Comme c'est assez subtile à voir on vous a mis un exemple où on a forcé l'utilisation du bruit de Perlin.



Figure 20: utilisation abusive du bruit de Perlin pour une amplitude de 1.5

## 8 Interaction avec un cube

La collision avec le cube est calculée en vérifiant si le point après déplacement est à l'intérieur des limites du cube.



Figure 21: Collision avec le cube.

Tout d'abord, on calcule les limites du cube (minimale et maximale) sur chaque axe ( $x$ ,  $y$ ,  $z$ ) à partir de la position du cube et de sa taille. Ensuite on calcule le point après déplacement en ajoutant le vecteur de déplacement au point original. Enfin, on vérifie si le point après déplacement est toujours à l'intérieur des limites du cube. Si c'est le cas, cela signifie que le point est toujours à l'intérieur du cube après le déplacement, donc il n'y a pas de collision. Si le point après déplacement est à l'extérieur des limites du cube, cela signifie qu'il y a eu une collision.

Pour la résolution de collision, on détermine simplement quel coordonnées nous amène dans le cube et on l'annule pour cette frame. Exemple : si la particule à un déplacement  $\text{vec3}(0.5,1.,2.)$  (ici on exagère mais dans le code se sont des valeurs très faible), et que l'on détecte que si on applique ce déplacement à la particule elle va se retrouver dans le cube alors on va chercher de quel coordonnées cela provient. Si on détermine que c'est x alors pour cette frame on appliquera le déplacement  $\text{vec3}(0.,1.,2.)$  à notre particule.

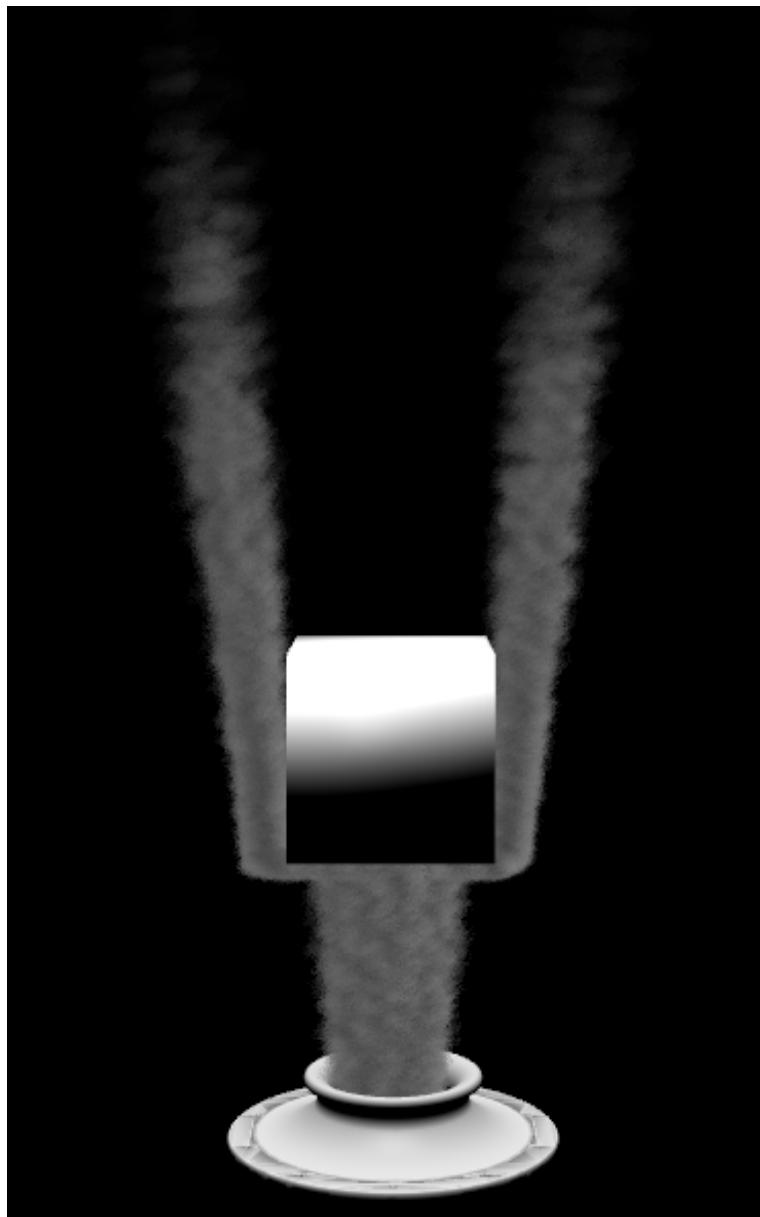


Figure 22: Autre exemple de collision avec le cube.

## 9 Mécanique des fluides

La fumée possède comme tout fluide certaines propriétés : une densité, une viscosité et une dynamique du flux. Afin de simuler le mouvement de la fumée, nous devons nous intéresser à la mécanique des fluides. Dans notre cas il s'agit de reproduire le mouvement de la fumée dans la réalité, c'est-à-dire: la manière dont elle se propage et se diffuse dans l'air, forme des tourbillons, et réagit aux obstacles ainsi qu'aux variations de température. Les principes de la mécanique des fluides permettent de simuler ces interactions, y compris les collisions, la friction, et le rebond. Cela ajoute du réalisme et du dynamisme aux simulations. Pour cela, il existe plusieurs algorithmes implémentables

### 9.1 Navier-Stokes

Les équations de Navier-Stokes sont souvent utilisées pour décrire le mouvement de fluides. Ici elles peuvent servir à reproduire les phénomènes tels la diffusion, la diffusion des forces et les turbulences nécessitant un champ de vitesse et un champ de pression représentés sous forme d'une grille

### 9.2 Gausse-Seidel

L'algorithme de Gauss-Seidel dans notre contexte permet de résoudre les problèmes de pression et de diffusion dans les différentes cellules du champ. Les valeurs sont constamment mise-à-jour en fonction des valeurs voisines. Il sert également à résoudre l'équation de Poisson pour assurer que le fluide reste incompressible, c'est-à-dire qu'il ne change pas de volume au cours du temps

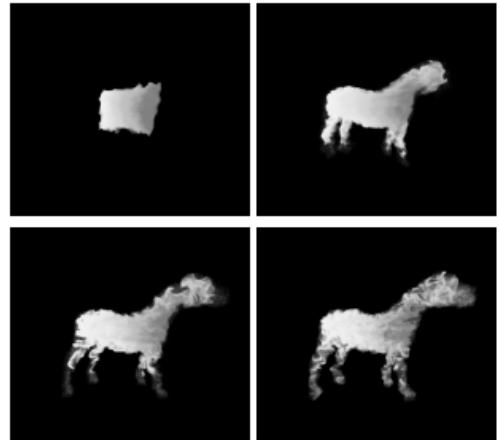
Dans notre simulation les champs sont calculés avec des vecteurs de valeurs aléatoires, pour exprimer des variations ces champs sont constamment recalculés.

## 10 Mesh

L'objectif est de reproduire une forme en fumée un peu comme dans le film Seigneur des Anneaux quand Gandalf crée un bateau avec de la fumée.



(a) exemple de mesh en fumée



(b) second exemple présenté dans l'article 1

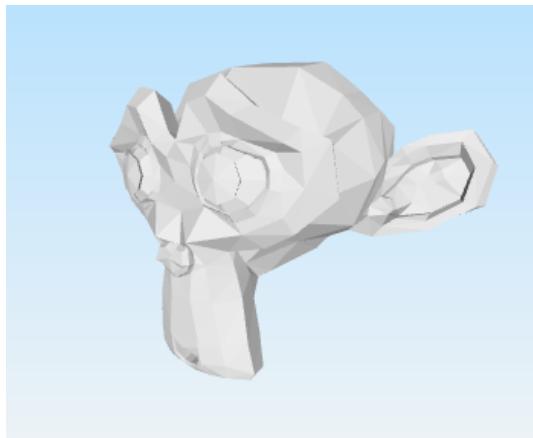
### 10.1 Méthode Utilisé

Dans un premier temps on va charger un modèle, on utilise ici des modèles .off. Après les avoir chargés pour générer le mesh sous forme de fumée on utilise deux méthodes :

- pour chaque sommet du maillage on va tout simplement générer une particules de fumée à la même position, de plus nous augmentons la taille des particules.
- pour celles ci en plus des sommets on a une fonction qui va générer pour chaque triangle du maillage un certain nombres de particules de façon aléatoire dans le triangle, avec une taille des particules beaucoup plus petites.

On applique un déplacement aléatoire pour toutes les particules.

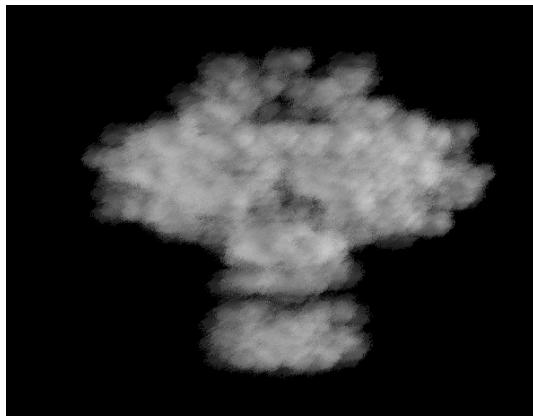
Voici un exemple de mesh avec les deux techniques que l'on viens de vous exposer :



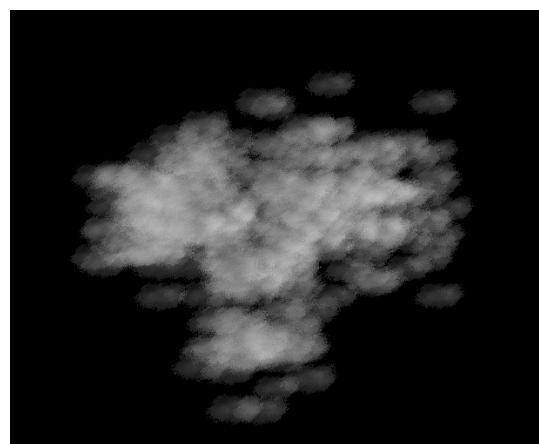
(a) mesh suzanne de coté



(b) mesh suzanne de face



(a) mesh de suzanne avec grosse particules



(b) mesh de suzanne avec grosse particules  
après dispersion

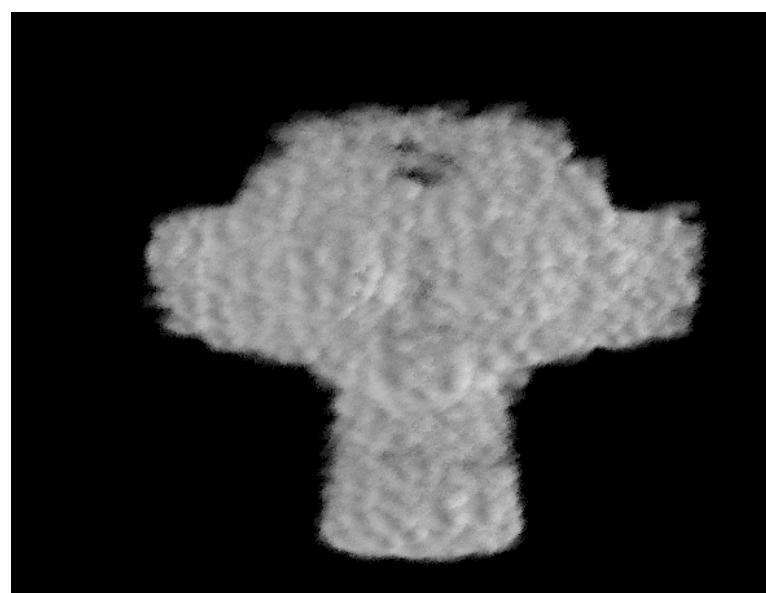


Figure 26: mesh de suzanne utilisant la deuxième méthode

Visuellement il manque quelque chose, notamment un éclairage qui permettrait de mieux voir les formes de suzanne (vous verrez dans la section suivante pourquoi nous avons pas cette fonctionnalité).

## 10.2 Flotabilité

Chaque particule possède une flottabilité. Cette flottabilité pour les mesh est gérée en utilisant la formule de la flottabilité suivante :

$$\text{acc} = 1. \times (1.225 - \text{particles.densite}[i]) \times 9.81 \times \Delta t / 100 \quad (1)$$

Dans cette formule :

- 1.225 est la densité de l'air en kg/m<sup>3</sup>,
- particles.densite[i] est la densité de la particule,
- 9.81 est l'accélération due à la gravité en m/s<sup>2</sup>,
- $\Delta t$  est le temps écoulé depuis la dernière mise à jour de la position de la particule.

La flottabilité est ensuite appliquée à la vitesse de base de la particule avec :

$$\text{particles.baseVelocity}[i] = -\text{acc} \quad (2)$$

Le signe négatif est utilisé pour indiquer que la force de flottabilité agit dans la direction opposée à la gravité.

# 11 Scène et contrôle utilisateur

## 11.1 Scene

Pour la scène on fait le choix de mettre en place une boîte autour de la fumée et d'implémenter un modèle de phong.

### 11.1.1 Modèle d'illumination de Phong

Phong est une technique de rendu utilisée en infographie 3D pour modéliser de manière réaliste la façon dont la lumière interagit avec les surfaces. Elle a été développée par Bui Tuong Phong en 1973 et est particulièrement connue pour sa simplicité et son efficacité. Il existe deux principaux aspects du modèle de Phong : l'illumination et l'interpolation.

Le modèle d'illumination de Phong combine trois composantes principales pour déterminer la couleur finale d'un pixel :

1. **Composante ambiante** : Elle représente la lumière ambiante uniforme qui éclaire tous les objets de la scène de manière égale, sans direction particulière. Elle permet de simuler l'éclairage diffusé dans l'environnement et donne une base de luminosité aux objets, même lorsqu'ils ne sont pas directement éclairés par une source de lumière.

$$I_{ambiante} = K_a * I_a$$

où  $K_a$  est le coefficient de réflexion ambiante de la surface, et  $I_a$  est l'intensité de la lumière ambiante.

2. **Composante Diffuse** : Elle modélise la lumière diffusée qui frappe directement la surface d'un objet et se disperse de manière uniforme dans toutes les directions. Cette composante dépend de l'angle entre la lumière incidente et la normale de la surface, suivant la loi de Lambert.

$$I_{diffuse} = k_d * I_d * (\vec{L} \cdot \vec{N})$$

où  $k_d$  est le coefficient de réflexion diffuse,  $I_d$  est l'intensité de la lumière diffuse,  $\vec{L}$  est le vecteur directionnel de la lumière, et  $\vec{N}$  est la normale à la surface.

3. **Composante Spéculaire** : Elle représente la lumière qui se réfléchit de manière spéculaire (comme un miroir) sur une surface

brillante. Cette composante dépend de l'angle entre le vecteur de réflexion de la lumière et la direction de l'observateur, avec une intensité qui diminue exponentiellement en fonction de cet angle.

$$I_{specular} = k_s * I_s * (\vec{R} \cdot \vec{V})^n$$

où  $k_s$  est le coefficient de réflexion spéculaire,  $I_s$  est l'intensité de la lumière spéculaire,  $\vec{R}$  est le vecteur de réflexion de la lumière,  $\vec{V}$  est le vecteur directionnel de l'observateur, et  $n$  est le coefficient de brillance (plus  $n$  est grand, plus la surface est brillante).

Conclusion :

Le modèle de Phong est très populaire en raison de son équilibre entre simplicité de mise en œuvre et réalisme visuel. Il permet de simuler des effets de lumière convaincants avec un coût computationnel relativement faible, ce qui le rend idéal pour de nombreuses applications en graphisme temps réel, comme les jeux vidéo et les simulations interactives.

### 11.1.2 Calcul des normales pour une particules

Calculer les normales pour chaque particule dans une simulation de fumée peut être très coûteux en termes de calcul, surtout parce que les particules évoluent constamment au cours du temps. Pour simplifier ce processus tout en maintenant une précision acceptable, on peut utiliser une technique basée sur une grille de densité. Voici les étapes détaillées de cette méthode :

1. Génération d'une Grille de Simulation : Diviser l'espace de simulation en une grille tridimensionnelle de cellules. Chaque cellule est un volume de l'espace qui contiendra des informations sur la densité locale de la fumée.
2. Calcul de la Densité de la Fumée :

Pour chaque particule de la simulation, déterminer à quelle cellule de la grille elle appartient en fonction de sa position. Incrémenter la densité de cette cellule pour chaque particule qu'elle contient. La densité d'une cellule est donc proportionnelle au nombre de particules qu'elle contient.

3. Calcul des Gradients de Densité :

Une fois les densités calculées pour toutes les cellules de la grille, calculer les gradients de densité. Le gradient de densité dans chaque

cellule peut être approximé en regardant les différences de densité entre cette cellule et ses voisines adjacentes. Le gradient de densité est un vecteur qui pointe dans la direction où la densité augmente le plus rapidement. Cela fournit une approximation du vecteur normal.

#### 4. Normalisation des Vecteurs de Gradient :

Les vecteurs de gradient calculés doivent être normalisés pour obtenir les normales unitaires. La normalisation implique de diviser chaque composante du vecteur par sa norme (longueur).

#### 5. Attribution des Normales aux Particules :

Chaque particule se voit attribuer la normale de la cellule à laquelle elle appartient. Ainsi, chaque particule utilise la normale de la densité locale pour les calculs d'éclairage.

On n'a malheureusement pas eu le temps d'implémenter cette méthode malgré le fait qu'on utilise déjà une grille pour la simulation des mouvements.

## 11.2 Interaction utilisateur avec ImGui

Notre programme ne se compose que d'une seul scène, les éléments vont être soumis à des modifications mais tout le programme utilise cette unique scène. Au lancement du programme nous nous retrouvons dans la boîte énoncé plus tôt, et on peut observer au sol le générateur d'où va être expulsé la fumée. Il est possible d'ajouter le cube qui va altérer les déplacements des particules de la fumée. La lumière est également paramétrable (intensité et position). Enfin la grille permettant d'appliquer Navier-Stokes et Gauss-Seidel voit sa résolution paramétrable m'est intentionnellement non visible.

Nous avons choisi d'utiliser ImGui, une bibliothèque d'interface utilisateur graphique afin de proposer un paramétrage en temps réel de la simulation. Cela permet d'effectuer une multitude de test, de voir les effets de chaque paramètres sans avoir à modifier et relancer le programme à chaque fois. La fenêtre principale ImGui est décomposée en plusieurs menus.

L'interface est découpé en 4 parties :

- Grille : permet de modifier les paramètres de la grille qui permet de gérer le mouvement type fluide de la fumée.



Figure 27: menu 1

- Environnement : dans cette partie on contrôle tous ce qui concerne l'éclairage, donc la position de la lumière, son intensité et sa couleur.

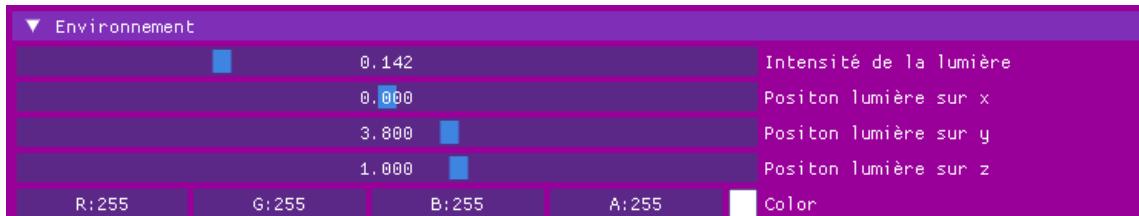


Figure 28: menu 2

- Force externe : ici on va gérer les paramètres des forces externes : vent, gravité, couleur de la fumée, collision avec un cube et mouvement de celui-ci.

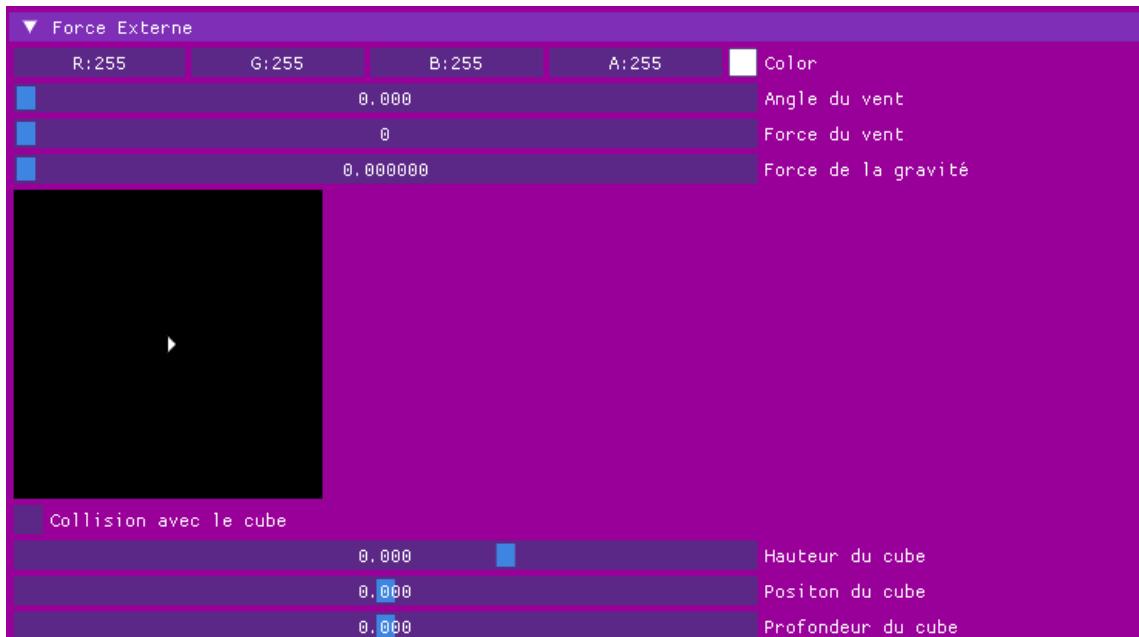


Figure 29: menu 3

- Paramètres de la fumée : enfin dans ce menu on gère le nombre de particules, la vitesse d'expulsion, leurs tailles, leurs durée de vie, leur transparence. Mais aussi le lancement de la fumée ainsi que son arrêt

et la possibilité d'afficher un mesh (petit ou gros). On a aussi le nombres de particules affichés.

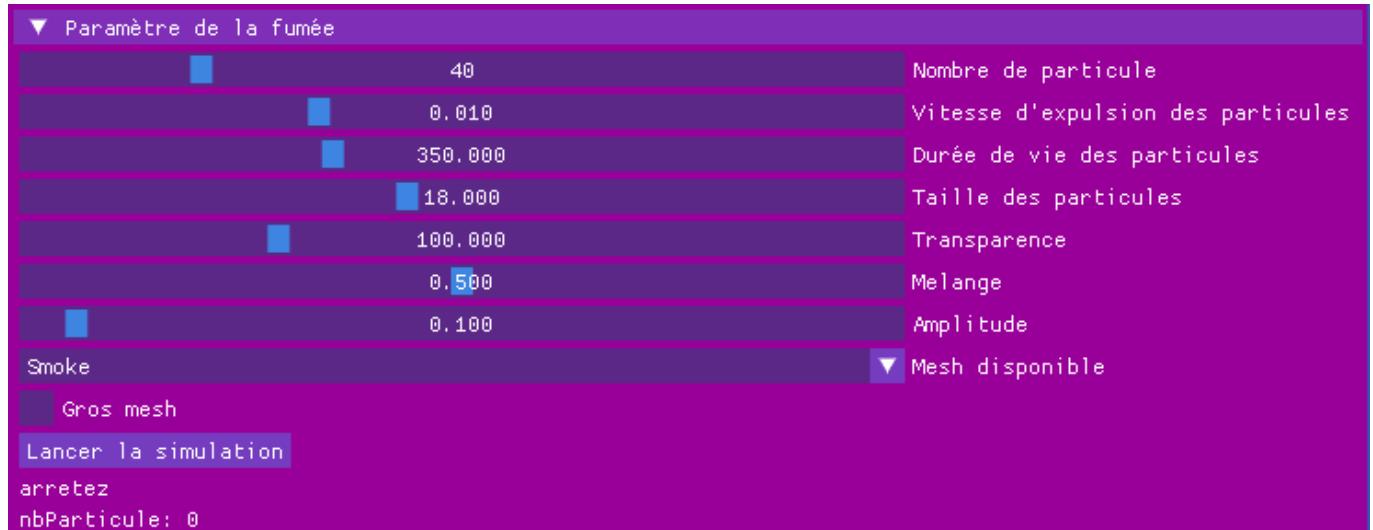


Figure 30: menu 4

## 12 Résultats

### 12.1 Démonstration des forces

Maintenant que nous avons vu toutes les forces qui agissent sur les particules, nous pouvons faire un résumé de tous les déplacements configurables via leurs champs de la fenêtre imgui dans cette simulation.

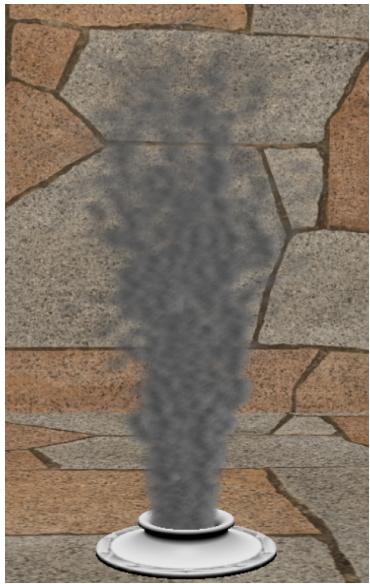
- L'accélération de la vitesse d'expulsion : permet une ascension plus rapide.
- La gravité : force qui s'applique vers le bas ce qui ralentit les particules dans l'axe Y et peut même les faire retomber.
- Le vent : paramétriser la force du vent et son angle permet d'appliquer des forces sur les axes X et Z.
- La résolution de la grille et la force du vecteur scalaire : la résolution permet d'avoir des cellules plus petites et donc plus de zones où les variations s'appliquent. La force du vecteur permet des variations plus importantes.



(a) résultat avec les paramètres par défaut



(b) vitesse d'expulsion accélérée



(c) Mécanique des fluides  
Navier-Stokes



(d) Vent + gravité

Figure 31: Il y'a un tas de résultats différents que l'on peut obtenir, puis ce qu'il y'a beaucoup de mouvement il est bien plus intéressant d'observer la simulation en démonstration qu'avec des images

## 12.2 Analyse des résultats

Sur la figure a on observe le résultat de la simulation lorsque l'on ne modifie aucun paramètre. La fumée est convaincante, ce type de fumée pourrait s'apparenter à une sortie de cheminée. Néamoins on discerne un peut trop un paterne dans sa texture.

La seconde figure possède une fumée expulsé à une vitesse plus importante, dans ce cas la dispersion est trop légère et mérirerait d'être améliorer car on à juste l'impression d'avoir une colonne de fumée régulièrre.

La figure B est une tentative de simulation de la mécanique des fluide, c'est le meilleur résultat que l'on arrive a obtenir avec cette méthode. On à un avant goût de ce que la méthode peut donner mais en mouvement on discerne les changements de direction qui sont trop périodique. Cela pourrait être améliorer en optimisant cette partie et les calculs effectués. Nous sommes mitigés sur ce résultat.

La dernière figure est un mélange de l'application de la force du vent et de la gravité afin de formé un arc de fumé. Le rendu n'est pas si mal mais aurait pu être améliorer en appliquant le vent de manière progressive en fonction de la durée de vie de la particule ou de sa hauteur par exemple car on à encore cette impression d'avoir un arc régulier et qui s'applique trop fort dès le début de l'émission

## 13 Problèmes rencontrés

### 13.1 Base de code

La première difficulté que nous avons rencontrée est arrivée plus tôt que nous le pensions. A cause de notre premier choix de base de code qui n'était pas le plus judicieux, nous avons perdu beaucoup de temps sur la mise en place de l'environnement. Finalement après avoir décidé de changer de base de code, nous avons pu repartir et travailler de manière plus efficace puisque celle-ci était plus apte à au développement de notre projet.

### 13.2 Particules parasites

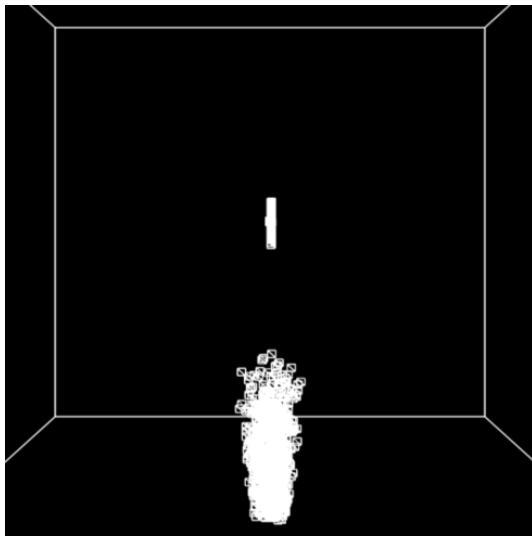
La partie qui nous a posé le plus de soucis est sans aucun doute les particules parasites qui sont arrivés après la disposition de l'environnement, au moment où nous avions nos premières particules (sous forme de point simple à ce moment là) qui se déplaçait comme le souhaitons. Cependant nous avons observés que toutes les particules n'étaient pas au bon endroit, certaines possédaient d'autre point de départ et une direction dans un axe complètement différent (Y ou Z au lieu de l'axe X). Après plusieurs jours de débogage nous avons émis l'hypothèse que le problème se situait au niveau de l'envoie aux shaders. Nous avons donc modifié la façon dont l'envoie ce faisait, c'est pour cela que nous avons modifié la structure des particules.

```
struct Particle{
    glm::vec3 position;
    float life = lifeglobal;
    vec3 deplacement;
    float baseVelocity = 0.01f;
};
```

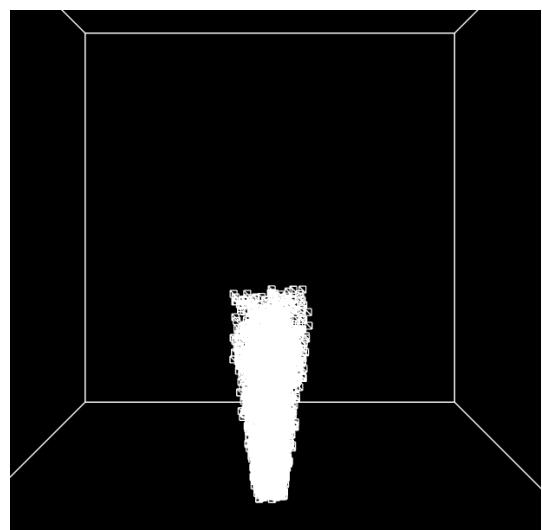
(a) Structure causant problème.

```
struct Particle{
    std::vector<glm::vec3> position;
    std::vector<float> life;
    std::vector<glm::vec3> deplacement;
    std::vector<float> baseVelocity;
    std::vector<float> densite;
};
```

(b) Structure après correction.



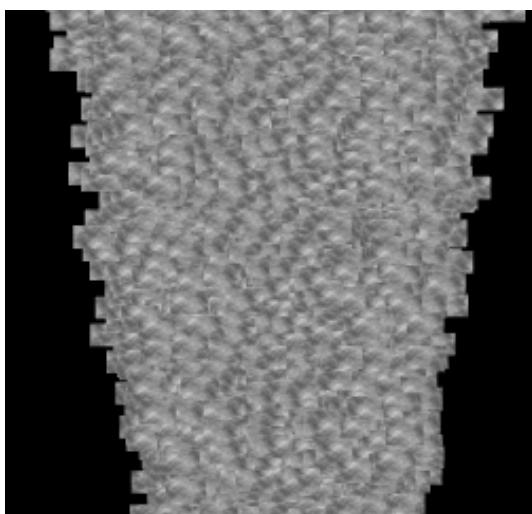
(a) Image contenant les parasites.



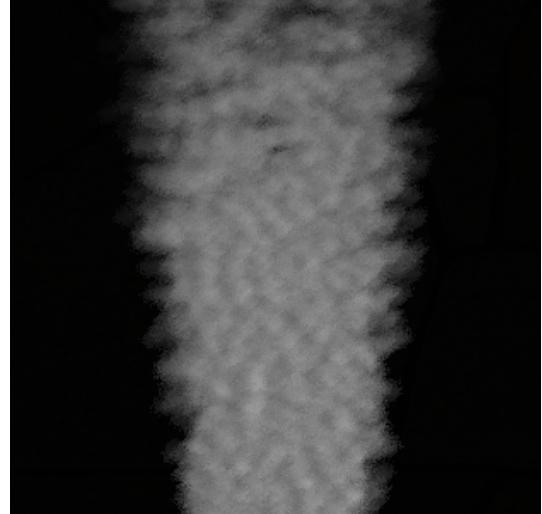
(b) Image après correction du problème.

### 13.3 Chargement de la texture

Enfin, nous avons rencontré un dernier problème, lors de l'intégration des textures, le chargement de la texture était particulièrement problématique, en raison de la gestion de la transparence. Nous avons dû nous assurer que les textures étaient correctement chargées et que les canaux de transparence étaient pris en compte. Pour résoudre ce problème, nous avons modifié le fragment shader afin d'incorporer correctement le canal alpha des textures, nous avons également modifié la fonction load (cf. 7.3), avant modification à la place de la valeur 4 il y avait 4 et à la place de RGBA il y avait RGB. Cela a permis une meilleure intégration des textures transparentes, comme illustré dans les images suivantes :



(a) texture sans utilisation de la transparence



(b) texture avec utilisation de la transparence

## 14 Conclusion

### 14.1 Objectifs Atteint ?

Ce projet visait à développer une simulation de fluide interactif en temps réel, en se concentrant sur la génération et le contrôle de la fumée à l'aide de particules. Nous nous accordons pour dire que l'objectif a été partiellement atteint. Nous avons obtenu une fumée avec une texture et des propriétés transparentes convaincantes, des mouvements non complets mais qui donne déjà une impression de fluide. Toutes les collisions escomptées ne sont pas présentes mais on a déjà un avant goût que cela peut donner avec celle du cube. L'initiative de l'implémentation de "mesh en fumée vivant" s'est révélée plus complexe que nous l'avions anticipée, cette partie a donc été mise de côté pour le bien de l'objectif principal. En somme si on prend chaque aspect individuellement, tout n'est pas parfait, mais le rendu final est convaincant.

### 14.2 Ce que nous avons appris

Nous retenons ce de projet que la reproduction virtuel de mécanisme s'appliquant au monde réel est une tâche délicate qui nécessite d'étudier de multiples paramètres et d'implémenter des algorithmes relativement complexes. L'application des lois de la physique demande de nombreux tests et une documentation précise du domaine que l'on souhaite représentés. Cela aura été une bonne expérience, qui nous aura appris à repenser notre façon de penser de départ car comme nous l'avons expliqués, plusieurs fois nous avons dû réadapter tout une partie de notre programme pour le bien de la suite du projet.

### 14.3 Amélioration futures

Parmi les améliorations possibles, on retrouve les points évoqués précédemment :

- des textures plus réalistes grâce à des combinaisons plus efficace.
- une meilleure implémentation de la mécanique des fluides pour des mouvements plus précis
- une interaction plus riche avec les objets du décor et une réponse à la collision plus complète tel qu'un rebond.

- un calcul des normales pour chaque particule ce qui rendrait plus esthétique avec le l'illumination de Phong mise en place.
- basculé plus de calcul sur gpu pour augmenter la vitesse de traitement.

## 15 Références

### 15.1 Recherches

- Simulation de fluide réaliste et Raytracing à partir de zéro
- Fast Fluid Dynamics Simulation on the GPU
- How to write an Eulerian fluid simulator with 200 lines of code.
- LearnOpenGL
- Linear-Time Smoke Animation with Vortex Sheet Meshes

### 15.2 Algorithmes

- Navier-Stokes\_Wikipédia
- Gauss-Seide\_Wikipédia
- Méthode de Gauss-Seidel

### 15.3 Articles

- Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Based Methods
- Visualization of smoke using particle systems f
- Target Particle Control of Smoke Simulation