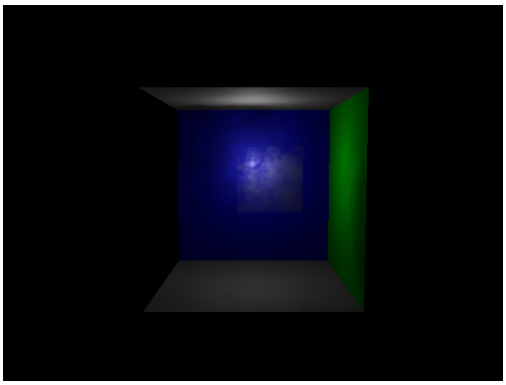
Rapport Final

Samuel Fournier
20218212

Dans le cadre du cours
IFT 3150



Département d'informatique et de recherche opérationnelle
Université de Montréal
27 novembre 2024

Introduction

L'objectif de ce projet est d'implémenter un volume (aussi appelé média) participatif dans un algorithme de traçage de rayons. Lors de la session d'hiver 2024, dans le cadre du cours IFT 3355 Infographie, j'ai eu à implémenter un *ray tracer* rudimentaire. Ce *ray tracer* ne pouvait rendre que quelques primitives (sphères, cylindres, plans et triangles). Par conséquent, avec le soutien du professeur Pierre Poulin, j'ai implémenté une nouvelle primitive, le volume participatif.

1 *Ray Tracing*

En infographie, il existe plusieurs méthodes pour générer le rendu d'une scène 3D, dont le *z-buffer*, le *scanline renderer*, l'arbre BSP, etc. Une de ces méthodes parmi les plus populaires est le traçage de rayons. Cet algorithme simule la façon dont la lumière provenant du Soleil (ou tout autre source de lumière) interagit avec les objets dans le monde. En effet, une source de lumière "tire" des rayons de lumière vers des objets (le Soleil vers la Terre par exemple). Ensuite, ces rayons de lumière sont soit absorbés, réfléchis, réfractés ou les trois en même temps. C'est ce phénomène physique qui donne la couleur aux objets qui nous entourent et qui nous permet de les voir. L'algorithme du *ray tracer* simule ces phénomènes, mais en suivant le chemin inverse, de la caméra vers la lumière, en se basant sur le principe d'Helmholtz. Pour chaque pixel d'une image, en partant de la caméra (notre œil dans la scène), on tire des rayons dans notre scène 3D. Ensuite, on détermine quels objets ces rayons frappent (s'ils frappent un objet). Une fois que l'on a déterminé un point d'intersection, on calcule la couleur à ce point. Pour effectuer ce calcul, on utilise des modèles de réflexion (*shading*) comme le modèle de Blinn-Phong [2] (celui vu dans le cours IFT 3355 et qui a été utilisé dans l'algorithme). L'algorithme ressemble à:

for $y \leq pixel_y$ **do**
  for $x \leq pixel_x$ **do**
    $ray \leftarrow newRay(origin, direction)$
    $trace(ray)$
    if $intersect == True$ **then**
      if object is reflective **then**
        if $ray\_depth < scene.max\_ray\_depth$ **then**
          $reflectedRay \leftarrow newRay(origin, reflecteddirection)$
          $trace(reflectedRay)$
        end if
      else if object is refractive **then**
        if $ray\_depth < scene.max\_ray\_depth$ **then**
          $refractedRay \leftarrow newRay(origin, refracteddirection)$
          $trace(refractedRay)$
        end if
      end if
      $outColor \leftarrow shade(ray, intersectPos)$
    else
      $outColor \leftarrow (0, 0, 0)$
    end if
  end for
end for

Bien évidemment, l'algorithme complet est un peu plus complexe que ça (particulièrement la fonction *shade*), mais ce pseudo-code donne un aperçu de la logique générale du fonctionnement de mon implémentation. Si vous voulez en apprendre plus sur cet algorithme, veuillez consulter les ressources suivantes dans la bibliographie à la fin du rapport [4, 5, 7, 11].

2 Le volume participatif

Un volume participatif est un objet qui affecte la quantité de lumière qui le traverse. Dans mon implémentation, ce volume représente de la fumée. La primitive utilisée pour représenter le volume est une grille de voxels [12]. En réalité, cette grille n'est rien de plus qu'un simple cube subdivisé en un certain nombre de voxels selon les axes x, y et z. Chaque voxel dans le volume a une densité initialisée par possiblement différentes méthodes. La densité de chaque voxel peut avoir une valeur constante entre 0 et 1, choisie arbitrairement avant l'exécution du programme. La densité peut aussi être initialisée par une fonction de bruit de Perlin [10]. Une troisième méthode initialise la densité de chaque voxel en suivant un gradient linéaire où la densité d'un voxel à $y = 0$ (minimum) vaudra 1 et la densité d'un voxel à $y = 1$ (maximum) vaudra 0. Une quatrième méthode est similaire à la troisième, mais en suivant un gradient exponentiel en fonction de la hauteur ($\exp\{-y\}$). Finalement, la dernière méthode consiste à initialiser les densités dans une sphère centrée sur la grille. Cette méthode peut utiliser une des quatre méthodes qui viennent d'être mentionnée.

Une fois notre nouvelle primitive définie, il faut spécifier à l'algorithme comment interagir avec cette primitive. Premièrement, il faut déterminer si on frappe la primitive ou pas. Le code pour l'intersection est très simple. En effet, comme notre primitive n'est qu'un cube qui contient des voxels, on doit tout simplement implémenter l'intersection entre une droite orientée et un cube. L'implémentation de ce code a déjà été vu en détail dans les rapports bi-hebdomadaires, mais je fais quand même un survol ici.

Un rayon est une fonction paramétrique dans l'espace de la forme (ici, le cube):

$$\begin{aligned}\vec{r}(t) &= \vec{o} + t\vec{d} \qquad \text{où } \vec{o} \text{ est l'origine du rayon et } \vec{d} \text{ sa direction} \\ &= \begin{bmatrix} o_x + td_x \\ o_y + td_y \\ o_z + td_z \end{bmatrix}\end{aligned}$$

Par conséquent, une fois dans l'espace local de la grille, on cherche les valeurs de t tel que:

$$\begin{aligned} o_x + td_x &= 0 \\ o_x + td_x &= 1 \\ o_y + td_y &= 0 \\ o_y + td_y &= 1 \\ o_z + td_z &= 0 \\ o_z + td_z &= 1 \end{aligned}$$

pour un cube dont les dimensions sont $x, y, z \in [0, 1]$.

Pour les prochaines étapes, j'introduis une nouvelle notation: $t0_{x,y,z}$ représente la valeur t lorsque le rayon frappe les plans $x = 0$, $y = 0$ ou $z = 0$ respectivement et $t1_{x,y,z}$ représente la valeur t lorsque le rayon frappe les plans $x = 1$, $y = 1$ ou $z = 1$. Pour obtenir nos $tMin$ et $tMax$, on résout les équations. En x par exemple, on obtient:

$$\begin{aligned} o_x + t0_x d_x = 0 &\iff t0_x d_x = -o_x \iff t0_x = \frac{-o_x}{d_x} \\ o_x + t1_x d_x = 1 &\iff t1_x = 1 - o_x \iff t1_x = \frac{1 - o_x}{d_x} \end{aligned}$$

Ensuite, on trouve les minimums et les maximums parmi les $t0_{x,y,z}$ et les $t1_{x,y,z}$. Le maximum des minimums est le point d'entrée du rayon et le minimum des maximums est le point de sortie.

Quelques problèmes peuvent survenir lors de ce calcul. Les deux que j'ai rencontrés étaient:

- l'origine $\vec{o}$ est dans le volume;
- la direction $\vec{d}$ d'une des composantes est nulle.

Heureusement pour nous, ces deux problèmes se gèrent très facilement.

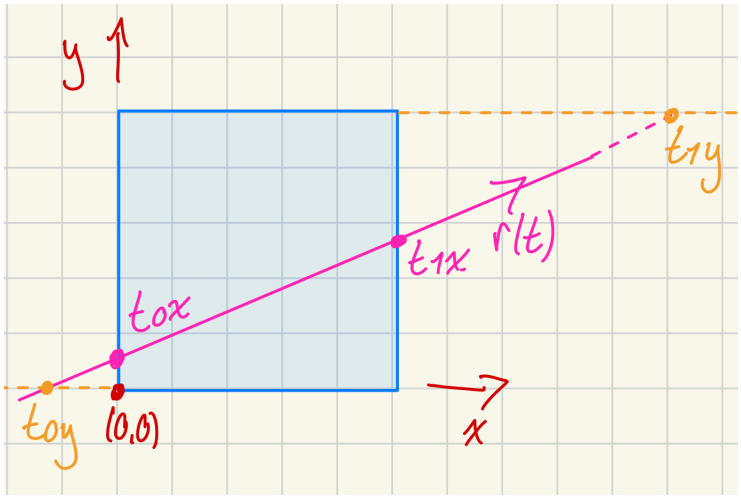


Figure 1: Test d'intersection en 2D entre un rayon et une boîte.

2.1 L'origine est dans le volume

Dans ce scénario, on remarque que nous ne voulons pas déterminer de points d'entrée. En effet, en fonction du signe de la direction, on va obtenir des $t0$ ou des $t1$ négatifs. Pour le démontrer, supposons que l'origine du rayon est (0.5, 0.5, 0.5) et sa direction est (-1, 0.4, -0.76). Si on reprend les formules développées plus haut et on observe une fois de plus la composante x du rayon, on obtient:

$$t0_x = \frac{-o_x}{d_x} = \frac{-0.5}{-1} = 0.5$$
$$t1_x = \frac{1 - o_x}{d_x} = \frac{1 - 0.5}{-1} = -0.5$$

On voit que le $t1_x$ est négatif. Cela indique que le "point d'entrée" de notre rayon est 0.5 unité derrière. Alors, lorsque l'on considère les différentes valeurs pour déterminer le point d'entrée et de sortie, on devra ignorer celle-ci. En fait, comme l'origine du rayon se situe entièrement dans le volume, nous n'avons pas de point d'entrée à déterminer.

2.2 La direction d'une composante est nulle

Lorsque cela arrive, on remarque assez facilement que l'on va voir un problème lors du calcul des t. En effet, comme on divise par la direction, une direction nulle va causer une division par 0. Heureusement (et étrangement), je n'ai pas eu beaucoup de problèmes reliés à la division par 0. Le projet a été codé en C++ et dans ce langage, lorsqu'une division par 0 survient, il ne lance pas d'exception, mais ne fait qu'assigner à la variable $\pm infinity$ (en fonction du signe du résultat). Le seul problème qui est survenu avec cette façon de gérer la division par 0 était lors du test d'intersection entre le medium et un rayon se dirigeant vers la lumière. Dans ce cas particulier, on sait que l'origine est à l'intérieur du volume, alors on ne cherche pas de point d'entrée. Pour trouver le point de sortie, on utilise la direction du rayon pour déterminer quelle valeur de t on doit utiliser pour la comparaison finale. Si la direction était 0, le code fonctionnait correctement, car, comme mentionné plus haut, la division donnait $infinity$. Par contre, si la direction était -0 (en C++ 0 == -0 est **True**), alors la division donnait une valeur négative très petite ($-infinity$). Alors, pour éviter ce problème, si la direction d'une composante était 0 ou -0, la valeur de t était mise à **DBL_MAX**. À première vue, cette solution peut sembler risquer, mais elle ne l'est pas. Comme la variable t est une valeure flottante, il est possible d'obtenir une valeure qui vaut zéro, mais est négative au niveau de sa représentation en bit.

Par conséquent, que la direction soit 0 ou -0, on veut que la valeur de t résultante soit **TRÈS** grande. Mais comme la division par -0 peut donner une une valeure **TRÈS** petite, on doit faire cet ajustement. Une autre solution aurait été de simplement vérifier si c'est un 0 négatif et changé son signe si jamais ce l'est avec la fonction *std::signbit(val)* de la librairie standard de C++.

3 Parcours du volume participatif

Maintenant que nous avons trouvé les points intersectés par le rayon (le point d'entrée et de sortie $tMin$ et $tMax$ respectivement), il faut traverser notre grille pour déterminer la quantité de lumière qui est bloquée par le volume le long de notre rayon. Pour ce projet, j'ai implémenté deux types d'algorithmes de parcours: le *Digital Differential Analyzer* [9] (DDA) et le *ray marcher*.

3.1 Digital Differential Analyzer (DDA)

Mon implémentation de l'algorithme du DDA est basé sur celle par Amanatides et Woo [1, 3]. L'idée générale de l'algorithme est simple. Nous voulons parcourir notre rayon en allant d'intersection à intersection dans la grille. Cet algorithme est très similaire à celui qui a été utilisé pour déterminer si le rayon frappait notre volume. La première étape est de déterminer les $tMax_{x,y,z}$ au sein de la grille. Le calcul reste inchangé, cependant le plan frappé ne sera pas forcément 0 ou 1, mais une valeur qui dépend de la taille d'un voxel. Une fois que l'on a trouvé les premiers $tMax$, il faut trouver ce qu'Amanatides et Woo appellent les $tDelta$. Les $tDelta$ représentent la quantité qu'il faut additionner à t pour que le rayon rencontre le prochain plan. Par exemple, supposons que nous avons une grille de 2 x 2 x 2 voxels. Par conséquent, la taille de ceux-ci est 0.5 (comme c'est un cube unitaire, chaque dimensions d'un voxel est 1/nb voxels en x, y et z). De plus, supposons que le point d'entrée du rayon est sur le plan $x = 1$. Par conséquent, la quantité qu'il faudra additionner à t pour que le rayon frappe le plan $x = 0.5$ est:

$$
\begin{aligned}
o_x + td_x &= 1 \\
o_x + (t + \Delta t)d_x &= 0.5 \\
\implies o_x + (t + \Delta t)d_x + 0.5 &= 1 = o_x + td_x \\
\implies o_x + td_x + \Delta td_x + 0.5 &= o_x + td_x \\
\iff \Delta td_x + 0.5 &= 0 \\
\iff \Delta td_x &= -0.5 \\
\iff \Delta t &= \frac{-0.5}{d_x}
\end{aligned}
$$

Bref, le Δt ne fait que représenter la distance à parcourir pour obtenir le prochain plan en x, y ou z et il dépend de la taille d'un voxel ainsi que la direction du rayon. Une fois que l'on a calculé tout ça, on utilise une boucle **while** pour itérer le long de notre rayon. À chaque itération, on observe quel t est le plus petit (entre x, y et z), on le met à jour avec la valeur $tDelta$ appropriée et on avance au point associé. La condition d'arrêt de notre boucle est lorsque notre position est la même que la position finale (celle à la sortie). Voici un pseudo-code de cet algorithme:

$start \leftarrow \vec{r}(tMin)$
$end \leftarrow \vec{r}(tMax)$
$tMaxx \leftarrow \frac{(currentVoxel_x \times sizeVoxel_x) - o_x}{d_x}$
$tMaxy \leftarrow \frac{(currentVoxel_y \times sizeVoxel_y) - o_y}{d_y}$
$tMaxz \leftarrow \frac{(currentVoxel_z \times sizeVoxel_z) - o_z}{d_z}$
$tDeltax \leftarrow \left| \frac{sizeVoxel_x}{d_x} \right|$
$tDeltay \leftarrow \left| \frac{sizeVoxel_y}{d_y} \right|$

$tDeltaz \leftarrow \left| \frac{sizeVoxel_z}{d_z} \right|$
$currentPos \leftarrow start$
while $currentPos \leq end$ **do**
  if $tMax_x < tMax_y$ and $tMax_x < tMax_z$ **then**
    $tMax_x \leftarrow tMax_x + tDelta_x$
    $currentPos \leftarrow \vec{r}(tMax_x)$
  else if $tMax_y < tMax_z$ **then**
    $tMax_y \leftarrow tMax_y + tDelta_y$
    $currentPos \leftarrow \vec{r}(tMax_y)$
  else
    $tMax_z \leftarrow tMax_z + tDelta_z$
    $currentPos \leftarrow \vec{r}(tMax_z)$
  end if
  $evaluateTransmittance(currentPos)$
  $evaluateScattering(currentPos)$
end while

Bien évidemment, il y a quelques opérations de plus, comme le calcul de l'atténuation de la lumière le long du rayon, mais je vais entrer plus en détail sur ce sujet dans une autre section. L'objectif de cette section n'est que d'expliquer l'idée derrière l'algorithme de Woo et Amanatides.

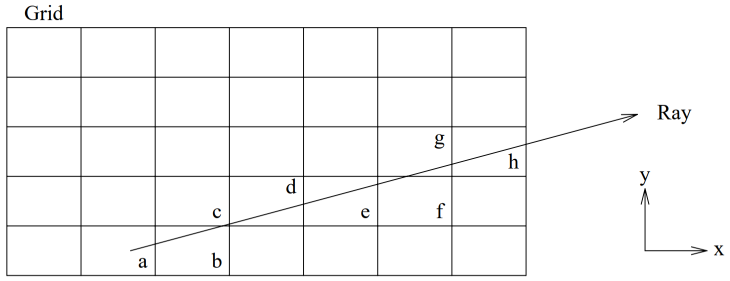


Figure 1

Source: L'article de Woo et Amanatides [1]
L'image dépicte les plans (a, b, c, ...) que le rayon va intersecter

3.2 Ray Marching

L'algorithme du *ray marching* est d'une certaine façon assez similaire au DDA de Woo et Amanatides. En effet, dans les deux cas, l'objectif de l'algorithme est de "marcher" le long du rayon. Ce qui les différencie cependant, est comment on définit le pas. Dans l'algorithme du DDA, un pas est la distance entre les intersections de la grille de voxel (les $tDelta$). Par conséquent, cette distance pourrait être "très grande" ou "très petite". Un avantage de cette approche est le fait que le DDA est garantit de ne rater aucun voxel le long du rayon même si le rayon ne fait que toucher un très petit coin du voxel. Dans le cas du *ray marcher*, on initialise une taille de pas fixe et on marche le long du rayon en prenant des pas de cette taille. Bref, le DDA est un *ray marcher* avec une taille de pas variable plutôt que constante. L'algorithme du *ray marcher* est plus simple que celui du DDA. En premier, on doit définir une taille de pas que l'on appelle **step** (comme on a un cube unitaire, il est important que cette valeur soit inférieure à 1). Ensuite, on découpe notre rayon en **n** intervalles. Une fois que c'est fait, on utilise une boucle **for** pour itérer le long de l'intervalle. À chaque itération, on calcule le nouveau t et on met à jour la position le long du rayon.

$step \leftarrow 0.05$
$interval \leftarrow ceil(\frac{tMax - tMin}{step})$
$step \leftarrow \frac{tMax - tMin}{interval}$

for $n < interval$ **do**
$t \leftarrow tMin + (n + jitter()) \times step$
$newPos \leftarrow \vec{r}(t)$
$evaluateTransmittance(currentPos)$
$evaluateScattering(currentPos)$
$n \leftarrow n + 1$
end for

Je tiens à mentionner quelques détails dans cette implémentation. Premièrement, la valeur de 0.05 pour **step** est complètement arbitraire, mais c'est la valeur qui a été utilisé tout le long du projet. Deuxièmement, on recalcule **step** pour normaliser la valeur pour les intervalles calculés. Comme **interval** est de type **int**, lorsque l'on le calcule, il va se faire arrondir (vers le haut). Par conséquent, la valeur de **step** initial ne sera plus appropriée, car l'intervalle sera plus grand que sa valeur réelle à cause de l'arrondissement. Finalement, la fonction **jitter()** ne fait qu'ajouter ou enlever une valeur aléatoire à notre pas.
Dans le cadre du projet j'ai implémenté quatre façon de calculer le nouveau point. Les quatre façons sont: *ray marcher* avec pas régulier, *ray marcher* avec pas régulier et **jitter**, *ray marcher* au centre du rayon et *ray marcher* au centre du rayon avec **jitter**.

3.2.1 Pas régulier

Dans ce type, on prend des pas réguliers le long du rayon. On se rend de frontière à frontière de façon régulière. Le pas est calculé de la façon suivante:

$$t = n \times step$$

3.2.2 Pas régulier avec jitter

Dans ce type, on prend des pas réguliers le long du rayon, mais on ajoute un léger déplacement aléatoire entre [0, 1). Le point est calculé de la façon suivante:

$$t = (n + jitter) \times step$$

3.2.3 Au centre du rayon

Dans ce type, on prend des pas réguliers, mais au lieu de se rendre de frontière à frontière, on se rend au centre de l'intervalle. Le point est calculé de la façon suivante:

$$t = (n + 0.5) \times step$$

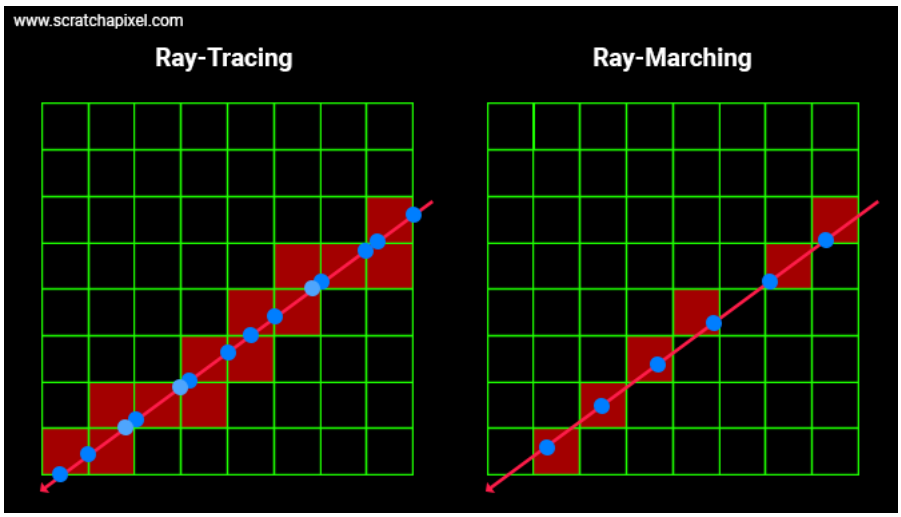


Image qui représente la différence entre le DDA et le ray marching

4 Atténuation de la lumière par le volume participatif

Après avoir intercepté notre médium et commencer à le traverser, il faut calculer l'atténuation de la lumière le long du rayon. Le principe est très simple. Pour évaluer l'atténuation à chaque pas, on utilise la loi de Beer-Lambert [8]. Selon la loi, la transmission de lumière le long d'un rayon est évaluée selon la formule suivante:

$$T = \exp\{-d \times \sigma_a\}$$

Dans la formule, T représente la transmittance, d la distance parcourue et σ_a le coefficient d'absorption. Par conséquent, la transmittance est une valeur décroissante. Une transmittance de 1 indique que 100% de la lumière atteint le point (ou que 0% de la lumière est bloquée) et une transmittance de 0 indique que 0% de la lumière atteint le point (ou que 100% de la lumière est bloquée). La formule finale est légèrement plus complexe que celle présentée ci-dessus. Lorsque l'on évalue l'atténuation de la lumière, on évalue l'interaction entre les particules du médium et les rayons de nos lumières. Pour évaluer cette interaction, on doit évaluer quatre phénomènes:

1. Absorption
2. Out-Scattering
3. Émission
4. In-Scattering

4.1 Absorption

Ce concept est probablement le plus simple des quatre mentionnés. Lorsque la lumière parcourt notre médium, une certaine quantitée va être absorbée par le médium. Dans le code, cette quantité est représentée par le coefficient d'absorption σ_a.

4.2 Out-Scattering

Lorsque que nous avons de la lumière qui se dirige vers l'œil (notre caméra), il est possible que cette lumière se fasse dévier par les particules du volume dans une direction aléatoire autre que l'œil. Par conséquent, moins de lumière finit par atteindre l'œil nous donnant un résultat plus sombre. Dans le code, ce phénomène est représenté par le coefficient de scattering σ_s.

4.3 Émission

Il est possible qu'un volume (comme du feu) émet sa propre lumière. On parle dans ce cas d'émission. Malheureusement, ce concept n'a pas été abordé dans le projet, mais il serait intéressant d'éventuellement l'ajouter.

4.4 In-Scattering

Ce concept est très similaire à celui du out-scattering. Cependant, plutôt que de rediriger de la lumière se dirigeant vers la caméra (comme dans le out-scattering), on redirige de la lumière se dirigeant dans une autre direction que la caméra vers la caméra. Dans le code, ce phénomène est représenté par le coefficient de scattering σ_s.
Maintenant que l'on a vu les concepts importants au calcul de l'attenuation de la lumière, regardons plus en

détail les formules et le code!
Premièrement, le calcul de la transmittance. On se rappelle que la transmittance suit la loi de Beer-Lambert:

$$T = \exp\{-d \times \sigma_a\}$$

Pour considérer le scattering et la densité de notre position, la formule devient:

$$T = \exp\{-d \times (\sigma_s + \sigma_a) \times density\}$$
$$transmittance = transmittance \times T$$

Dans l'algorithme du **ray marcher**, d représente la taille du pas. La variable *density* est la densité associée au voxel contenant notre position courante. Une fois que l'on a calculé l'atténuation de notre échantillon, on multiplie notre transmittance réelle par cette valeur (la transmittance est initialisée à 1). Ensuite, on calcule la quantité de lumière qui se rend à notre œil. Pour trouvé cette quantité, on doit déterminer la quantité de lumière qui se rend à chaque point de notre rayon. Pour déterminer cette valeur, à chaque pas que l'on va prendre, on tire un nouveau rayon dans la direction de nos lumières et on le parcourt pour accumuler une opacité. Avec cette opacité, on est en mesure de déterminer la quantité de lumière qui se rend à notre point. À chaque pas, on évalue une nouvelle transmittance ($lightTransmittance$ dans le code) et on utilise une fonction de phase pour déterminer la direction de propagation de la lumière. Le calcul pour l'inténsité finale de la lumière est:

$$lightIntensity = lightColor \times lightTransmittance \times phaseFunction$$
$$result = lightIntesity \times transmittance \times density \times d \times \sigma_s$$

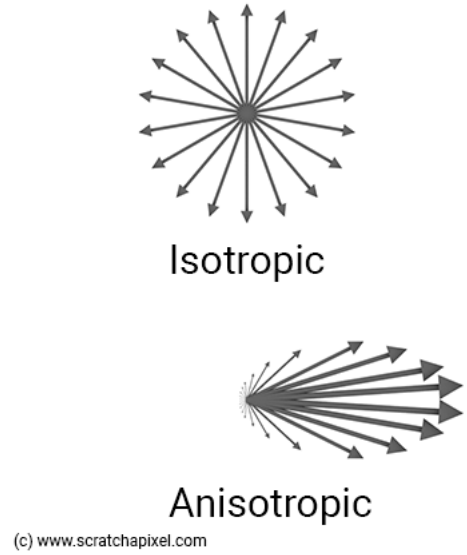
Une fois que l'on a fait tous ces calculs, la couleur finale sera:

$$out\_color = shade() \times transmittance + result$$

La fonction $shade()$ évalue l'ombrage au point d'intersection. Cette fonction peu traitée dans le rapport, car elle existait déjà et très peu de changements lui ont été fais.
Comme mentionné plus haut, il fallait utiliser une fonction de phase pour le calcul de la lumière. Une fonction de phase n'est qu'une fonction qui va éparpiller les rayons dans une certaine direction selon une fonction en particulier. La fonction la plus simple est la fonction de phase isotropique définie par:

$$p_{iso} = \frac{1}{4\pi}$$

Cette fonction éparpille les rayons de lumière de façon égale en suivant la forme d'une sphère.

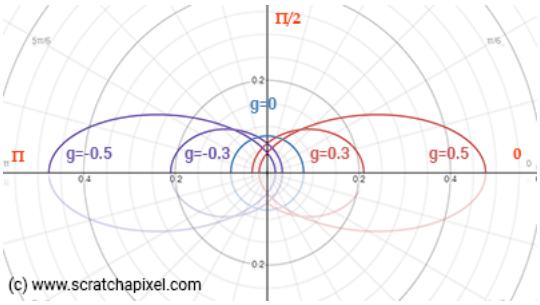


Comparaison entre isotropique et anisotropique

Il y a aussi les fonctions anisotropiques. Contrairement à la fonction isotropique, ces fonctions n'éparpillent pas les rayons de façon égale, mais vont plutôt favoriser une ou des directions particulières. La fonction de phase qui a été utilisée le long du projet est la fonction de Henyey-Greenstein. Cette fonction est définie par:

$$p_{HG} = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 + 2g\cos\theta)^{\frac{3}{2}}}$$

Dans cette formule, g est tout simplement une constante qui se situe dans l'intervalle [-1, 1]. Je tiens à remarquer que si $|g| = 1$ la fonction de phase vaut 0 et si $g = 0$, on obtient une fonction de phase isotropique. La valeur cosθ est le cosinus de l'angle entre le rayon provenant de la caméra et le rayon qui se rend vers la lumière.



Henyey-Greenstein avec plusieurs valeurs de g

Je tiens à préciser certains aspects de l'algorithme. Premièrement, pour réduire la quantité d'artéfacts visuels sur l'image, on obtient la densité grâce à de l'interpolation trilinéaire. Ce processus revient à faire une somme pondérée des densités des huit voxels qui entourent le point. De plus, pour éviter que le **ray marching** soit trop long, j'utilise une méthode **Russian Roulette** pour mettre fin à l'algorithme avant qu'il atteigne la fin du médium. Bref, si la transmittance accumulée atteint un certain seuil, on génère une valeur aléatoire entre 0 et 1 et on vérifie si cette valeur est inférieure à $\frac{1}{d}$. La constante d n'est qu'une valeur choisie arbitrairement (2 dans mon cas). Si jamais on gagne à la roulette, on met fin au **ray marching** sinon, on multiplie la transmittance par d pour compenser et on continue l'algorithme.
Finalement, voici l'algorithme pour évaluer l'attenuation de la lumière:

$transmittance \leftarrow 1$
$colorResult \leftarrow (0, 0, 0)$
$sampleAttenuation \leftarrow \exp\{-step \times (\sigma_a + \sigma_s) \times density\}$
$transmittance \leftarrow transmittance \times sampleAttenuation$
for *light* in *lights* **do**
  $lightRay \leftarrow Ray(position, normalize(light.position - position))$
  if $intersect(lightRay)$ **then**
    $Continue$
  end if
  $lightTransmittance \leftarrow 1$
  for $n < lightIntervals$ **do**
    $lightTransmittance \leftarrow lightTransmittance \times lightSampleAttenuation$
  end for
  $lightIntensity \leftarrow lightColor \times lightTransmittance \times phaseFunction$
  $colorResult \leftarrow colorResult + lightIntensity \times transmittance \times density \times step \times \sigma_s$
  if $transmittance < 1e - 3$ **then**
    if $rand() < \frac{1}{d}$ **then**
      $Stop$
    else

$transmittance \leftarrow transmittance \times d$
end if
end if
end for

Certaines étapes ont été omises pour garder le pseudo-code simple et court. Cependant, les parties qui ne sont pas présentes dans le pseudo-code sont soit des calculs intermédiaires qui n'affectent pas la logique du code ou des morceaux de code qui ont déjà été couverts précédemment.

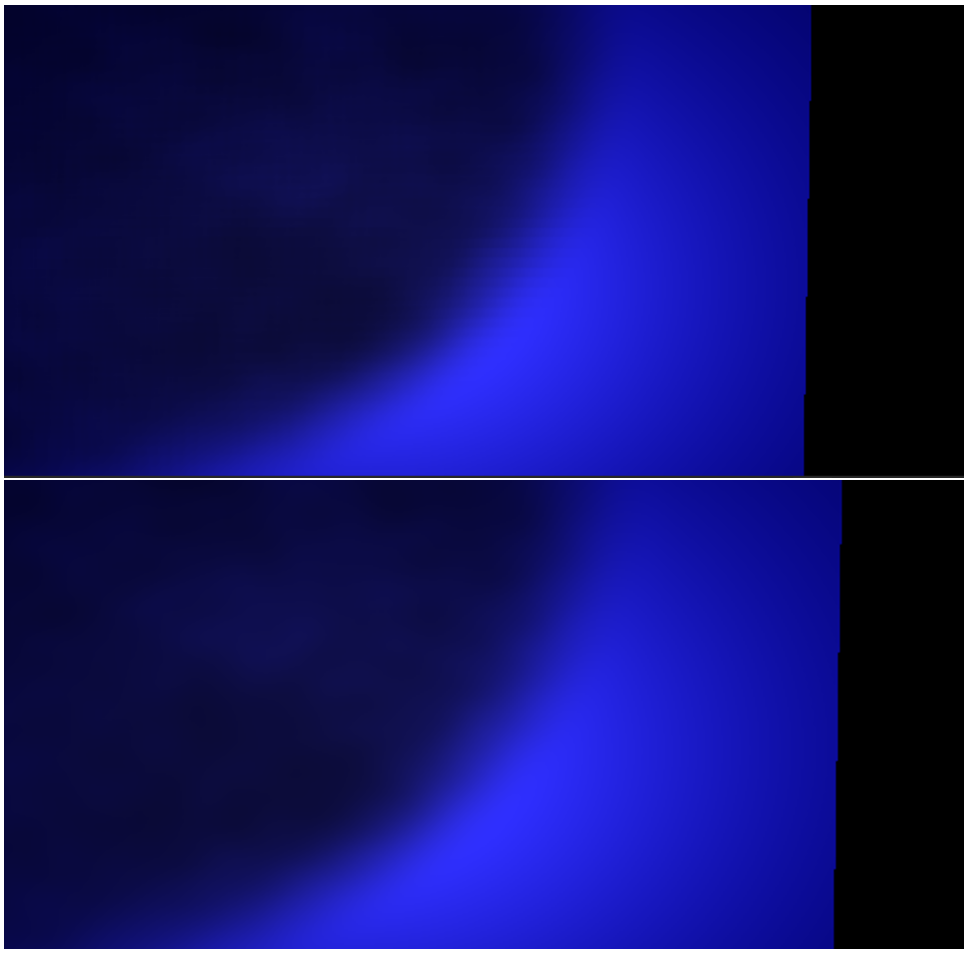


Figure 2: Image du haut: Sans interpolation trilinéaire. Image du bas: Avec interpolation trilinéaire.

5 *shade*()

Comme je l'ai mentionné précédemment, il y a eu très peu de changements apportés à la fonction shade(). Autrefois, le calcul de l'ombre en un point s'effectuait de manière très binaire. Lors de ce calcul, on vérifiait simplement s'il existait un objet bloquant la lumière. En somme, la contribution était soit de 0 (si un objet bloquait la lumière), soit de 1 (si rien ne bloquait la lumière).
Le seul changement nécessaire consistait à faire évoluer cette approche binaire vers une approche continue. Désormais, si un objet bloque la lumière, la contribution reste à 0. Dans le cas contraire, on vérifie si le rayon d'ombre traverse le médium. Si c'est le cas, la quantité de lumière atteignant le point est égale à l'opacité

cumulée le long du rayon. Si le rayon d'ombre ne rencontre ni objet ni médium, alors la contribution demeure 1.
Mis à part cette modification, le calcul du shading demeure identique. Toutefois, l'intensité de la lumière utilisée dans le modèle de Blinn-Phong est maintenant égale à l'intensité originale de la source lumineuse, multipliée par le facteur d'opacité ainsi déterminé.

6 Conclusion

En conclusion, ce projet m'a permis d'en apprendre beaucoup à propos des techniques de rendu de volumes. En effet, cette technique peut être généralisée et utilisée pour faire le rendu d'un liquide comme de l'eau [6].

Il y a plusieurs améliorations possibles que je pourrais éventuellement faire à se projet. La première et la plus évidente serait d'implémenter l'émission du volume participatif. Malheureusement, comme je l'ai déjà mentionné, je n'ai pas pus implémenter cette fonctionnalité dans mon code à temps pour la remise. Une seconde amélioration serait de traduire mon code C++ en *shader language*. En ce moment, mon *ray tracer* ne fait que générer une seule image par exécution du code. Cependant, il pourrait être intéressant d'avoir un rendu en temps réel. Pour faire cela, il va falloir que j'utilise le GPU et que j'écrive la logique de mon code dans des *vertex shaders* et des *fragment shaders* si j'utilise un API graphique comme **OpenGl** ou **Vulkan** ou dans des *compute shaders* si je travaille avec le moteur de jeu **Unity**. La dernière amélioration que je crois pourrait être intéressante serait d'implémenter la capacité au *ray tracer* de faire le rendu de plusieurs types de fluide. En ce moment, je ne peux que générer de la fumée. Cependant, comme je l'ai mentionné plus haut, la méthode pour générer d'autres types de fluides est sensiblement la même.

Pour finir ce rapport, voici quelques rendus que j'ai obtenus grâce à mon projet!

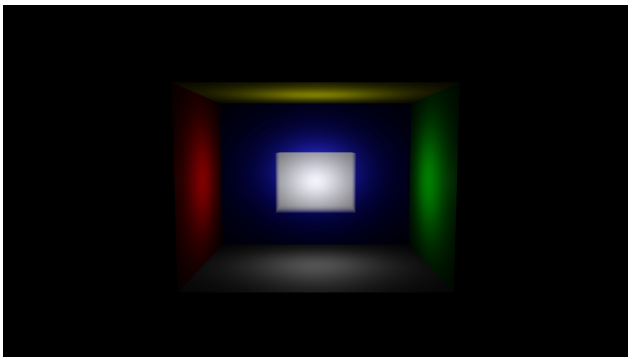


Figure 3: Densité uniforme de 1

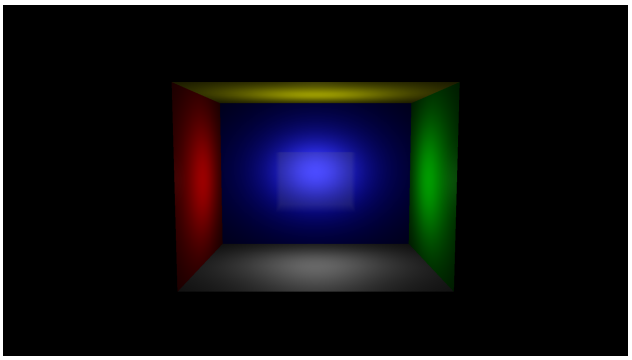


Figure 4: Densité uniforme de 0.07

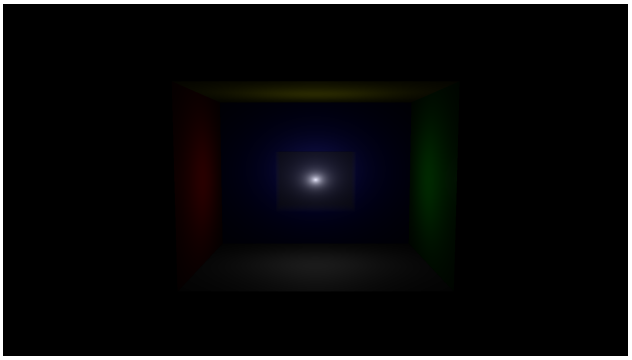


Figure 5: Paramètre de Henyey-Greenstein de -0.65

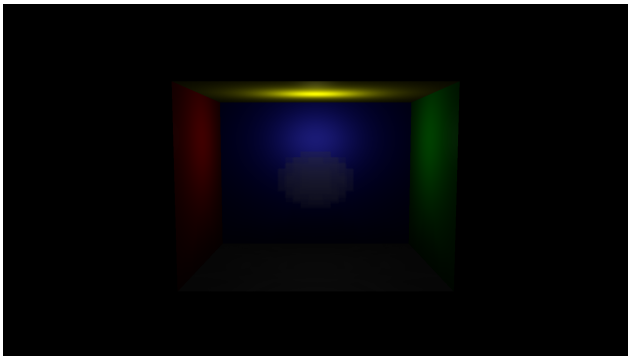


Figure 6: Sphère sans interpolation tri-linéaire

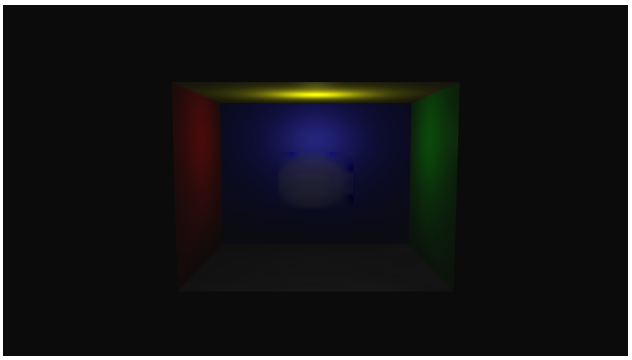


Figure 7: Sphère avec interpolation tri-linéaire

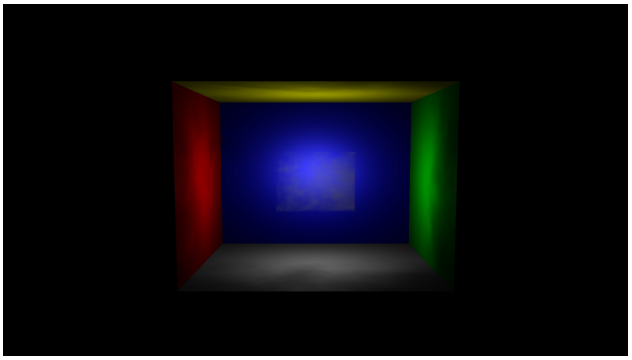


Figure 8: Grille générée avec du bruit de perlin

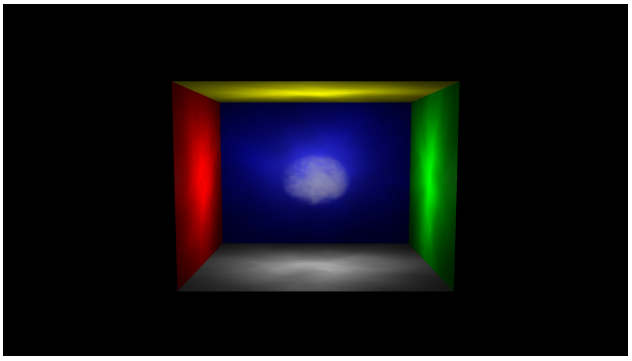


Figure 9: Sphère générée avec du bruit de perlin

References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proceedings of EuroGraphics*, 87, 08 1987.

[2] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '77, page 192–198, New York, NY, USA, 1977. Association for Computing Machinery.

[3] Gyurgyik C and Kellison A. An overview of the fast voxel traversal algorithm. https://github.com/cgyurgyik/fast-voxel-traversal-algorithm/blob/master/overview/FastVoxelTraversalOverview.md, 2022.

[4] Josh's Channel. How ray tracing (modern cgi) works and how to do it 600x faster. https://www.youtube.com/watch?v=gsZiJeaMO48, 2022.

[5] Sebastian Lague. Coding adventure: Ray tracing. https://www.youtube.com/watch?v=Qz0KTGYJtUk, 2023.

[6] Sebastian Lague. Coding adventure: Rendering fluids. https://youtu.be/kOkfC5fLfgE?si=aP0OiqEs7PTszgTI, 2024.

[7] Steve Hollasch Peter Shirley, Trevor David Black. Ray tracing in one weekend, August 2024. https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[8] Wikipedia contributors. Beer–Lambert law — Wikipedia, the free encyclopedia, 2024. [Online; accessed 12-December-2024].

[9] Wikipedia contributors. Digital differential analyzer (graphics algorithm) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Digital_differential_analyzer_(graphics_algorithm)&oldid=1236307484, 2024. [Online; accessed 3-December-2024].

[10] Wikipedia contributors. Perlin noise — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Perlin_noise&oldid=1258165795, 2024. [Online; accessed 3-December-2024].

[11] Wikipedia contributors. Ray tracing (graphics) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ray_tracing_(graphics)&oldid=1253778958, 2024. [Online; accessed 3-December-2024].

[12] Wikipedia contributors. Voxel — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Voxel&oldid=1260713506, 2024. [Online; accessed 3-December-2024].