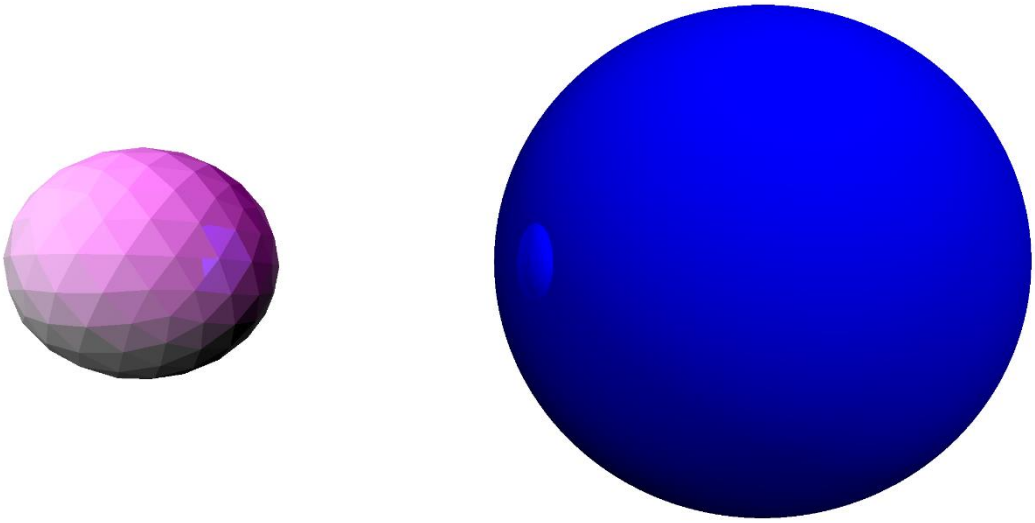
Collège Voltaire

# Réalisation d'un moteur de rendu 3D

Maître accompagnant : Thomas Speer



Valuthy Karunakaran
2021

# TABLE DES MATIÈRES

# INTRODUCTION

Les images en 3D sont présentes partout autour de nous. Les jeux-vidéo, le cinéma — Que ce soit dans les films animés ou dans les effets spéciaux — et tous les domaines utilisant l'image comme moyen de communication utilisent des méthodes diverses et variées. Par exemple les jeux-vidéo nécessitent de pouvoir afficher les images rapidement pour fournir une expérience de jeu plaisante, tandis que le cinéma ou le graphisme 3D disposent d'un temps de calcul presque infini. Ces derniers peuvent donc utiliser des techniques plus complexes, plus réalistes et plus couteuses en énergie, alors que les jeux-vidéo prendront beaucoup de raccourcis pour s'approcher le plus possible d'une image de qualité sans sacrifier les performances.

Je m'intéresserai dans le cadre de ce travail de maturité à une méthode plutôt utilisée dans le cinéma et dans le graphisme 3D : le *raytracing*, que l'on pourrait traduire par « tracés de rayons ». Ce dernier donne des résultats généralement plus réalistes que la *rastérisation*, processus souvent utilisé dans les jeux-vidéo, surtout au niveau des jeux de lumières : ombres, réflexions et réfractions.

Le but final de cette réalisation technique est de créer un programme informatique utilisant des droites (*rays*) pour détecter les objets présents sur une scène et générer une image qui donne une impression de 3D. Je présenterai à travers ce document le processus suivi pour arriver à l'image sur la page de garde ainsi que les problèmes rencontrés et comment ils ont été résolus.

Nous commencerons par voir comment la méthode de raytracing fonctionne dans mon programme à travers différents explications techniques et mathématiques. Au fur et à mesure, des images générées par le programme seront présentées pour illustrer les explications. Finalement, nous ferons un point sur les fonctionnalités que je n'ai pas réussi à ajouter et pourquoi cela n'a pas été possible.

## PHASE DE RECHERCHE

Avant de travailler avec l'ordinateur pour débuter les expérimentations concernant le projet, j'ai commencé par me renseigner sur le raytracing. Je connaissais déjà l'existence de la technique et son fonctionnement général grâce à une vidéo sur YouTube, donc la majorité de mes recherches se portaient sur les solutions mathématiques ou les outils informatiques que je pouvais utiliser pour implémenter l'algorithme. J'ai pu assez simplement trouver la formule de l'intersection rayon-sphère, mais je voulais éventuellement pouvoir afficher des modèles 3D quelconques. Je savais que les moteurs de rendu 3D classiques comme OpenGL utilisent de nombreux triangles pour représenter des formes. J'avais donc aussi besoin d'une formule pour l'intersection rayon-triangle, que j'ai trouvé grâce à un article publié en 1997.

# LE RAYTRACING

Le raytracing est un algorithme permettant généralement de détecter des objets en utilisant des droites. Il est par exemple utilisé dans d'anciens jeux de tirs pour détecter les « balles » qui atteindraient leur cible. Pour ce faire, une droite — que nous appellerons désormais rayon — est utilisée pour trouver une potentielle intersection avec les *hitbox*[1] des cibles comme illustré dans la figure *1*. Un rayon est défini avec une *origine* et une *direction* normalisée, c'est-à-dire que la norme du vecteur représentant la direction est égale à 1.0.

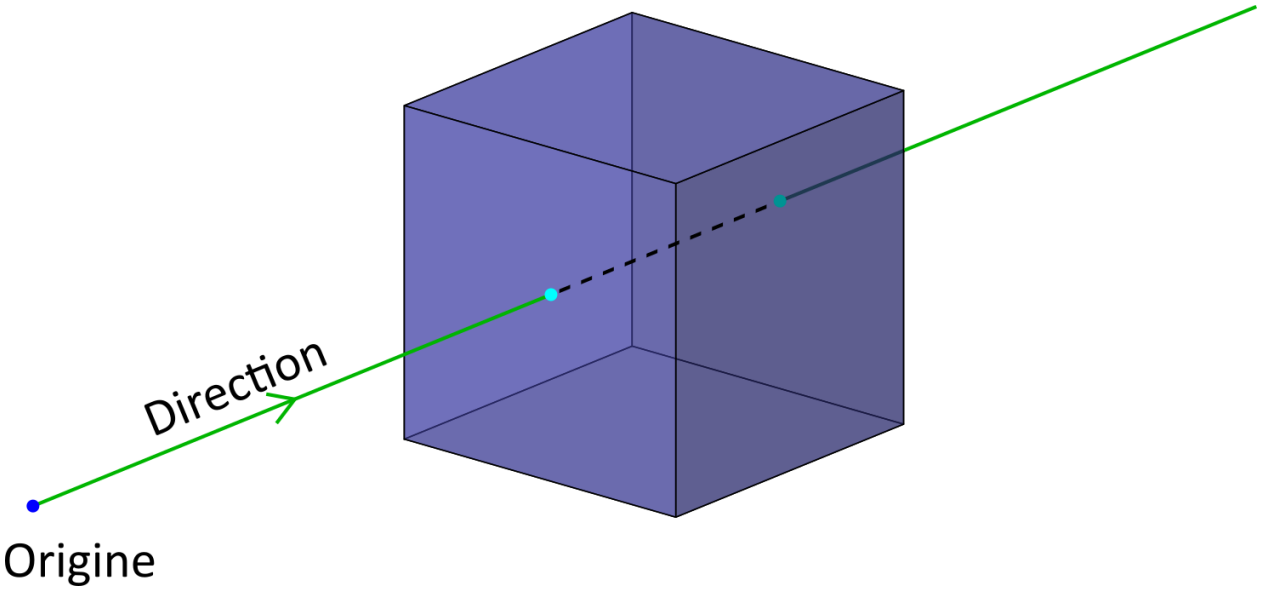



**Figure 1 - Intersection entre une hitbox sous forme de cube et un rayon**

Nous ferons la même chose, nous allons donc détecter nos objets sur la scène à l'aide de rayons, mais nous allons en utiliser beaucoup plus, un par pixel à l'écran. Le fait de savoir si un rayon intersecte avec un objet nous permettra de savoir si et comment le pixel correspondant doit être coloré.

Il faut d'abord définir un rayon avec une origine $\vec{o}$ et une direction $\hat{d}$. Puisque $\hat{d}$ est un vecteur directeur, il est normalisé, c'est-à-dire qu'il a une norme qui est égale à *1.0*. Ainsi, on peut poser l'équation paramétrique de notre rayon :

$$\begin{cases} x = o_x + d_x \cdot t \\ y = o_y + d_y \cdot t \\ z = o_z + d_z \cdot t \end{cases}$$

Ici, grâce au vecteur directeur normalisé, *t* représente la distance depuis l'origine d'un point dans l'espace *(x ; y ; z)* donné.

Prenons pour la position *(0 ; 0 ; 0)* comme origine commune de tous les rayons. La caméra est donc positionnée dans ce cas à l'origine de la scène. Pour la direction, cela se complique, car elle doit être générée correctement en fonction du champ de vision de la caméra et la définition de l'image sur laquelle on veut faire le rendu. Pour ce faire, on peut imaginer un plan (voir figure *2*) qui est placé devant la caméra et qui a la même taille que l'image qu'on souhaite générer (un pixel représente donc $1u^2$ dans la scène).

[1] Une *hitbox* est un objet souvent invisible au joueur qui a une forme plus simple que l'objet auquel il est lié et qui facilite donc les calculs d'intersections avec ce dernier.

On peut maintenant faire « aller » la direction d'un rayon vers son pixel correspondant en utilisant les coordonnées *(x ; y)* dans l'image de ce dernier. Il manque maintenant plus que la coordonnée *z* de $\hat{d}$ que nous appellerons *p*.

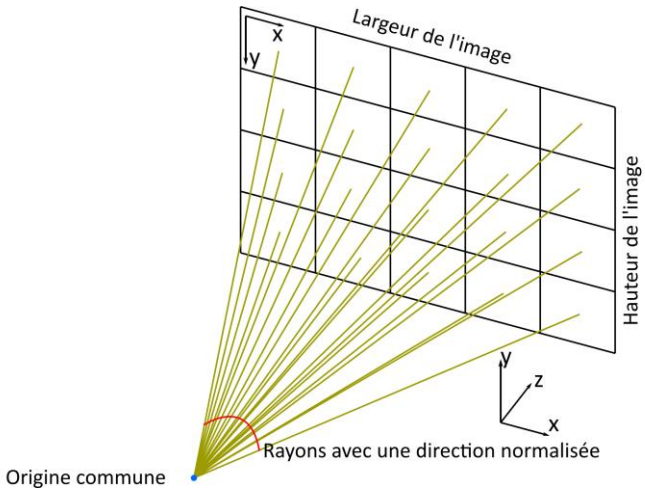



**Figure 2 - Disposition des rayons formant la caméra**

## CHAMP DE VISION ET *P*

Le champ de vision de la caméra dépend du $p$ choisi et inversement. Les fonctions trigonométriques permettent de déterminer un $p$ (Voir figures *3* et *4*) respectant le champ de vision et la définition de l'image choisie. Dans mon cas, j'ai choisi arbitrairement d'indiquer seulement un champ de vision horizontal. Soient $\alpha$ la moitié du champ de vision de la caméra, $L$ la largeur de l'image et $p$ la profondeur recherchée, on peut poser le système d'équations suivant :

$$\begin{cases} \mathrm{p} = \cos(\alpha) \cdot \mathrm{h} \\ \frac{L}{2} = sin(\alpha) \cdot h \end{cases}$$

En résolvant pour trouver *p*, on trouve :

$$p = \frac{cos(\alpha) \cdot L}{2 \cdot sin(\alpha)}$$

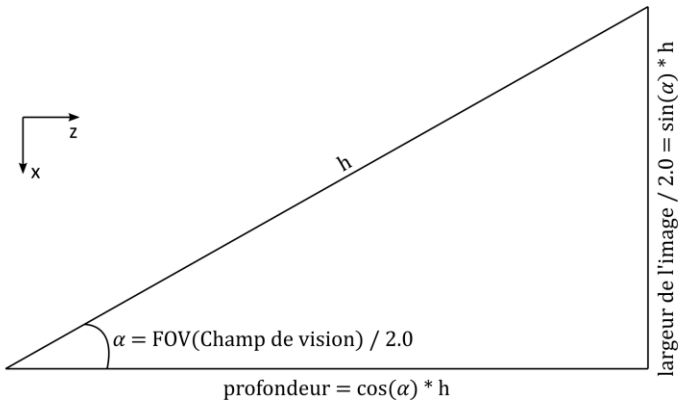



**Figure 3 – Relations trigonométriques permettant de trouver la profondeur**

On peut pour l'instant définir la direction comme $\begin{pmatrix} x_0 \\ y_0 \\ p \end{pmatrix}$, ou $x_0$ et $y_0$ sont les coordonnées du pixel correspondant dans l'image. Néanmoins, il faut décaler la valeur de $x_0$ et $y_0$ en leur soustrayant respectivement la moitié de la largeur et de la hauteur de sorte à ce qu'ils soient dans la moitié des cas dans le négatif. On ajoute également *0.5* à la valeur obtenue pour centrer la direction sur le pixel, on dirige ainsi correctement les rayons pour qu'ils respectent le champ de vision et qu'ils soient bien disposés par rapport à l'image virtuelle (celle représentée sur la figure *2*). On évite donc la situation aux figures *4* et *5*.

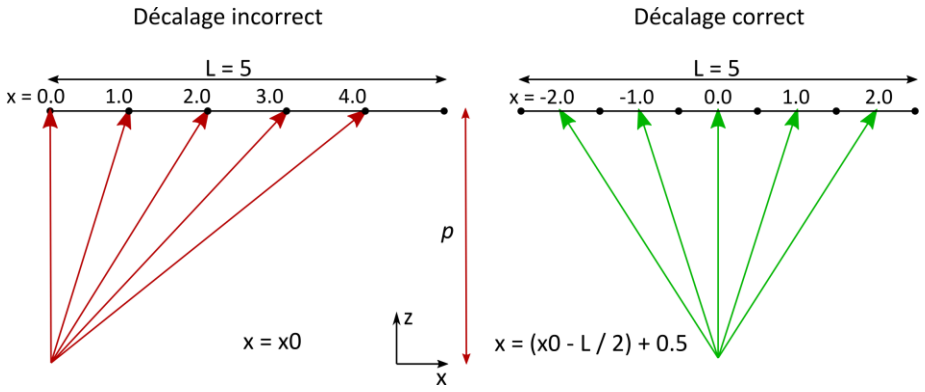



**Figure 4 - Vue du dessus de la disposition des rayons avec et sans décalage horizontalement (même principe que verticalement)**

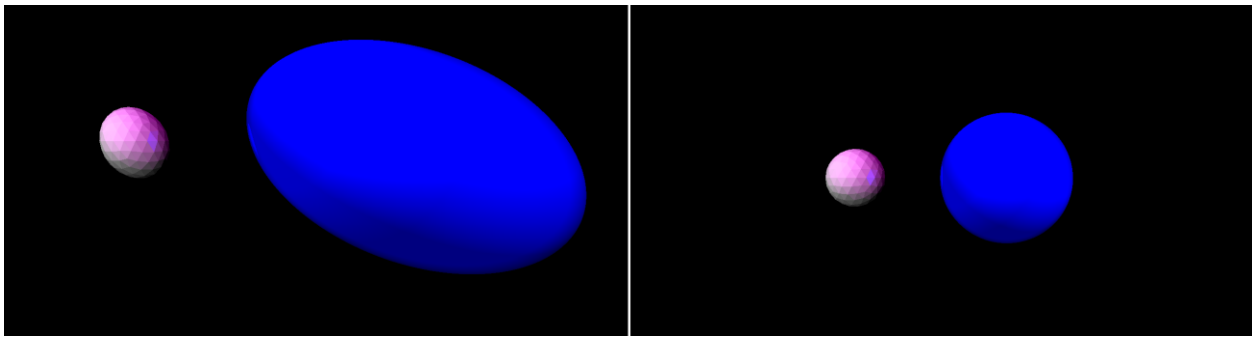


**Figure 5 - Comparaison du résultat à l'image sans décalage(gauche) et avec(droite)**

Avec $H$ la hauteur de l'image, nous avons :

$$\vec{d} = \begin{pmatrix} x_0 - \frac{L}{2} + 0.5 \\ y_0 - \frac{H}{2} + 0.5 \\ p \end{pmatrix}$$

On le normalise pour respecter la définition du rayon donnée plus haut :

$$\hat{d} = \frac{\vec{d}}{||\vec{d}||}$$

Maintenant, nous avons des rayons initialisés avec les bonnes directions, mais pour pouvoir détecter les objets, il faut une méthode permettant trouver les intersections entre les rayons et ces derniers.

# INTERSECTION RAYON-SPHÈRE

On peut définir mathématiquement une sphère avec une position ($\vec{p}$) de son centre et son rayon ($r$). Ainsi, on peut définir l'équation cartésienne d'une sphère :

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0$$

où $\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \vec{p}$ et $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ un point dans l'espace. Cette équation inclut dans la sphère tous les points de l'espace pour lesquels l'égalité est vraie. On reprend notre rayon d'une origine $\vec{o}$ et d'une direction $\hat{d}$ et son équation paramétrique : $\begin{cases} x = o_x + d_x \cdot t \\ y = o_y + d_y \cdot t \\ z = o_z + d_z \cdot t \end{cases}$, en substituant $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ par cette dernière, on obtient :

$$(o_x + d_x \cdot t - x_0)^2 + \left(o_y + d_y \cdot t - y_0\right)^2 + (o_z + d_z \cdot t - z_0)^2 - r^2 = 0$$

En développant tout, on remarque qu'on peut mettre en évidence par rapport à $t$ et $t^2$ :

$$\left({d_x}^2 + {d_y}^2 + {d_z}^2\right) \cdot t^2 + 2\left(\left(d_x \cdot o_x + d_y \cdot o_y + d_z \cdot o_z\right) - \left(d_x \cdot p_x + d_y \cdot p_y + d_z \cdot p_z\right)\right) \cdot t + \left({p_x}^2 + {p_y}^2 + {p_z}^2 - 2 \cdot \left(o_x \cdot p_x + o_y \cdot p_y + o_z \cdot p_z\right) + {o_x}^2 + {o_y}^2 + {o_z}^2\right) - r^2 = 0$$

On remarque ainsi l'apparition des coefficients $a$, $b$, et $c$ d'une équation quadratique, donc on peut très simplement calculer $t$ en utilisant la formule de Viète.

## CAS POSSIBLES DE VIETE

On peut avoir 3 cas possible avec cette formule : 2 solutions ($\Delta > 0$), 1 solution ($\Delta = 0$) et aucune solution ($\Delta < 0$). Cela s'interprète respectivement comme une sphère percée en 2 points par le rayon, une sphère qui est frôlée par le rayon et une sphère qui n'est pas traversée par le rayon. Dans le cas où il y a une seule solution, il faut s'assurer qu'elle est positive, car une solution négative signifie que la sphère est derrière la caméra. Dans le cas où il y a 2 solutions, on ne prend que les solutions positives et parmi elles, on choisit la plus petite.

## UTILISATION DE *T*

La distance depuis l'origine du rayon est une donnée importante. Tout d'abord c'est elle qui permet de savoir quel objet est le plus proche de la caméra, donc la couleur de quel objet doit être prise en compte dans la suite des calculs. $t$ est aussi utile pour déterminer un point d'impact, qu'on peut définir comme suit :

$$\vec{\imath} = \vec{o} + \hat{d} \cdot t$$

# ET LA LUMIÈRE FUT…

Pour avoir de l'ombre, qui rendrait l'image un peu plus vivante, il faut d'abord de la lumière. On peut définir une source de lumière avec une position dans l'espace, une couleur et une intensité. Lorsqu'on détecte un objet avec un rayon, pour savoir si cette zone est ombrée ou non, on va relancer un rayon depuis le point d'impact ($\vec{\imath}$) en direction de *toutes* les sources de lumières présentes. Si on détecte à nouveau un objet avec le nouveau rayon pour une source donnée, on en conclut que sa lumière ne parvient pas au point d'impact initial. On additionne les couleurs des sources valides (les couleurs sont stockées et interprétées

comme des vecteurs dans le code) et on obtient la couleur finale de la lumière incidente qui arrive au point d'impact (synthèse additive).

Pour calculer la couleur perçue sur un objet d'une couleur *a* éclairée par de la lumière de couleur *b*, il suffit de multiplier les composantes R, G et B d'*a* et *b* entre elles, c'est la synthèse multiplicative (aussi appelée synthèse soustractive). Par exemple, un objet rouge *(1.0, 0.0, 0.0)* éclairé par une lumière jaune *(1.0, 1.0, 0.0)* sera perçu comme rouge, car un objet rouge absorbe par définition toutes les longueurs d'onde sauf le rouge, il absorbe dans ce cas la partie verte de la lumière. Un objet jaune éclairé par une lumière bleue paraîtra noir, car on multiplie *(1.0, 1.0, 0.0)* avec *(0.0, 0.0, 1.0)*, chacune des composantes vaut *0.0*. Ceci est bien-sûr un modèle, car dans la vie réelle les objets ne sont pas parfaitement jaunes et la lumière pas toujours parfaitement absorbée.

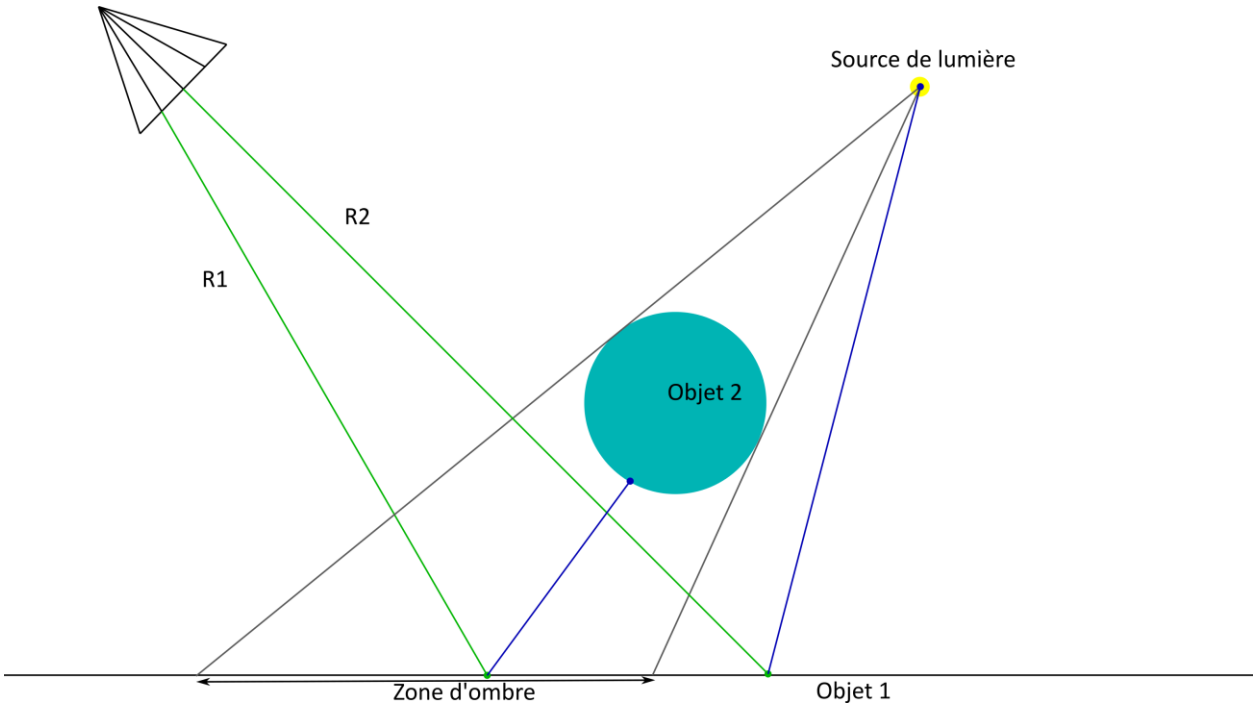



Figure 6 - Schéma de la méthode d'ombrage

# IMPLÉMENTATION DU PROTOTYPE

Un programme informatique se crée en écrivant le code source dans un des nombreux langages de programmation existants. J'avais dès le début du projet 3 langages en tête : *Python*, *Lua* et *Rust*. Pour simplifier grossièrement, Python et Lua(interprétés) sont des langages donnant des programmes lents mais simples à écrire, tandis que Rust(compilé) est beaucoup plus rapide mais aussi plus complexe à écrire. Python et Lua sont donc de parfaits candidats pour un prototype ou pour tester rapidement un algorithme, c'est pourquoi j'ai commencé la phase d'expérimentation avec Lua en utilisant LÖVE2D. C'est un choix purement subjectif, j'avais déjà utilisé LÖVE2D auparavant pour faire des petites « expériences » graphiques.

L'écriture du prototype était assez simple, j'ai trouvé une bibliothèque permettant de créer des vecteurs et appliquer des opérations vectorielles qui m'a été très utile. J'ai rapidement mis en place le squelette de base du programme (gestion de la fenêtre, lien entre chaque rayon et son pixel associé, etc.). Puis j'ai pu

assez rapidement implémenter les bases du raytracing comme la génération des rayons et la détections de sphères.

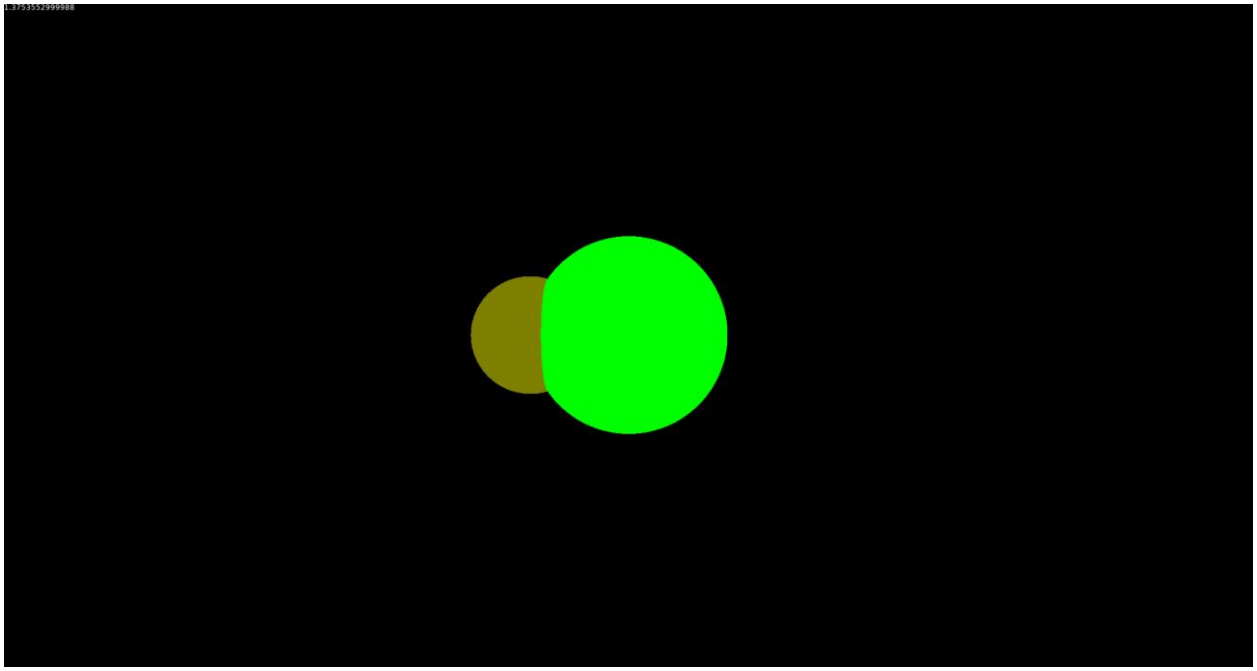


Figure 7 - Première version du prototype

On voit sur la figure *7* 2 sphères que j'ai placé sur la scène, celle de gauche est à la position *(-5 ; 0 ; 32)* avec un rayon de *3* et l'autre sphère se situe en *(0 ; 0 ; 32)* avec un rayon de *5*. La couleur pour chaque rayon/pixel est simplement la couleur assignée à la sphère. Le but de cette configuration était de vérifier que les sphères sont bien détectées et surtout que les positions des sphères sont correctement interprétées, c'est-à-dire que les parties de l'image au premier plan sont bien ceux qui sont plus proches de la caméra dans la scène. On utilise donc les *t* obtenus pour les 2 sphères pour savoir laquelle est au premier plan.

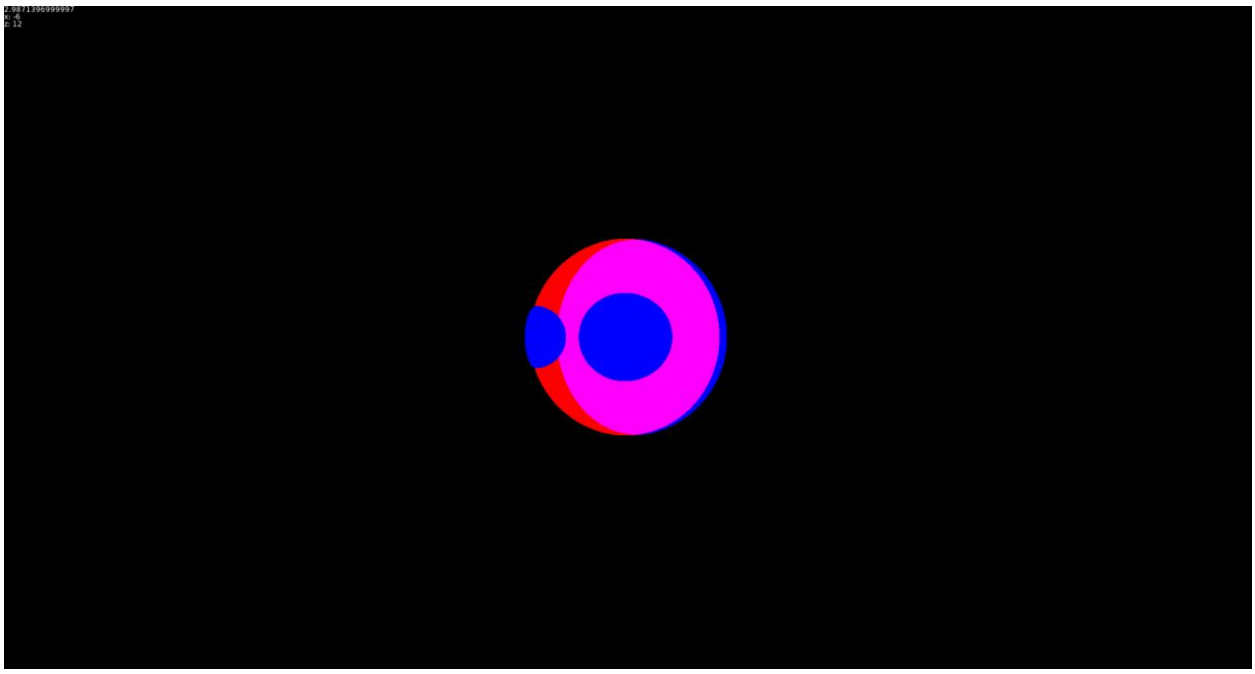


Figure 8 - Ombres portées et source de lumière

Ici, il y a une sphère de couleur bleue et de rayon 1 en *(-3 ; 0 ; 20)*, une sphère de couleur blanche et de rayon 5 en *(0 ; 0 ; 32)*. Deux sources de lumières sont présentes, une de couleur rouge en *(-6 ; 0 ; 12)* et une autre de couleur bleue en *(10 ; 0 ; 17)*. On utilise donc ici l'algorithme décrit précédemment pour faire apparaître une ombre et on remarque l'effet de la synthèse additive.

# COULEUR DIFFUSE ET DISTANCE

A partir de la version montrée sur la figure (*10*), on peut ajouter des opérations très simples à la couleur d'une lumière incidente qui permettront d'avoir un effet 3D beaucoup plus convaincant. En prenant en compte la distance de la source de lumière et l'angle à laquelle la lumière frappe la surface, on peut avoir une surface éclairée de façon plus réaliste. Définissons donc deux facteurs *dist_f* et *diff_f*, qui représentent respectivement l'effet de la distance jusqu'à la source et l'effet de l'angle d'incidence de la lumière.

*dist_f* utilise la loi en carré inverse de l'intensité lumineuse. Cette loi, qui a son équivalent pour d'autres types de « quantités » comme la gravité ou le son, dit que l'intensité lumineuse en un point diminue avec le carré de la distance entre ce point et la source. Cela implique donc d'ajouter une propriété définissant l'intensité lumineuse pour les sources de lumières.

$$dist\_f = \frac{1}{\mathrm{d}^2}$$

*diff_f* est une valeur qui approche 1.0 lorsque l'angle entre la normale[2] de la surface et le vecteur partant du point d'impact vers la source approche 0.0. Lorsqu'une surface est frappée par la lumière de face, elle paraît plus lumineuse que quand la lumière arrive en longeant la surface. La normale d'une sphère en un point peut être calculée en utilisant le point d'impact ($\vec{\imath}$) et la position de la sphère ($\vec{p}$) :

$$\hat{n} = \frac{\vec{\imath} - \vec{p}}{||\vec{\imath} - \vec{p}||}$$

Soient $\vec{l}$ la position de la source de lumière et $\alpha$ l'angle entre la normale et la direction de la source, le vecteur indiquant la direction de la lumière est calculé de la façon suivante :

$$\widehat{ld} = \frac{\vec{l} - \vec{\imath}}{||\vec{l} - \vec{\imath}||}$$

et $\alpha$ peut être calculé grâce à la définition suivante du produit scalaire :

$$\vec{u} \cdot \vec{v} = ||\vec{u}|| \cdot ||\vec{v}|| \cdot cos(\alpha)$$

$\hat{n}$ et $\widehat{ld}$ sont normalisé, donc $\hat{n} \cdot \widehat{ld} = cos(\alpha) \Leftrightarrow \hat{n} \cdot \widehat{ld} \in [-1; 1]$ et $\hat{n} \cdot \widehat{ld} = 1$ quand $\alpha = 0$ et $\hat{n} \cdot \widehat{ld} = 0$ quand $\alpha = \frac{\pi}{2}$. On définit finalement $diff\_f = \hat{n} \cdot ld$ en restreignant sa valeur entre 0 et 1 car on ne veut pas qu'il soit négatif.

La couleur de la lumière incidente d'une source donnée est multipliée par l'intensité de cette dernière et les facteurs *dist_f* et *diff_f* pour obtenir une couleur finale qui sera additionnée aux couleurs finales des autres sources qui auront subi les mêmes multiplications. On applique cette somme à la couleur de l'objet par synthèse multiplicative et on l'appelle la couleur diffuse $\overrightarrow{cd}$ en $\vec{\imath}$.

[2] Une normale est un vecteur normalisé (sa norme vaut 1) et perpendiculaire à la surface associée

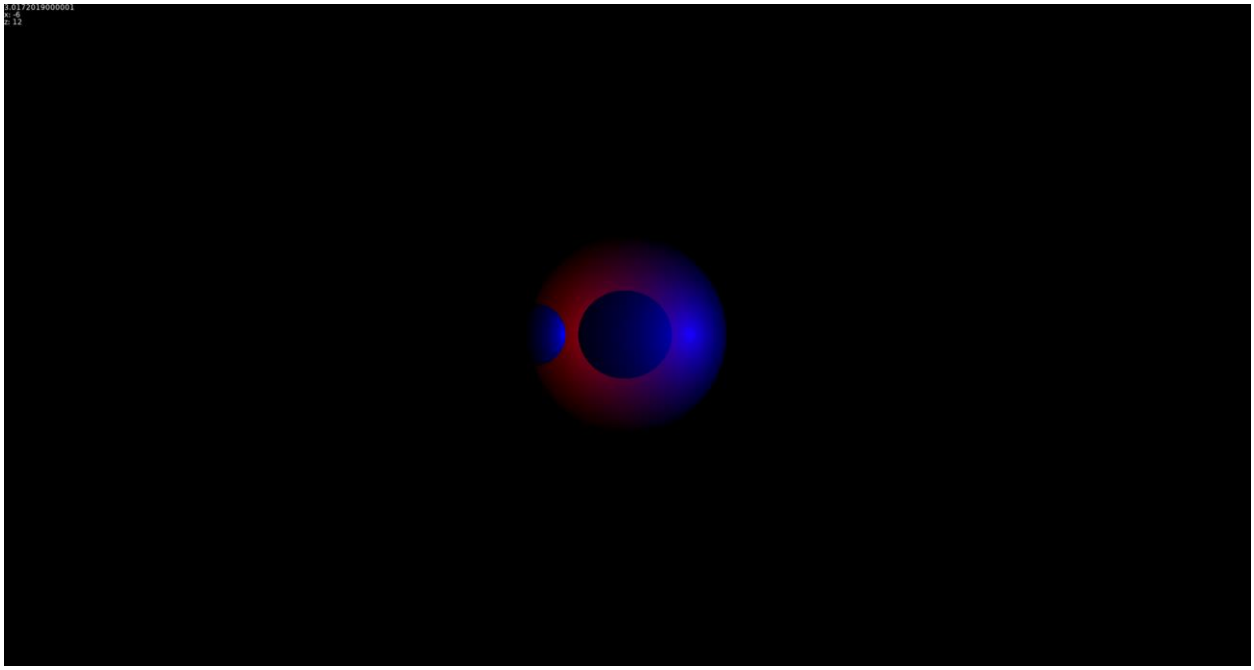


Figure 9 - Seconde version du prototype

Sur la figure *9*, les sphères ont gardé leur position et leur rayon, la grande boule est de couleur blanche *(1.0, 1.0, 1.0)* et la petite de couleur bleue *(0.0, 0.0, 1.0)*. Le tout est éclairé par les deux mêmes lumières d'intensité *200.0*. L'effet 3D se ressent largement plus grâce à l'utilisation de *dist_f et diff_f*.

# INTERSECTION RAYON-TRIANGLE

Pour pouvoir afficher des modèles 3D quelconques, il faut pouvoir détecter une intersection avec un rayon. En graphisme 3D, les modèles sont dans la plupart des représentations numériques des collections de triangles. En ayant à disposition une méthode d'intersection entre un rayon et un triangle, on peut l'appliquer sur tous les triangles d'un modèle pour savoir s'il y a des intersections. Comme avec la sphère, il faut trouver la valeur du *t* dans l'équation vectorielle du rayon.

Commençons par définir le triangle avec les points $\vec{A}, \vec{B}, \vec{C}$. On peut à partir de ces points définir $\overrightarrow{AB} = \vec{B} - \vec{A}$ et $\overrightarrow{AC} = \vec{C} - \vec{A}$. La surface du triangle peut être exprimé comme suit :

$$T(u,v) = \vec{A} + \overrightarrow{AB} \cdot u + \overrightarrow{AC} \cdot v$$

où $(u,v) \in [0;1]$ et $u + v \leq 1$ pour que $T$ forme un triangle. On peut donc le mettre en égalité avec l'équation vectorielle du rayon :

$$\vec{o} + \vec{d} \cdot t = \vec{A} + \overrightarrow{AB} \cdot u + \overrightarrow{AC} \cdot v \Leftrightarrow \vec{o} - \vec{A} = \overrightarrow{AB} \cdot u + \overrightarrow{AC} \cdot v - \vec{d} \cdot t$$

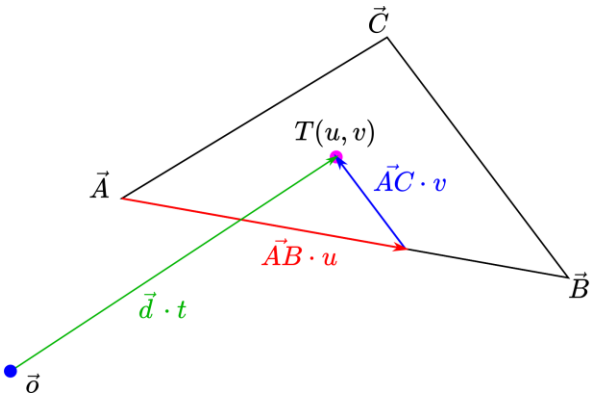



Figure 10 - Schéma de l'intersection rayon-triangle

Désignons $\vec{o} - \vec{A}$ par $\vec{P}$, puisque ce ne sont pas des inconnus, pour simplifier l'équation :

$$\vec{P} = \overrightarrow{\mathrm{AB}} \cdot u + \overrightarrow{AC} \cdot v - \vec{d} \cdot t$$

En passant à l'équation paramétrique, on obtient :

$$\begin{cases} P_x = AB_x \cdot u + AC_x \cdot v - d_x \cdot t \\ P_y = AB_y \cdot u + AC_y \cdot v - d_y \cdot t \\ P_z = AB_z \cdot u + AC_z \cdot v - d_z \cdot t \end{cases}$$

Puisque ce système d'équations a autant de solutions que d'équations et qu'il ne peut pas y avoir plus d'une solution, car ce serait géométriquement impossible, la règle de Cramer peut être utilisée pour trouver *u*, *v* et *t*. On écrit donc les équations comme une transformation d'un vecteur par une matrice donnant comme résultat $\vec{P}$ :

$$\begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = \begin{bmatrix} AB_x & AC_x & -d_x \\ AB_y & AC_y & -d_y \\ AB_z & AC_z & -d_z \end{bmatrix} \begin{pmatrix} u \\ v \\ t \end{pmatrix}$$

Désignons $\begin{bmatrix} AB_x & AC_x & -d_x \\ AB_y & AC_y & -d_y \\ AB_z & AC_z & -d_z \end{bmatrix}$ par **M**.

Selon la règle de Cramer :

$$u = \frac{\begin{vmatrix} P_x & AC_x & -d_x \\ P_y & AC_y & -d_y \\ P_z & AC_z & -d_z \end{vmatrix}}{|M|}$$

$$v = \frac{\begin{vmatrix} P_x & AB_x & -d_x \\ P_y & AB_y & -d_y \\ P_z & AB_z & -d_z \end{vmatrix}}{|M|}$$

$$t = \frac{\begin{vmatrix} P_x & AB_x & AC_x \\ P_y & AB_y & AC_y \\ P_z & AB_z & AC_z \end{vmatrix}}{|M|}$$

Il faut remarquer que si $|M| = 0$, il n'y a pas de solutions à cause de la division par 0, donc pas d'intersection et le rayon est parallèle au triangle. Dans l'implémentation, les déterminants sont calculés selon la définition suivante :

$$\vec{a} \cdot (\vec{b} \times \vec{c}) = \begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{vmatrix}$$

On a donc :

$$|M| = \overrightarrow{AB} \cdot (\overrightarrow{AC} \times -\vec{d})$$

$$u = \frac{\vec{P} \cdot (\overrightarrow{AC} \times -\vec{d})}{|M|}$$

$$v = \frac{\vec{P} \cdot (\overrightarrow{AB} \times -\vec{d})}{|M|}$$

$$t = \frac{\vec{P} \cdot (\overrightarrow{AB} \times \overrightarrow{AC})}{|M|}$$

Les formules peuvent être réarrangées grâce aux propriétés des produits vectoriels et scalaires pour minimiser l'utilisation des produits vectoriels. Nous avons donc :

$$|M| = \overrightarrow{AB} \cdot (\vec{d} \times \overrightarrow{AC})$$

$$u = \frac{\vec{P} \cdot (\vec{d} \times \overrightarrow{AC})}{|M|}$$

$$v = \frac{\vec{d} \cdot (\vec{P} \times \overrightarrow{AB})}{|M|}$$

$$t = \frac{\overrightarrow{AC} \cdot (\vec{P} \times \overrightarrow{AB})}{|M|}$$

Ici, il y a seulement 2 produits vectoriels à effectuer au lieu de 4 au départ. Nous avons donc la distance $t$ depuis l'origine du rayon à laquelle ce dernier touche le triangle, ainsi que $u$ et $v$ qui permettent de localiser le point d'impact par rapport aux points du triangle. $u$ et $v$ seront très utiles plus tard, notamment lorsqu'on voudra appliquer des textures sur nos modèles 3D.

# RENDU DE MODÈLES

Comme dit plus tôt, les modèles 3D sont constitués de nombreux triangles. Avec la méthode d'intersection avec un triangle en poche, on peut désormais déterminer les intersections avec des modèles. En intersectant le rayon avec tous les triangles faisant partie d'un modèle, 2 triangles sont généralement détectés. Ici, même raisonnement qu'avec la sphère, on prend la distance *t* la plus petite des solutions non-négatives. Le calcul de la lumière diffuse suit le même principe que la sphère, il n'y a que le calcul de la normale qui diffère.

Les modèles sont chargés en mémoire à partir de fichiers au format *obj*. Le fichier contient une liste de positions ayant chacun un indice associé, une liste de vecteurs normaux ou de coordonnées UV (Voir explications pour les textures) sont aussi présents optionnellement toujours avec des indices associés.

Ensuite vient une autre liste qui regroupe les données précédentes en triangles à l'aide des indices.

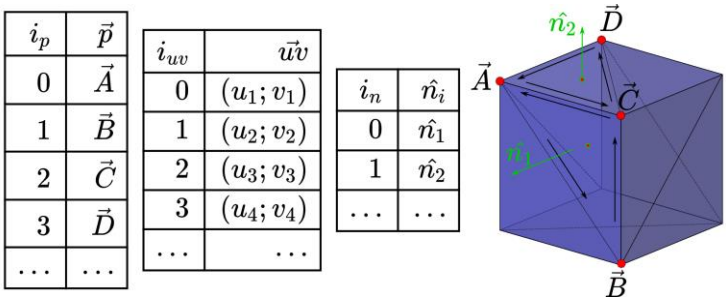



**Figure 11 - Représentation schématique d'un modèle chargé en mémoire**

Avec les données des points et normales fournies dans la figure (*11*), on peut définir 2 des 12 triangles qui composent le cube simplement avec des indices, par exemple, si $T_1 = (0; 1; 2; i_n = 0)$, cela signifie qu'il est composé des points $\vec{A}$, $\vec{B}$ et $\vec{C}$ dans cet ordre précis et que la normale qui correspond est $\widehat{n_1}$. Dans la même logique, $T_2 = (0; 2; 3; i_n = 1)$. Les UVs ont les mêmes indices que les positions et chaque position est donc associée à une coordonnée UV. Remarquez que les points sont définis en tournant dans le sens anti-horaire vue de l'extérieur.

Une méthode pour calculer la normale en un point est d'utiliser le produit vectoriel, celle-ci donnera donc la même normale pour tous les points du triangle. Par contre, il est indispensable d'avoir les triangles définis en tournant dans le sens anti-horaire comme dans la figure *11* pour que $\hat{n}$ ait le bon sens. Pour un triangle $\widehat{ABC}$ donné :

$$\hat{n} = \frac{\overrightarrow{AB} \times \overrightarrow{AC}}{||\overrightarrow{AB} \times \overrightarrow{AC}||}$$

L'autre méthode consiste à lire la normale directement depuis le fichier *obj*. Selon les modèles et comment ils ont été exportés, on peut avoir jusqu'à 3 normales par triangle. Dans ce cas, les coordonnées $u$ et $v$ sont utiles, car elles permettent d'interpoler une normale en $\vec{\imath}$ par rapport aux autres. Soient $\widehat{n_A}$, $\widehat{n_B}$, $\widehat{n_C}$ les normales en $\vec{A}$, $\vec{B}$ et $\vec{C}$, alors $\hat{n} = (\widehat{n_B} - \widehat{n_A}) \cdot u + (\widehat{n_C} - \widehat{n_A}) \cdot v + \widehat{n_A}$. On arrive aux résultats sur les figures *12* et *13* en utilisant $\hat{n}$ dans les calculs de *diff_f*.

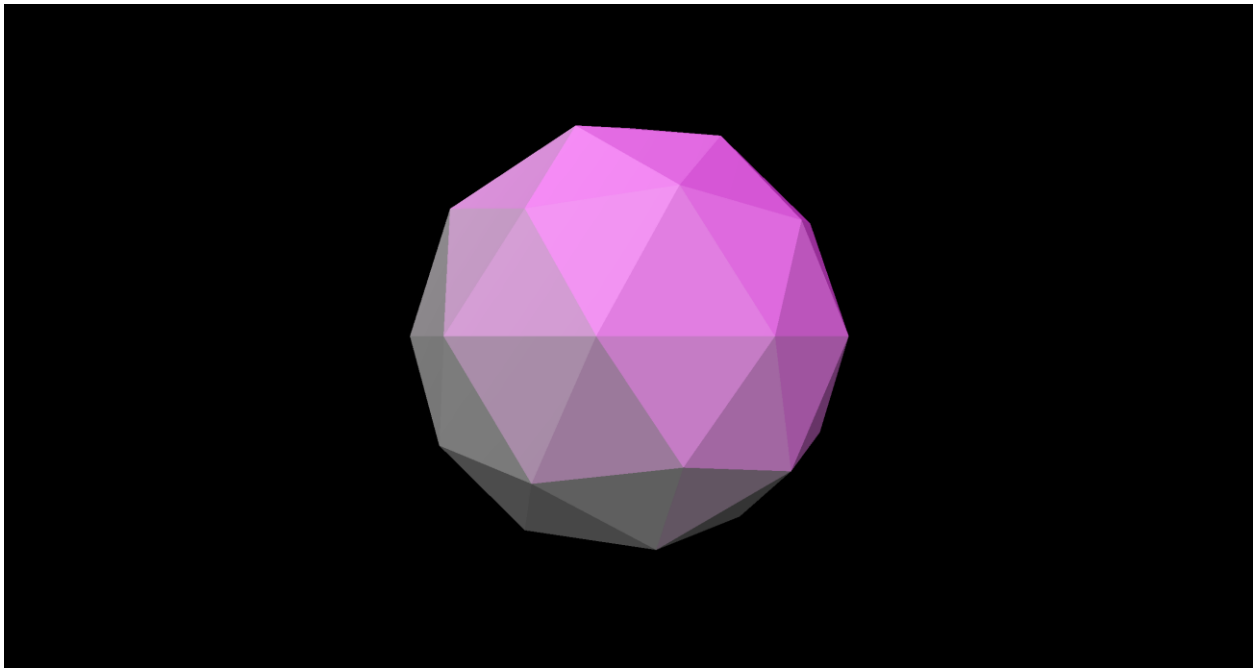


Figure 12 - Méthode 1 de calcul de la normale

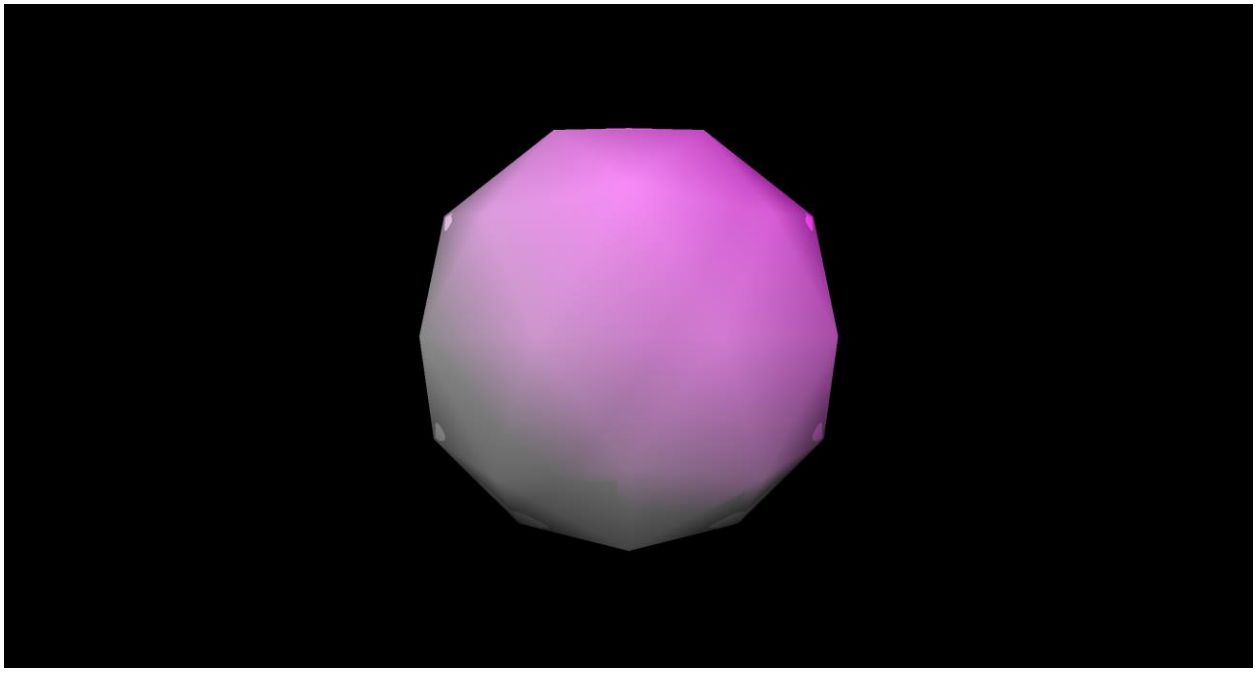


Figure 13 - Méthode 2 de calcul de la normale

Il est étonnant de constater que la seconde méthode arrive à un résultat très lissé malgré un nombre de triangle assez bas. C'est donc la méthode privilégiée dans les jeux vidéo par exemple.

## REFLÉXIONS ET RÉFRACTIONS

Dans la réalité, les photons partent des sources de lumières pour rebondir sur les surfaces ou les traverser, mais dans notre cas, nous allons reconstituer le chemin parcouru par le photon en partant de l'arrivée. Les réflexions et réfractions sont décrites par les lois de Snell-Descartes et les formules fonctionnent indépendamment du sens dans lequel on raisonne (source – arrivée ou arrivée – source). Pour simplifier le

modèle, la réfraction et la réfraction seront calculés séparément, contrairement à la réalité où les deux phénomènes peuvent survenir à chaque changement de milieu. Un rayon provenant d'une source se séparerait normalement en 2 à la première interface entre médiums rencontrée et ces deux nouveaux rayons subiraient la même chose. Dans le modèle, pour chaque premier point d'impact (qu'on appellera $\overrightarrow{i_1}$), un rayon retracera le chemin suivi par un photon exclusivement réfléchie et un autre rayon retracera le chemin suivi par un photon exclusivement réfracté. Pour moduler les effets de la diffusion, réflexion et réfraction, des facteurs propres aux objets doivent être définis.

Jusqu'à maintenant, le rayon s'arrêtait sur son premier obstacle. Il est assez facile depuis cette situation de relancer depuis $\overrightarrow{i_1}$ un rayon ayant une direction réfléchie par rapport à la surface, obtenue en utilisant la loi de la réflexion de Snell-Descartes qui dit que le rayon incident et le rayon réfléchi ont le même angle par rapport à la normale et que ces 3 vecteurs sont sur le même plan, pour tomber sur un autre point et de prendre la couleur en ce point comme base pour la suite des calculs et avoir des reflets sur nos objets. On peut relancer en chaîne le rayon autant de fois qu'on le veut, sachant que plus loin on va, plus on s'approche de la réalité. Néanmoins, il est nécessaire de limiter le nombre de rebonds, car il y des cas où les rayons rebondiraient un nombre excessif de fois, ce qui ralentirait fortement le programme ou au pire le ferait s'écraser.

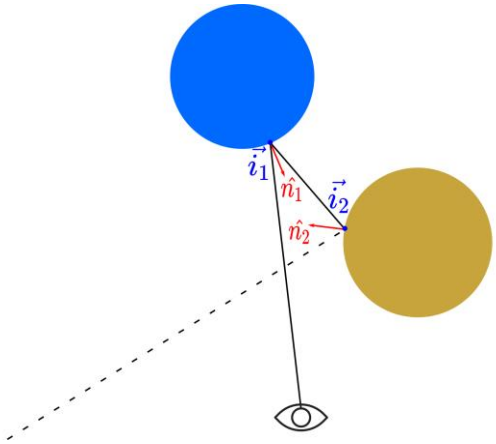



**Figure 14 - Traçage du chemin du photon réfléchi**

Avec $q$ le nombre de rebonds détectés, on calcule d'abord la couleur diffuse $\overrightarrow{cd_{\mathrm{q}}}$ en $\overrightarrow{i_q}$. Puis pour chaque $\overrightarrow{i_k}$ avec $k \in \mathbb{Z}$ partant de $q-1$ jusqu'à 1, on applique le facteur $dist\_f$ correspondant à la distance $||\overrightarrow{i_k} - \overrightarrow{i_{k+1}}||$ et le facteur de réflexivité de l'objet $o_k$ à $\overrightarrow{cd_{\mathrm{k}}}$, puis on y applique la couleur de $o_k$ par synthèse multiplicative. On peut y aussi ajouter après ces opérations le $\overrightarrow{cd_{\mathrm{k}}}$ en $i_k$ multiplié par le facteur de diffusion de $o_k$ pour un meilleur effet visuel.

Dans la figure (*13*), le premier rayon part de la caméra et détecte le point d'impact $\overrightarrow{i_1}$ sur le cercle bleu, on lance le second rayon ayant une direction réfléchie par rapport à $\widehat{n_1}$ depuis $\overrightarrow{i_1}$. Le second rayon détecte le cercle orange, on relance encore un rayon avec la direction réfléchie, mais ce dernier part dans le vide. On prend donc la couleur diffuse ($\overrightarrow{cd}$) du cercle orange qui est l'objet de la dernière intersection valide et on

va repartir en arrière (donc dans le bon sens par rapport à la réalité) pour y appliquer des mentionnées ci-dessus. Soient $\hat{I}$ la direction du rayon incident et $\hat{R}$ La direction du rayon réfléchi :

$$\hat{R} = \hat{I} - 2(\hat{n} \cdot \hat{I})\hat{n}$$

A l'aide du produit scalaire, on peut projeter un vecteur sur un autre. Soit $\vec{a}$ et $\vec{b}$ des vecteurs et $\alpha$ l'angle entre eux, la projection de $\vec{a}$ sur $\vec{b}$ se calcule comme suit :

$$Proj_{\vec{b}}(\vec{a}) = \frac{(\vec{a} \cdot \vec{b})\vec{b}}{||\vec{b}||^2}$$

Si $\hat{a}$ et $\hat{b}$ sont normalisés, l'expression devient simplement

$$Proj_{\hat{b}}(\hat{a}) = (\hat{a} \cdot \hat{b})\hat{b}$$

Grâce à cette formule, on peut facilement illustrer la formule de la direction réfléchie présentée au-dessus.

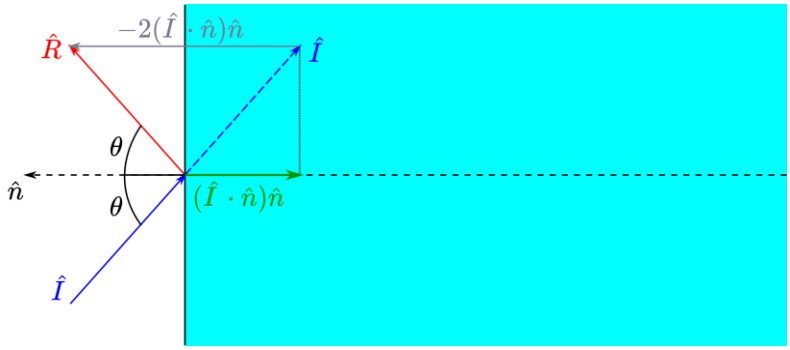



Figure 15 - Illustration de la formule de la réflexion

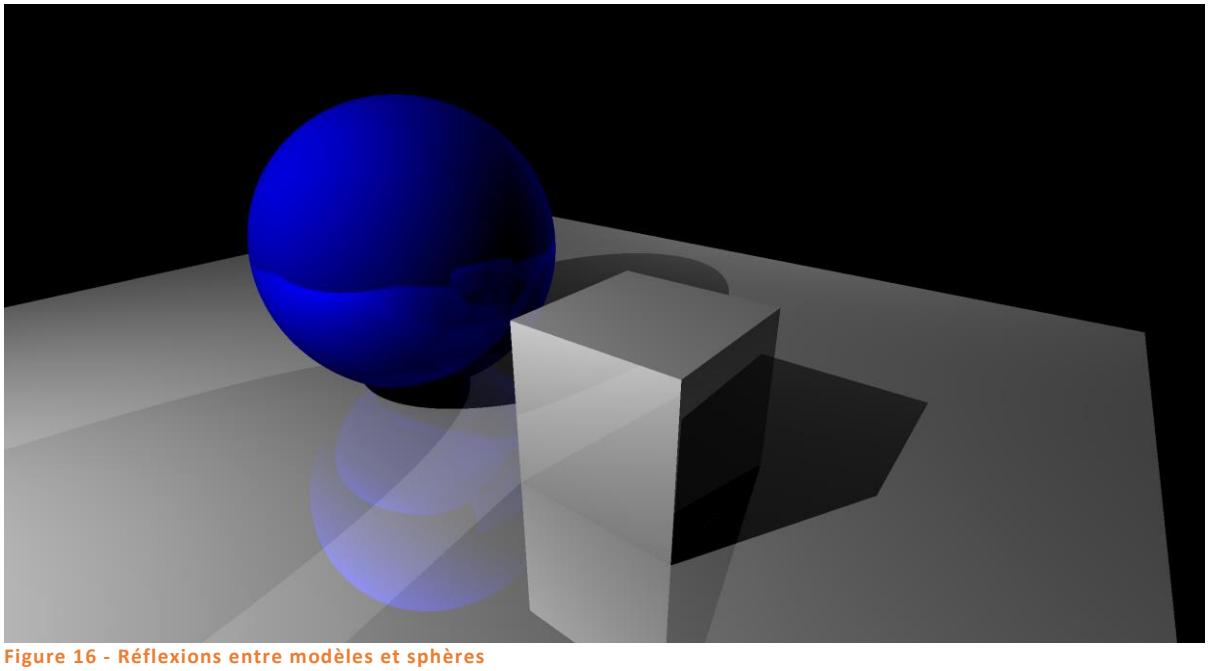


Figure 16 - Réflexions entre modèles et sphères

Pour la réfraction, le principe est le même, sauf pour quelques aspects dont le changement de direction qui se fait cette fois selon la loi de la réfraction de Snell-Descartes. Cette dernière dit que lorsqu'un rayon change de milieu, par exemple quand il passe de l'air à l'eau ou inversement, il est dévié à cause de la différence de la vitesse à laquelle circule la lumière entre les 2 milieux. La direction réfractée est calculée

grâce à la dérivation de la formule suivante : $n_1 \cdot sin(\theta_1) = n_2 \cdot sin(\theta_2)$, où $n = \frac{c}{v}$ avec $v$ la vitesse de la lumière dans le milieu.

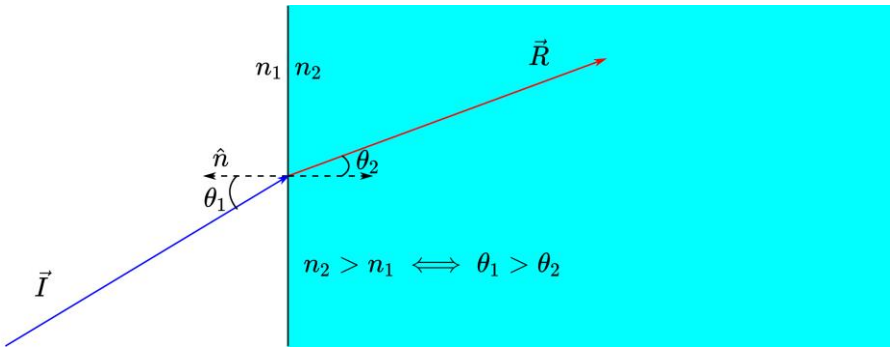



Figure 17 - Schéma de la loi de la réfraction

A noter que le programme utilise des fonctions inclus dans GLSL pour calculer les direction réfléchies ou réfractées. La fonction `reflect(dir, normal)` utilise en interne la formule présentée plus haut, tandis que `refract(dir, normal, eta)` utilise un algorithme plus optimisé défini comme suit :

```
// Ici, eta est le ratio entre l'indice de réfraction du premier milieu et du second
k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I));
if (k < 0.0)
	R = vec4(0.0); // Dans le cas où l'angle critique est atteint
else
	R = eta * I - (eta * dot(N, I) + sqrt(k)) * N;
```

La démonstration de ce dernier étant trop complexe, elle ne sera pas faite ici.

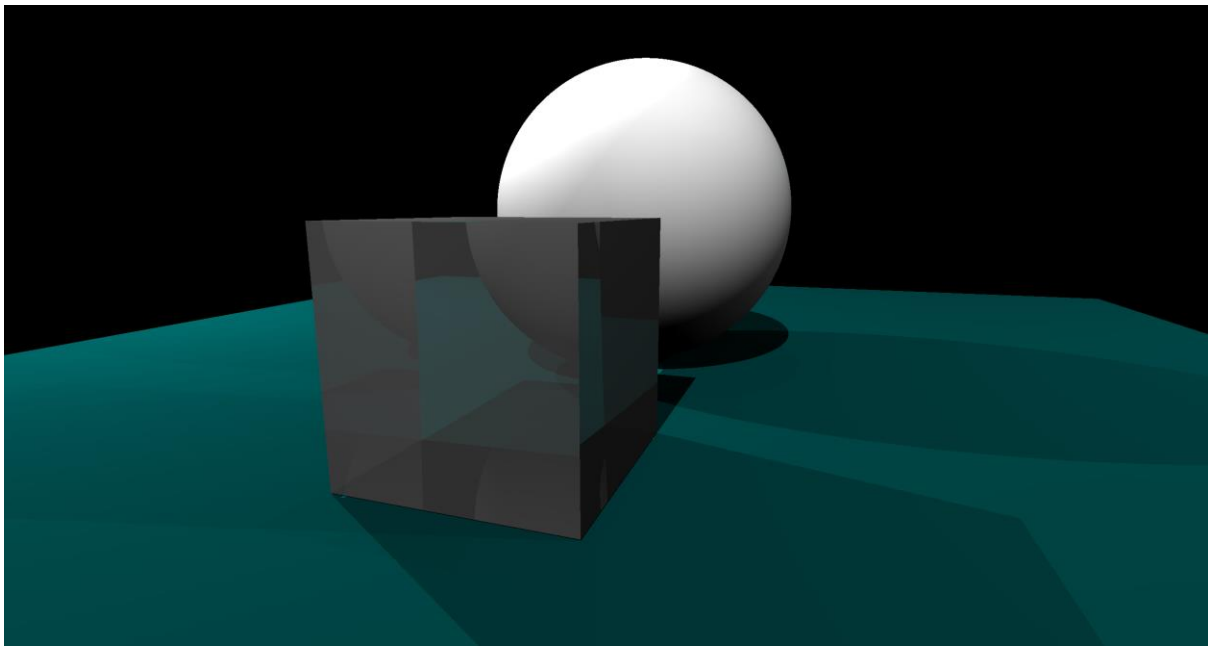


Figure 18 - Un cube réfractant l'image d'une sphère

Il faut aussi tenir compte du fait que lorsqu'on se trouve à l'intérieur d'un objet, la normale change de signe et $eta$ doit avoir la valeur inverse. Si $\hat{I} \cdot \hat{n} > 0$, alors ça signifie que $\hat{I}$ est en train de quitter l'objet, puisque l'angle entre $\hat{I}$ et $\hat{n}$ est inférieur à $\pi/2$. Dans le cas contraire, $\hat{I}$ entre dans l'objet.

Finalement, la couleur finale du pixel est la somme des couleurs données par la réflexion et la réfraction multiplié respectivement par $w$ et $w\ -\ 1$, où $w$ est l'opacité correspondant à l'objet $o_1$.

# TEXTURAGE DES OBJETS

Une texture en graphisme 3D est une image, qui est « posée » sur l'objet pour lui donner plus d'effet, ça peut être une texture de briques appliqué à un mur, une texture de peau appliqué à un personnage ou bien une texture appliquée sur une sphère pour représenter une planète.

Nous avons une texture qui est une image en 2D. Il faut donc arriver à convertir une coordonnée en 3D en une coordonnée en 2D. L'utilisation de textures dans le shader nécessite que cette coordonnée 2D que nous appellerons $\overrightarrow{uv}$ possède des composantes comprises entre 0 et 1. Soient $w$ la largeur de l'image, $h$ la hauteur et $\vec{m}$ une coordonnée de l'image :

$$\overrightarrow{uv} = \begin{pmatrix} \frac{m_x}{w} \\ \frac{m_y}{h} \end{pmatrix} \Leftrightarrow \vec{m} = \begin{pmatrix} u_x \cdot w \\ u_y \cdot h \end{pmatrix}$$

Pour les sphères, déterminons la latitude appartenant à $[0; \pi]$ et la longitude appartenant à $[0; 2\pi]$ pour trouver $\overrightarrow{uv}$. Ces derniers se mesurent en une unité d'angles (radians dans notre cas). Soient $\hat{N}$ le vecteur normalisé pointant vers le « pôle nord » de la sphère, $\hat{E}$ un vecteur normalisé perpendiculaire à $\hat{N}$ et $\hat{n}$ la normale en un point d'impact $\vec{\imath}$, nous avons :

$$lat = cos^{-1}\left(\hat{n} \cdot \hat{N}\right)$$

C'est simplement l'angle entre le pôle nord et $\hat{n}$. De la même manière, la longitude est l'angle entre $\hat{E}$ et $\hat{n}$ projeté sur le plan duquel $\hat{N}$ est la normale.

$$Proj_{\hat{N}}(\hat{n}) = \frac{\hat{n} - \hat{N}\left(\hat{n} \cdot \hat{N}\right)}{||\hat{n} - \hat{N}\left(\hat{n} \cdot \hat{N}\right)||}$$

Avec $Proj_{\hat{N}}(\hat{n})$, on peut maintenant calculer l'angle avec $\hat{E}$ :

$$long = cos^{-1}\left(\mathrm{Proj}_{\hat{N}}(\hat{n}) \cdot \hat{E}\right)$$

On interpole maintenant $lat$ et $long$ pour qu'ils soient compris dans $[0; 1]$ :

$$\mathrm{uv}_x = \frac{lat}{\pi} \qquad \mathrm{uv}_y = \frac{long}{2\pi}$$

Dans le cas d'un modèle, les UVs sont comme mentionnés ci-dessus inclus dans le fichier *obj*. Comme chaque point du triangle aura une coordonnée UV associée (chaque position pointe donc vers un pixel de la texture), la méthode appliquée est la même que la seconde méthode pour retrouver la normale d'un modèle. On utilise $u$ et $v$ du calcul d'intersection rayon-triangle pour interpoler le UV sur $\vec{\imath}$. Soient $\overrightarrow{\mathrm{UV}_A}$, $\overrightarrow{UV_B}$ et $\overrightarrow{UV_C}$ les coordonnées UV de $\vec{A}$, $\vec{B}$ et $\vec{C}$ :

$$\overrightarrow{\mathrm{uv}} = \left(\overrightarrow{\mathrm{UV}_B} - \overrightarrow{UV_A}\right) \cdot u + \left(\overrightarrow{\mathrm{UV}_C} - \overrightarrow{UV_A}\right) \cdot v + \overrightarrow{UV_A}$$

La couleur donnée par la texture pour $\overrightarrow{uv}$ peut maintenant être utilisée partout où la couleur propre de l'objet était utilisée.

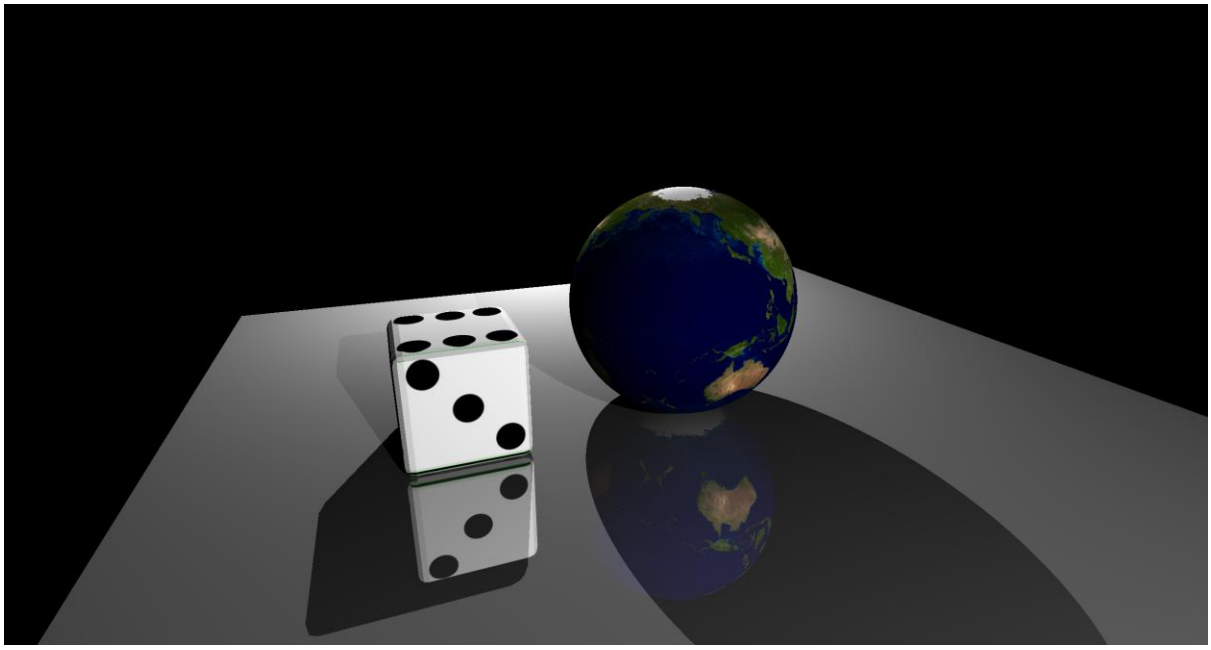


Figure 19 - Application de textures sur un dé et une terre (sphère)

# IMPLÉMENTATION DE LA VERSION FINALE

Le passage à la version finale assure une augmentation des performances. Tous les calculs concernant le raytracing dans la version prototype se faisaient uniquement sur le CPU, il n'y avait que l'affichage des pixels à l'écran qui était fait à travers le GPU par LÖVE2D. Dans l'ordinateur, le CPU a généralement peu de cœurs (4 dans mon cas). Un cœur est une unité qui va pouvoir effectuer des opérations à la chaîne. Mon prototype fonctionne sur un seul cœur car il n'est pas programmé pour répartir les calculs sur les différents cœurs, il n'exploite donc pas toute la puissance de calcul disponible contrairement à la plupart des programmes sur un ordinateur (navigateurs web, logiciels de graphisme, jeux-vidéos).

Le GPU en revanche possède des centaines de cœurs qui servent généralement dans des tâches graphiques et qui sont très optimisés pour une sélection d'opérations (arithmétiques, trigonométriques, vectorielles, racines carrées) très utilisées dans les applications graphiques. Les cœurs GPU sont moins polyvalents que les cœurs CPU, mais ils sont plus nombreux et surtout plus performants dans ce qu'ils peuvent faire. On peut les utiliser dans ce qu'on appelle des *shaders*. Un shader est un même programme qui va être lancé sur plusieurs cœurs du GPU en donnant à chaque cœur des entrées différentes. Originellement, il était utilisé en graphisme 3D pour ombrer une surface (d'où le nom) puisque l'ombre devait être appliqué à plusieurs pixels à l'écran alors que le processus était le même. Les shaders sont donc le moyen parfait de paralléliser des algorithmes.

Comme langage, j'ai choisi Rust car c'est un langage compilé — donc plus performant que Lua — qui m'intéressait, le projet était donc une bonne occasion d'en apprendre plus dessus. J'ai créé donc une bibliothèque en Rust qui permet à travers Vulkan, une bibliothèque graphique, de lancer n'importe quel

programme de shader et qui est capable de lui passer les données nécessaires. Les shaders sont écrits en *GLSL*, un langage très proche du *C*.

### GAINS DE PERFORMANCES

Après l'implémentation du rendu de sphères avec gestion de la lumière comme dans le prototype montré à la figure (*9*), les différences de performances sont énormes. Avec les mêmes sphères et lumières à une définition d'image égale, le prototype génère en moyenne 2 images par seconde, alors que la nouvelle en génère 1000.

### MÉTHODE DE DÉBOGAGE

L'utilisation de shaders implique un accès moins facile à la mémoire, puisque c'est la mémoire du GPU et non du CPU, le programme est donc plus difficile à analyser en cas de fonctionnement anormale. Après un peu de recherches, je suis tombé sur le logiciel RenderDoc, qui permet de capturer à chaque fois capturer une seule image et qui fournit un instantané de la mémoire du GPU. Il permet également l'utilisation d'une fonction *print*, qui permet d'«imprimer» des message que je trouvais pertinents pour qu'ils soient visible dans RenderDoc.

### DIFFICULTÉS RENCONTRÉS

Pour envoyer des données au shader, il faut respecter un format très spécifique et assez technique. Il faut faire attention à la taille et l'alignement des données à l'octet près. Lorsque ce formatage n'était pas correct, les données dans le GPU devenaient totalement chaotiques, ce qui causait des bugs totalement improbables qui étaient assez difficiles, dont la cause était assez difficile à repérer.

Malheureusement, l'implémentation de la réfraction est assez peu fiable, la situation à la figure *18* est correcte, mais j'ai dû beaucoup « bricoler » pour y arriver. L'algorithme donne des résultats inattendus lorsqu'on a une sphère transparente ou lorsqu'il y a une texture appliquée sur l'image réfractée.

Je voulais également avoir un système qui à partir d'un script pourrait générer une vidéo animée avec les objets défini dans ce dernier. Cela n'a malheureusement pas été possible à cause d'un manque de temps et parce que la bibliothèque Rust que j'utilisais ne permettait pas de définir certaines données au moment de l'exécution car cela devait se faire au moment de la compilation.

## BILAN PERSONNEL

L'utilisation de Rust pour développer la bibliothèque « squelette » était très laborieuse à cause de son compilateur très à cheval sur les erreurs lié à la mémoire, c'est-à-dire que dès qu'il y a la moindre chance que la mémoire soit modifiée indirectement de façon involontaire, ce qui arrive souvent dans d'autres langages compilés, le compilateur refuse de fonctionner. Cela peut être une force, car si le programme se lance, on est presque sûr que les bugs qui surviennent ne sont pas dus à une obscure subtilité de la gestion de la mémoire, mais cela rend le développement plus long. Quand je l'ai choisi comme langage, je savais dans quoi je m'embarquais, mais cela m'a pris une dizaine d'heures rien que pour développer cette bibliothèque.

J'ai surtout choisi Rust car je voulais profiter de ce projet pour en apprendre plus dessus et dans ce sens, je trouve que c'est une assez bonne chose que j'aie eu à passer autant de temps à l'explorer, malgré la frustration que cela créait parfois.

L'aspect mathématique m'a beaucoup plu, car contrairement à ce qu'on fait généralement en cours, surtout dans un chapitre comme l'algèbre linéaire, j'ai eu l'occasion d'avoir un aperçu de l'application dans des choses concrètes de concepts qu'on étudie.

# CONCLUSION

Le monde du graphisme 3D est très varié et utilise beaucoup d'astuces permettant d'approximer la réalité. La technique du raytracing simplifie l'implémentation de certains phénomènes comme la réflexion ou la réfraction, mais elle souffre de sa performance très limitée. Cela prend des secondes à afficher un modèle complexe qui peut être affichée en des millisecondes par une méthode de rendu 3D en temps réel comme la rastérisation. Malgré cela, les notions utilisées restent relativement simples, le concept le plus complexe utilisé étant de mon point de vue la règle de Cramer. Là est donc la force du raytracing, dans sa simplicité de fonctionnement.

# OUTILS UTILISÉS

- Internet est un outil indispensable quand on traite de n'importe quel sujet en informatique. J'ai dû utiliser des sites web pour apprendre à mettre en place un espace de travail sur mon ordinateur (installer les bons programmes et les configurer correctement), pour apprendre des choses concernant la programmation, pour trouver les bibliothèques[3] qui pourraient m'être utiles ainsi que leur documentation et finalement pour trouver les outils mathématiques qui pouvaient me servir à résoudre les problèmes que je rencontrais.
- Visual Studio Code est avant tout un éditeur de code, il permet au programmeur d'écrire le code d'un programme. Il y a aussi un terminal intégré facilement accessible.
- LÖVE2D est une bibliothèque pour Lua permettant la création de programmes utilisant des graphismes 2D, c'est-à-dire qu'on peut facilement dessiner des formes géométriques, des images ou du texte. Il est donc premièrement fait pour le développement de jeux-vidéo, mais il peut aussi être utilisé pour des projets plus expérimentaux comme dans mon cas.
- Git est un outil pour gérer les différentes versions d'un programme, je l'ai utilisé pour avant tout avoir une sécurité et revenir à une version précédente lorsque je sentais que la version actuelle était « irrécupérable ».
- Blender est un logiciel de graphisme 3D, permettant de manipuler des modèles et faire des rendus d'images ou des animations vidéo. Il m'a permis de créer et exporter des petits modèles 3D avec la bonne configuration du format des fichiers pour qu'ils puissent être chargés par le programme.
- Rust est un langage de programmation qui est fourni avec plusieurs programmes, dont :
    - *rustc*, le compilateur
    - *cargo*, un système permettant de centraliser des bibliothèques Rust et le programme qui permet de les inclure dans des projets facilement

[3] Essentiellement un bout de code déjà écrit qui simplifie le développement

# BIBLIOGRAPHIE ET LIENS

Voici la bibliographie, des liens utiles et des liens cliquables de la version PDF :


https://www.youtube.com/watch?v=9RHGLZLUuwc – Vidéo présentant le raytracing

https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf – l'article présentant la méthode d'intersection rayon-triangle utilisé dans ce travail

https://love2d.org/ - Site officiel du logiciel utilisé pour créer le prototype

https://github.com/themousery/vector.lua – Bibliothèque qui m'a été utile durant le développement

https://git-scm.com/ - Site officiel de git, le logiciel qui m'a permis de gérer les différentes versions du code

https://www.lua.org/ - Site officiel du langage Lua, utilisé dans le prototype

https://www.rust-lang.org/ - Site officiel du langage Rust, utilisé dans la version finale

https://github.com/vulkano-rs/vulkano - Bibliothèque Rust permettant d'utiliser la bibliothèque graphique Vulkan

https://www.vulkan.org/ - Site officel de Vulkan

https://ricktu288.github.io/ray-optics/simulator/ - Une simulation d'optique en 2D qui m'a été très pratique pour tester des situations et les comparer avec mon programme

https://www.blender.org/ - Site officiel de Blender, un logiciel de graphisme 3D

https://renderdoc.org/ - Site officiel de RenderDoc, un logiciel de débogage graphique