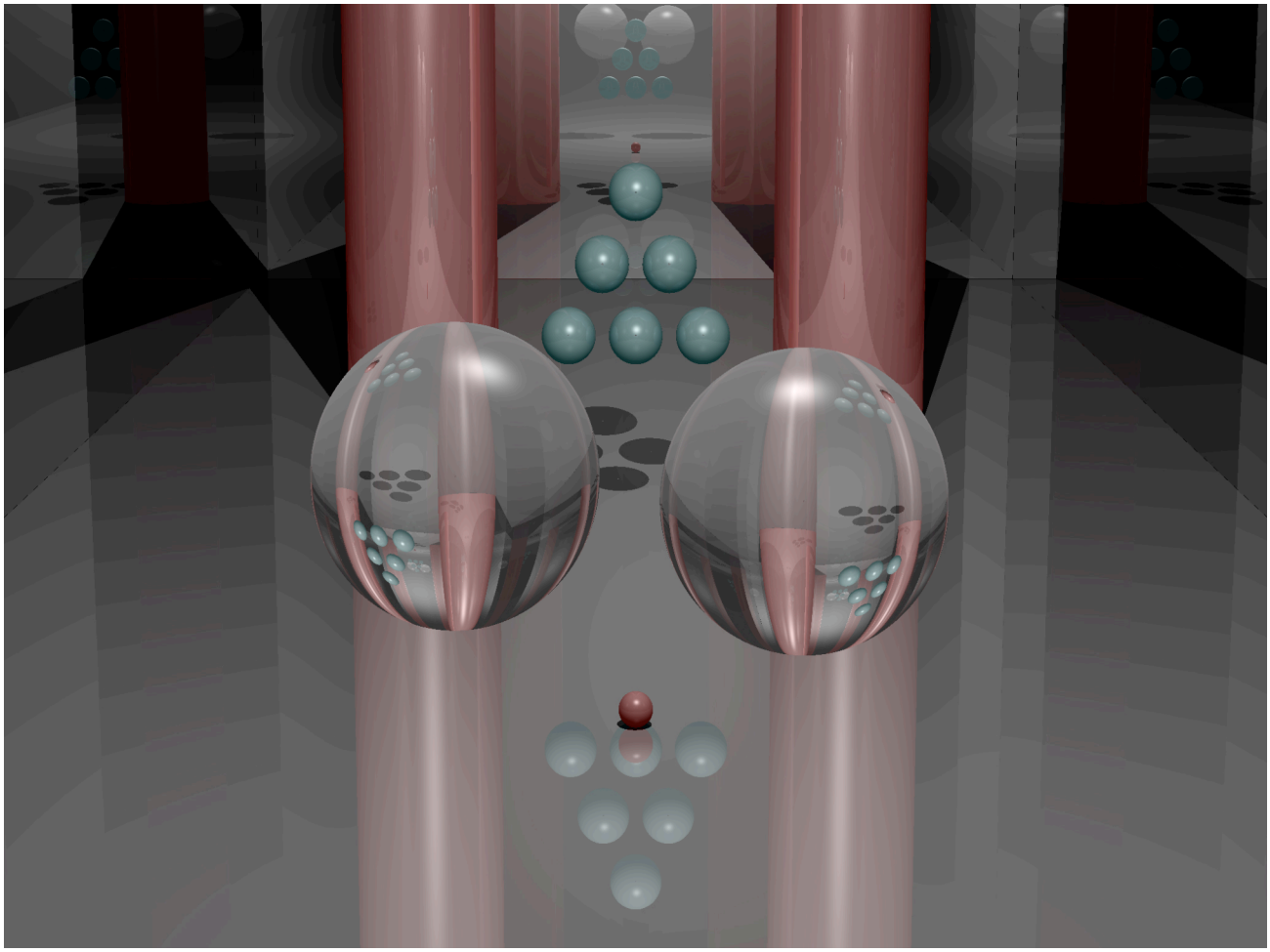
Corgié Grégory *ETI 2*

Projet Module Synthèse d'Images
Raytracing



Année 2004-2005

Sommaire

I/ Introduction

En 1980, Turner Whitted publia un article proposant une nouvelle méthode de rendus permettant de reproduire des images mettant en scène des objets réfléchissants. Ce principe, connu sous le nom de raytracing, est assez simple.

En schématisant, le monde qui nous entoure est composé d'objets et de source de lumières. De plus, chaque objet possède diverses propriétés comme une couleur par exemple. Mais physiquement qu'est-ce qu'une couleur ?

La lumière visible est composée d'une multitude de longueurs d'ondes qui correspondent à des couleurs (640 au bleu et 860 au rouge). La couleur d'un objet correspond aux longueurs d'ondes issues de la lumière qui l'éclaire moins celles que l'objet absorbe. Ainsi, un objet est bleu s'il absorbe toutes les longueurs d'ondes du spectre visible sauf la longueur d'onde correspondant au bleu.

Il apparaît dès lors évident qu'en absence de lumière, les couleurs n'existent plus. Physiquement, lorsque vous regardez un objet, le trajet de la lumière peut se modéliser par un rayon de lumière qui irait frapper l'objet puis une partie de ce rayon serait redirigé (avec un spectre appauvri) vers votre oeil. Cette méthode n'est pas implémentable car de nombreux calculs seraient nécessaires pour reproduire une scène quelconque alors que beaucoup d'entre eux seraient inutiles. En effet, il n'est pas concevable de suivre chaque rayon issu d'une source de lumière car la plupart d'entre eux ne parviendront jamais jusqu'à l'œil de l'observateur.

C'est alors que Turner Whitted proposa une idée qui se basait sur le principe de Fermat. Ce principe dit que le chemin emprunté par un rayon de lumière pour aller d'un point A vers un point B est rigoureusement le même que celui qu'il emprunterait pour aller de B vers A. Il serait donc plus simple de suivre un rayon issu de l'œil de l'observateur, de voir s'il touche un objet et de vérifier dans ce cas quelle source de lumière l'éclaire et en déduire la couleur du point envisagé. L'intérêt de la méthode vient du fait qu'ici tous les rayons sont utiles.

Ce projet a donc pour but de concevoir un moteur de Raytracing permettant de construire des images mettant en scène des éléments simples tout en tenant compte des phénomènes physiques.

II / Principe de base

Toute scène 3D doit, pour être représentée sur un écran, être projetée sur un plan que l'on appellera Vue.

Cette vue sera composée d'un nombre de pixel qui dépend de la résolution adoptée. La résolution étant le nombre de pixels considérés en largeur fois le nombre de pixels composant l'image en hauteur. Par exemple, pour une image de résolution 640*480, la vue sera composée de 307 200 pixels.

Le principe du moteur de raytracing est simple et peut se résumer par la démarche suivante. Pour chaque pixel, on lance un rayon, on regarde si ce rayon touche un objet. Dans ce cas, on détermine la couleur de l'objet en ce point et on l'attribue au pixel correspondant (voir Fig1.).

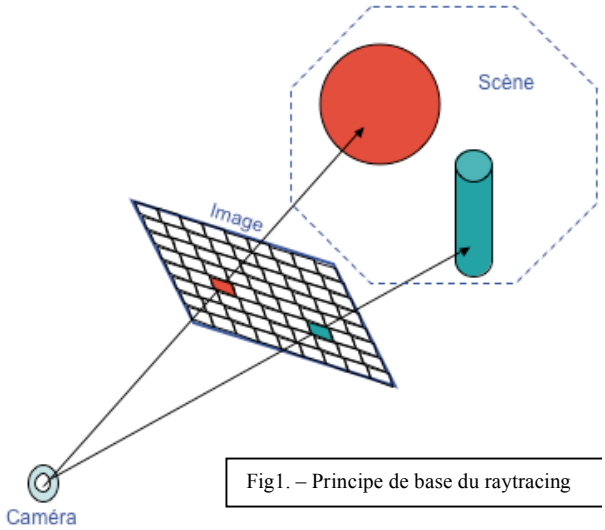


Fig1. – Principe de base du raytracing

III / Opérations de prétraitements

Avant de s'intéresser au cœur de l'algorithme, nous allons détailler quelques éléments complémentaires de l'algorithme de raytracing.

III.1/ Le Fichier de configuration

Pour réaliser une scène, on a opté pour un fichier de configuration de type texte. Ainsi, fabriquer une scène devient beaucoup plus rapide et moins fastidieux. Lors de l'exécution du programme main(), celui-ci va ouvrir et analyser lors de sa phase d'initialisation le fichier « Scene.txt ».

Dans celui-ci se trouve un ensemble de valeurs classées dans un ordre précis. Attention, le non-respect de la syntaxe engendrera dans le meilleur des cas une scène incohérente et dans le pire des cas une erreur dans l'exécution.

La lecture des données n'est sensible qu'aux espaces et aux retours à la ligne. En effet, pour séparer 2 variables, on peut utiliser N espaces et/ou N retours à la ligne ce qui permet d'améliorer grandement la lisibilité du fichier.

Le programme récupère **dans cet ordre** les valeurs des variables suivantes :

```
Résolution.x    Résolution.y
PositionCamera.x          PositionCamera.y          PositionCamera.z
PointRegardé.x   PointRegardé.y   PointRegardé.z

TypeBrouillard      DistanceBrouillard

NombreSphere              NombrePlan              NombreCylindre          NombreLumiere

Rayon1      Centre1.x  Centre1.y  Centre1.z   TEXTURE       MATERIAU
...
```

RayonN CentreN.x CentreN.y CentreN.z TEXTURE MATERIAU
PtPlan1.x PtPlan1.y PtPlan1.z Normale1.x Normale1.y Normale1.z TEXTURE MATERIAU ... *PtPlanN.x PtPlanN.y PtPlanN.z NormaleN.x NormaleN.y NormaleN.z TEXTURE MATERIAU* *Normale1.x Normale1.y Normale1.z PtAxe1.x PtAxe1.y PtAxe1.z Rayon TEXTURE MATERIAU* *NormaleN.x NormaleN.y NormaleN.z PtAxeN.x PtAxeN.y PtAxeN.z Rayon TEXTURE MATERIAU* *PosLum1.x PosLum1.y PosLum1.z Couleur1.rouge Couleur1.vert Couleur1.bleu CoefDif* ... *PosLumN.x PosLumN.y PosLumN.z CouleurN.rouge CouleurN.vert CouleurN.bleu CoefDif*

Bien entendu si NombreSphere est égal à 4, il doit y avoir exactement 4 séries de valeurs décrivant les sphères. On place à part la série de valeurs regroupées sous l'intitulé TEXTURE et MATERIAU pour plus de lisibilité.

TEXTURE :

- Pour un objet sans textures mais uniquement avec une couleur unique, TEXTURE sera du type :

1 Couleur.rouge Couleur .vert Couleur.bleu

- Pour un objet faisant appel à une texture procédurale la syntaxe sera différente et nécessitera la définition de type (compris en 1 et 4) que nous verrons plus tard.

2 type C1.r C1.g C1.b C2.r C2.g C2.b C3.r C3.g C3.b

MATERIAU :

L'étiquette MATERIAU regroupe un ensemble de valeurs propres au matériau.

CoefDiffusion	CoefReflexion	CoefRefraction	IndiceMilieu	TypeBump	ValBump

Exemple de définition une sphère avec une texture procédurale de type Bois :

1.0 0.0 2.0 9.0 2 2 0.5 0.5 0.5 0.0 0.0 1.0 1.0 1.0 0.0 0.3 0.2 0.0 1.5 0 0.0

III.2/ Configuration de la caméra

Comme vu précédemment, l'utilisateur doit spécifier quatre valeurs pour configurer la caméra :

- La position de la caméra : PosC
- Le point regardé : PtRegarde
- Le vecteur Haut : $\vec{H}$
- La résolution de l'image : ResX et ResY (640 pixels*480 pixels en règle générale)

Trois autres paramètres sont fixés par défaut :

- La distance entre la caméra et la vue : DistVue = 1.0;

- La longueur de la vue : LongVue = 0.35;
- La largeur de la vue : LargVue = 0.5;

Ces paramètres suffisent pour calibrer la vue de la scène car ils permettent de calculer les autres paramètres essentiels.

- La direction dans laquelle la caméra observe la scène : $\vec{U}$

$$\vec{U} = Pt\,\mathrm{Re}\,gardé - PosC$$

- Le vecteur droite : $\vec{D}$

$$\vec{D} = \vec{U} \wedge \vec{H}$$

- La position du coin haut gauche de l'image : PosHGVue

$$PosHGVue - PosC = DistVue * \vec{U} + \frac{LongVue}{2} * \vec{H} - \frac{L\arg Vue}{2} * \vec{D}$$

Ensuite, il reste à calculer le vecteur directeur de chaque rayon partant de la caméra en direction de chaque pixel composant l'image :

- Vecteur directeur du rayon lancé : $\vec{V}$

V = (PosHGVue – PosC)+ D*LargVue/ResX*PosX – H*LongVue/ResY*PosY

avec 0<PosX<ResX et 0<PosY<ResY

Dès lors la caméra est configurée et nous pouvons alors nous intéresser au cœur du moteur de raytracing.

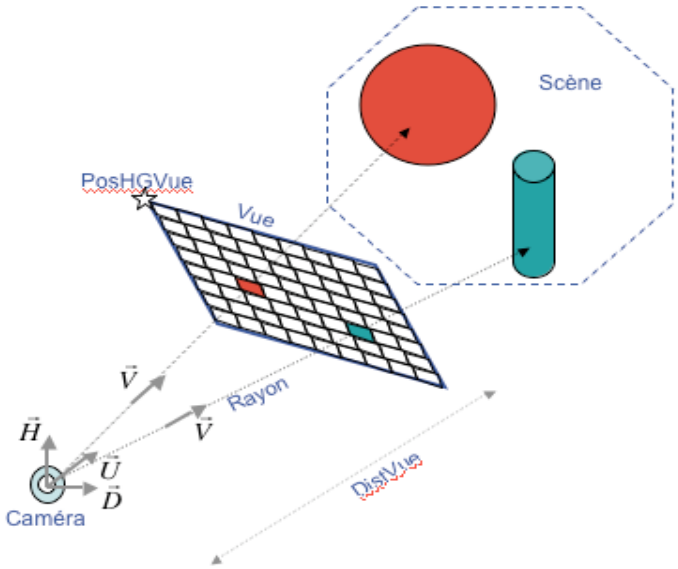


Fig.2 – Variables utilisées pour la configuration la caméra

IV / Le Cœur du moteur de Raytracing

La fonction de raytracing admet 5 paramètres et renvoie une couleur :

```
CCouleur RayTrace( CScene& Scene, CRayon ray, int _depth, float coef, int dedans)
```

On passe à la fonction :

1. Une scène
2. Un rayon constitué d'une origine et d'un vecteur directeur

3. La profondeur du rayon en cours
4. Un coefficient correspondant à la valeur de la contribution de la couleur trouvée. Ce coefficient est compris entre 0.0 et 1.0.
5. Un entier permettant de savoir si le rayon en cours est à l'intérieur ou à l'extérieur d'un objet.

Un résumé de l'algorithme de raytracing du moteur pourrait être le suivant :

```
Main

Pour chaque pixel
    On forme un rayon primaire
    On cherche l'intersection la plus proche
    S'il y a une intersection
            CalculCouleur(Profondeur+1)
    On attribue la couleur calculée au pixel


CalculCouleur(Profondeur)

On forme le rayon lumineux vers une source
On cherche une intersection
    Si l'objet est réfléchissant
            On forme le rayon réfléchi
            On cherche l'intersection la plus proche
                    S'il touche un objet
                            CalculCouleur(Profondeur+1)
    S'il l'objet est transparent
            On forme le rayon réfracté
            On cherche l'intersection la plus proche
                    S'il touche un objet
                            CalculCouleur(Profondeu+1)
On calcule la couleur du pixel
```

IV.1/ La gestion des intersections

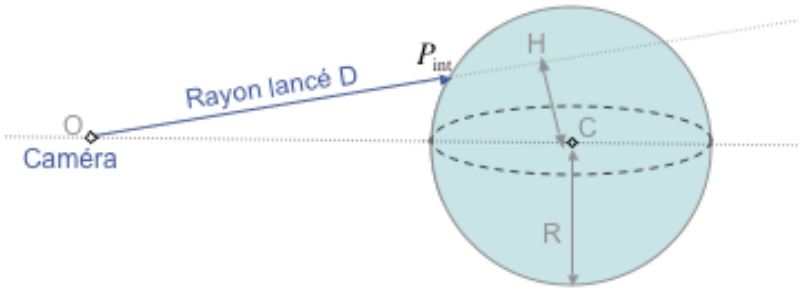
Chaque objet possède une méthode de calcul d'intersection propre. Un rayon représente le trajet d'un photon, et n'a donc pas d'épaisseur. Il est modélisé sous la forme d'une demi-droite, c'est-à-dire un point de départ *Origine* et un vecteur directeur *VectDir*. Les points appartenant à la demi-droite vérifient donc l'équation :

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} Origine.x \\ Origine.y \\ Origine.z \end{pmatrix} + \begin{pmatrix} VectDir.x \\ VectDir.y \\ VectDir.z \end{pmatrix} * t \qquad \text{avec } t > 0$$

IV.1.1/ Intersection avec une sphère

L'une des premières formes à avoir été implémentée, car étant la plus simple, fut la sphère. Une sphère possède un centre, un rayon et son équation est de la forme $x + y + z = R^2$.

Pour calculer l'intersection, deux approches sont possibles. Soit un calcul algébrique se basant sur les équations mathématiques, soit une approche géométrique. Cette seconde méthode est bien moins gourmande en calcul car elle nécessite moins d'opérations. Nous avons donc naturellement opté pour celle-ci.



$\overline{OH} = \vec{OC}.\vec{D}$ où OC et D sont connus

Si $\overline{OH} < 0$, il n'y a pas d'intersection

Sinon, on calcule $CH^2 = OC^2 - OH^2$
Si $CH^2 > R^2$, pas d'intersection

$d^2 = R^2 - CH^2$

On a alors deux solutions $\boxed{\begin{matrix} t_1 = \overline{OH} + \sqrt{d^2} \\ t_2 = \overline{OH} - \sqrt{d^2} \end{matrix}}$

On conserve ensuite la valeur la plus petite des deux et strictement positive de t.
Il ne nous reste plus qu'à déterminer les coordonnées du point d'intersection à l'aide de la définition du rayon :

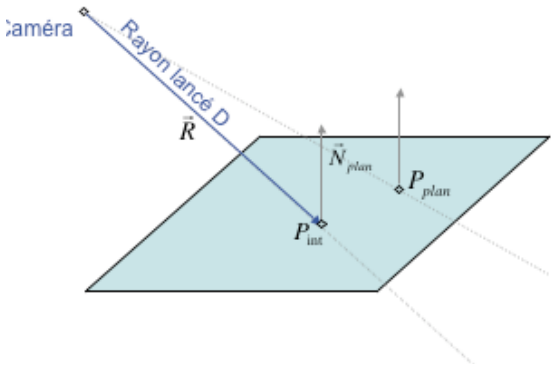
$$\begin{pmatrix} x_{int} \\ y_{int} \\ z_{int} \end{pmatrix} = \begin{pmatrix} Origine.x \\ Origine.y \\ Origine.z \end{pmatrix} + \begin{pmatrix} VectDir.x \\ VectDir.y \\ VectDir.z \end{pmatrix} * t_{min} \quad \text{avec } t_{min} > 0.$$

Un autre paramètre très utile doit être calculé à ce moment-là, il s'agit de la normale à l'objet en ce point. Ici, la normale n'est pas dure à trouver, il suffit de construire le vecteur $\vec{CP}_{int}$ et de le normaliser.

<u>*Rem :*</u> A cause des approximations de calcul, une forme géométrique possède une surface avec certaine épaisseur certes très petite mais lors de calculs d'intersections, il se peut que l'on retombe sur le point d'origine du rayon émis (lors d'une réflexion sur une surface par exemple). C'est pourquoi, on ne conserve que des valeurs de t > 0.01.

IV.1.2/ Intersection avec un plan

Pour le plan, une approche calculatoire a été retenue. Un plan est défini par l'équation
$A.x + B.y + C.z + D = 0$.



Sachant que la normale au plan est $\vec{N}ormale = \begin{pmatrix} A \\ B \\ C \end{pmatrix}$

Un rayon $\vec{R}$, tel que $\vec{R} = \vec{R}_O + \vec{R}_d \times t$, appartient au plan si $\vec{N}ormale.(\vec{R}_O + \vec{R}_d \times t) + D = 0$

D'où $\boxed{t = -\dfrac{\vec{N}ormale.\vec{R}_O + D}{\vec{N}ormale.\vec{R}_d}}$

On en déduit le point d'intersection en réinjectant t dans l'expression du rayon et la normale au point d'intersection est donnée directement par l'équation du plan.

IV.1.3/ Généralisation d'intersection aux quadriques

L'équation générale d'une quadrique est

$$F = A.x^2 + B.y^2 + C.z^2 + D.x.y + E.x.z + F.y.z + G.x + H.y + I.z + J = 0$$

En injectant les composantes d'un rayon dans celle-ci, nous obtenons une équation du second degré en t. $A_q.t^2 + B_q.t + C_q = 0$, avec

$$A_q = A.x_d^{\ 2} + B.y_d^{\ 2} + C.z_d^{\ 2} + D.x_d.y_d + E.x_d.z_d + F.y_d.z_d$$
$$B_q = 2A.x_o.x_d + 2B.y_o.y_d + 2C.z_o.z_d + D(x_o.y_d + y_o.x_d) + E(x_o.z_d + x_d.z_o) + F(y_o.z_d + y_d.z_o) + G.x_d + H.y_d + I.z_d$$
$$C_q = A.x_o^{\ 2} + B.y_o^{\ 2} + C.z_o^{\ 2} + D.x_o.y_o + E.x_o.z_o + F.y_o.z_o + G.x_o + H.y_o + I.z_o + J$$

On en déduit deux solutions

$$t_o = \frac{-B_q - \sqrt{B_q^{\ 2} - 4.A_q.C_q}}{2.A_q}$$

$$t_1 = \frac{-B_q + \sqrt{B_q^{\ 2} - 4.A_q.C_q}}{2.A_q}$$

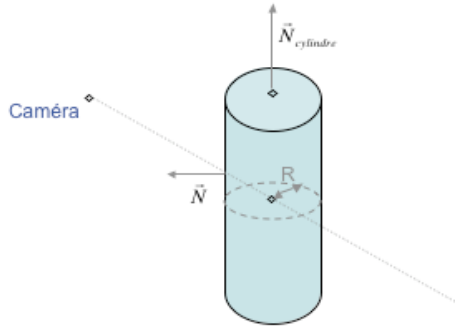
Ensuite la démarche algorithmique à suivre est la suivante :

1. Si $A_q = 0$, alors *la solution* est $t = -\frac{C_q}{B_q}$
2. Sinon, on calcule le discriminant $B_q^{\ 2} - 4.A_q.C_q$. Si celui-ci est négatif, il n'y a *pas d'intersection*.
3. Si le discriminant est positif, on calcule t_o et t_1. On conserve la valeur positive la plus petite des deux et l'on calcule le point d'intersection (x_i, y_i, z_i).
4. On calcule la normale en ce point en se rappelant que ses composantes sont les dérivées partielles respectives de F en ce point vaut

$$\vec{N}ormale = \begin{pmatrix} \partial F/\partial x \\ \partial F/\partial y \\ \partial F/\partial z \end{pmatrix}$$, soit en développant $$\vec{N}ormale = \begin{pmatrix} 2A.x_i + D.y_i + E.z_i + G \\ 2B.y_i + D.x_i + F.z_i + H \\ 2C.z_i + E.x_i + F.y_i + I \end{pmatrix}$$

On normalise ce résultat, et l'on retourne $\vec{N}ormale$ si $\vec{N}ormale.\vec{R}_d > 0$, sinon on inverse $\vec{N}ormale$.

IV.1.4/ Intersection avec un cylindre



Un cylindre étant une quadrique simple, il suffit d'utiliser les résultats précédents et de les appliquer à ce cas particulier.

Pour des raisons de simplicité de calculs, le moteur ne gère que des cylindres parallèles à l'axe des x , à celui des y ou celui des z. Les cylindres peuvent êtres infinis si on affecte zéro au paramètre *hauteur*.

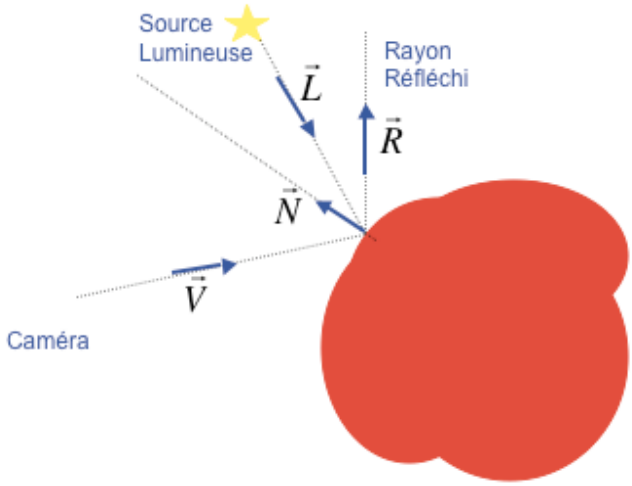
Prenons l'exemple d'un cylindre parallèle à x d'équation $y^2 + z^2 - 1 = 0$. Il suffit de suivre la démarche donnée au paragraphe précédent avec $B = 1, C = 1, J = -1$ et les autres coefficients nuls.

Dans ce cas, la normale vaudra $\vec{N}ormale = \begin{pmatrix} 0 \\ 2y_i \\ 2z_i - 1 \end{pmatrix}$

IV.2/ Détermination de la couleur finale

Une couleur n'est visible que si elle reçoit de la lumière. C'est pourquoi, si un pixel n'est pas dans l'ombre d'un objet, on détermine la source de lumière qui l'éclaire.

Toute source de lumière possède un coefficient représentant son intensité. Ce coefficient est compris entre 0.0 et 1.0.



IV.2.1/ La lumière ambiante

Comme il est impossible de prendre en compte tous les rayons dérivés (rayons réfléchis, transmis etc…) dans la scène, on considère que ceux-ci constitue une lumière ambiante. Cette lumière, de faible intensité (environ 0.2) constitue un éclairage uniforme sur chaque objet constituant la scène.

Sa contribution à la couleur finale est : $C_{base} += C_{objet} \times L_{ambiante}$

IV.2.2/ La lumière diffuse

Pour l'instant, si l'on considère uniquement la lumière ambiante, les objets semblent plats. En effet, les objets sont d'une couleur unie et semble être en 2D. La composante diffuse d'une illumination représente le fait que l'énergie d'un faisceau lumineux s'étale d'autant plus sur la surface d'un objet si son angle d'incidence est proche de la tangente à l'objet.

Pour représenter cet effet, notre moteur suit la valeur de Beer-Lambert :

$$C_{base} += C_{lumiere} \times \vec{L}.\vec{N} \times K_{diffusion}$$

Cette formule fait intervenir l'angle entre le rayon lumineux et la normale au point à travers le produit scalaire. Ainsi, on obtient un dégradé représentatif de la diffusion de la lumière.

IV.2.3/ La lumière spéculaire

La lumière spéculaire représente une des propriétés des matériaux réfléchissants : la brillance. La brillance est en fait l'image de la source que l'œil (la caméra) perçoit via l'objet. Cette image est un cône formé par le rayon lumineux et le rayon réfléchi avec une intensité qui dépend de l'angle entre ces deux vecteurs.

$$C_{base} += C_{lumière} \times (-\vec{L}.\vec{R})^N$$

IV.3/ Les lois de Descartes

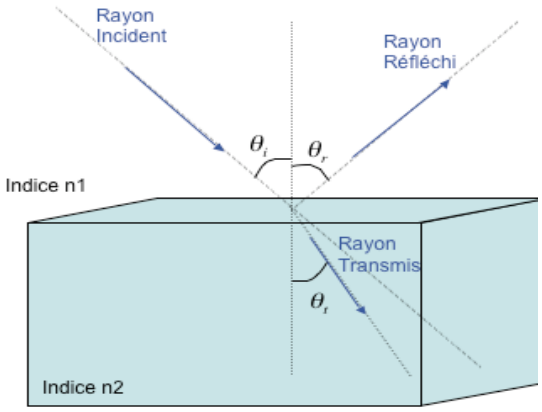
IV.3.1/ La réflexion

La loi de la lumière la plus évidente à mettre en place est le phénomène de réflexion. Si la surface d'un objet est réfléchissante, l'algorithme de lancer de rayons doit non seulement calculer la valeur de l'illumination sur la surface, mais aussi ajouter les reflets des autres objets de la scène. Pour ce faire, il suffit uniquement de connaître l'angle formé par le rayon réfléchi, puis de lancer un nouveau rayon dans cette direction pour déterminer ce qui se reflète.

On calcule le vecteur directeur du rayon réfléchi est calculé suivant la formule suivante:

$$\vec{R} = \vec{V} - 2 \times (\vec{V}.\vec{N}) \times \vec{N}$$

Une fois le vecteur directeur déterminé, il suffit de renouveler l'algorithme de lancer de rayon en lui passant en paramètre le rayon réfléchi. Le rayon réfléchi a pour origine le point d'intersection et suit la direction $\vec{R}$.



IV.3.2/ La réfraction

La seconde loi de Descartes concerne la déviation d'un rayon lumineux lors d'un passage dans les matériaux.

La réfraction permet de prendre en compte la transparence des objets de la scène. Un rayon qui vient frapper un objet transparent va être dévié en fonction de l'indice de réfraction du matériau de cet objet, mais aussi de celui de celui du matériau d'où arrive le rayon (ici le milieu extérieur est toujours l'air, dont l'indice de réfraction est égal à 1).

On détermine l'angle de déviation du rayon transmis lors d'un changement de matériau par la formule :

$$\vec{T} = \frac{n_1}{n_2}\vec{V} - (\frac{n_1}{n_2}\cos\theta_i + \sqrt{1-\sin^2\theta_t})\vec{N}$$

avec $\begin{cases} \cos\theta_i = \vec{V}.\vec{N} \\ \sin^2\theta_t = (\frac{n_1}{n_2})^2(1-\cos^2\theta_i) \end{cases}$

IV.3.3/ La transparence

On entend par transparence que tout objet réfractant (qui laisse passé la lumière) va modifier la structure même de la lumière. En effet, un filtre rouge transparent éclairé par une lumière blanche laissera passer uniquement les composantes rouges de la lumière.

Dans le programme, après le passage dans un objet transparent, un rayon de lumière prend la couleur de cet objet et est atténué par son coefficient de réfraction.

V / Les effets spéciaux pour plus de réalisme

V.1/ Les textures procédurales

Une texture procédurale est une texture qui est calculée lors du rendu d'une image, son principal avantage est qu'elle n'a pas besoin de coordonnées de mapping et qu'elle recouvre n'importe quelle surface ou forme et ce, dans les 3 dimensions.

Un objet n'a plus une seule couleur, mais peut en avoir une multitude. Chaque pixel qui le compose a une couleur propre liée à ses coordonnées. Il faut donc faire attention au temps de calculs car les textures procédurales engendrent beaucoup plus d'opérations pour l'algorithme de raytracing.

V.1.2/ Le damier *(type=1)*

Un des effets que l'on retrouve le plus souvent dans les moteurs de raytracing est le damier car il est facile à mettre en place et nécessite peu de calculs. Dans notre moteur, il est possible de rentrer les deux couleurs qui composent le damier dans la définition du matériau

Rappel : La définition des textures s'écrit

2 type C1.r C1.g C1.b C2.r C2.g C2.b C3.r C3.g C3.b

Petite ambiguïté dans notre moteur seules les couleurs C1 et C2 sont les couleurs prises en compte. La couleur C3 est obligatoire à rentrer mais est inutile (simplification dans l'implémentation).

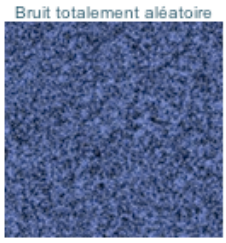
On considère des carrés de couleur de largeur 1.

V.1.2/ Le bruit de Perlin

Afin de modéliser des motifs complexes (nuages, marbres …), il est nécessaire d'avoir à disposition des algorithmes capables de les générer de façon automatique. Pour introduire une grande diversité et par ailleurs un meilleur réalisme, ces algorithmes se doivent d'introduire de l'aléatoire dans ces motifs. C'est là qu'intervient le bruit de Perlin.

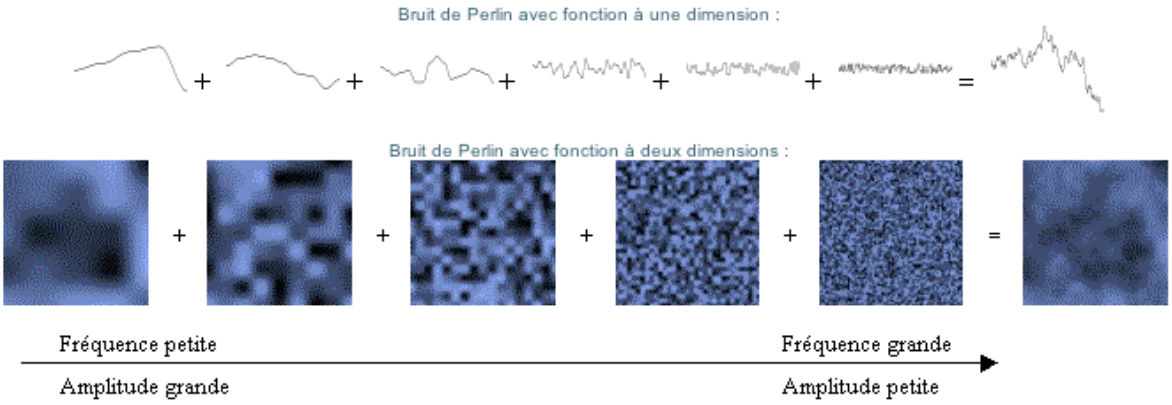
Ken Perlin est chercheur au « Department of Computer Science » au sein de l'université de New York. Il est aussi le directeur du « Media Research Laboratory » . Récompensées à de nombreuses reprises, ces recherches portent principalement sur

l'animation, le graphisme et le multimédia.



Ce bruit est qualifié de cohérent et c'est ici que réside tout l'intérêt de son utilisation. En effet, il ne s'agit pas de générer des motifs totalement aléatoires, car sinon nous obtiendrions un motif "tacheté" très peu réaliste. En réalité, le bruit de Perlin est une somme de signaux aléatoires de fréquences et d'amplitudes différentes sachant que plus un signal est de fréquence élevée, plus son amplitude est faible.

Dans une image, les basses fréquences constituent le squelette alors que les hautes fréquences représentent plutôt les détails. Dès lors, il semble tout à fait normal d'associer les amplitudes élevées aux signaux de basse fréquence.



La fonction utilisée dans notre programme a été directement récupérée sur le site de Ken Perlin (http://mrl.nyu.edu/~perlin/). La fonction *noisef*(x,y,z) requiert trois coordonnées et renvoie un entier.

V.1.2.1/ Le bruit de Perlin seul *(type=2)*

On défini trois seuils v1,v2,v3. Ces seuils sont fixés de manière purement expérimentale et il convient de les modifier jusqu'à obtenir l'effet souhaité. Pour rappel, une texture possède trois couleurs C1, C2, C3.

Soit *n* la valeur absolue de l'entier retourné par la fonction de Perlin. A partir de *n* on détermine la couleur du point avec la démarche suivante :

- Si n < v1, on retourne la couleur C1 de la texture.
- Si v1 < n < v2, on fait une interpolation linéaire et l'on retourne la valeur $C_1 \times \frac{n-v1}{v2-v1} + C_2 \times \frac{v2-n}{v2-v1}$.
- Si v2 < n < v3, on retourne $C_2 \times \frac{n-v2}{v3-v2} + C_3 \times \frac{v3-n}{v3-v2}$
- Si n > v3, on retourne C3.

Cet effet donne des rendus visuellement non réalistes mais assez intéressants.

V.1.2.2/ L'effet Bois *(type=3)*

Cependant à partir de ce bruit, nous pouvons reconstituer des textures très réalistes. Cette fois-ci, n est calculé d'une manière différente en faisant intervenir la partie entière :

$$\begin{aligned} T &= 20 \times noisef(x,y,z) \\ n &= |T - E(T)| \end{aligned}$$

Ici on retourne une interpolation sinusoïdale des deux premières couleurs de la texture, soit :

$$\begin{aligned} ft &= n \times \pi \\ f &= (1 - \cos(ft)) \times 0.5 \\ C_{retournée} &= C_1 \times (1 - f) + C_2 \times f \end{aligned}$$

Rem : On aurait très bien pu utiliser la méthode d'interpolation à trois seuils comme pour la texture avec le bruit de Perlin seul. Il s'agit juste ici, d'étudier différentes méthodes.

V.1.2.3/ L'effet Marbre *(type=4)*

Pour restituer le visuel de marbre, nous allons une nouvelle fois tirailler le bruit de Perlin de la manière suivante :

$$n = \cos(x + noisef(x,y,z))$$

A titre d'exemple, on pourra retourner l'interpolation linéaire des deux premières couleurs de la texture, soit :

$$C_{retournée} = C_1 \times (1 - n) + C_2 \times n$$

V.2/ Le Bump Mapping

Cette technique permet d'ajouter à un objet à priori lisse un aspect rugueux. Pour se faire, nous allons perturber les normales des objets et donc, comme les normales rentrent en compte dans la plupart des calculs de luminosité, l'éclairage à la surface des objets. Ainsi deux points très proches pourront présenter un contraste très fort et ainsi créer un relief en surface.

Deux types de Bump Mapping ont été prévu dans le programme et encore une fois, la perturbation se fera à l'aide d'un bruit de Perlin. On associe au Bump Mapping une valeur ε.

V.2.1/ Perturbation uniforme dans l'espace *(TypeBump=1)*

Pour chaque point intersecté par un rayon, on crée trois coefficients issus de la normale $\vec{N}$:

$$\begin{aligned} x &= noisef(\vec{N}_x - \varepsilon, \vec{N}_y, \vec{N}_z) - noisef(\vec{N}_x + \varepsilon, \vec{N}_y, \vec{N}_z) \\ y &= noisef(\vec{N}_x, \vec{N}_y - \varepsilon, \vec{N}_z) - noisef(\vec{N}_x, \vec{N}_y + \varepsilon, \vec{N}_z) \\ z &= noisef(\vec{N}_x, \vec{N}_y, \vec{N}_z - \varepsilon) - noisef(\vec{N}_x, \vec{N}_y, \vec{N}_z + \varepsilon) \end{aligned}$$

Ensuite, il suffit ensuite d'ajouter ces coefficients aux composantes de la normale : $N_x = N_x + x,\ N_y = N_y + y,\ \ N_z = N_z + z$

V.2.2/ Perturbation d'une surface plane *(TypeBump=2)*

Ce type de perturbation permet de perturber la surface d'un plan, idéal pour retranscrire une surface aqueuse.

On ne peut pas utiliser la méthode précédente pour perturber les normales d'un plan, car elles sont toutes égales entre elles. Il faut donc faire intervenir un nouveau paramètre pour les distinguer. Nous nous servons alors des coordonnées des points d'intersection et l'on pose :

$$\begin{aligned} X &= N_x + PtInter_x \\ Y &= 50 \times N_y + 100 \times PtInter_y \\ Z &= N_z + PtInter_z \end{aligned}$$

on calcule alors

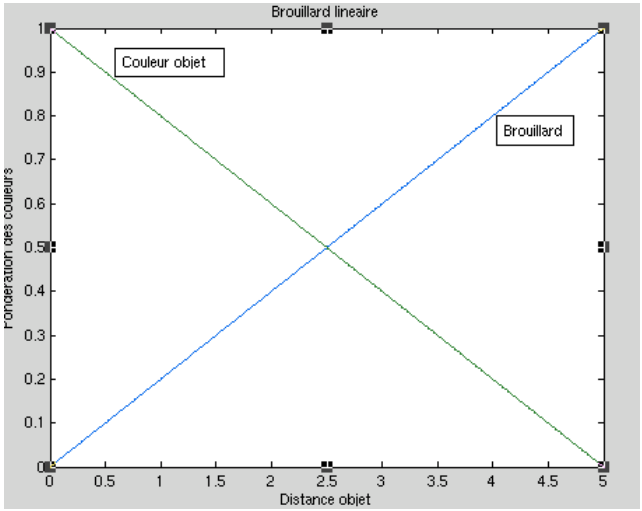
$$\begin{aligned} x &= noisef(X-\varepsilon,Y,Z) - noisef(X+\varepsilon,Y,Z) \\ y &= noisef(X,Y-\varepsilon,Z) - noisef(X,Y+\varepsilon,Z) \\ z &= noisef(X,Y,Z-\varepsilon) - noisef(X,Y,Z+\varepsilon) \end{aligned}$$

Ensuite, de même que précédemment on ajoute à la normale les coefficients trouvés : $N_x = N_x + x,\ N_y = N_y + y,\ \ N_z = N_z + z$. Ici nous amplifions la déformation sur l'axe des y, il convient donc de l'utiliser sur des plans coplanaires au plan $(O,\vec{x},\vec{z})$.

V.3/ Le brouillard

Le moteur est aussi capable de gérer les effets de brouillard. Le brouillard dépend de la distance entre l'objet et l'observateur et est de couleur blanche.

V.3.1/ Le brouillard linéaire *(TypeBrouillard=1)*



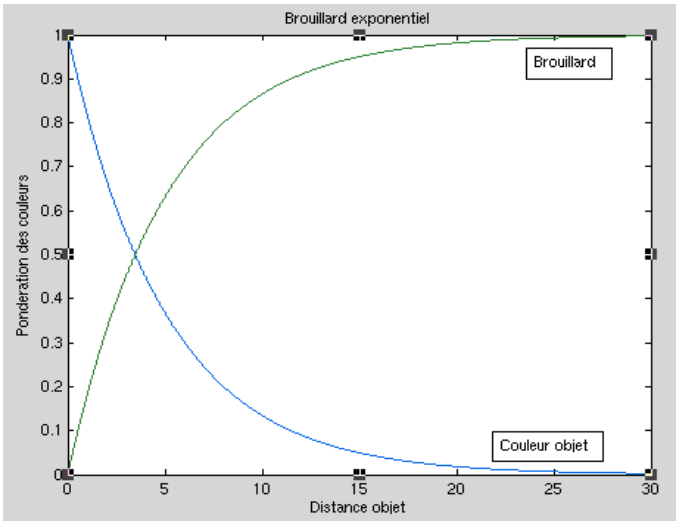
Pour ce type de brouillard, la couleur finale du point envisagé est

$$C_{Finale} = C_{Brouillard} \times (1 - \frac{dObjet}{dBrouillard}) + C_{Objet} \times \frac{dObjet}{dBrouillard}$$

Evidemment, si $dObjet > dBrouillard$, seule la couleur du brouillard rentre en compte.

Le brouillard sur l'image croit régulièrement et atteint son maximum dès $dBrouillard$.

V.3.2/ Le brouillard exponentiel *(TypeBrouillard=2)*



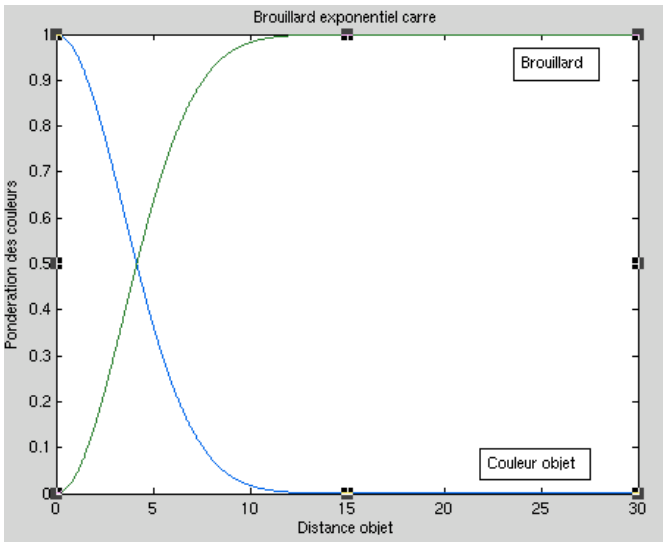
Ce brouillard suit la formule :

$$C_{Finale} = C_{Brouillard} \times (1 - e^{-(\frac{dObjet}{dBrouillard})}) + C_{Objet} \times e^{-(\frac{dObjet}{dBrouillard})}$$

La distance de vue est très grande car on parvient à distinguer des objets très loin dans le brouillard. En effet, le brouillard est total seulement au bout de $5 \times dBrouillard$.

Ce brouillard ajoute au rendu un effet très réaliste.

V.3.3/ Le brouillard exponentiel carré *(TypeBrouillard=3)*



La formule de cet effet est :

$$C_{Finale} = C_{Brouillard} \times (1 - e^{-(\frac{dObjet}{dBrouillard})^2}) + C_{Objet} \times e^{-(\frac{dObjet}{dBrouillard})^2}$$

L'intérêt de ce type de brouillard est qu'il laisse une plage de vision très claire de largeur $dBrouillard$. Cette plage est suivie d'une plage de transition de même largeur puis d'un brouillard très dense à une distance de $2 \times dBrouillard$.

VI / Structures utilisées

Le projet a été écrit en C++. Les différentes classes implémentées sont les suivantes :

- Le cœur du moteur *main*
- Les classes propres à la scène :
  - *CEntite* tout objet d'une scène dérive de cette classe.
  - *CScene,*
  - *Les différents objets CObjet*
    - *CCylindre*
    - *CPlan*
    - *CSphere*
  - Les différents matériaux *CMateriau*
    - *CMatSimple*, textures simples
    - *CMatTextProc*, textures procédurales
  - Les lumières *CLumiere*

        - *CSource*
    - *Les caméras CCamera*
        - *CCameraActive* définie la caméra servant de point d'observation

- Les classes de soutien :
    - Les classes mathématique *CVect3d*, *CCoord3d*
    - *CIntersection*, *CRayon*
    - La classe du bruit de Perlin *CPerlin*
    - Les couleurs *CCouleur*

VI.1/ La scène

La classe *CScene* est la classe qui contient toutes les informations utiles à la construction du rendu. En particulier, elle stocke :

- Sous forme de vecteurs les objets, les caméras et les matériaux.
- La valeur de la lumière ambiante
- Le type et la distance de brouillard

En effet, la méthode de calcul du brouillard a été placée dans cette classe, car les informations nécessaires ne sont accessibles qu'à ce niveau.

VI.2/ Les objets

Une classe abstraite *CObjet* a été créée et toutes les formes dérivent de celle-ci. Cette classe permet d'associer un matériau à une forme géométrique et de définir la méthode virtuelle pure de calcul d'intersection. L'intérêt de cette classe est qu'elle permet de constituer des listes d'objets différents.

VI.3/ Les couleurs

Les couleurs sont stockées dans une classe dédiée *CCouleur*. Elles sont définies selon le modèle RGB (Rouge Vert Bleu) et acceptent des valeurs entre 0 et 255. En cas de dépassement de la valeur haute (300 par exemple), la couleur est automatiquement corrigée et est plafonnée à la valeur haute (255).

Le rendu final est au format '.raw' qui est un format brut. Il s'agit en fait d'une suite de valeurs de couleur. Pour chaque pixel, il y a les trois valeurs des couleurs associées.

VI.4/ Les lumières

Une classe abstraite *CLumiere* a été implémentée pour les mêmes raisons que la classe *CObjet*. Certes, le moteur ne gère qu'un type de lumière (omnidirectionnelle) mais ainsi de nouveaux types de lumière pourraient être ajoutées.

Cette classe définie les méthodes virtuelles pures de calcul de l'éclairage de diffusion et de l 'éclairage spéculaire. Ces méthodes sont implémentées dans les classes dérivées.

VI.5/ Les classes d'outils mathématiques

Les classes *CVect3d* et *CCoord3d* sont présentes et permettent de définir des opérations géométriques très utiles comme le produit scalaire, le produit vectoriel ou encore la construction d'un vecteur à partir de deux points.

Si la classe *CCoord3d* n'est pas énormément utilisée, elle permet de bien faire la distinction entre un objet vecteur et un point.

Rem : Au début du projet, une classe *CMatrice* avait été prévue, mais au final elle n'a pas été exploitée.

VI.6/ La classe d'intersection

Dans la classe *CIntersection,* sont stockés les coordonnées du point d'intersection, la normale en ce point, le matériau de l'objet touché et la distance de l'objet à la caméra. Dans la fonction de rendu, cette collection d'information est très utile.

La perturbation de la normale pour l'effet de BumpMapping se fait dans cette classe.

Conclusion

Notre moteur de Raytracing est, certes très loin d'être complet, mais il a le mérite de prendre en compte les effets de base. De plus, il s'agit d'une base solide pour l'ajout facile de nouvelles méthodes.

Parmi elles, on peut citer les suivantes :

- Le traitement de l'anti-aliasing : il s'agit de lancer 4 rayons pondérés pour chaque pixel. Ainsi, l'effet d'escalier sur le bord des formes disparaîtrait.
- L'ajout de nouvelles formes géométriques
- L'ajout de sources de lumières : des sources unidirectionnelles par exemple.
- La géométrie constructive CSG (Constructive Solide Geometrie) : on pourrait alors ajouter ou retrancher des formes entre elles.
- L'implémentation d'une interface graphique pour afficher les rendus directement sur une fenêtre et faciliter la création d'une scène.
- Le plaquage de texture sur les formes.
- L'ajout d'un effet pour flouter les ombres
- L'atténuation de la lumière avec la distance
- L'optimisation du code
- L'approche à l'aide de threads pour calculer les images
- Etc …

De plus, lors des phases de test, une vidéo à été réalisée, pour se faire, il a suffi de créer 250 images avec le moteur en modifiant à chaque fois la position de la caméra. La vidéo dure 10 secondes (25 images/seconde) et la caméra tourne autour de la scène suivant un cercle (on aurait tout naturellement pu faire suivre une spline à la caméra).

Cependant, il faudrait optimiser la production des images car, pour faire la vidéo, une compression à l'aide d'un logiciel d'image au format '.jpg' a été nécessaire. Or, cette compression est une opération manuelle qu'il faut réitérer pour chaque image. Cette opération est donc longue et très fastidieuse ce qui explique l'unique vidéo créée.

Bibliographie

A propos des moteurs de Raytracing

- Implémentation d'un moteur de Raytracing.
  http://www.flipcode.com/articles/article_raytrace01.html
- Site présentant un moteur de Raytracing, auteurs Thomas Bonfort, Delphine Chaigneau, Olivier Galizzi et Laure Heigeas.
  http://heigeas.free.fr/laure/ray_tracing/index.html
- Projet d'implémentation d'un moteur de Raytracing, Gruson Sébastien et Rabat Cyril.

A propos des principes de Raytracing

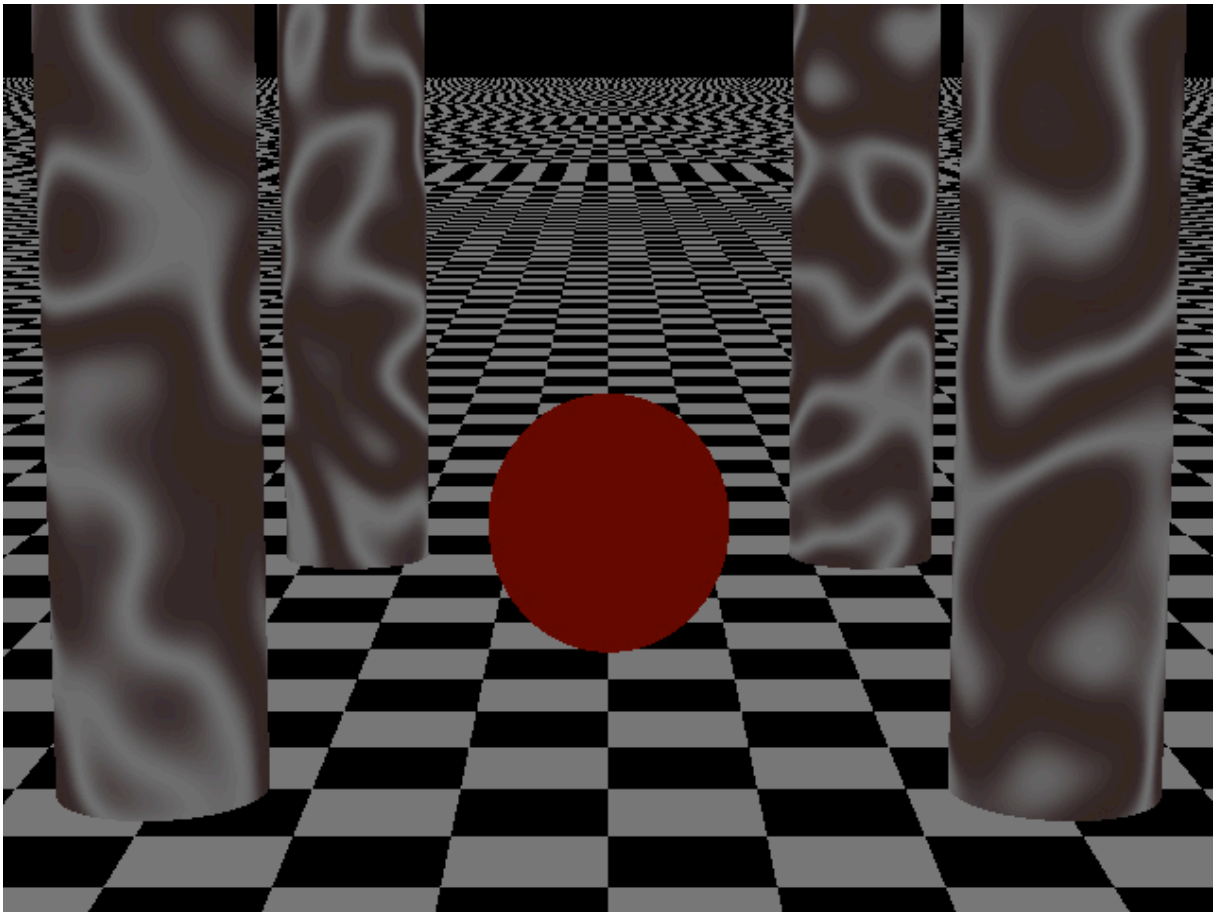
- Recherche d'intersection avec les formes géométriques courantes :
  http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter0.html
- Forum de discussion sur le Raytracing :
  http://forum.hardware.fr/forum.hardware.fr/hardwarefr/Programmation/Article-un-raytracer-de-base-en-C--sujet-40224-1.html
- Article sur les phénomènes de réflexions et réfractions, « *Reflections and Refractions in Raytracing* », Bram de Greve.

A propos du bruit de Perlin

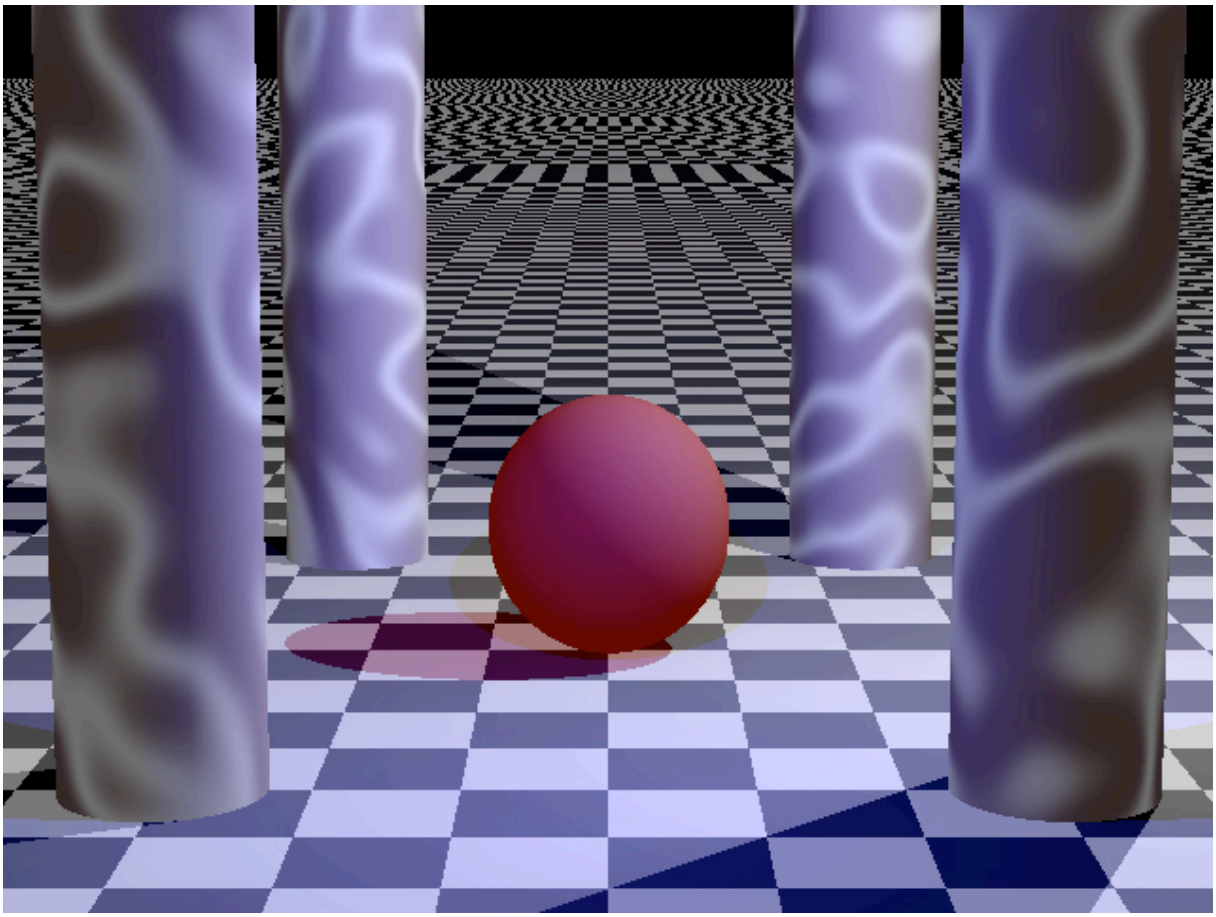
- Site officiel de Ken Perlin : http://mrl.nyu.edu/perlin/
- Site sur les textures procédurales liées au bruit de Perlin :
  http://freespace.virgin.net/hugo.elias
- FAQ sur le bruit de Perlin par Matt Zucker :
  http://www.robo-murito.net/code/perlin-noise-math-faq.html

Annexes

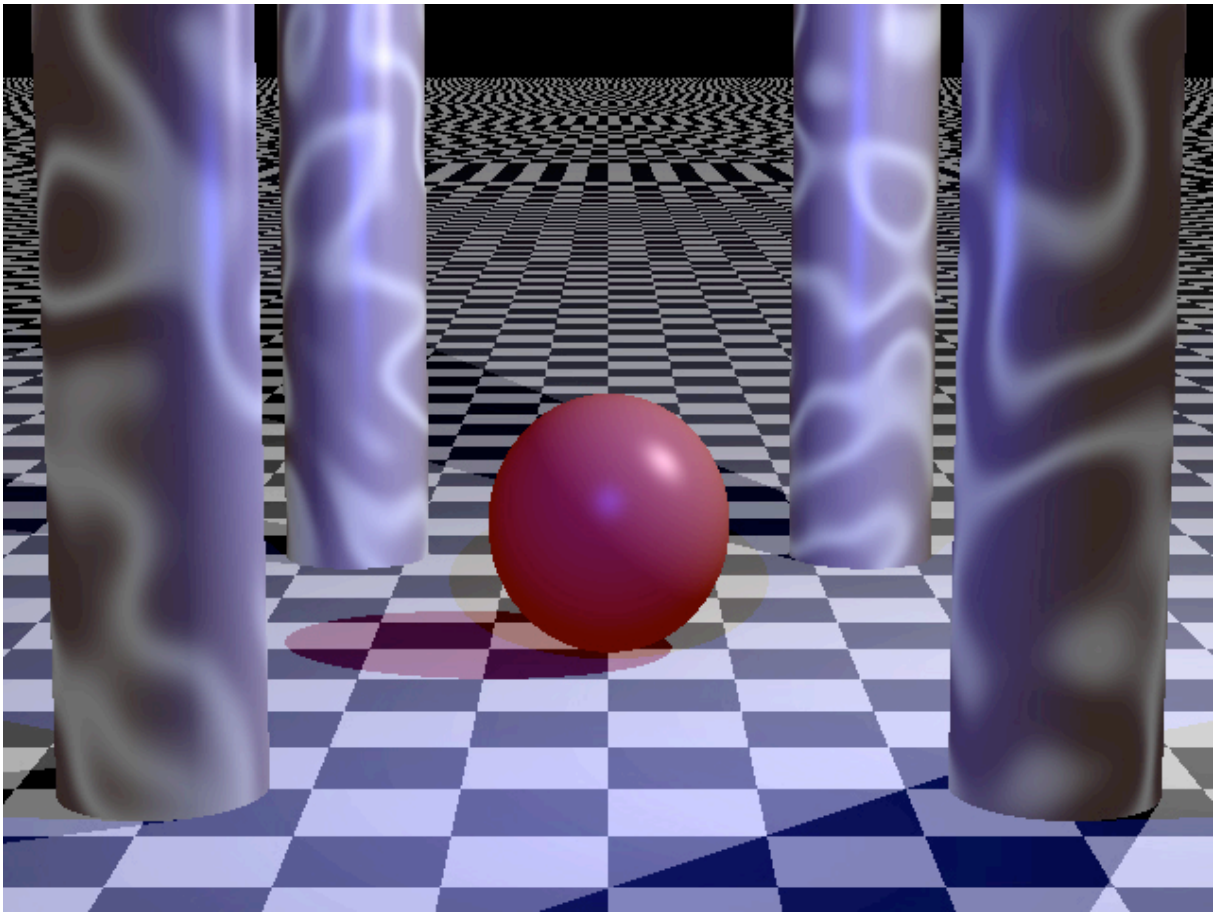
Images issues du moteur



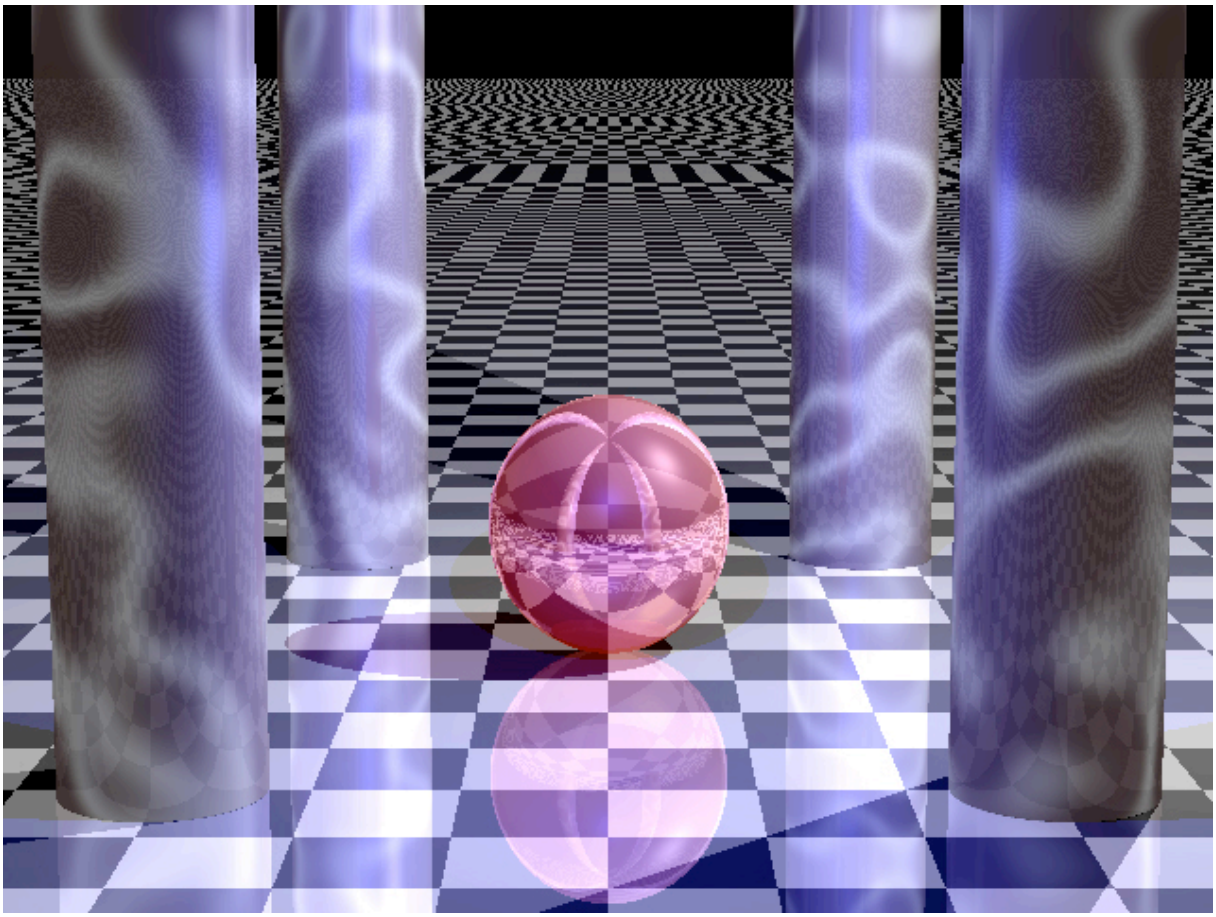
Lumière Ambiante seule



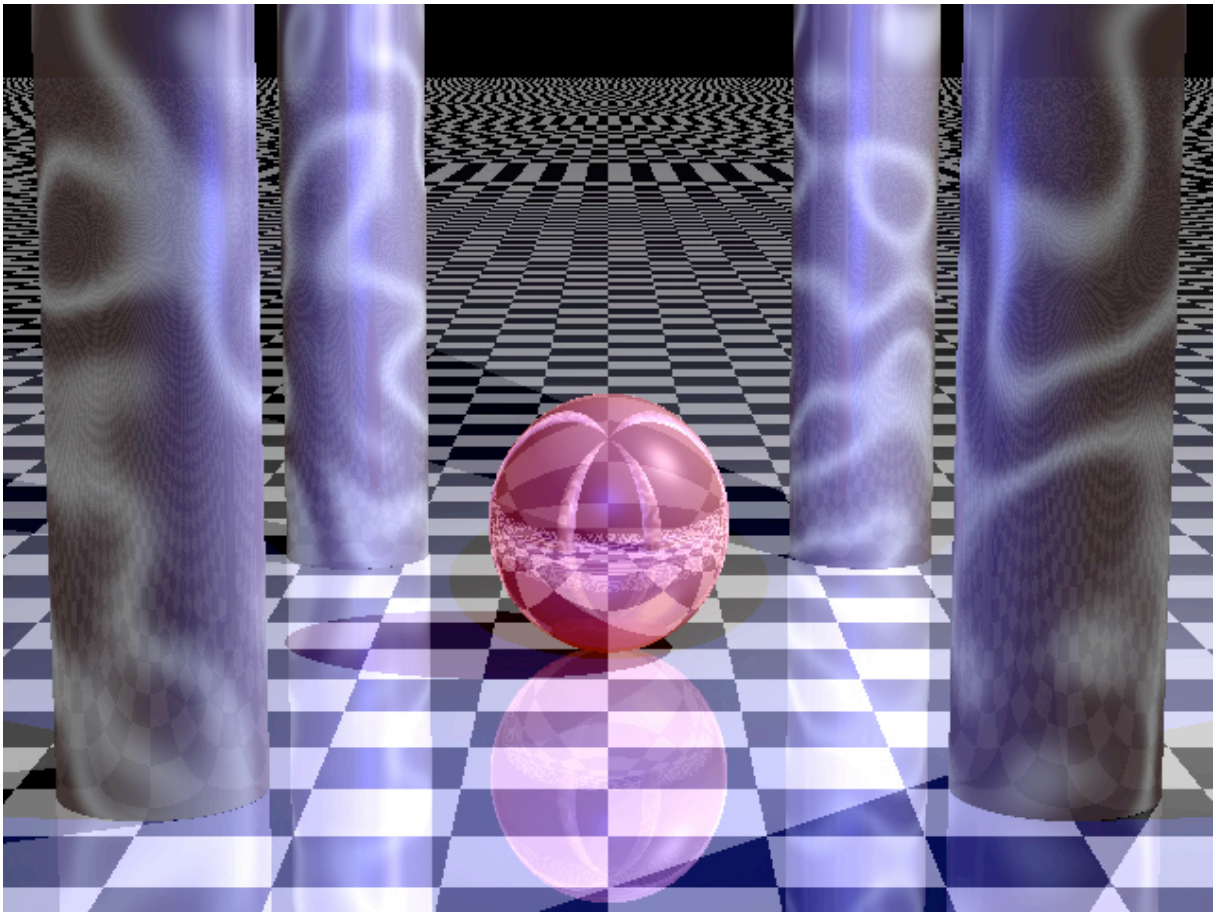
Ajout de la lumière de diffusion



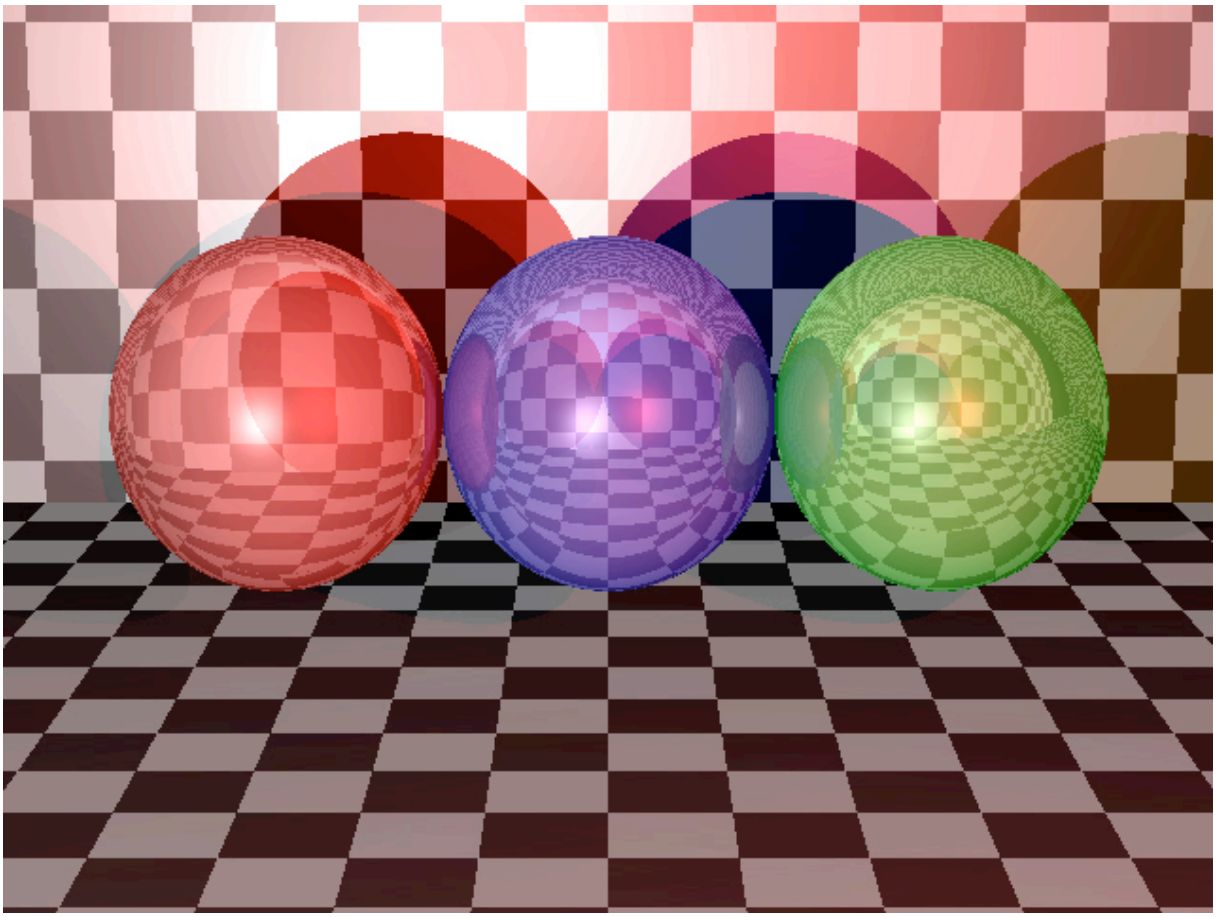
Ajout de la lumière spéculaire



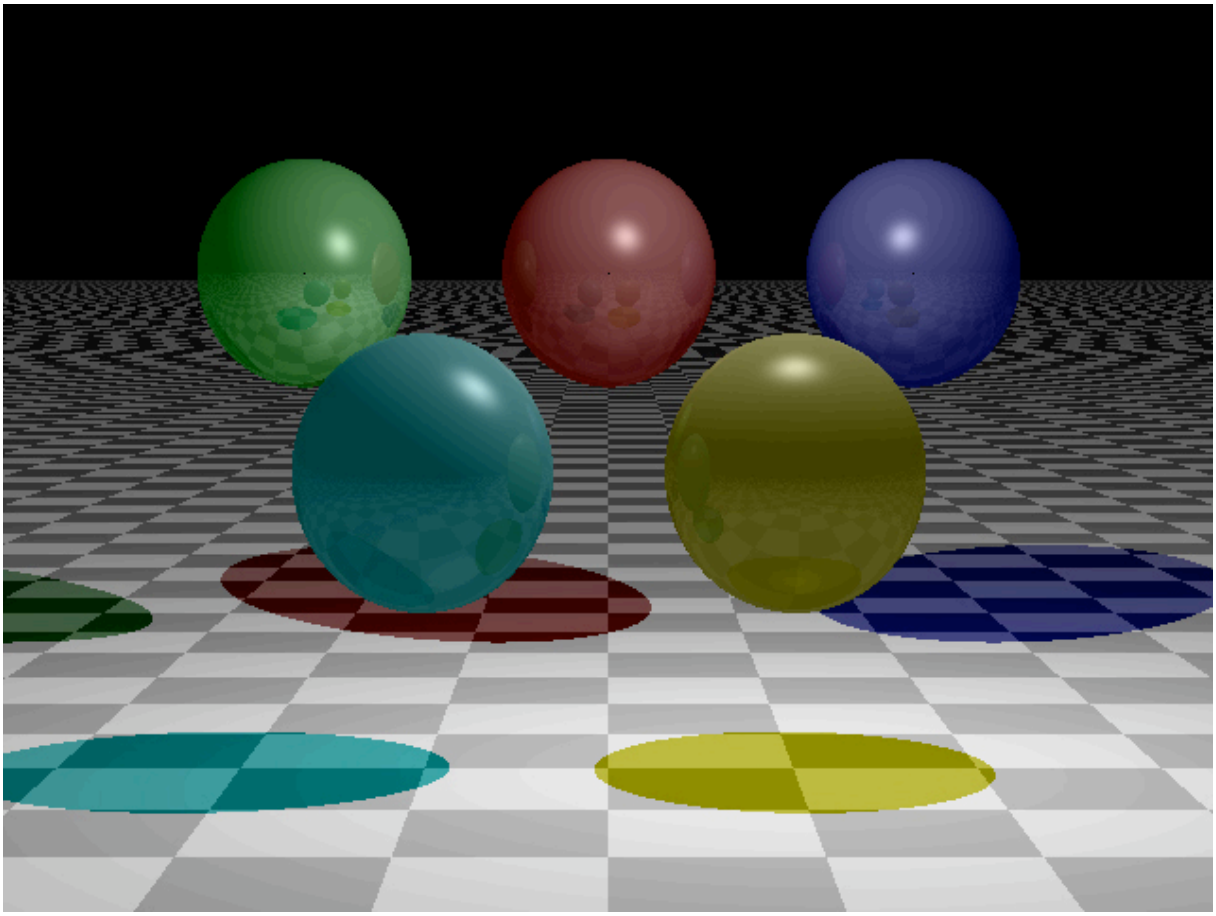
Ajout du phénomène de réflexion



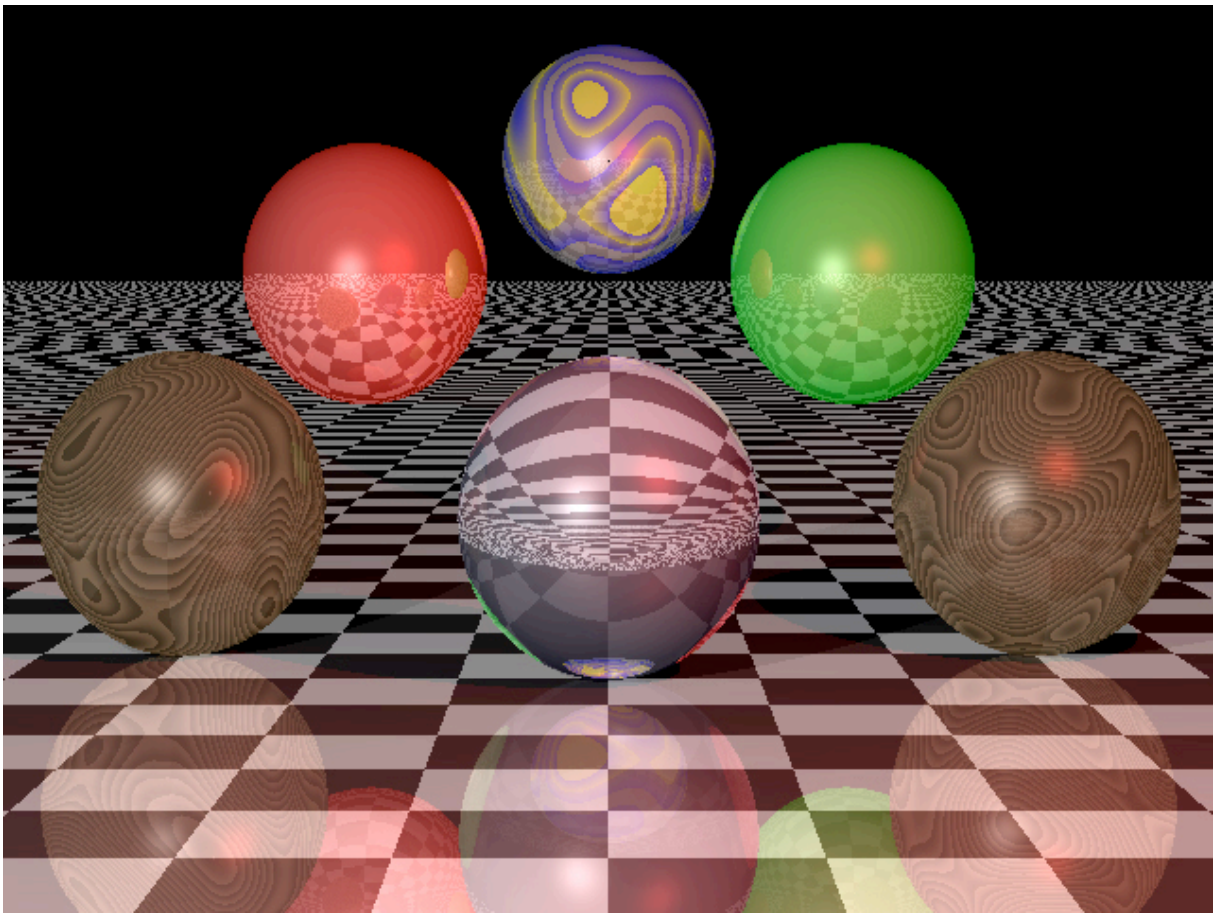
Ajout du phénomène de réfraction



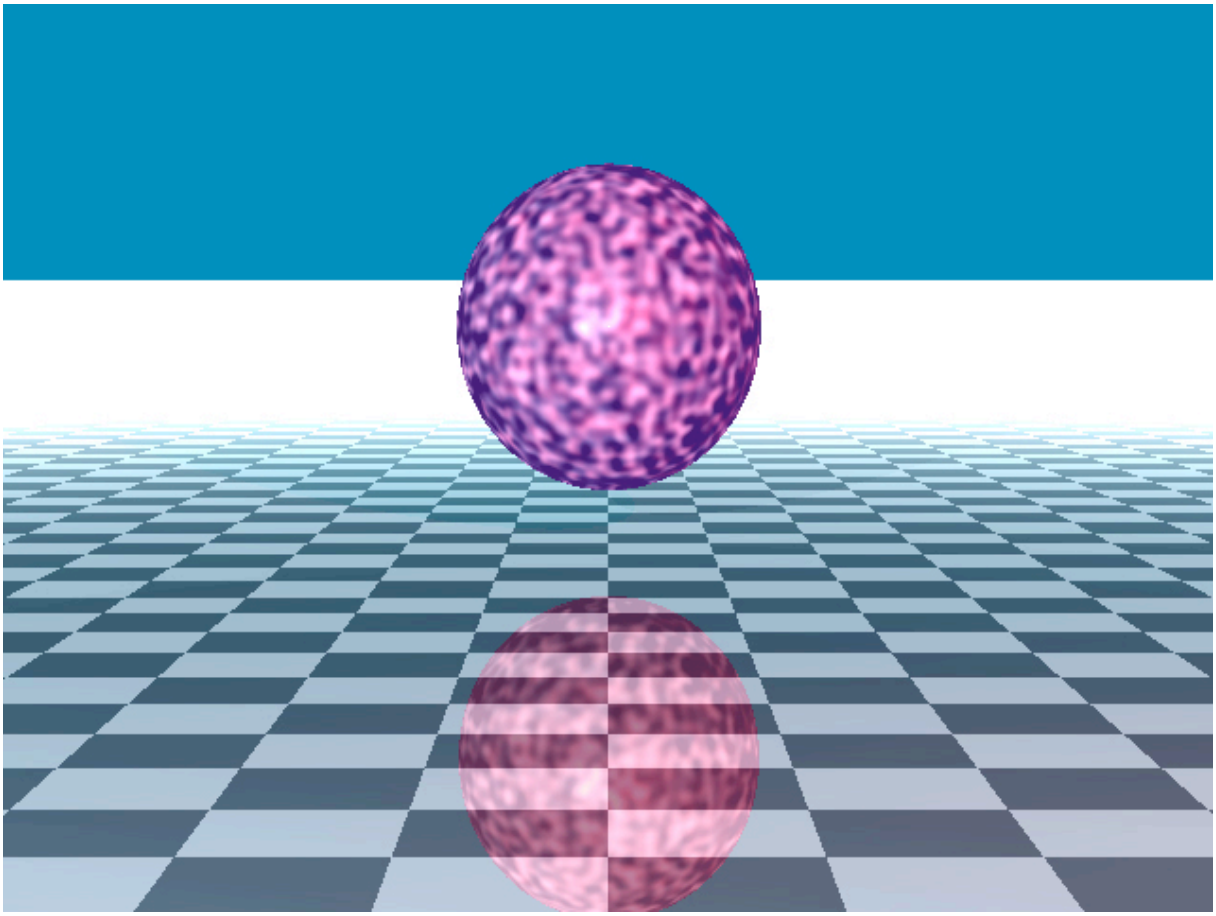
Phénomène de réfraction avec des indices différents



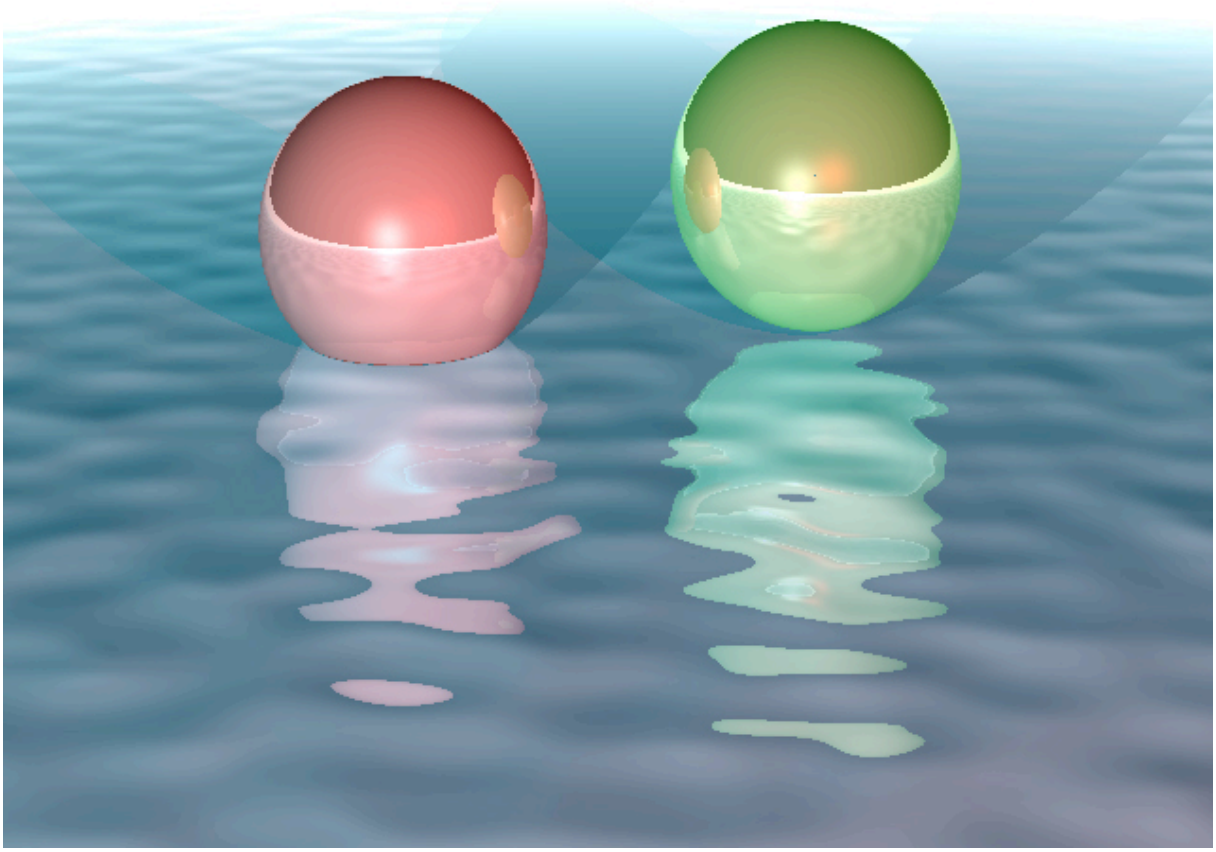
Phénomène de transparence



Exemple de textures procédurales



Effet de Bump Mapping



Effet d'eau avec le Bump Mapping

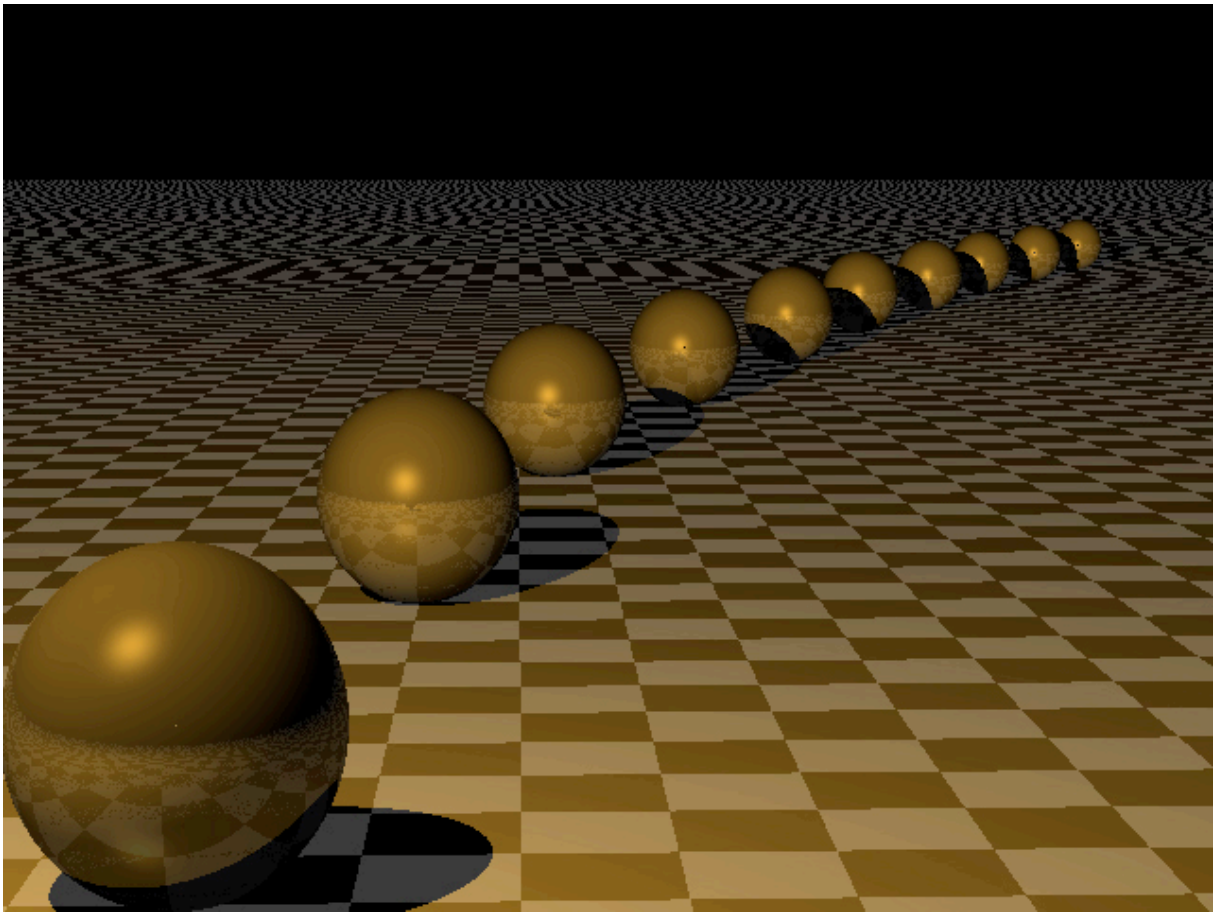
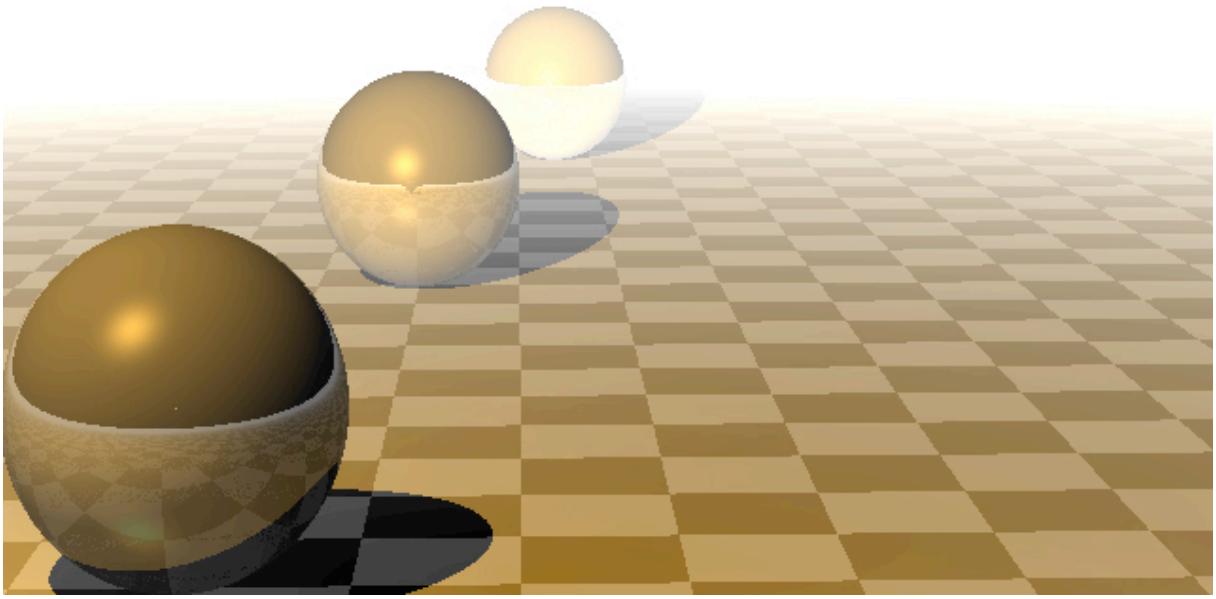
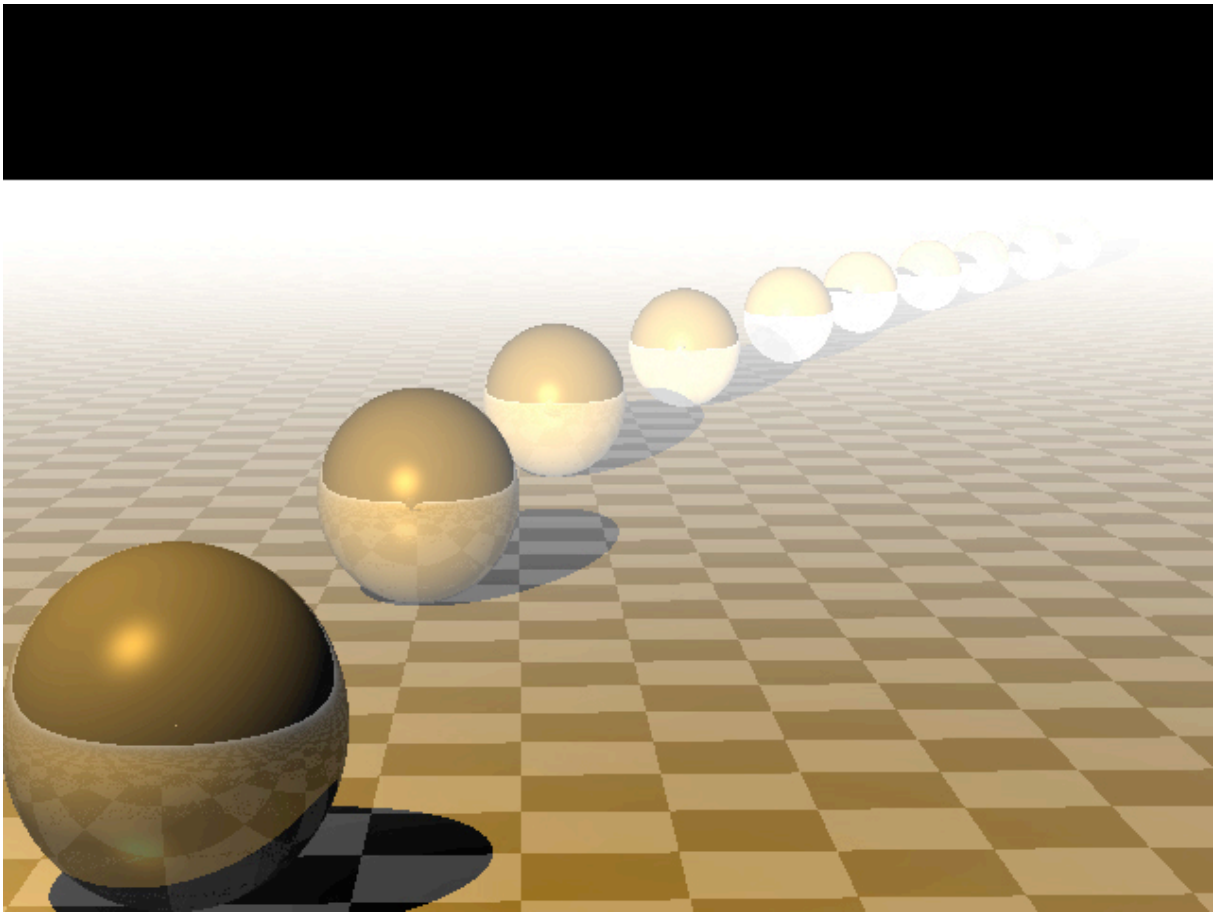


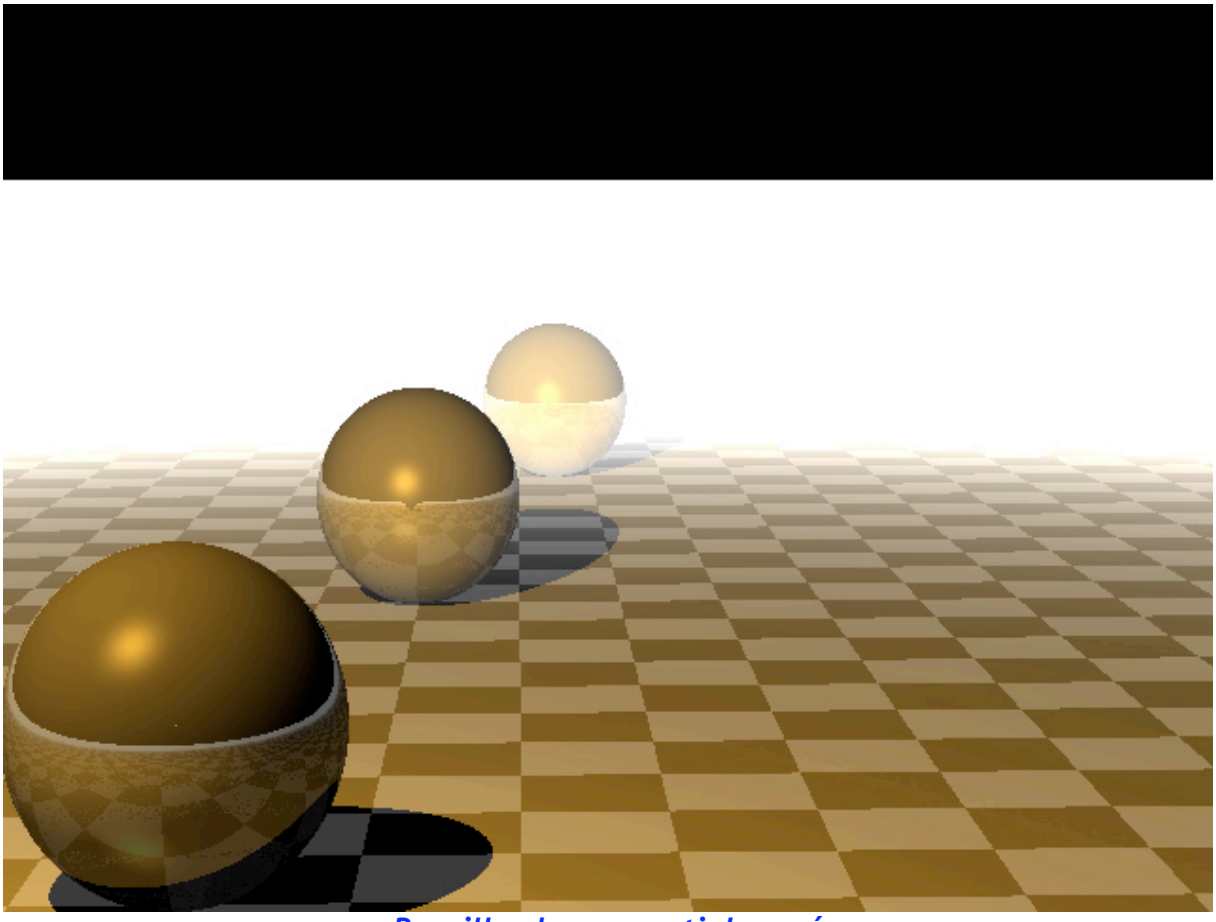
Image de base sans brouillard



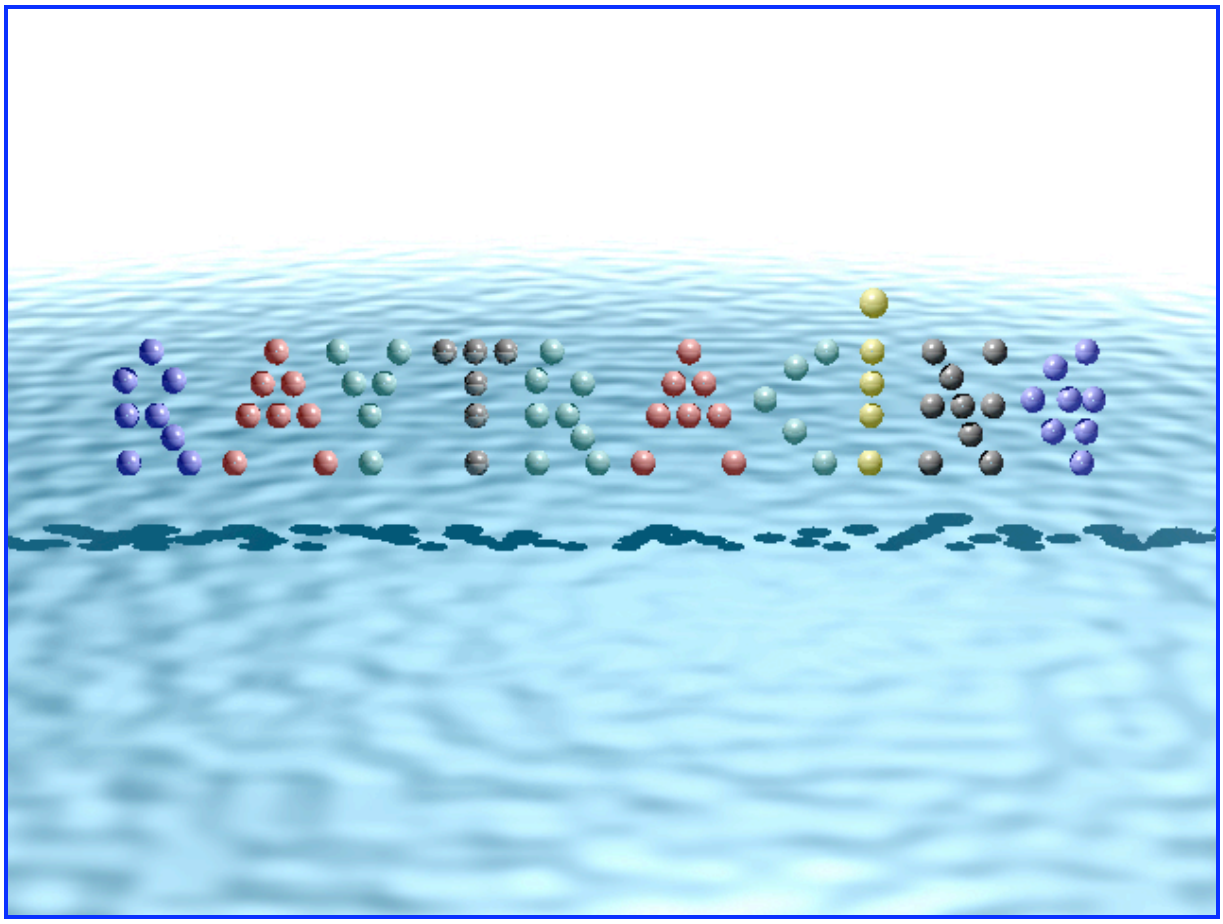
Brouillard linéaire



Brouillard exponentiel



Brouillard exponentiel carré



Raytracing ...