PROJET
ILLUMINATION GLOBALE
PARTIE 2

Lilian Buzer

1. Suite du projet

Nous vous proposons maintenant des options pour la suite de votre projet. Ce document est découpé en deux parties.

Dans la première partie, nous vous proposons trois options pour améliorer le moteur de rendu 3D pour construire une image plus photoréalistique :

- La technique des Virtual Point Lights qui simulent les rayons de niveau 2 en créant des milliers de sources lumineuses issues de rayons de niveau 1.
- La technique du Pathtracing qui consiste non plus à lancer 1 seul rayon par pixel mais des milliers de rayon et à suivre leurs parcours dans la scène. Le pixel résultant portera la couleur correspondant au cumul de leurs contributions
- La technique du Raytracing permettant de gérer les matières transparentes et les objets réfléchissants.

Pour permettre ces améliorations, nous vous donnerons des conseils pour effectuer des lancés de rayons aléatoires et faire des calculs en multithread avec C#.

Dans la seconde partie, nous vous proposons des options pour ajouter des effets graphiques dans votre projet. Ces modifications sont plus simples mais peuvent être effectuées en complément de la première partie. Voici la liste des options proposées :

- Gestion des Mesh
- Ombres dégradés / Soft Shadow
- Surfaces de révolution

Idéalement il faudrait choisir une option dans la partie 1 et une option dans la partie 2 ou trois options dans la partie 2 pour obtenir la note maximale. Dans tous les cas, pensez à discuter de vos choix avec l'intervenant pour garantir que vous ne partiez pas sur une fausse route.

2. Amélioration du moteur

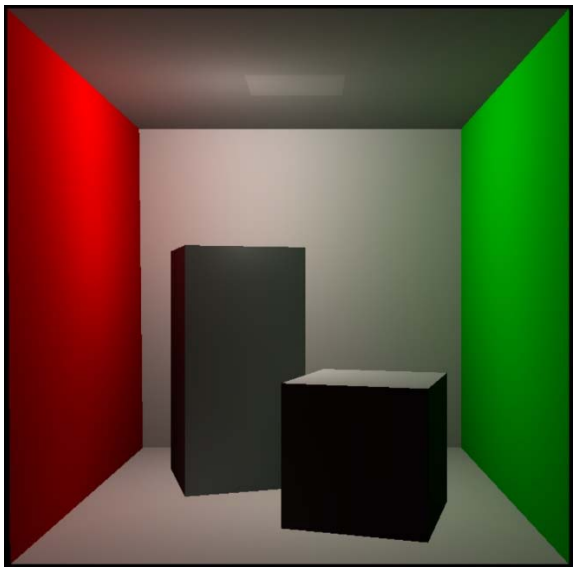
A ce niveau, vous avez parcouru une bonne partie du projet. Nous vous offrons l'opportunité d'intégrer une dernière option à choisir parmi plusieurs suivant vos affinités. Les deux premières options, VPL et Pathtracer, doivent être considérées en premier choix car elles représentent la pointe des technologies.

Virtual Point Lights

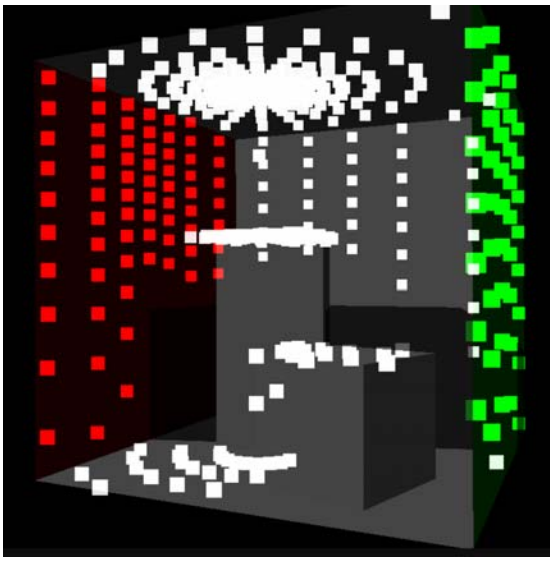
La méthode des Virtual Point Lights permet de simuler des rayons lumineux de niveau 2 et ceci sans rajouter beaucoup de lignes de code au moteur actuel. Cependant, l'augmentation de la quantité de calcul va vous pousser à optimiser le code de manière drastique. Ce sera l'occasion d'utiliser le profiler de Visual Studio (demander à l'intervenant).

L'idée est la suivante, nous allons prendre la source lumineuse principale et nous allons lancer aléatoirement des rayons depuis cette source dans la scène. Chaque rayon va heurter un objet. Nous calculons alors la couleur de l'intersection. L'astuce consiste à transformer ce point de couleur en une lampe ponctuelle pour simuler des rayons lumineux de niveau 2. En production, on peut aller jusqu'à la création de 100 000 nouvelles lampes.

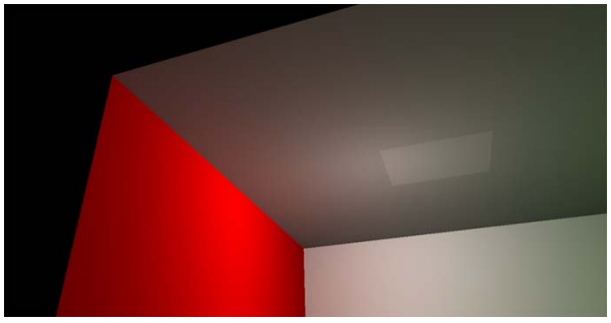
Dans une deuxième passe, nous reprenons le moteur précédent et nous lançons le calcul de la scène en utilisant les lampes principales et les VPL annexes. Nous utilisons uniquement les modèles d'illumination de niveau 1 habituels. Nous allons ainsi arriver à simuler un éclairage de niveau 2.



Scène utilisant les VPL



Mise en place des VPL et calcul de leurs couleurs



Remarquez la présence de dégradés dans la luminosité.

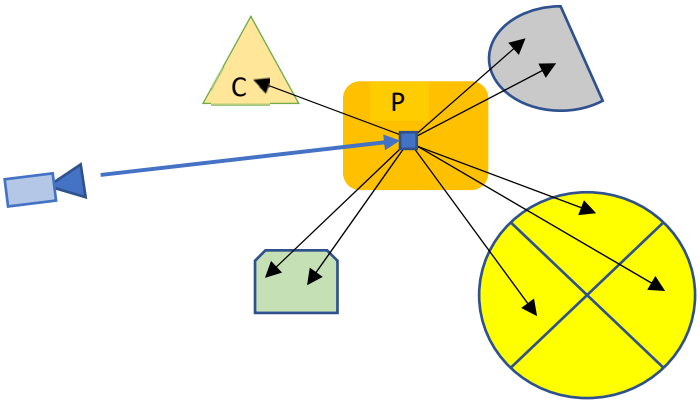
PathTracing

Dans le modèle d'illumination diffus de niveau 1, on ne considère que la contribution directe des rayons lumineux issues des lampes sur les intersections entre les rayons issus de la caméra et les objets. Cependant, les rayons lumineux proviennent de toutes les zones de la scène !

Pour améliorer notre modèle diffus, nous allons à partir du point d'intersection, moyenner des couleurs provenant d'un échantillonnage de plusieurs rayons éclairant ce point :

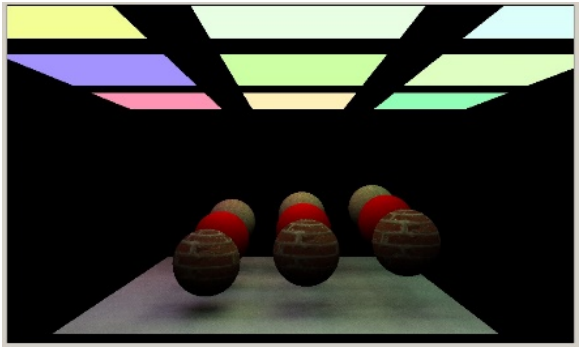
Calcul du diffus - méthode PathTracer

Depuis un point courant P = intersection entre un rayon issu de la caméra et un objet
Calculer N la normale au point P
Diffus = Couleur(0,0,0)
Pour i = 1 à n
  Trouver un vecteur unitaire aléatoire R tq N.R > 0
  Lancer le rayon R et rechercher son intersection I
  Si I est sur un objet : C = couleur de I en prenant diffus de niv 1 + spéculaire de niv 1
  Si I est sur une lampe : C = couleur de la lampe
  Diffus += (N.R)*C
Diffus /= n
retourner Diffus

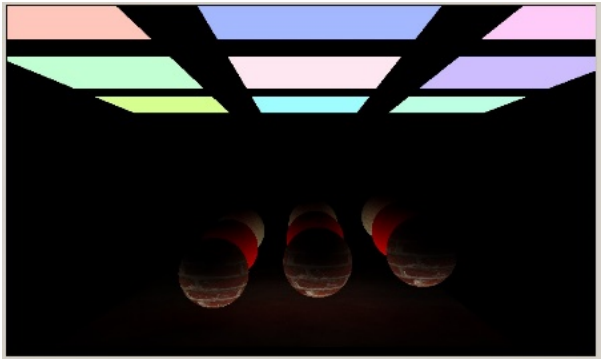


Dans le modèle du PathTracing, les lampes doivent être modélisées comme des objets de la scène (par une sphère ou un rectangle). L'idée ici est que les sources de lumière ont une surface et plus elles sont grandes, plus elles éclairent la scène.

Exemple de notre méthode :

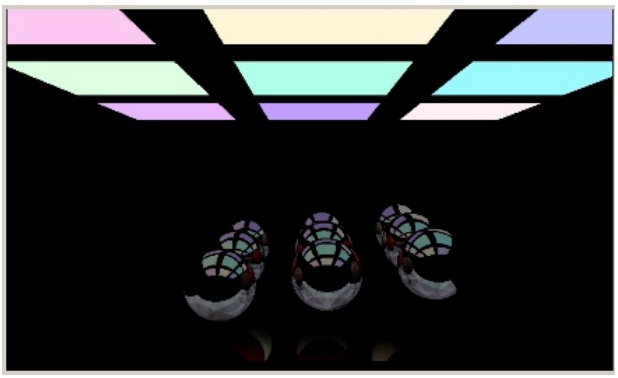


Contribution des lampes au diffus du Pathtracing. On exécute l'algorithme précédent en ne comptabilisant que la contribution des lampes. Pour chaque point des sphères et du sol, on lance des rayons pour le Pathtracer qui vont apporter une contribution uniquement s'ils touchent une des lampes au plafond. Ainsi le bas des boules n'est pas éclairé car les rayons lancés depuis cet endroit n'atteignent pas les lampes.



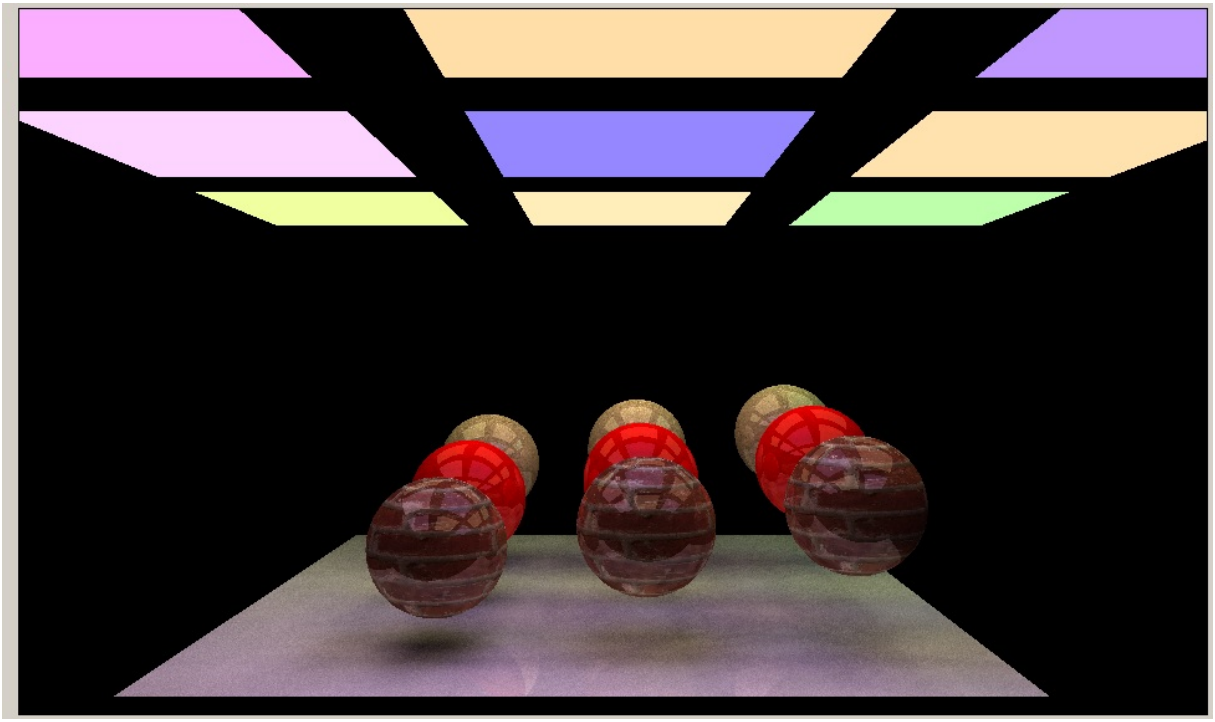
Contributions des objets au diffus de type Pathtracing. On exécute l'algorithme précédent en ne comptabilisant que la contribution des objets. Le sol a disparu car tous les rayons du pathtracer relancé depuis le sol n'atteigne aucun endroit éclairé directement par une lampe.

Par contre, le bas des boules subit l'influence de l'éclairage indirect du sol. Les rayons émis depuis ces zones touchent le sol qui lui est directement éclairé par les lampes du plafond. C'est une contribution forte car sans elle, le bas des sphères resterait sombre.



Effet des spéculaires

Résultat final correspondant à l'addition des trois images précédentes :



Notez le grain qui apparaît dans l'image, il est propre à la méthode du Pathtracing. Pour le faire diminuer, il faut augmenter la taille de l'échantillon de vecteurs aléatoires… d'un facteur 10, mais cela veut dire que la méthode prend 10 fois plus de temps.

Raytracer

Le Raytracer est une approche intéressante. Très à la mode dans les années 90/2000, elle permet de représenter des réflexions et des réfractions complexes tout à fait impressionnantes. Cette méthode est basée sur la récursivité. Lorsqu'un point d'intersection est trouvé entre un rayon et un objet, nous calculons sa couleur de la manière suivante :

$$Coul = Coul_{Diff/Spec/Amb\ Niv\ 1} + \rho_1 Coul_{Réflexion} + \rho_2 Coul_{Réfraction}$$

Ainsi un rayon qui heurte une surface nécessite un calcul de Diff/Spec/Amb habituel. Ensuite, on génère deux nouveaux lancés de rayons depuis le point courant : un pour la réflexion et un pour la réfraction (si l'objet est transparent). La direction du rayon réfléchi se construit par la loi de Descartes avec la symétrie autour du vecteur normal. Pour calculer le rayon réfléchi, il suffit de reprendre la formule donnée dans le premier poly pour le calcul du spéculaire.

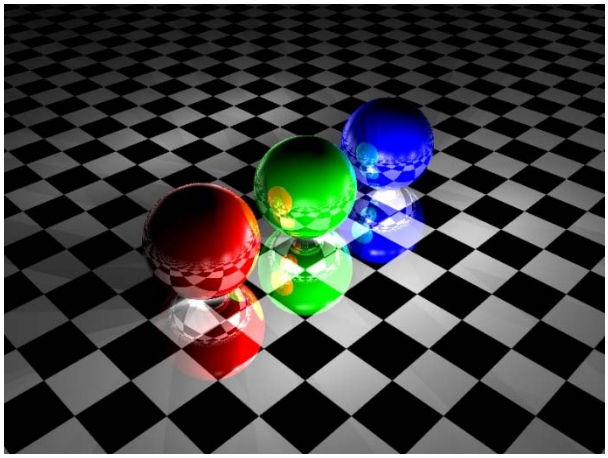
Pour les surfaces transparentes, la direction du rayon réfracté se déduit à partir de la loi de Fresnel : $n_1 sin_{i1} = n_2 sin_{i2}$ où n désigne l'indice de Fresnel du matériau concerné. Pour gérer la formule de Fresnel, il faut déterminer le milieu d'origine et le milieu que va traverser le rayon, la seule connaissance de l'intersection ne vous indique pas le sens de traversée du rayon. Voici quelques valeurs utiles :

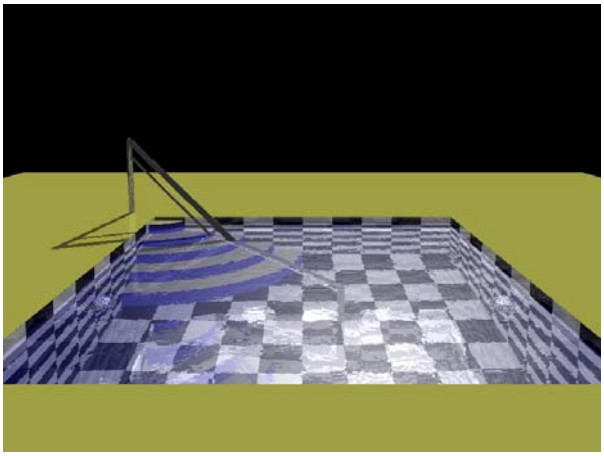
Milieu	Indice de Fresnel
Verre	1.6
Air	1
Eau	1.33

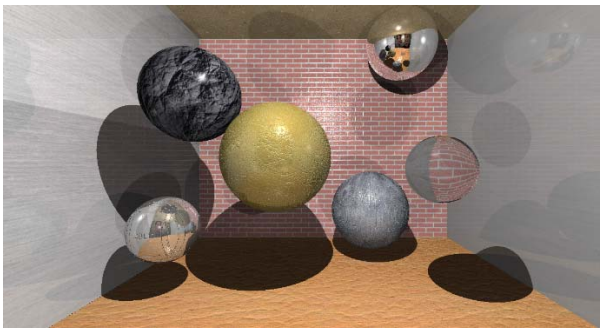
A la couleur de réflexion et la couleur de réfraction, on ajoute des coefficients ρ_1 et ρ_2 entre 0 et 1 de façon à indiquer si le matériau est pas (=0), peu, très ou totalement (=1) réfléchissant ou transparent.

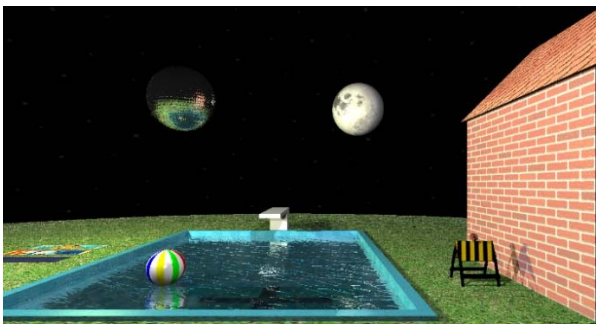
Le danger dans cette approche peut être l'explosion due à la récursivité. Si la scène contient une multitude d'objets transparents et brillants, le rayon initial peut générer 2 puis 4 puis 8 puis … trop de rayons. Pour éviter cela, on peut mettre une borne sur la profondeur maximale de la récursivité égale à 5 ou 6 ce qui est largement suffisant.

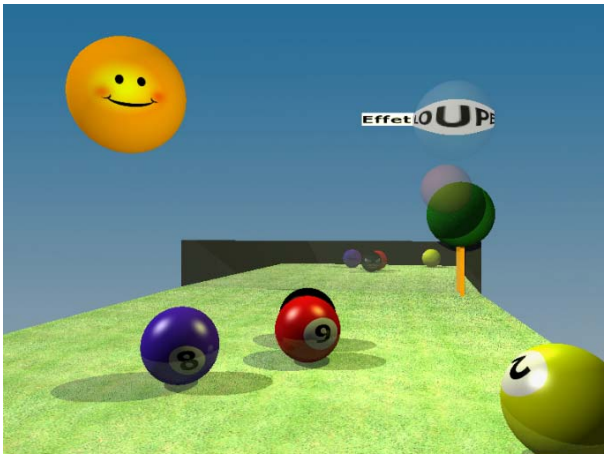
Pour rendre hommage aux générations d'ESIEEns s'étant illustrés dans le projet de synthèse d'image, voici quelques exemples des leurs créations :

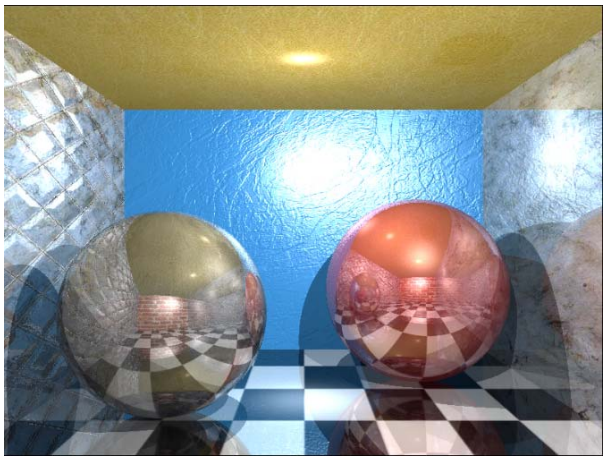


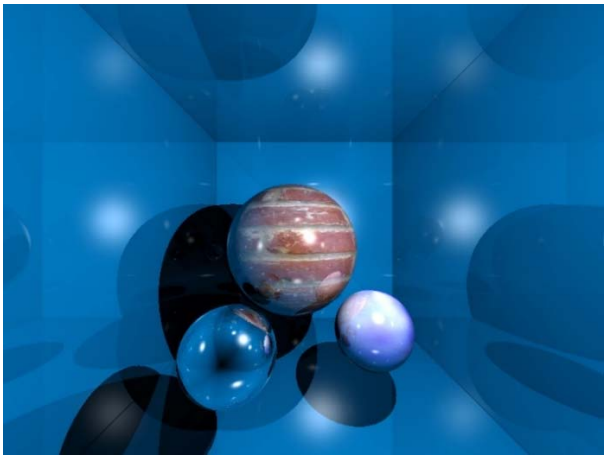


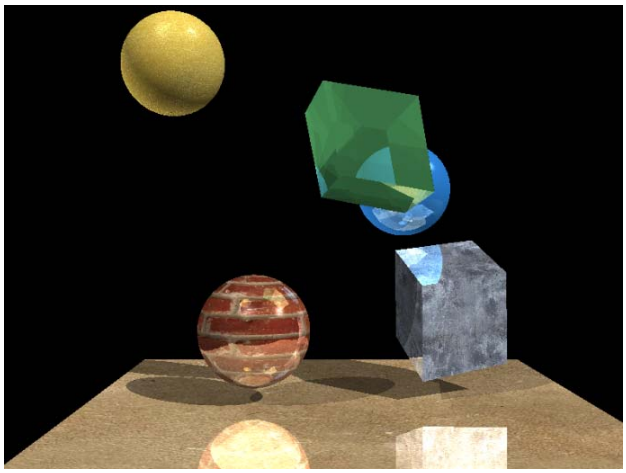


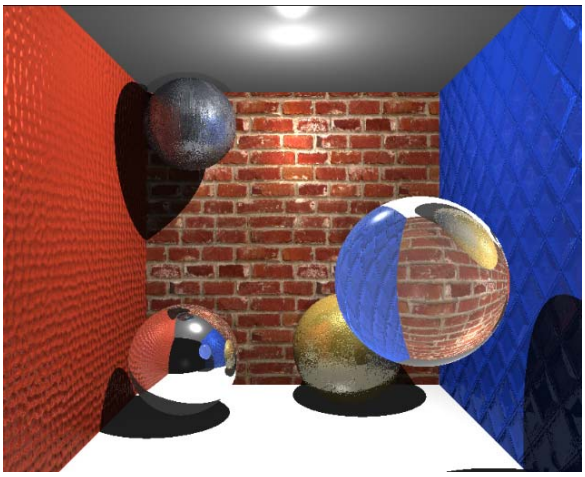






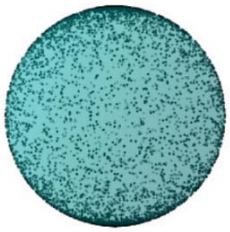




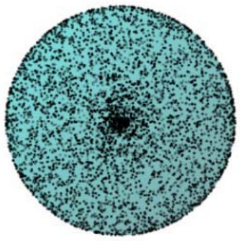


Génération aléatoire de directions

Pour les méthodes VPL/Pathtracer, vous devez tirer une direction de vecteur aléatoire. Pour cela, certains d'entre vous vont utiliser les coordonnées sphériques : tirage aléatoire des valeurs *u* et *v* et calcul d'une direction. Une telle approche conduit à une distribution incorrecte des vecteurs sur la sphère unitaire :

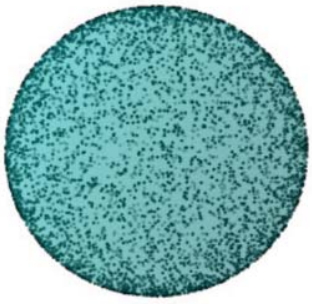


Vue de face

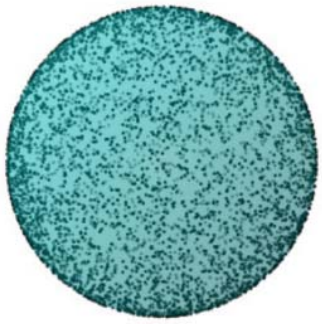


Vue de dessus

En effet, le problème se situe près des pôles. Si l'on prend la zone [0°,360°]x[85°,90°] (le pôle nord) comparée à la zone [0°,360°]x[0°,5°] (l'équateur), nous allons statistiquement générer la même quantité de vecteurs, cependant, sur une surface très petite (le pôle) comparée à une surface très grande autours de l'équateur. Il y aura donc une densité supérieure sur la région des pôles. Dans l'idée, nous aimerions avoir une densité uniforme de ce type :



Vue de face



Vue de dessus

Ce problème s'avère être un problème difficile des mathématiques. Il n'y a pas de méthode idéale. Cependant, au vue des besoins, il a fallu fournir des méthodes approchées permettant de construire une bonne approximation. Nous vous proposons de calculer un set de vecteurs aléatoires en début de programme à partir de la fonction suivante :

```
for(int i = 0 ; i < nb ; i++)
{
   double theta = 2*PI*rand(0.0,1.0);
   double phi = acos(2*rand(0.0,1.0)-1.0);
   x[i] = cos(theta)*sin(phi);
   y[i] = sin(theta)*sin(phi);
   z[i] = cos(phi);
}
```

Parallélisation des algorithmes de rendu

Les algorithmes VPL comme Pathtracer sont intrinsèquement fortement parallélisable. En effet, les données de scène ne varient pas durant le rendu et elles sont accédées uniquement en lecture, plusieurs threads peuvent donc y avoir accès sans avoir à gérer aucun aspect de concurrence.

Ensuite, le calcul de la couleur d'un pixel est totalement indépendant des autres pixels. Ainsi, il suffit de découper la zone de rendu et de distribuer ces zones à chaque thread. Chaque thread a alors la responsabilité de calculer sa zone de rendu. Une fois son travail terminé, chaque thread retourne un morceau de l'image. L'ensemble des différents morceaux calculés produit l'image résultat.

Une fois que vous avez écrit le moteur 3D, il est donc très facile de le multithreader. En effet, la double boucle en largeur et en hauteur du Raycasting peut se transformer facilement pour gérer une zone rectangulaire de l'image :

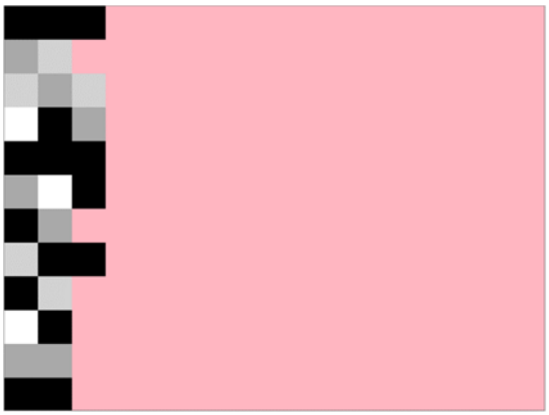
```
Raycasting() :
    Pour x allant de 0 à LargeurEcran
        Pour y allant de 0 à HauteurEcran
            Coul = Rayon passant par le pixel (x,y)
            Afficher pixel (x,y) avec la couleur Coul
```

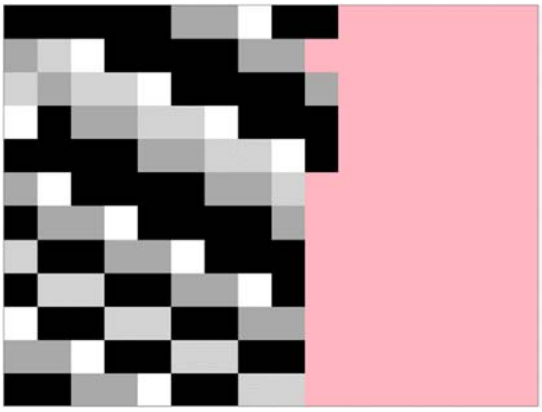
```
RaycastingMultiThread(xmin, xmax, ymin, ymax) :
    Pour x allant de xmin à xmax
        Pour y allant de ymin à ymax
            Coul = Rayon passant par le pixel (x,y)
            Afficher pixel (x,y) avec la couleur Coul
```

Traditionnellement, sur un CPU possédant 6 cœurs, il est judicieux de créer seulement 6 threads. Le système deviendra peu réactif car tous les cœurs seront occupés à 100%. Si vous voulez continuer à utiliser votre machine pendant les calculs, laissez un cœur de libre.

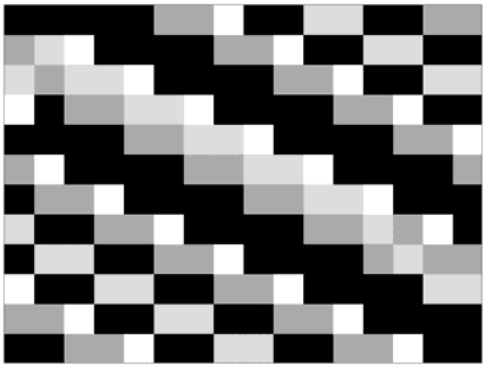
Comment découper la zone de rendu et la distribuer aux différents threads ? Pour 4 threads, la méthode la plus simple consisterait à découper l'image en 4 bandes verticales. Cependant, cette méthode a un désavantage certain. En effet, si trois des bandes de l'image sont calculées en 1 minute et que la 4^{ième} prend 5 minutes, alors 3 cœurs pendant les 4 minutes restantes ne seront pas utilisés. Pour remédier à cela, il est plus judicieux de découper l'image en petits carrés et de stocker ces zones dans une liste. Chaque thread à son démarrage va démarrer une boucle consistant à demander une zone et à la calculer. Une fois cette zone terminée, il l'envoie à l'affichage et traite une autre zone en attente dans la liste. Cette approche permet d'équilibrer la charge de calcul.

Nous avons simulé un exemple avec 4 threads, chaque thread dispose de sa propre couleur : noir, gris foncé, gris clair, blanc. Chaque thread remplit juste sa zone avec une couleur unie, pour différencier leurs performances, le thread noir est 4x plus rapide, le gris foncé 3x, le gris clair 2x et le noir est notre référence. Voici le rendu de notre simulation :





Les zones ont été créées de haut en bas et de gauche à droite. Les threads les traitent et les affichent donc dans cet ordre-là. Dans la colonne la plus à droite, des carrés sont présents alors que des carrés sont encore roses. Cela provient du fait que le thread noir est le plus rapide, ainsi lorsque le dernier carré noir est affiché, le thread blanc est encore en train de calculer sa zone et elle n'est pas encore affichée, d'où la présence d'un carré rose, signe du retard d'un autre thread.



Dans cet exemple, le thread noir était le plus rapide, il a donc calculé le plus de zone de rendu dans notre image finale. Le thread blanc était le plus lent et ainsi les zones blanches sont les moins nombreuses.

Lors de ce test, les threads ont quasiment fini leur travail en même temps. Le choix de la taille des carrés était donc optimal.

Multithread en C#

La mise en place du multithread dans les langages modernes (C++, Java, C#) a été grandement facilitée. En effet, il est possible aujourd'hui de créer un thread juste en utilisant un appel de fonction. Cela est certes très pratique, mais il ne faut pas oublier que cette fonction va s'exécuter dans un thread tiers.

Il y a quelques notions à connaître avant de se lancer dans la programmation multithread car nous sommes dans une application graphique/fenêtrée et des règles s'appliquent. Les différents composants (widgets) de la fenêtre appartiennent au thread principal, celui-ci gère d'ailleurs tous les évènements que reçoit la fenêtre : déplacement, agrandissement, click bouton… Les threads fils peuvent librement accéder aux ressources de la scène 3D car ils le font uniquement en lecture. Aucune précaution particulière n'est à prendre dans ce cas.

La liste des zones à dessiner, elle, est une ressource partagée, elle doit donc être synchronisée entre les différents threads enfants. Pour cela, C# fournit une classe très pratique **ConcurrentBag** représentant une liste gérant la synchronisation entre threads, nous utiliserons donc cette classe.

ConcurrentBag, doc et exemples : https://docs.microsoft.com/fr-fr/dotnet/api/system.collections.concurrent.concurrentbag-1

Chaque fois qu'un thread a traité une zone de l'image, il serait judicieux de l'afficher. Cependant, cela sous-entendrait qu'un thread fils modifie en écriture un objet appartenant au thread principal de la fenêtre et là c'est l'accident assuré. Pour résoudre ce problème, nous utilisons une solution très simple qui consiste à envoyer un évènement au thread principal qui sera donc exécuté par le thread de l'interface. Cette astuce se fait grâce à la méthode **Invoke**. Le thread fils se sert de cette méthode pour « retourner » un bitmap contenant le rendu de la zone traitée. Le système se charge alors de transférer l'objet au thread principal.

Faire un appel « safe call » vers le thread de la GUI grâce à la méthode Invoke : https://docs.microsoft.com/en-us/dotnet/desktop/winforms/controls/how-to-make-thread-safe-calls-to-windows-forms-controls?view=netframeworkdesktop-4.8

Un code complet, à titre d'exemple pour le multithread, est disponible pour vous aider. Demandez-le-moi si vous recherchez un modèle pour vous familiariser avec cette approche.

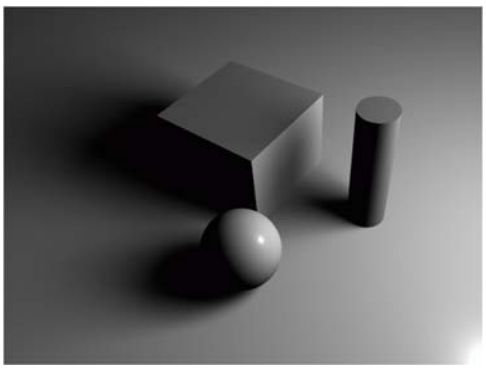
3. Effets graphiques

Mesh renderer

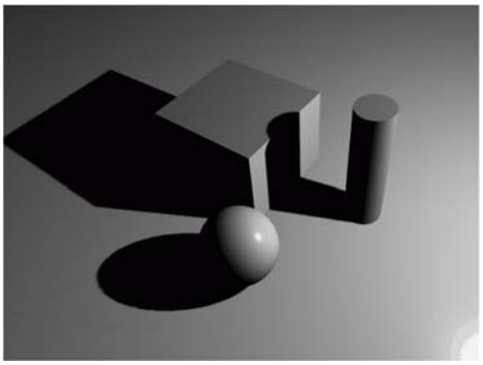
Vous pouvez intégrer dans votre projet un lecteur d'objets 3D. Nous vous conseillons fortement le format historique OBJ qui contient les géométries (triangles) et les coordonnées uv des textures. Il est standardisé et relativement simple et bien documenté. Il est cependant difficile de trouver des objets complets (mesh low poly + texture). Demandez de l'aide à l'intervenant.

Vous trouverez des informations suffisantes sur le format OBJ sur Wikipédia.

Soft Shadow



Soft Shadow



Hard Shadow

Le processus d'occultation que vous avez mis en place produit des ombres dures (hard shadow) : leur contour est franc et leur intérieur est noir. Ces ombres sont réalistes un jour d'été en plein soleil : elles correspondent à des ombres sous une lampe directionnelle de forte intensité. Mais ce n'est pas le cas des scènes d'intérieur.

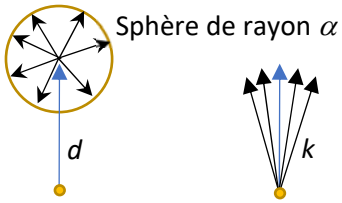
Pour simuler des ombres plus réalistes, il suffit lors du test d'occultation de faire tilter le vecteur de quelques degrés. En tirant plusieurs rayons, une dizaine par exemple, et en moyennant les résultats obtenus. On obtient des ombres et des transitions plus douces. C'est l'algorithme dit des soft shadows.

Comment faire tilter un vecteur autours de sa position d'origine ? Notons *d* le vecteur direction actuel (normalisé), prenons un rayon *r* tiré aléatoirement. Pour déplacer le vecteur *d*, nous allons calculer le nouveau rayon *k* :

$$k = d + \alpha . r \quad \text{où } \alpha \in [0,1[$$

Il suffit après ce calcul de renormaliser *k* si besoin.

Plus la valeur de α est grande plus le tilt est important, on peut prendre une valeur de 0.2 par exemple.



Surface de revolution

Nous construisons une surface en faisant tourner un profil *R(v)* autour de l'axe vertical.

$$Rev(u,v) = \begin{cases} x(u,v) = \mathrm{R}(v) \,.\cos\,(u) \\ y(u,v) = \mathrm{R}(v) \,.\sin\,(u) \\ z(u,v) = v \end{cases}$$

Nous pourrions travailler comme pour la sphère, mais l'étape d'inversion des coordonnées sera difficile à résoudre. Le plus simple consiste à échantilloner cette surface en créant une collection de rectangles

Un beau vase dans le projet !

Si vous fixez bien les détails, vous remarquerez les facettes sous forme de quad à la surface du vase.