

01/04/2017

Cours et Projet de Synthèse d'image



Lilian BUZER

1. LA COULEUR

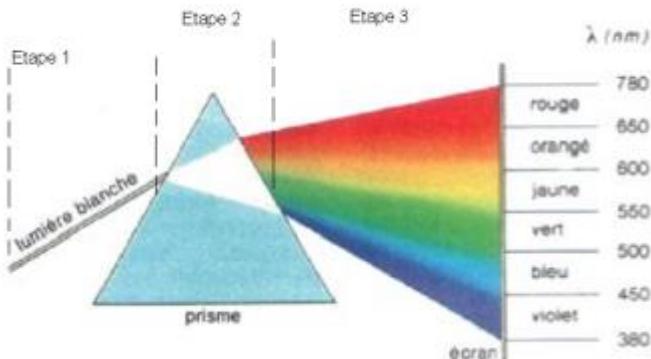
Définition physique

Couleur : sensation qui résulte de l'interaction de la lumière visible avec les cellules de notre œil.

Nous insistons sur le terme « sensation » car cela souligne que cette perception relève d'un sens et elle est ainsi subjective. Cela sous-entend que la même scène perçue par deux personnes différentes sera différente pour chacun. De même entre les deux yeux d'une même personne, la perception est différente, un œil pourra être plus sensible dans les bleus et l'autre dans les verts. Ainsi, même si l'on oublie les défauts optiques, l'œil reste un capteur non étalonné / standardisé faisant varier la perception du monde réel d'une personne à l'autre.

Lumière : on définit la lumière comme une grandeur physique. Elle est décrite par l'intensité de chaque composante spectrale.

Historique : en 1666 Newton fait ses premières expériences sur la diffusion de la lumière à travers un prisme. En faisant traverser un fin rayon de soleil à travers un prisme, le rayon se décompose en une suite de couleur correspondant aux composantes monochromatiques.



Chaque rayon de lumière sortant du prisme est monochromatique. Il s'agit d'une lumière « pure » : non issue d'un mélange de plusieurs couleurs monochromatiques.



Remarquez que les couleurs de l'arc en ciel reproduisent exactement le même effet que le prisme de Newton, il s'agit bien d'un phénomène de décomposition de la lumière en composante.

Quizz : quelles couleurs ne sont pas présentes dans l'arc en ciel ?

Chaque couleur monochromatique est identifiée par sa longueur d'onde. L'œil humain perçoit de la longueur d'onde 380nm (violet) à 750nm (rouge). Au-delà de cette plage, l'œil n'est plus sensible à ces lumières, on parle ici des infra-rouges et des ultra-violets.

L'œil

L'œil est un capteur composé de cônes (~ 7 millions) spécialisés dans la perception de la couleur et de bâtonnets (~ 100 millions) spécialisés dans la perception de l'intensité de la lumière.

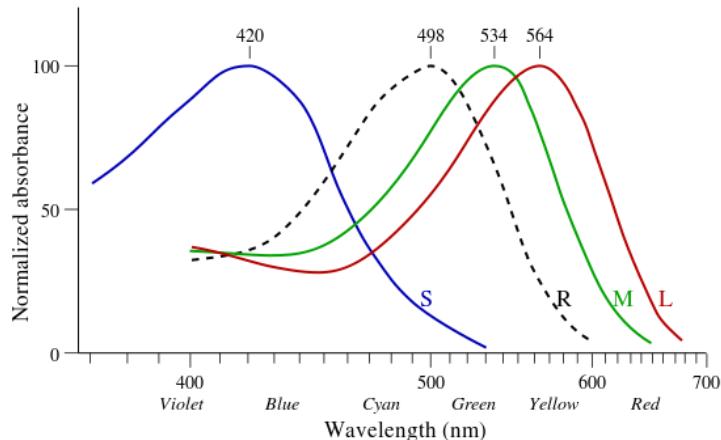
Quizz :

- Pourquoi la nuit tous les chats sont gris ?
- Pourquoi dit-on que les photos couleurs ont tendance à être perçues floues comparées à une version N&B ?
- Pourquoi les étoiles ont des couleurs et pas la voie lactée ?

Réponses :

- La nuit, la lumière est trop faible pour produire une excitation nerveuse des cônes, ces capteurs restent donc inactivés. Les bâtonnets eux, 100 fois plus sensibles, sont encore actifs au prix d'une vision en teinte de gris.
- Il n'y a pas photo ! Avec une densité 10 fois supérieures, les bâtonnets fournissent une image en 100 Mégapixels ! Avec une densité de 7 Mégapixels, les cônes fournissent une qualité d'image d'un APN des années 2010. Aujourd'hui l'APN le plus avancé arrive à 80MPix.
- Les étoiles les plus brillantes, peuvent être vues colorées car suffisamment brillantes pour exciter les cônes. Cependant, la voie lactée, une sorte de nuage diffus formé par les étoiles les plus éloignées de notre galaxie n'est pas assez brillante pour apporter une couleur, elle reste d'un teint grisâtre.

La perception de la lumière depuis notre œil est partielle. En fait, il existe trois types de cônes chacun spécialisé dans une gamme de couleur : les bleus, les verts et les rouges. Les bleus par exemple ont une sensibilité maximale autour de 420nm, une sensibilité deux fois plus faible à 380nm et à 460nm et quasiment plus rien au-delà de 530 nm.



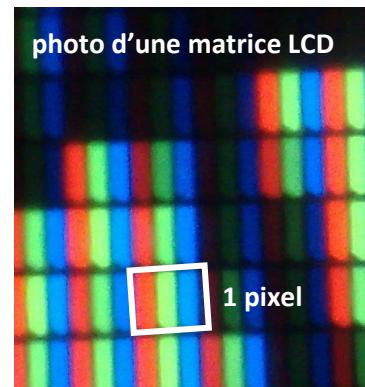
Ainsi, un œil convertit un rayon de lumière en trois signaux électriques. Voici un exemple des signaux transmis au cerveau :

Longueur d'onde	Cônes bleues	Cônes verts	Cônes rouges
420	100%	43%	40%

534	1%	100%	84%
564	0%	70%	100%

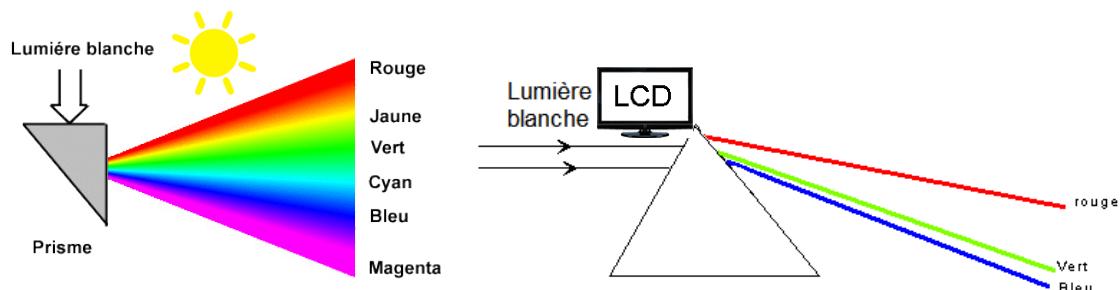
Le cerveau ne connaît jamais le spectre complet d'une lumière, il déduit « une » couleur à partir des informations électriques qu'il reçoit depuis les cônes RVB. Par exemple, si le cerveau reçoit une lumière monochromatique produisant une excitation nerveuse correspondant à la moyenne des excitations nerveuses des longueurs d'onde 534 et 564 : [0% 85% 92%], il en déduit que la longueur d'onde reçue se trouve au milieu de 534nm et 564nm soit 549nm.

On peut ainsi leurrer l'œil et le cerveau. C'est le principe des écrans LCD que nous connaissons. En effet, pour reproduire des couleurs, l'écran se compose de multitudes de cellules (pixels) composées de 3 lampes monochromatiques. Il est uniquement possible de régler l'intensité de chaque lampe et il est impossible de faire varier leur couleur. Cependant, ce système est suffisant pour leurrer l'œil. En effet, pour reproduire une couleur on choisit d'éclairer plus ou moins un triplé de lampe R,V,B qui produit la même excitation nerveuse au niveau des cônes que la couleur souhaitée. De cette façon l'œil voit la couleur choisie alors qu' « elle n'existe pas ».



Deux lumières sont dites *homochromes* lorsque leurs spectres physiques sont différents mais que la vision humaine ne les différencie pas.

Prenons un exemple simple, lorsque nous prenons la lumière blanche du soleil, le spectre du visible est complet et le passage dans un prisme décompose ce blanc en un éventail de couleur monochromatique correspondant aux couleurs de l'arc en ciel. Prenons un écran LCD et générions du blanc, cette lumière blanche est généralement calibrée pour être homochrome à celle du soleil. Pour notre œil, aucune différence, cependant, si nous faisons passer cette lumière blanche provenant du LCD à travers un prisme, nous aurons cependant une décomposition en 3 raies monochromatiques uniques et non un spectre complet comme celui du soleil.



Synthèse additive et soustractive

Il y a deux façons de modéliser le mélange des couleurs : la synthèse additive et soustractive.

Synthèse additive : elle correspond à la superposition de deux sources lumineuses. Dans cette approche, les effets se cumulent car la lumière est considérée comme de l'énergie lumineuse que l'on cumule. Il s'agit par exemple de l'écran LCD avec ses trois lampes monochromatiques :



La superposition du Rouge et du Vert produit du Jaune, du Rouge et du Bleu produit du Magenta et du Vert et du Bleu du Cyan. La superposition des trois composantes de base crée du blanc.



Synthèse soustractive : ce modèle correspond à l'absorption de la lumière. Il vous est familier car il correspond aux tubes de peinture de l'école primaire avec les trois primaires Cyan Magenta Jaune ou aux cartouches d'encre des imprimantes couleurs (CMYK = cyan magenta yellow black). Comme vous le savez, la superposition des pigments des trois primaires n'est pas idéale et donne du marron... C'est ainsi que l'on rajoute la cartouche de noir dans une imprimante afin d'avoir un noir parfait.

Nous nous intéressons nous plutôt à la couleur d'un objet. Ainsi, pourquoi un objet est bleu ? Parce qu'il émet du bleu ? Pas exactement, un objet est visible car il renvoie la lumière des sources lumineuses (lampes, soleil, bougie), il ne produit pas de lumière directement. Ainsi, un objet est bleu, car positionné sous une lumière blanche, il absorbe la composante rouge et verte. La logique consiste ici à se préoccuper de ce qui est absorbé (soustrait).

Exercices :

Couleur de la lampe	Couleur de l'objet (sous une lumière blanche)	Couleur de l'objet vue de l'extérieur
Blanc	Blanc	Blanc
Cyan	Blanc	Cyan
Blanc	Magenta	Magenta
Cyan	Bleu	Bleu
Cyan	Jaune	Vert
Rouge	Magenta	Rouge
Jaune	Magenta	Rouge
Bleu	Vert	Noir
Cyan	Rouge	Noir

Loi de Grassmann

Les lois de Grassmann sont des lois régissant la superposition, l'addition, des couleurs en colorimétrie. Elles sont à la base de tout calcul colorimétrique. Les études menées par Hermann Grassmann au XIXe siècle sur la perception humaine des couleurs l'ont mené à énoncer 3 lois en 1853. Ces lois expriment le principe de la trichromie : une sensation colorée peut être égalisée par un mélange additif de trois couleurs primaires convenablement choisies.

Première loi : Trivariance visuelle

$$C \cong r \times [R] + v \times [V] + b \times [B]$$

Toute sensation colorée peut être reproduite par un mélange additif de trois couleurs primaires convenablement choisies. Le symbole \cong correspond à une égalité suivant le principe de l'homochromie. Ainsi deux lumières $C_1 \cong C_2$ peuvent être perçues de façon identique tout en ayant des compositions spectrales différentes. Dans la formule : r, v et b correspondent à des coefficients appelés *composantes* et sont positifs dans le système $[R], [V], [B]$.

Deuxième loi : Additivité des couleurs

La sensation colorée provoquée par un mélange additif de deux ou plusieurs lumières colorées est égal à la somme des composantes de chacune des lumières. Ainsi, si nous avons : $C_1 \cong r_1 \times [R] + v_1 \times [V] + b_1 \times [B]$ et $C_2 \cong r_2 \times [R] + v_2 \times [V] + b_2 \times [B]$ alors la superposition de ces deux lumières est égale à l'addition de leurs composantes :

$$C_1 + C_2 \cong (r_1 + r_2) \times [R] + (v_1 + v_2) \times [V] + (b_1 + b_2) \times [B]$$

Troisième loi : Linéarité des couleurs

Si une lumière colorée baisse ou augmente en intensité, il faut, pour l'égaliser, modifier les trois primaires dans les mêmes proportions. Ainsi, si nous avons : $C \cong r \times [R] + v \times [V] + b \times [B]$ et k un réel :

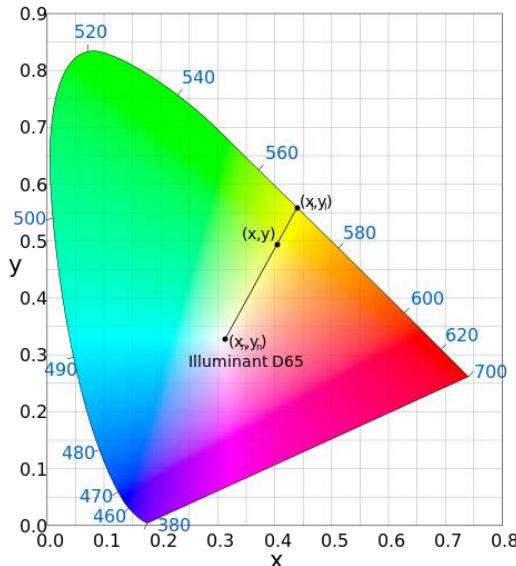
$$k.C \cong k.r \times [R] + k.v \times [V] + k.b \times [B]$$

Diagramme de chromaticité

Présenter pour culture générale, chapitre non utile au projet.

On suppose que la couleur est une propriété indépendante de la luminosité et on l'appelle **chromaticité**. Ainsi, un rouge clair et un rouge foncé représentent la même chromaticité : le rouge et la luminosité devient un paramètre annexe. Nous avons vu précédemment que nous pouvions représenter toute couleur comme une combinaison de trois composantes. Nous choisissons donc ici une composante correspondant à la luminosité. Il nous faut alors deux composantes pour décrire la chromaticité et cela peut se faire sur une feuille : espace à deux dimensions !

En colorimétrie, un diagramme de chromaticité est une représentation cartésienne où les points représentent la chromaticité des stimuli de couleur.



Définition de l'espace CIE_xyY

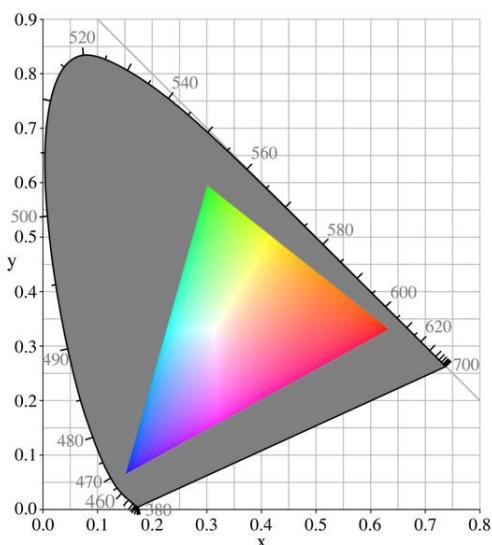
La zone intérieure représente toutes les chromaticités perceptibles. Le bord supérieur, en forme de fer à cheval, représente l'ensemble des couleurs monochromatiques du violet au rouge. C'est pourquoi le bord supérieur est gradué selon la longueur d'onde de 380 nm à 700 nm.

Si deux points représentent chacun une couleur, le segment qui les joint représente les mélanges de ces deux couleurs en proportions variables.

Le point de coordonnées $x = y = 1/3$ représente le blanc D65 utilisé comme référence.

Le segment qui relie le point blanc à celui d'une couleur pure représente toutes les couleurs possédant une même teinte plus ou moins lavées de blanc. Elles ont toutes la même longueur d'onde dominante.

En synthèse des couleurs, synthèse additive ou synthèse soustractive, le gamut, ou gamut de couleur est la partie de l'ensemble des couleurs qu'un certain type de matériel permet de reproduire. Il peut notamment s'agir d'un appareil photographique, d'un scanner informatique, d'un écran d'ordinateur, d'un rétroprojecteur, d'un vidéoprojecteur, d'un procédé d'imprimerie ou d'une imprimante. Ainsi, le gamut donne l'ensemble des couleurs possibles. Pour un système d'affichage par synthèse additive, tel qu'un écran d'ordinateur, le triangle formé par les trois points correspondant aux trois couleurs primaires représente le gamut du système. Seules les couleurs représentées par un point situé à l'intérieur du triangle peuvent être reproduites.



Un gamut typique d'écran LCD : le gamut correspondant à un écran d'ordinateur ou un téléviseur ne couvre pas la totalité de l'espace des couleurs visibles. Le gamut d'une imprimante est généralement encore plus réduit que celui d'un écran.

Mise en place informatique

Nous considérons que les composantes R,V et B d'une couleur sont comprises entre 0 et 1. La valeur 0 correspond à l'absence de lumière. La valeur 1 correspond à l'intensité maximale que peut traduire l'œil. L'œil étant un capteur, comme tout capteur, il sature au-delà d'une certaine valeur. Ainsi la couleur [1 1 1] est le blanc de luminosité maximale perceptible par l'œil. S'il y avait plus de lumière, un phénomène d'écrêtage se produit et la valeur du capteur ne dépasse pas 1.

Si un objet éclairé par une lampe L_1 est de couleur C_1 et si le même objet éclairé par une lampe L_2 est de couleur C_2 alors cet objet éclairé par les lampes L_1 et L_2 en même temps sera perçu avec une couleur correspondant à C_1+C_2 suivant le principe d'additivité de la lumière. Ainsi pour connaître la couleur d'un objet dans une scène, on calcule sa couleur en allumant chaque lampe indépendamment et la couleur finale correspond à la somme des couleurs générées par chaque lampe.

$$\text{Couleur OBJET dans la scène} = \sum_{L \text{ lampe de la scène}} \text{Couleur de l'objet éclairé uniquement par } L$$

Nous rappelons qu'un objet de couleur bleue est un objet qui absorbe le rouge et le vert. Si l'on regarde les composantes du bleu [0 0 1], on s'aperçoit qu'elles s'interprètent comme un filtre. Ainsi 0 correspond à je ne laisse pas passer de lumière et 1 à je laisse passer 100% de la lumière. Ainsi lorsqu'une lampe de couleur $[a b c]$ éclaire un objet de couleur $[x y z]$, l'objet est perçu dans la scène comme ayant la couleur :

$$\text{Lampe } \begin{bmatrix} a \\ b \\ c \end{bmatrix} \text{ éclairant un objet } \begin{bmatrix} x \\ y \\ z \end{bmatrix} \gg \text{une couleur } \begin{bmatrix} a \times x \\ b \times y \\ c \times z \end{bmatrix}$$

Nous retrouvons nos précédents résultats :

$$\text{Lampe Magenta } \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \text{ éclairant un objet jaune } \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \gg \text{du rouge } \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

La classe Couleur : Elle est mise à votre disposition dans le projet. Elle inclut l'opérateur * qui gère la multiplication des composantes, membre à membre, comme ci-dessus. Les opérateurs multiplication et division par une constante sont présents ainsi que l'addition de deux couleurs. La fonction d'écrêtage est appelée automatiquement lorsque vous traduisez une couleur en couleur écran Windows pour effectuer un affichage à l'écran.

Exercice :

Calculez la couleur perçue d'un objet de couleur [20% 50% 40%] éclairé par deux lampes [80% 60% 40%] et [40% 40% 40%]

$$\begin{bmatrix} 0.2 \\ 0.5 \\ 0.4 \end{bmatrix} \times \begin{bmatrix} 0.8 \\ 0.6 \\ 0.4 \end{bmatrix} + \begin{bmatrix} 0.2 \\ 0.5 \\ 0.4 \end{bmatrix} \times \begin{bmatrix} 0.4 \\ 0.4 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.16 \\ 0.3 \\ 0.16 \end{bmatrix} + \begin{bmatrix} 0.08 \\ 0.2 \\ 0.16 \end{bmatrix} \rightarrow \begin{bmatrix} 0.24 \\ 0.5 \\ 0.32 \end{bmatrix}$$

2. Sources fournies

Classe Couleur

Paramètres

R – V – B (valeurs comprises entre 0 et 1)

Constructeurs

Couleur(float R, float V, float B)

Couleur(Couleur c)

Opérateurs

Couleur operator + (Couleur a, Couleur b)

Couleur operator - (Couleur a, Couleur b)

Couleur operator - (Couleur a)

Couleur operator * (Couleur a, Couleur b)

Couleur operator * (float a , Couleur b)

Couleur operator * (Couleur b, float a)

Couleur operator / (Couleur b, float a)

Classe BitmapEcran

Méthodes statiques

DrawPixel(int x, int y, Couleur c) // dessine un pixel avec la couleur donnée

int GetWidth() // récupère la largeur de la zone d'affichage

int GetHeight() // récupère la hauteur de la zone d'affichage

Classe Texture

Constructeur

Texture(string f) // construit la texture à partir du fichier image en paramètres

Méthodes

Couleur LireCouleur(float u, float v) // couleur aux coordonnées $(u,v) \in [0,1] \times [0,1]$

Bump(float u, float v, out float dhdu, out float dhdv) // coefs de BUMP dh/du dh/dv

Struct V3

Constructeurs

```
V3(V3 t)  
V3(float _x, float _y, float _z)  
V3(int _x, int _y, int _z)
```

Paramètres

```
float x,y,z;           // coordonnées du vecteur
```

Méthodes

```
float Norm()          // retourne la norme du vecteur  
float Norme2()        // retourne la norme^2 du vecteur  
void Normalize()      // normalise le vecteur
```

Opérateurs

```
V3 operator +(V3 a, V3 b)  
V3 operator -(V3 a, V3 b)  
V3 operator *(float a, V3 b)  
V3 operator *(V3 b, float a)  
V3 operator /(V3 b, float a)  
V3 operator ^ (V3 a, V3 b)    // produit vectoriel  
float operator * (V3 a,V3 b) // produit scalaire
```

ProjetEleve

Méthode statique

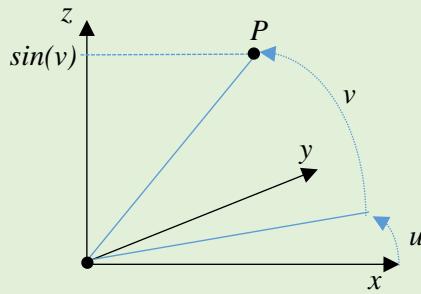
```
Go()    // point d'entrée de votre projet, c'est là que commence votre travail !!!
```

Cette fonction est remplie d'un code exemple sans utilité pour le projet mais qui a le mérite d'afficher un dessin à l'écran. Un écran noir... c'est si déprimant pour commencer un projet 😊

3. Les sphères

Coordonnées sphériques : Il vous ait demandé d'utiliser le repère donné pour le projet, l'axe X désigne l'axe horizontal orienté vers la droite. L'axe Y désigne la profondeur orientée positivement vers le fond et l'axe Z désigne l'axe vertical orienté vers le haut.

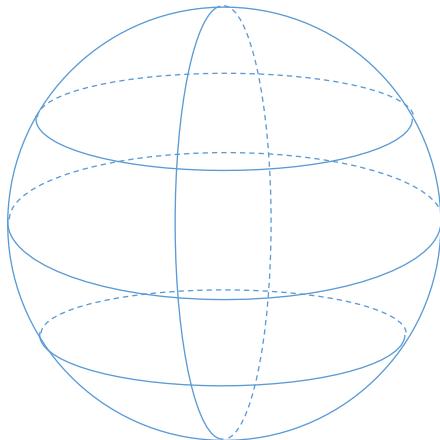
Nous utilisons dans ce projet une modélisation paramétrique des surfaces. Pour les sphères, nous vous imposons de travailler avec les formules suivantes et de les reprendre telles quelles :



Pour positionner un point P sur une sphère de rayon 1, nous pouvons le faire à partir de deux angles u et v . Prenons une métaphore à partir d'un canon d'artillerie. Pour positionner le tir, on fait d'abord tourner le canon sur un axe vertical pour qu'il pointe vers sa cible. L'angle u correspond à l'orientation du canon. Ensuite ; les canonniers élèvent l'extrémité du canon pour régler la distance de tir. L'angle v correspond à cet angle d'élévation. Les deux rotations peuvent s'effectuer dans n'importe quel ordre, au final le canon sera placé de la même manière.

$$S(u, v) = \begin{cases} x(u, v) = \cos(v) \cdot \cos(u) \\ y(u, v) = \cos(v) \cdot \sin(u) \\ z(u, v) = \sin(v) \end{cases}$$

Exercice : Placez les points suivants sur la surface de la sphère unitaire



$u \setminus v$	$\pi/2$	$\pi/4$	0	$-\pi/4$	$-\pi/2$
0	A	B	C	D	E
$\pi/2$	F	G	H	I	J
π	K	L	M	N	O
$3\pi/4$	P	Q	R	S	R

Espace de définition

Pour décrire la sphère entière, nous nous proposons d'utiliser comme espace de définition :

$$(u, v) \in [0, 2\pi] \times [-\pi/2, \pi/2]$$

Lorsque v varie, P décrit un demi-cercle dans le plan Oxz . Lorsque u varie sur $[0, 2\pi]$, ce demi-cercle effectue un tour complet autours de l'axe z ce qui permet à P de décrire entièrement la surface de la sphère. Il vous est demandé d'utiliser obligatoirement cette espace de définition.

Position et taille

Pour décrire une sphère quelconque dans une scène, il faut ajouter des paramètres supplémentaires : le rayon R et les coordonnées du centre C . Ainsi, nous obtenons pour une sphère $M(u,v)$ la formulation suivante :

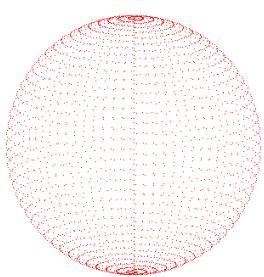
$$M(u, v) = R \times S(u, v) + C$$

Construction d'une surface par échantillonnage

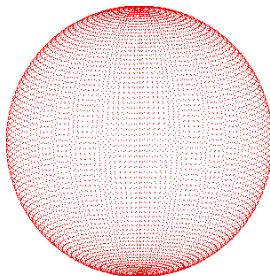
Afin de développer un moteur de rendu ainsi, nous pouvons modéliser des surfaces parfaites grâce aux surfaces paramétriques. Ainsi pour une sphère, nous allons échantillonner sa surface à partir d'un pas fixé. Ce pas sera choisi approximativement afin d'obtenir une bonne performance et un bon rendu.

```
for(float u = 0 ; u ≤ 2π ; u += pas)
    for (float v = -π/2 ; v ≤ π/2 ; v += pas)
        P = S(u, v);
        Desinnez P;
```

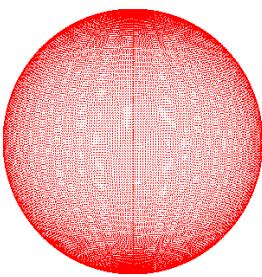
Voici des exemples obtenus pour une sphère de 150 pixels de diamètre :



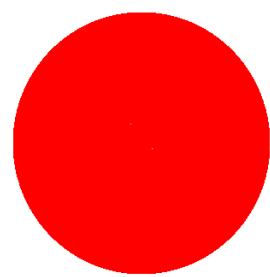
pas de 0.08



pas de 0.04



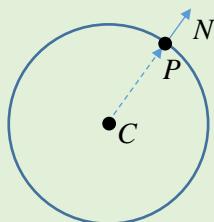
pas de 0.02



pas de 1/R

Normale en un point

Il est nécessaire de calculer la normale en un point de la surface d'un objet afin d'effectuer les calculs d'illumination. Pour la sphère, nous sommes dans une configuration particulière où il est inutile de se lancer dans des calculs complexes (malgré les essais répétés des élèves chaque année). Il suffit de remarquer que pour un cercle ou une sphère, la normale est orientée suivant la direction du rayon joignant le centre et le point courant.

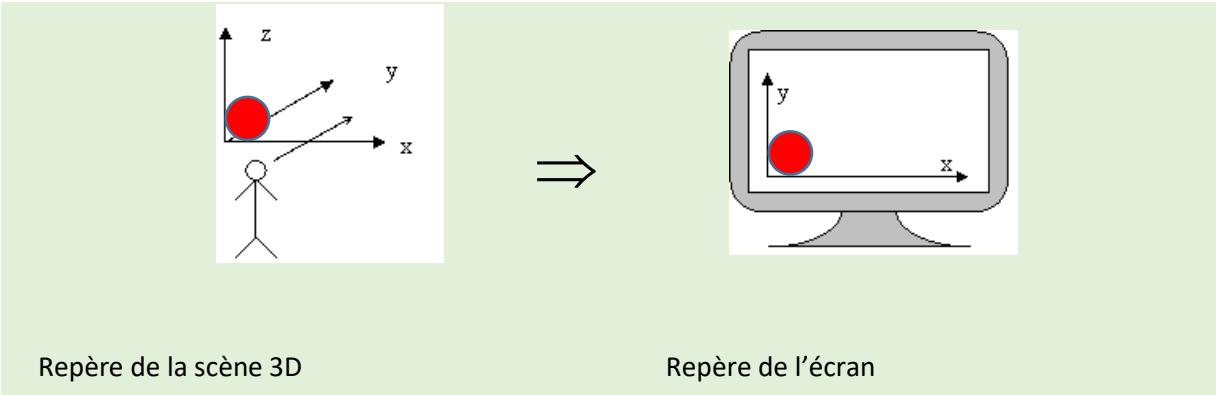


$$\text{Normale}(u, v) = S(u, v)$$

Ainsi, quelle que soit la valeur du Rayon, l'expression de la normale n'en tient pas compte.

4. L'écran et la scène

Pour certaines raisons, nous resteront discrets, nous cherchons à construire notre moteur de rendu sans utiliser un seul calcul basé sur les matrices ou sur des transformations complexes. Cela reste possible sans nuire à la qualité du rendu final.



Pour commencer nous allons utiliser la projection la plus simple qui soit : la projection orthogonale. Ainsi, nous n'avons pas de gestion de l'effet de perspective. Un objet qui s'éloigne apparaît aussi large que lorsqu'il est en devant de scène. De cette façon, les coordonnées de l'axe horizontal et de l'axe vertical de la scène 3D vont s'injecter dans les coordonnées des axes de l'écran. A noter que l'origine de la scène 3D correspond à l'origine du repère écran, c'est-à-dire le point (0,0) en bas à gauche.

La coordonnées Y de profondeur n'est pas utilisée durant la projection. Nous verrons comment construire l'effet de perspective/profondeur lorsque nous mettrons en place le Raycasting.

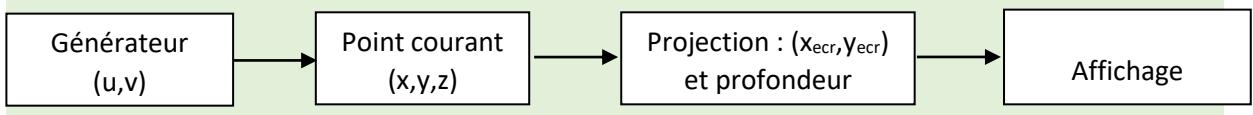
Il est bon de rappeler que les coordonnées de la scène 3D sont des nombres à virgule et que les coordonnées de l'écran sont des entiers. Il serait de mauvais goût que les types se mélangent.

Mise en place informatique

Unités : afin d'éviter toute conversion et de simplifier la gestion des échelles, l'unité choisie sera le pixel. Ce choix vous permet de dimensionner les objets dans votre scène exactement comme si vous les voyez à l'écran : une sphère de rayon 100 est donc dessinée comme une sphère de rayon 100 pixels à l'écran.

La fonction `DrawPixel(x,y,coul)` de la classe `BitmapEcran` vous permet de dessiner sur l'écran de l'application.

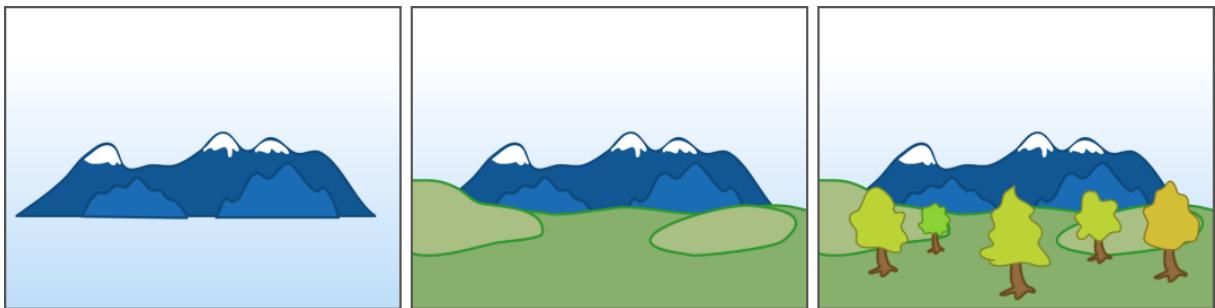
Dessinez votre première sphère à l'écran de rayon 200 pixels et faite en sorte qu'elle soit entièrement visible à l'écran :



5. Z-BUFFER

Historique

Dès que vous êtes dans un jeu, il faut gérer plusieurs objets dans la scène et se pose alors immédiatement la question de « qui est affichée avant ? ». Dans les tout premiers jeux, la réponse était presque automatique car il était possible d'afficher les objets du fond vers l'avant. Cette méthode s'appelle « l'algorithme du peintre », métaphore du peintre qui généralement dessinait d'abord le fond (ciel, montagnes), puis le sol, la végétation et en dernier les personnages au premier plan :



Quelques jeux célèbres ayant poussé au maximum les capacités de cet algorithme :



Megadrive – Bomber Raid - 1988

Le classique de la vue par-dessus. Un petit avion bombarde tout ce qui se trouve sous son passage. Ici, une version encore plus simplifiée de l'algorithme du peintre : les layers. Nous avons un modèle 4 couches : le fond (mer et sable), le décors au sol (arbres, tanks), les objets volants (avions, tirs) et le HUD (head up display) comportant le score, la vie et les bonus...

ATARI ST – Checkmate 1990

Le jeu d'échec utilise un algorithme un peu plus compliqué qui nécessite la mise en place d'un ordre d'affichage entre les différents objets. Ici les pièces sont affichées tout simplement de gauche à droite et du fond vers l'avant. Cela garantit que les pièces les plus proches seront affichées en dernier.





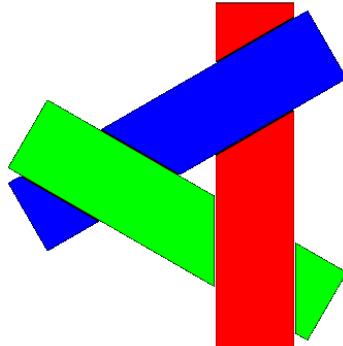
Operation Wolf – Taito – Borne d'arcade 1987

Inutile de présenter l'objectif du jeu. Dans l'ordre d'affichage : les montages et le sol (pour le fond), les bâtiments, dont les ouvertures peuvent contenir des personnes, ensuite les personnes appartenant au plan moyen (char, soldats) et pour finir les personnes les plus proches qui occupent la moitié basse de l'écran. Une bonne vingtaine de plan en tout pour rendre l'effet le plus poussé possible et des personnages dont la taille varie en fonction de l'éloignement. On peut difficilement faire plus avancé

pour cet algorithme, il faudra passer ensuite à la vrai 3D.

L'ordonnancement

Dans l'algorithme du peintre, il faut construire un ordre d'affichage (=ordonnancer en informatique) des différents objets pour respecter la profondeur. Il semble possible de pouvoir construire cet ordonnancement dans une scène composée de triangles. Pourtant, des configurations pathologiques peuvent apparaître. Cependant, dans le dessin de droite, nous avons trois surfaces qui empêchent la construction de tout ordonnancement. En effet, la surface rouge est devant la verte et la verte est devant la bleue. La bleue doit donc être affichée en premier et la rouge en dernier. Mais si l'on fait cela, la rouge sera devant la bleue et ce n'est pas le cas.



Bref, pour 1000 triangles, on peut espérer s'en sortir et ne pas avoir de malchance, mais dans une scène à 10 000 000 de triangles, l'ordonnancement devient vite complexe et source d'erreur d'affichage. La technologie qui suivit (le Z-Buffer) supplanta la méthode du peintre et permit d'éviter de se confronter à la construction d'un ordonnancement.

La transparence

Le Z-buffer permet-il de se passer de l'ordonnancement ? Non, il existe des configurations où il faut reconstruire un ordre d'affichage, c'est notamment le cas lorsque des objets transparents sont présents dans la scène. Fut une époque où la présence d'une seule face transparente désactiver l'utilisation du Z-Buffer dans le driver OpenGL... fichre.

La transparence se caractérise par une constante α qui traduite la quantité de lumière que laisse passer la surface. Ainsi nous obtenons :

$$\text{Couleur} = (1 - \alpha) \times \text{Couleur de l'objet} + \alpha \times \text{Couleur de l'objet derrière}$$

Jusque-là, rien de complexe, tout devient plus difficile lorsque plusieurs objets transparents s'empilent les uns derrière les autres :

$$Coul\ Finale = (1 - \alpha_1) \times Coul\ Obj1 + \alpha_1 \times Coul\ Obj2 \quad Obj2\ derrière\ Obj1$$

$$Coul\ Obj\ 2 = (1 - \alpha_2) Coul\ Obj\ 2 + \alpha_2 \times Coul\ Obj\ 3 \quad Obj3\ derrière\ Obj2$$

Ce qui donne :

$$Coul\ Finale = (1 - \alpha_1) \times Coul\ obj\ 1 + \alpha_1 \times (1 - \alpha_2) Coul\ Obj\ 2 + \alpha_1 \times \alpha_2 \times Coul\ Obj\ 3$$

La couleur devant être rendue à l'écran dépend de la couleur des trois objets et pour appliquer correctement les différents coefficients, il faut connaître leur ordre d'empilement. Alors, cela sous-entend que dès qu'une vitre, un verre est présent dans la scène, on ne peut appliquer le Z-buffer ? Surtout pas, les performances graphiques seraient inexistantes. Il faut tout simplement utiliser un système en deux passes :

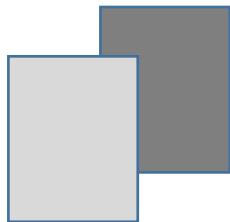
Passe 1 : Z-buffer appliqué sur les objets opaques (soit 99.9% des faces) – rendu des objets opaques.

Passe 2 : Pour les faces restantes, les transparentes, leur nombre est faible 1 à 100. Construire un ordonnancement et afficher les faces visibles au-dessus des faces opaques.

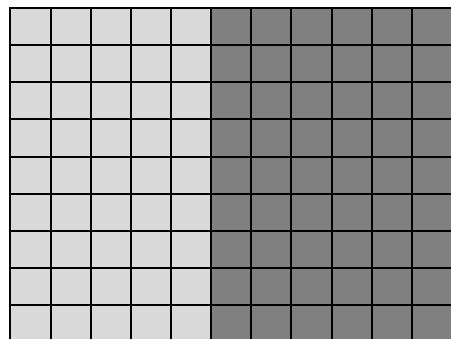
Algorithme du Z-Buffer

Un Z-Buffer est un tableau 2 dimensions. Chaque valeur dans le tableau est associée à un pixel de la zone d'affichage. Pour chaque pixel allumé à la position (x_ecran,y_ecran), on va stocker la composante en y (la profondeur) dans ZBuffer[x_ecran][y_ecran].

Lorsque je dessine un nouveau point dont les coordonnées écran sont identiques, je dois alors comparer les profondeurs. Si le nouveau point est devant (plus proche), il est alors affiché et la profondeur correspondante dans le ZBuffer est mise à jour. Dans le cas contraire, nous savons que le point associé à la valeur dans le Z-Buffer se trouve devant l'actuel, le point courant est donc masqué par celui-ci, il peut être oublié.



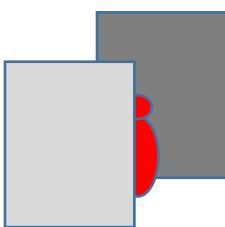
Scène 3D contenant
contenant deux murs



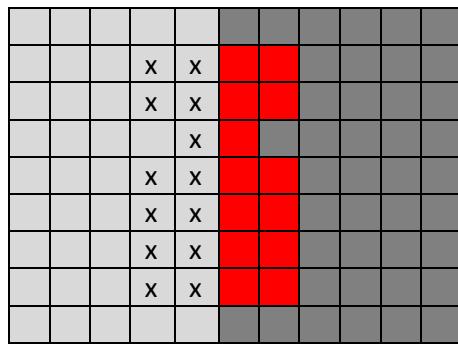
Affichage Ecran

1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9
1	1	1	1	1	1	9	9	9	9	9

Etat du Z-Buffer associé à l'écran



Affichage d'un personnage caché derrière le mur gauche

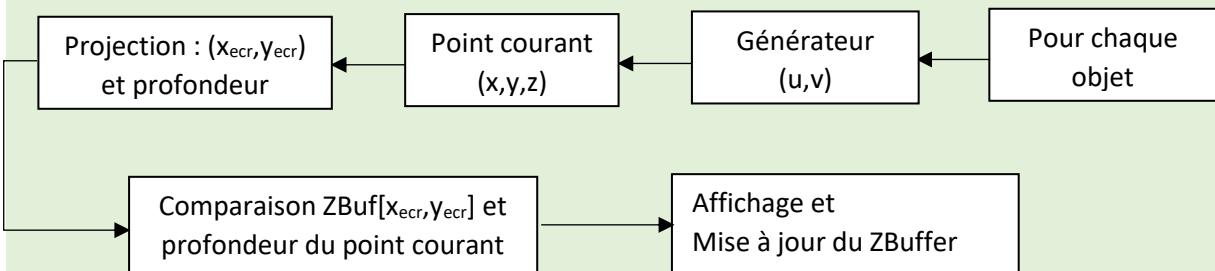


Affichage Ecran
x : point ignoré

1	1	1	1	1	1	9	9	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	4	9	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	4	4	9	9
1	1	1	1	1	1	9	9	9	9

Etat du Z-Buffer associé à l'écran

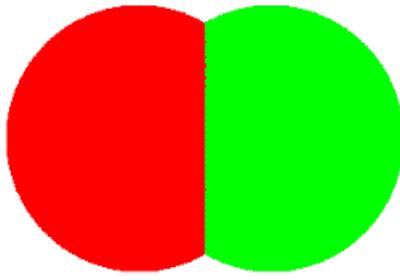
Cette façon de procéder permet de gérer aisément l'occultation entre objets. Avec l'algorithme du Z-Buffer, nous parcourons la liste des objets, pour chaque objet, nous démarrons la double boucle en (u,v) et nous construisons itérativement des points de la surface. Pour chaque point, nous calculons ses coordonnées et sa couleur. Après la projection, nous connaissons ses coordonnées écran et ainsi juste avant de passer à l'affichage, nous effectuons le test sur le Z-buffer. Nous sommes ici dans une logique où seul le point courant est connu. Lorsque nous faisons le test du Z-buffer, nous ne savons pas quels objets ont été dessinés avant, nous n'avons pas d'information sur ceux qui vont être dessinés après. Il n'y a plus de logique de d'ordonnancement, nous avons juste un point et par rapport au Z-buffer, nous cherchons à savoir s'il peut être affiché où pas. Au moment du test, nous comparons ainsi le point courant au dernier point passé à l'affichage.



Le Z-Buffer joue donc le rôle de mémoire et permet de conserver l'ensemble des profondeurs des points visibles à l'écran à un instant t de la boucle. Cette comparaison s'effectue entre deux points, à aucun moment on se préoccupe des objets, on ne sait pas à quel objet a appartenu le point stocké dans le Z-Buffer, la connaissance de l'objet qui a construit le point courant n'est pas utile, ces informations n'ont pas d'importance. Nous effectuons uniquement une comparaison entre deux points d'après leur profondeur respective.

Mise en place

Vous devez intégrer une classe représentant l'objet sphère avec ses attributs internes : rayon, centre, couleur... Dessinez deux sphères de rayon 100 et de coordonnées (200,0,200) et (350,0,200). Vous devriez obtenir le résultat suivant si le Z-Buffer fonctionne correctement :



6. Consignes et assistance

Pour votre projet, il vous est demandé :

- Nommage : donner des noms de variables explicites / clairs
- Variables : créer une variable par notion / besoin

En cas de demande d'assistance sur un problème survenant dans le code de votre projet, si votre source ne respecte pas ces deux règles de base, aucune assistance ne vous sera offerte.

On ne perd pas 30 minutes à comprendre un code de 10 lignes alors que 5 minutes étaient requises pour le rendre lisible. De plus, dans 80% des situations, la simple restructuration du code supprimera le/les problèmes d'origine. Alors, pourquoi s'en priver.

Nommage

Le nom d'une variable, d'une fonction ou d'une classe doit répondre à certaines questions : la raison de son existence, son rôle et son utilisation. Si un nom exige un commentaire, c'est qu'il ne répond pas à ces questions.

```
int d; // Temps écoulé en jours.
```

Le nom *d* ne révèle rien. Il n'évoque pas une durée écoulée, pas même des jours (*days*). Nous devons choisir un nom qui précise ce qui est mesuré et l'unité de mesure si possible :

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Voici un traitement, telle que l'on en trouve si souvent :

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

Pourquoi est-ce si compliqué d'en comprendre les intentions ? Il ne contient aucune expression complexe. Son utilisation des espaces et de l'indentation est correcte. Il ne fait référence qu'à trois variables et deux constantes. Il est même dépourvu de classes ou de méthodes polymorphes, il n'y a qu'une liste de tableaux.

Le problème vient non pas de la simplicité du code mais de son. Le code exige implicitement que nous connaissons les réponses aux questions suivantes :

1. Quelles sortes de choses trouve-t-on dans theList ?
2. Quelle est la signification de l'indice zéro sur un élément de theList ?
3. Quelle est la signification de la valeur 4 ?
4. Comment dois-je utiliser la liste renournée ?

Les réponses à ces questions ne sont pas données dans l'exemple de code, mais elles pourraient l'être. Supposons que nous travaillons sur un jeu du type démineur. Nous déterminons que le plateau de jeu est une liste de cellules nommée theList. Nous pouvons alors changer son nom en gameBoard. Chaque cellule du plateau est représentée par un simple tableau. Nous déterminons que l'indice zéro correspond à l'emplacement d'une valeur d'état et que la valeur d'état 4 signifie "marquée". En donnant simplement

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

La simplicité du code n'a pas changé. Il contient toujours exactement le même nombre d'opérateurs et de constantes, et les niveaux d'imbrication sont identiques. En revanche, il est devenu beaucoup plus explicite.

Créer des variables !!

Une chose = une variable

Très étrangement, c'est le point le plus dur pour les étudiants. Les symptômes sont multiples : ligne de code sur plusieurs lignes ou sortant de l'écran sur la droite. Copier coller récurrents ceci sur une même ligne, l'acharnement est sans limite parfois. Utilisation de l'opérateur ternaire ? pour se sortir d'une mauvaise passe.

Voici le genre de code que l'on peut rencontrer, à noter que le problème se cumule avec le précédent ce qui rend les choses encore plus indigestes :

```

if (x-y*alpha*k > 4 )
    m = (x-y*alpha*k)*( x-y*alpha*k) + c ;
if ( m > 0 )
    T[ x, (int) (x-y*alpha*k)*( x-y*alpha*k) + c + 10 ] = x-y*alpha*k ;

```

Le terme $x-y*alpha*k$ est réutilisé plusieurs fois, le code en devient réellement illisible et trouver un bug là-dedans revient à chercher une aiguille dans une botte de foin.

Généralement, un terme qui se répète correspond souvent à une nouvelle notion ou un résultat intermédiaire dans la chaîne de calcul qui aurait dû être explicitement stocké dans une variable nommée.

Il en est de même lorsque l'on change le type d'une variable, en général, cela désigne une autre notion et cela doit faire l'objet d'une nouvelle variable.

On aboutit finalement au code suivant :

```

float profondeur = x-y*alpha*k;
if (profondeur > dist_camera )
    y_3D = profondeur*profondeur+c ;
if ( y_3D > 0 )
{
    int y_ecran = (int) y_3D + 10 ;
    ZBuffer[ x_ecran,y_ecran] = profondeur ;
}

```

7. Les Lampes

Comment éclairer une scène ?

En synthèse d'images, le monde par défaut est sans éclairage, donc la scène est plongée dans la nuit noire. En posant une unique lampe, on ne peut reproduire un éclairage proche de la réalité. Utiliser une seule lampe vous amènera à un éclairage proche des éclairages que l'on fait sous une tante avec une lampe de poche placée sous le menton histoire de faire peur à ses camarades ☺

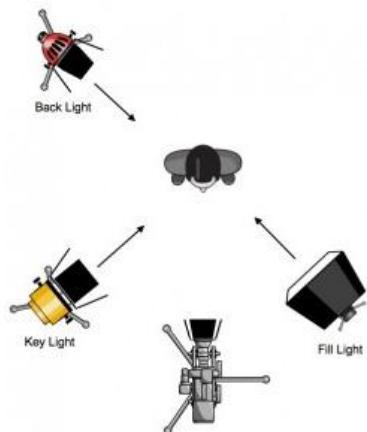




Plus sérieusement, voici une image de synthèse produite avec une unique lampe placée devant l'objet principal. Voici les différents défauts qui se produisent :

- L'objet est mal détourné, c'est-à-dire que ses bords sont sombres et se mélangent avec le fond
- Le défaut précédent a pour effet indirect de creuser l'objet : il a l'air plus petit que sa taille réelle
- Le visage reste fade, terne, sans saveur. Il n'attire pas l'attention. Il n'est pas mis en valeur, l'éclairage n'est pas flatteur.

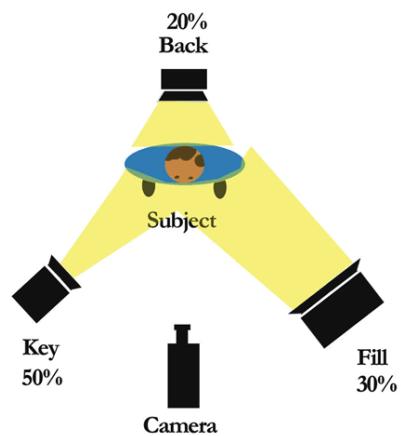
Il faut donc passer à un éclairage plus professionnel. Cette problématique n'est en rien propre à l'informatique, le monde du cinéma ou de la publicité ont largement travaillé la question. Nous vous proposons d'utiliser leur modèle le plus classique : « Three Points Lighting » qui a largement fait ses preuves.

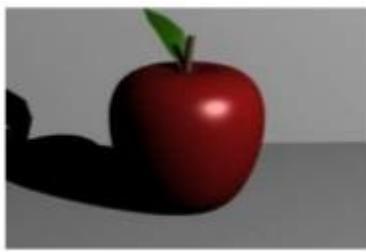


Dans un éclairage studio (cinéma, photo, plateau télé), 3 lampes principales sont nécessaires : la Key Light, la Fill Light et la Back Light.

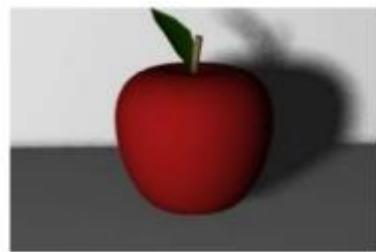
La Key Light est la lampe principale. Elle a pour fonction d'éclairer le sujet de trois quart. Pour compléter l'éclairage principal, on pose une deuxième lampe, la Fill Light, à droite qui éclaire les zones secondaires non éclairées par la Key Light. 95% des zones de l'objet sont en général éclairées par ces deux lampes. Certaines zones peuvent résister et peuvent ne pas recevoir de rayons depuis ces deux sources. On utilise alors en plus une lampe arrière, la Back Light, permettant de fournir un éclairage à 360° de l'objet.

Mais comment régler la puissance de ces lampes ? Si vous prenez trois lampes chacune à une puissance de 100%, vous pouvez avoir une zone de l'objet en surexposition car les pixels vont être éclairés à 200% voire 250% et donc tout sera blanc et brûlé. Il faut donc faire attention à ce que la somme des puissances de vos lampes flirte avec les 100% pour éviter les zones de saturation. Dans le schéma à droite, une répartition équilibrée a été faite entre les trois lampes, à noter que leur total fait 100%.

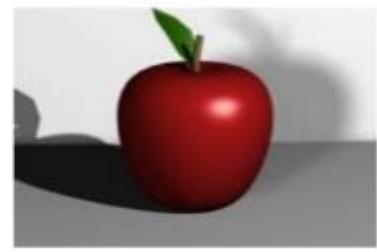




key light only



fill light only



key + fill

L'éclairage de gauche utilise seulement une lampe, la Key Light. L'éclairage de trois quart (à droite ici) est déjà plus agréable que l'éclairage de face. L'arrière de l'objet n'est pas détourné, il est fade. L'avant avec un léger reflet présente mieux. A noter l'ombre portée sur la gauche. Elle a pour effet d'apporter un effet de volume à l'objet, elle n'était pas présente sur l'éclairage de face du visage.

La Fill Light est une lampe plus douce. Elle a pour effet d'éclairer la zone gauche de l'objet non éclairé par la Key Light. Son intensité plus faible ne permet pas de faire un effet de reflet, l'ombre est aussi plus douce. L'image produite à parti des deux lampes est assez sympathique. Vous vous contenterez de cette approche Key+Fill dans votre projet, elle fournit des résultats très satisfaisants.

Cependant les puristes remarqueront que certaines zones sont un peu fades, la feuille et la base de la tige. Normal, la feuille est de trois quart par rapport à la caméra et les deux lampes sont mal placées, la base de la tige est mal éclairé lui aussi. On peut alors poser une Back Light légèrement surélevée pour rehausser ces zones.



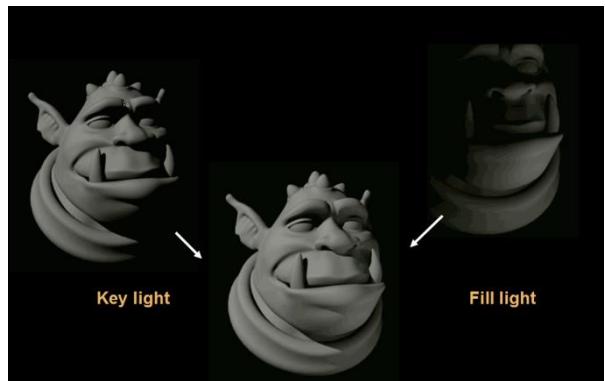
backlight only

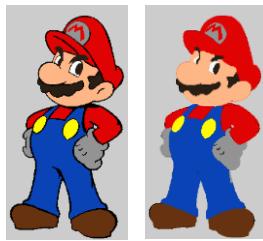


key + fill + back

L'effet de la Back Light passe quasiment inaperçue. Il ne faut pas oublier qu'elle se situe en arrière de l'objet et qu'elle éclaire surtout son dos non visible de la caméra. Cependant les zones peu mises en valeur par la Fill et la Key sont rehaussées. L'addition des trois lampes fournit une image homogène mettant en valeur l'objet dans la scène.

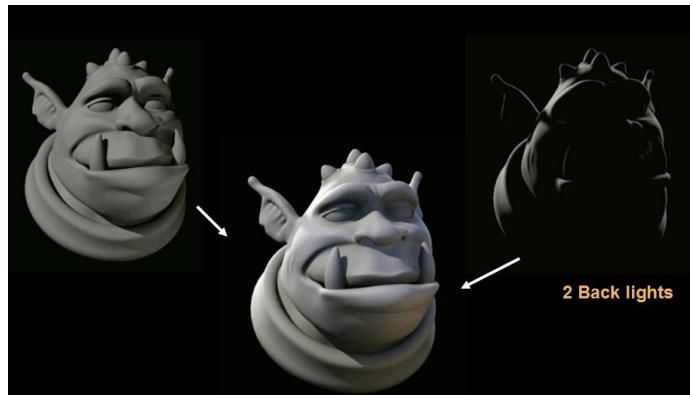
Pour terminer, analysons un peu la touche Dreamworks à partir d'un Shrek Like. La Key Light est placée de trois quart et en hauteur. Normal, elle symbolise ici l'éclairage principal du soleil. Pour compenser l'absence d'éclairage du menton à droite, on compense par une Fill Light diamétralement opposée à La Key Light. L'addition Key+Fill est réussi. Mais pas totalement complète !





Dans les bandes dessinées, on détourne les éléments par un trait noir, sans ce trait, relativement mince par rapport à la taille du personnage, les éléments deviennent difficilement lisibles, les contours sont plus difficiles à interpréter pour l'œil et pour le cerveau ☺ Les grands studios ne sont pas passés à côté de ce principe. Cependant, en synthèse 3D, il est plus simple de poser de la lumière qu'un trait noir !

Pour mieux détouurer et éclairer l'objet, on va utiliser 2 Back Lights placées à l'arrière du personnage et avec un angle légèrement décalé. L'objectif ici est de produire un éclat lumineux sur les bords du visage, 1 liseré à gauche et 1 autre à droite. Une fois ajouté à la Key + Fill, on obtient un pur visage de cinéma !



La lampe principale éclaire la joue gauche par le bas, effet type éclairage de nuit provenant du sol (bougie, feu). La joue droite reste éclairée, il y a donc utilisation d'une Fill Light. Remarquez que chaque frontière du visage sur la droite et sur la gauche est plus lumineuse que le centre des joues, cela n'est possible que par l'utilisation de deux Back Lights supplémentaires !

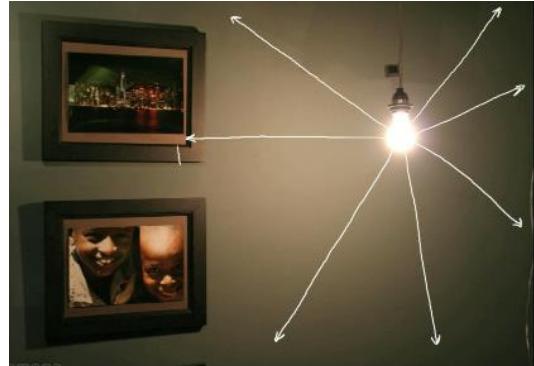
Les différents types de lampe



La lampe la plus courante est la **lampe directionnelle**, elle correspond à un éclairage extérieur provenant du soleil. Le soleil étant éloigné, lorsque ses rayons touchent la scène, ils sont parallèles. Ainsi les ombres sont orientées de la même manière. Les lampes directionnelles n'ont pas de position dans la scène, elles sont considérées comme étant placées à l'infini, comme le soleil. Seul compte leur orientation.

Pour démarrer votre projet, nous vous demandons d'utiliser une unique lampe directionnelle. En effet, son effet est clairement identifiable visuellement et il est facile de remarquer ce qui cloche. Une lampe directionnelle est aussi plus simple à débuguer sur le plan informatique.

Une **lampe ponctuelle** a une position dans la scène 3D. Ses rayons lumineux sont émis depuis cette source et partent dans toutes les directions. Les lampes ponctuelles produisent des éclairages plus complexes et sont plus difficiles à maîtriser. Elles sont souvent associées à une valeur de Decay (affaiblissement) car leur portée est relativement limitée. Elles permettent de modéliser des bougies, des lampes à incandescence, des spots... Les ombres tournent s'orientent en fonction de la disposition de la lampe et de l'objet.



Attention les lampes en synthèse d'images sont des sources de lumière, elles n'ont pas d'existence physique ! Si vous voulez qu'une lampe existe physiquement dans la scène, il faudra l'associer à objet 3D.

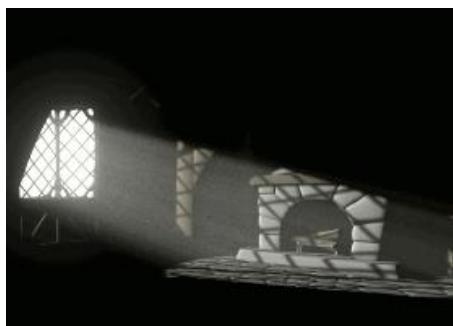
Si vous voulez en savoir plus sur les illuminations, les comportements de la lumière et les éclairages, je vous recommande l'excellent DVD de Jeremy Vickery : Practical light and Color.

Trajet des rayons lumineux

Dans la nature, les rayons lumineux ont un comportement bien plus complexes que dans les modèles que nous allons développer ici (le dernier chapitre du projet amène cependant à des modèles physiquement proches de la réalité). Quelle est la différence principale ? Nous modélisons dans ce premier volet uniquement des rayons lumineux de niveau 1, c'est-à-dire des rayons lumineux effectuant 1 rebond en tout. Un tel rayon part de la source lumineuse, rebondit sur un objet pour arriver jusqu'à la caméra. Un rayon de niveau 1 rebondit 1 fois dans la scène. Les rayons de niveau 1 constituent l'éclairage principal de la scène, mais ils ne permettent pas de représenter toute la scène.

Par exemple, lorsque vous regardez dans un miroir vous avez à faire à des rayons lumineux de niveau 2. En effet, ils sont émis par une source, rebondissent sur un objet, puis sur le miroir et arrivent afin à votre œil. Avec des rayons de niveau 1, il est ainsi impossible de modéliser une réflexion dans un miroir.

Il faut imaginer que les rayons lumineux dans une scène rebondissent sans cesse. Chaque rayon qui frappe un objet est généralement réémis avec une intensité plus faible cependant. Cet objet se comporte alors « comme une source d'éclairage secondaire ». Ainsi, un objet bleu posé sur une table réemet des rayons lumineux bleus dans la scène. Ces rayons bleus vont aller éclairer d'autres objets et rebondir à nouveau à leur surface. La trajectoire du rayon s'achève dans votre œil ! Pour obtenir une image photoréaliste, il faut pouvoir calculer et modéliser des rayons de niveaux 4/5/6. La plupart des jeux vidéo des années 2000 n'utilisaient que le niveau 1 !



Voici une scène d'intérieur éclairée par la lumière du jour. Si l'on ne conserve que les rayons de niveau 1, on illumine uniquement les objets présents en face des ouvertures. Le reste de la pièce est sombre, car il n'y a aucun éclairage direct qui atteint les objets.



Utilisons maintenant les rayons de niveau 2. Que se passe-t-il ? Le plafond de la pièce apparaît ? Pourquoi. En fait il faut examiner le sol. Il reçoit des rayons provenant de la fenêtre et en renvoie dans toute la pièce. Le sol se comporte alors comme une lampe secondaire qui va éclairer directement ce qui se trouve en face, c'est-à-dire : le plafond !



Lorsque nous activons les rebonds multiples, nous obtenons une scène bien plus lumineuse. Si nous examinons le sol, il est cette fois éclairé dans sa globalité. Ici, le plafond a joué son rôle de lampe indirecte en renvoyant au sol des rayons provenant du centre du sol, lui-même éclairé par le soleil. Il a donc fallu utiliser des rayons de niveau 3 pour avoir un sol éclairé de manière réaliste. Pour les charpentes sur le haut de la scène, il faudra sûrement compter sur des rayons de niveau 4 ou plus.

La lampe ambiante

Il est encore difficile de calculer des rayons de niveaux 3/4/5 en temps réel. Alors comment faire ? Les zones qui se trouvent dans les recoins et sous les tables, par exemple, ne peuvent recevoir d'éclairage direct. Une fois le calcul effectué, ces zones restent noires comme dans la première image de la pièce. Le noir, sauf sans un film d'épouvante, n'est pas recherché, il ne porte ni couleur, ni texture, ni information de relief. Si vous regardez les productions professionnelles (Disney, Pixar, Dreamworks) vous constaterez que les aplats de noirs sont volontairement évités, ce n'est pas flatteur pour l'image. Alors pour s'en sortir, l'informaticien triche, mais ici c'est pour la bonne cause.

En effet, on s'aperçoit que les rayons de haut niveau produisent en général des éclairages de faible intensité autour de 5% à 15%. Les variations de luminosité y sont faibles, alors, pourquoi ne pas mettre une lampe qui éclaire tous les objets de la scène à 10% par exemple. Quel est l'intérêt ? Une telle lampe est peu coûteuse à calculer, elle permet de relever les noirs et d'assurer que les zones les plus sombres sont éclairées de manière homogène. Comme elle ne fait aucune distinction entre les objets et qu'elles éclairent ainsi tout le monde, elle va cependant surcharger les objets lumineux. Mais, on peut corriger cet effet en diminuant l'intensité des autres lampes. Finalement, nous obtenons une bonne solution pour simuler des rayons de niveaux supérieurs.

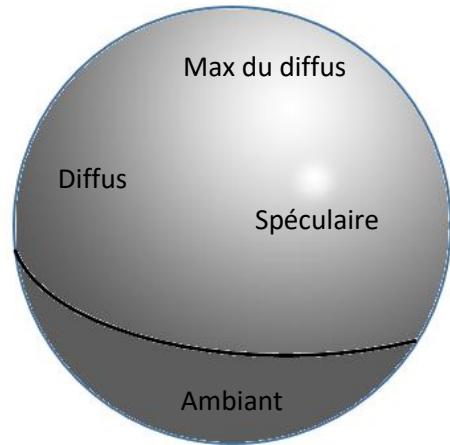
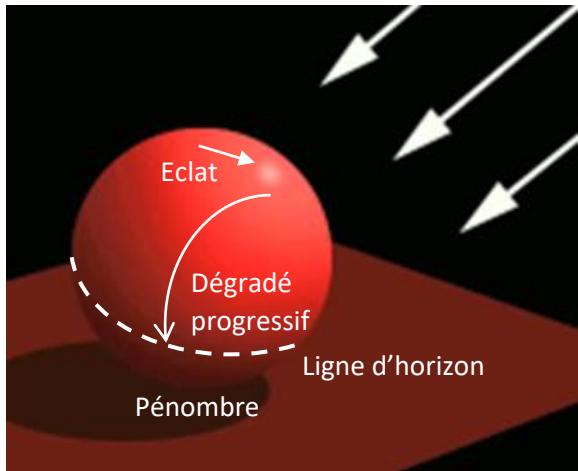
A noter que la lampe ambiante est unique, car deux lampes ambiantes se cumulent et se résume à une seule lampe ambiante. Il faut rester doux au niveau du réglage de l'intensité. En effet, la lampe ambiante est assez puissante puisqu'elle éclaire tout le monde et partout, dépasser 10% tend à produire des images très lumineuses, attention !

8. Les modèles d'illumination

Que peut signifier l'expression « modèle d'illumination » ? Un modèle d'illumination permet de modéliser les comportements de la lumière à la surface d'un objet. Différents comportements ? Pour éviter de démarrer par une formulation trop mathématique, nous allons étudier un exemple : la boule de billard.

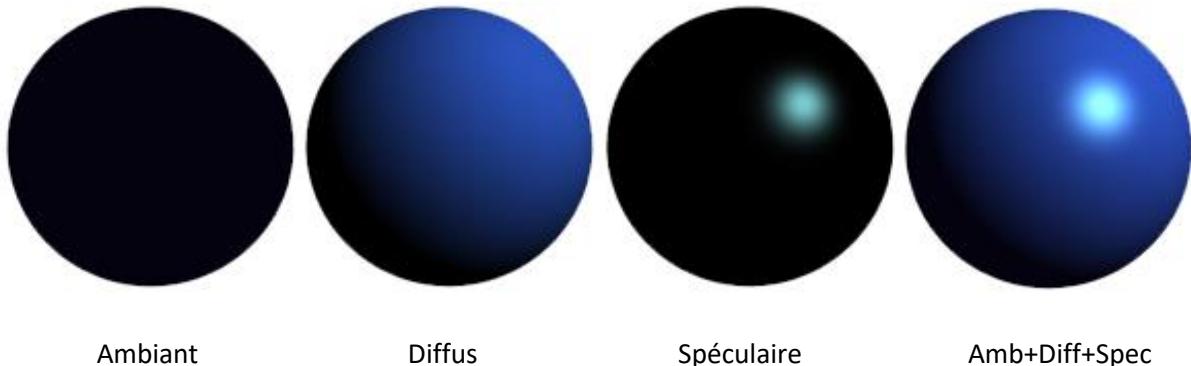
Sur cette photographie, nous avons une source principale correspondant à la lumière du soleil : les ombres sont de même direction et bien découpées. Sur la boule blanche nous remarquons qu'une zone est plongée dans la pénombre. Elle correspond à la partie de la boule qui ne reçoit pas directement des rayons du soleil. Nous remarquons une « ligne d'horizon » qui sépare la zone éclairée de la pénombre. Sur la boule bleue et la mauve, on remarque un dégradé allant du plus clair sur l'avant droit jusqu'à la zone de pénombre.





Le dégradé progressif qui va jusqu'à la ligne d'horizon est produit par le **modèle de réflexion diffuse** appelé aussi « le diffus ». Sur l'ensemble des boules, un éclat est présent et semble dirigé vers le soleil. L'éclat correspond au **modèle de réflexion spéculaire** appelé aussi « le spéculaire ». L'éclat se trouve parfois proche de la zone d'intensité maximale du diffus, mais leurs positions sont généralement différentes. La zone dans la pénombre n'est pas noire, elle porte un niveau de luminosité faible apporté par le **modèle de réflexion ambiant**, « l'ambiant ».

Voici les différents effets produits par chaque modèle. Le résultat final à droite est l'addition des différents résultats. Vous allez bien sûr programmer tout cela :



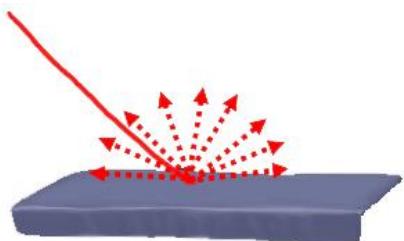
Le modèle de réflexion ambiant

Comme nous l'avons vu, ce modèle ne correspond pas à une réalité physique mais il existe car il permet de simuler les éclairages complexes des zones sombres, éclairage impossible à réaliser avec des rayons de niveau 1. La formule est la plus simple qui soit puisqu'il y a juste la couleur de l'objet et la couleur de la lampe ambiante à gérer. Ainsi, nous avons :

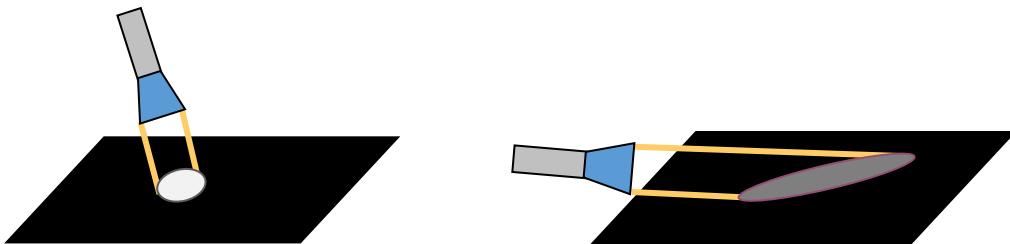
$$C_{Ambiant} = C_{L_Amb} \times C_{Objet}$$

Le modèle de réflexion diffuse

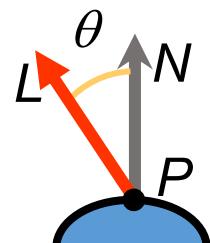
Il génère 90% de la luminosité d'une scène. C'est le modèle le plus commun et le plus classique. Dans ce modèle un rayon lumineux heurte une surface, se teinte suivant la couleur du point d'impact et réemet une multitude de rayons lumineux dans toutes les directions. Il faut noter qu'il n'y a pas de direction privilégiée. Ainsi en un point donné d'un objet, quelle que soit la position de la caméra autours de cet objet, la couleur produite par le modèle diffus ne change pas. Ainsi, lorsque l'on se promène dans une pièce, il y a une certaine stabilité des couleurs des différents objets, cela est dû au principe du modèle diffus.



Pour comprendre la formule du modèle diffus, il suffit de se ramener à une expérience de physique basique. Si l'on prend une lampe de poche et qu'on la place perpendiculairement à une surface, cette lampe crée une zone circulaire éclairée vivement. Plus on penche la lampe, plus la tâche de lumière s'étale. Comme la quantité de lumière produite est constante et que la surface éclairée augmente, cela implique que l'intensité de la zone éclairée s'affaiblit.



En synthèse d'images, nous considérons uniquement les informations présentes à un point de la surface de l'objet. Nous ne tenons pas compte du reste de la surface. Toutes les formules données sont donc exprimées pour un point donné P dont la normale N est orientée vers l'extérieur. Nous notons L le vecteur directeur des rayons lumineux. Par tradition, ce vecteur est orienté dans le sens inverse du rayon incident, cela permet d'avoir l'ensemble des vecteurs orientés du point courant vers l'extérieur.



La Loi de Lambert nous donne la formule de calcul du modèle diffus : l'intensité de la lumière ne dépend que de l'angle entre la source et la normale à la surface :

$$I_{Diffuse} = I_{Source} \cdot \cos(\theta)$$

Pour rappel, en travaillant avec les couleurs nous obtenons :

$$C_{Diffuse} = (C_{Source} \times C_{Objet}) \cdot \cos(\theta)$$

Il est parfois utile de rajouter un coefficient $\rho_{Diffuse}$ propre à l'objet qui indique la capacité de ce dernier à produire un effet de diffus. Ce coefficient multiplicatif compris entre 0 et 1 agit comme un

potentiomètre et permet de régler l'intensité du Diffus. En effet, pour un objet mât, le diffus est proche de 100% par contre pour un objet brillant ou métallique, le diffus est beaucoup plus faible.

Les fonctions arccos, arcsin doivent être évitées à cause de leur coût prohibitif. Il faut donc effectuer ce calcul sans jamais connaître la valeur de l'angle θ . Cela est possible grâce à la propriété du produit scalaire. Pour rappel :

$$u \cdot v = \|u\| \cdot \|v\| \cdot \cos(\theta)$$

Ainsi, si nous travaillons avec des vecteurs normalisés, nous obtenons :

$$C_{Diffuse} = (C_{Source} \times C_{Objet}) \cdot (N \times L)$$

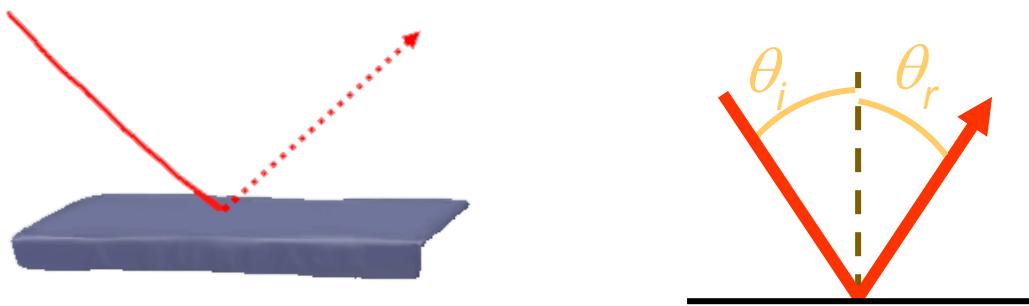
Le modèle de réflexion spéculaire



Une autoroute et un ralentissement sont une excellente occasion d'examiner des spéculaires. En effet, les peintures des carrosseries de voitures sont brillantes et sujettes aux reflets. La réflexion spéculaire correspond au comportement miroir d'une surface : le rayon lumineux frappe un point de la surface et rebondit totalement dans la direction symétrique par rapport à la normale. Ainsi lorsque vous observez des éclats de lumière sur les carrosseries, ce sont généralement les reflets des sources ponctuelles environnantes : lampes, phares qui se réfléchissent.



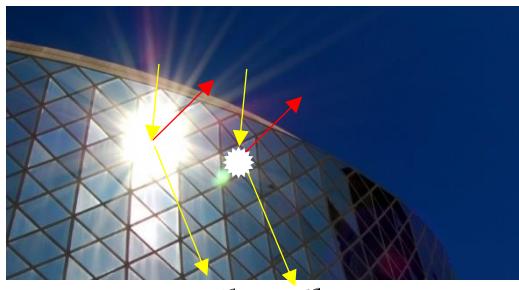
Le marché du luxe mise fortement sur le spéculaire. En effet, le comportement des reflets sur les surfaces est autant étudié que leur forme. Un bijou est essentiellement composé de spéculaire, c'est du métal poli : un miroir en somme !



La modèle de réflexion spéculaire est celui du miroir parfait. Les rayons ne font que rebondir en respectant la loi de Descartes : $\theta_i = \theta_r$



Dans le monde réel, comment faire la différence entre un diffus et un spéculaire ? Prenons ces trois photos d'avion, les couleurs sont rendues de la même manière, même si les appareils photos sont un peu différents. Cependant, d'une scène à l'autre, la position du photographe a changé. Cette propriété distingue le spéculaire, si l'objet et la source de lumière sont fixes, si vous vous déplacez légèrement, la position du spéculaire change de place. Le modèle diffus renvoie la lumière dans toutes les directions. Ainsi quelque soit votre position autours d'un objet, la lumière diffus ne varie pas.



Si nous examinons les vitres, une seule produit l'effet spéculaire. Il s'agit du vitrage qui se trouve à la position précise où les rayons du soleil rebondissent pour finir leur course dans l'objectif de la caméra.

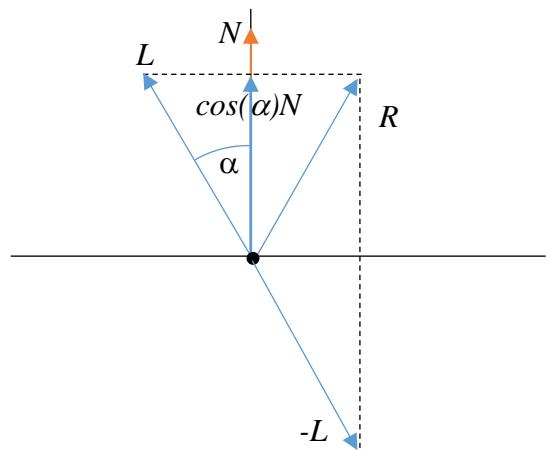
Si la caméra se déplace légèrement sur la droite, les rayons précédents ne peuvent plus l'atteindre. L'éclat se déplace alors sur la droite.

La logique est la même lorsque vous regardez quelqu'un ouvrir une fenêtre dans le bâtiment d'en face. A un bref instant, il va y avoir un flash de lumière. Cela correspond au moment où la vitre atteint la position de miroir parfait où elle revoie les rayons du soleil pile vers votre œil. Une fois cet angle dépassé, le flash disparaît.

Basiquement, le comportement du spéculaire est similaire à celui d'un miroir. Lorsque vous regardez dans un miroir, vous voyez les objets réfléchis à travers celui-ci. Si vous vous déplacez légèrement, vous voyez d'autres objets. Avec le spéculaire, n'importe quel objet peut jouer le rôle du miroir. Prenons cet exemple des carreaux sur le sol, il s'agit en fait des rayons qui proviennent du réfrigérateur et qui se réfléchissent au sol. Si nous nous déplaçons, ce reflet va changer de place.



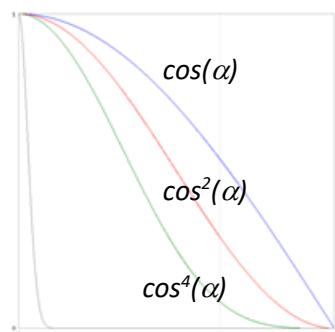
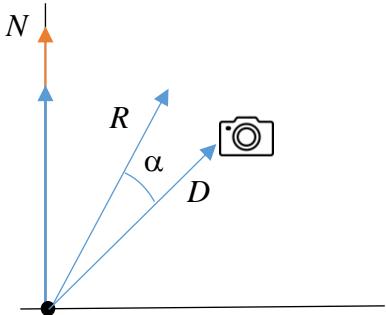
Pour calculer l'effet du spéculaire, il faut connaître la direction du rayon réfléchi R :



Nous supposons que le L et N sont des vecteurs normalisés. La projection de L sur la normale s'obtient facilement avec l'expression $\cos(\alpha).N$

Pour calculer R , il suffit de remarquer qu'il s'écrit à partir de la combinaison de $\cos(\alpha).N$ et de $-L$.

Maintenant, il faut connaître l'angle entre le rayon réfléchi et la direction de l'observateur. Pour cela il faut d'abord calculer le vecteur D entre le point courant et l'observateur. Si R et D sont colinéaires alors l'angle α est nul et l'effet du spéculaire est maximal. Lorsque D se décale de la direction R l'effet du spéculaire diminue rapidement. Cependant, comme nous l'avons déjà dit, il faut éviter le calcul de l'angle α .



Si nous examinons la courbe de $\cos(\alpha)$ entre 0° et 90° nous constatons qu'elle décroît lentement. Nous cherchons idéalement une courbe dont la valeur est à 100% lorsque l'angle est proche de 0° et à quelque % lorsque l'angle est faible. Si nous regardons les courbes de $\cos^2(\alpha)$ puis de $\cos^4(\alpha)$ nous constatons que nous obtenons une courbe qui se rapproche de plus en plus de ce que l'on cherche.

Ainsi, l'effet spéculaire se formule de la manière suivante :

$$C_{\text{Specular}} = C_{\text{Source}} \times (R \cdot D)^k$$

Le coefficient k s'appelle le « specular power ». Son utilisation est purement pragmatique. Plus k est grand pour l'éclat du spéculaire sera mince. Vous pouvez tester des valeurs entre 50 et 200.

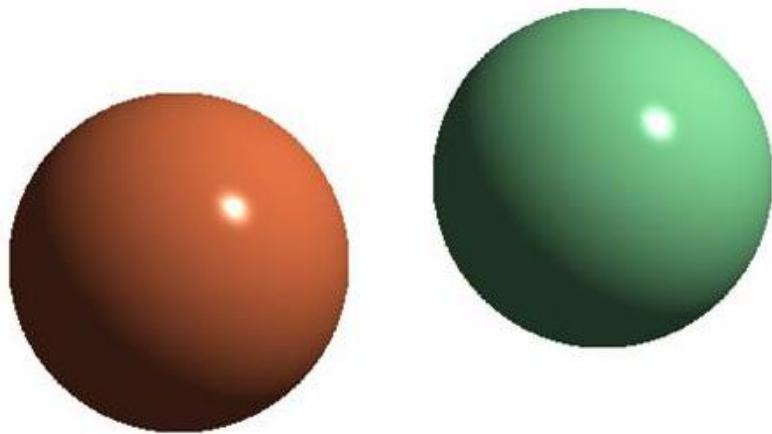
Attention pour pouvoir calculer le vecteur D , il faut donner une position à la caméra. Nous travaillons dans le repère de la grille de l'écran : 1 unité = 1 pixel. Il faut donc donner la position de votre œil dans ce repère. Généralement, vous êtes placé en face de l'écran, au milieu et reculé de 1 à 2 fois la largeur de la zone de dessin. Fournir une coordonnée en pixels à votre œil semble un peu déroutant mais c'est une manipulation assez simple en fait. Si votre écran fait 2000x1000 pixels, le x et le z seront égaux à 1000 et 500. Si vous êtes à une distance de 1.5xlargeur écran, votre œil est reculé de 3000.



Il faut donner une position à la caméra qui correspond à votre regard. Si par mégarde vous ne vous préoccupez pas de cette question et que vous mettez (0,0,0), cela équivaut à positionner votre œil dans le coin en bas à gauche de l'écran, ce qui ne correspond pas du tout à votre position. Pour le diffus, aucun soucis, mais pour les spéculaires, ils seront positionnés étrangement et inutile de chercher un bug informatique car il ne s'agit pas de cela. Si vous oubliez de reculer la caméra par rapport à la scène, ce sera un autre problème, car l'œil se trouvera probablement à l'intérieur d'une sphère ce qui engendrera des spéculaires tout à fait originaux.

Mise en place informatique

Appliquez à ces deux sphères les modèles de lumière ambiante, diffuse et spéculaire vus en cours. Choisissez la source de lumière que vous souhaitez ainsi que son affaiblissement. Nous présentons le résultat obtenu pour une source directionnelle (1,-1,1) sans affaiblissement :

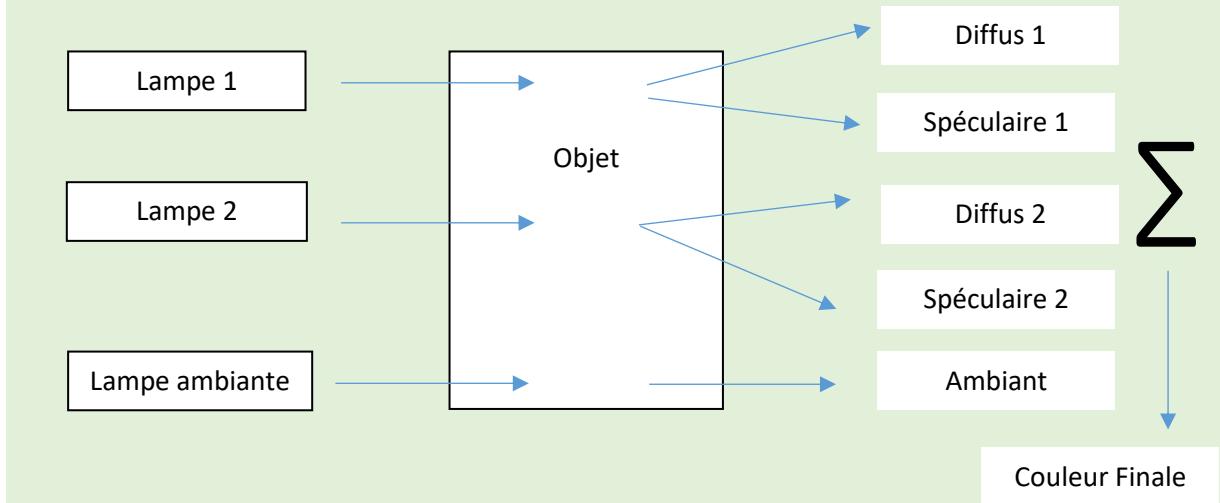


Modèles d'illumination et types de lampe

Il est parfois possible de mélanger les notions de diffus, spéculaire, ambiant et de directionnel. On entend alors parler de « lampe diffuse » et de « lumière spéculaire ». Remettons les choses à leur

place. Les lampes (directionnelles ou ponctuelles) produisent des rayons lumineux. Ces rayons lumineux lorsqu'ils atteignent une surface réagissent suivant le modèle de diffusion et/ou le modèle spéculaire. Il existe le cas particulier de la lampe ambiante qui utilise le modèle ambiant uniquement.

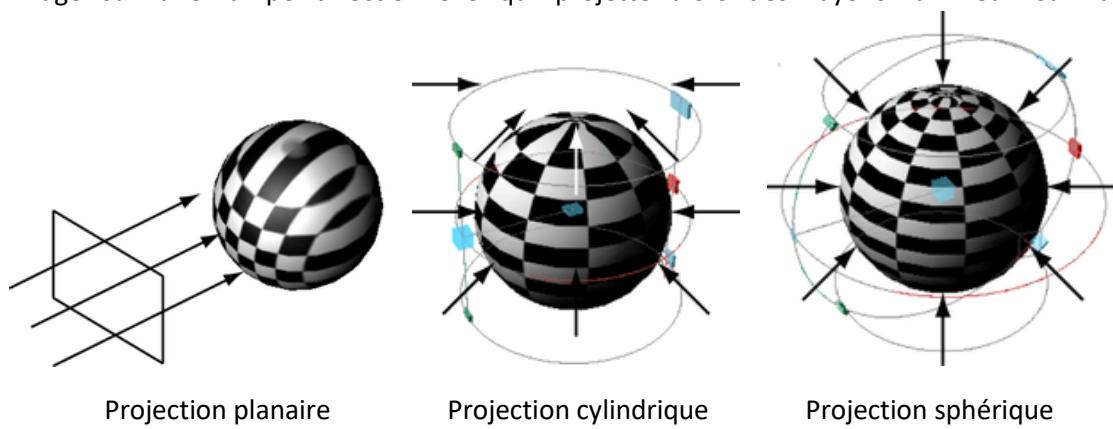
Lorsque plusieurs lampes sont présentes, les effets des différents modèles se cumulent :



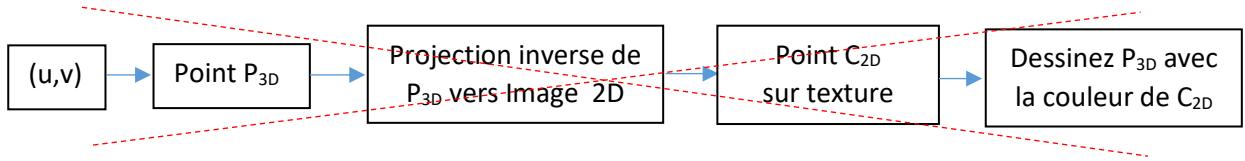
9. Texture

Une texture correspond à une image 2D. Dans le monde professionnel, la largeur et la hauteur de l'image sont souvent des puissances de 2 (256x128 par exemple) ceci afin d'optimiser les algorithmes sur GPU. Mais dans notre projet, ce n'est pas obligatoire. Afin d'être indépendant de la résolution de l'image d'origine, nous utilisons des coordonnées normalisées : [0,1]x[0,1] dans la classe Texture pour décrire la position courante sur l'image.

Il y a plusieurs moyens de « peindre » un objet à partir d'une image. Un des plus classiques est la projection. Vous pouvez voir plusieurs exemples ci-dessous où une texture de damier est projetée en mode planaire, cylindrique et sphérique sur une sphère. La projection planaire consiste à coller une image sur une lampe directionnelle qui projette alors des rayons lumineux sur la sphère.



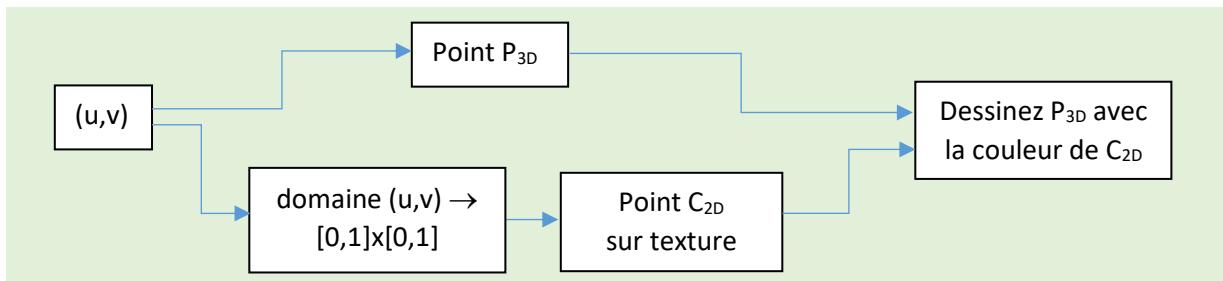
Pour gérer de telles projections, voici le flux de traitement :



Le problème réside dans les calculs engendrés lors de la correspondance géométrique : la projection inverse qui permet de savoir à quel endroit de la texture est associé le point 3D dans la scène. En général, c'est compliqué et ça finit par coûter cher.

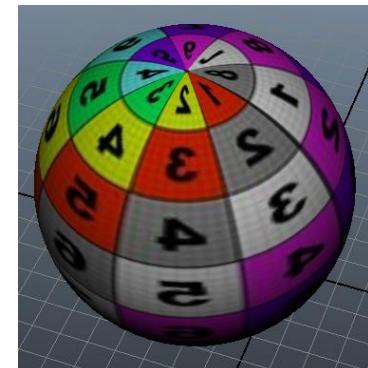
Nous allons donc choisir l'approche faite en production, beaucoup plus simple et qui a l'avantage de supprimer la recherche de la correspondance, ou du moins de la déléguer aux graphistes. Cette approche est parfois déroutante pour certains élèves car elle n'a pas d'interprétation naturelle sinon le fait qu'elle est très simple à calculer et à mettre en place.

Nous supprimons donc l'étape de projection inverse et nous partons d'un couple (u,v) . Avec ce couple (u,v) nous savons construire un point 3D. Nous allons maintenant appliquer une simple transformation pour faire correspondre le domaine des (u,v) avec le domaine $[0,1] \times [0,1]$ de la texture. Nous obtenons donc d'un côté un point 3D et de l'autre une couleur. Il nous reste juste à choisir l'approche suivante : dessinons ce point 3D avec cette couleur de texture.

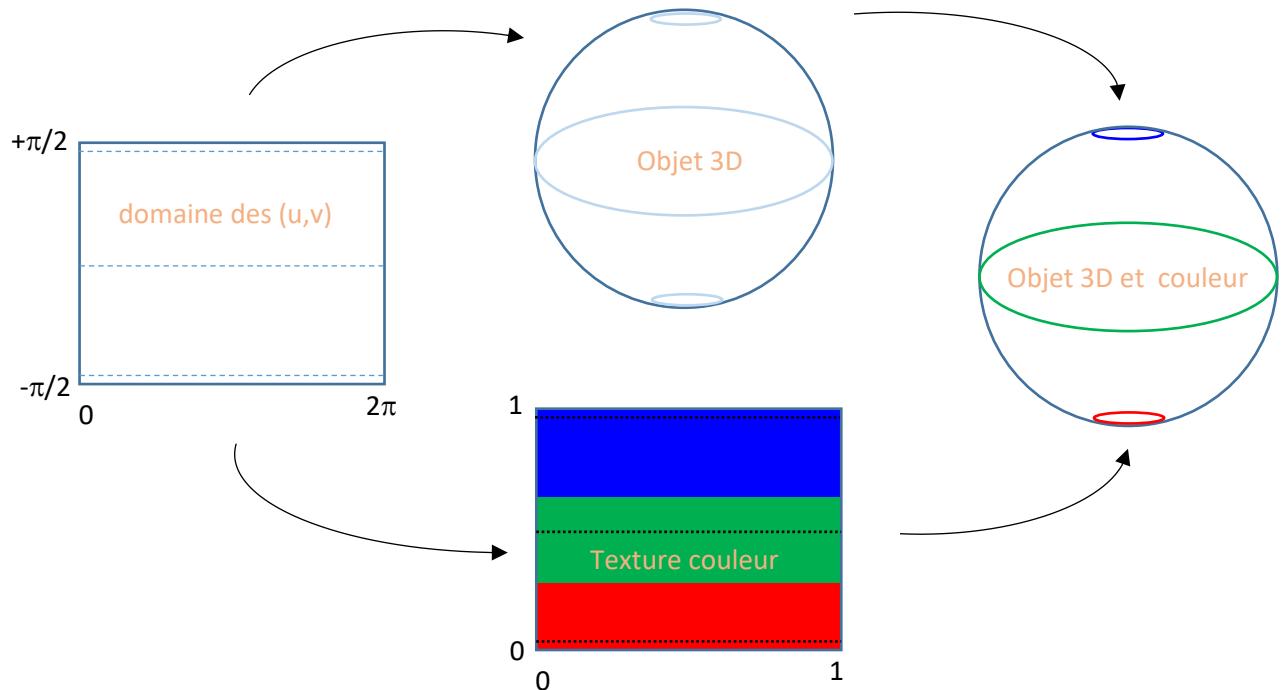


Voici le résultat obtenu avec cette approche à partir de la texture de test :

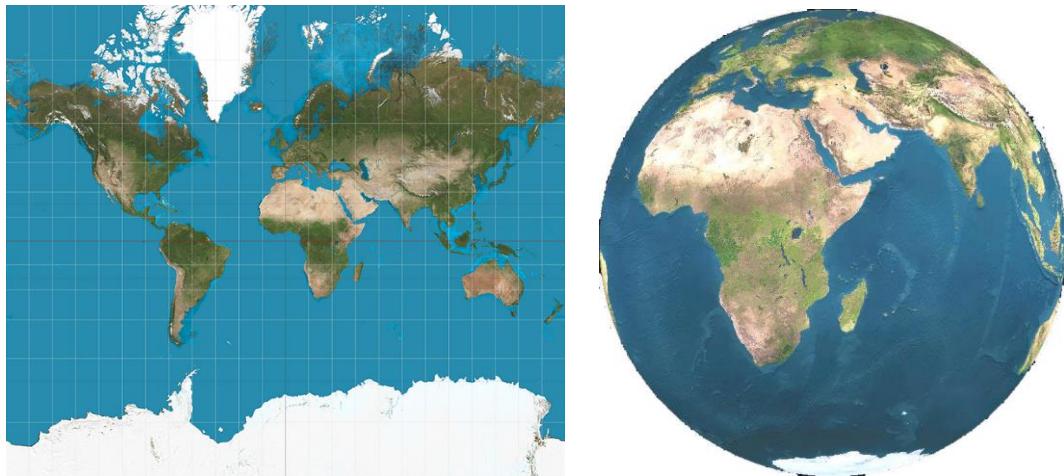
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	1
3	4	5	6	7	8	1	2
4	5	6	7	8	1	2	3
5	6	7	8	1	2	3	4
6	7	8	1	2	3	4	5
7	8	1	2	3	4	5	6
8	1	2	3	4	5	6	7



Comment voir les choses ? Reprenons notre espace (u,v) et choisissons quelques lieux spécifiques et regardons où se projettent la couleur à la fin :



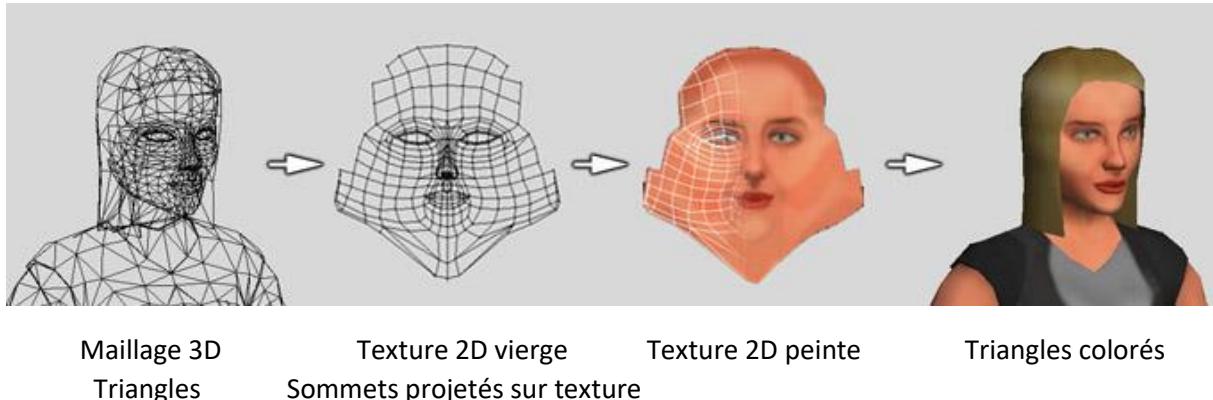
Cette projection pourra sembler familière à certains car elle correspond à la projection de Mercator, la projection utilisée sur les planisphères :



A gauche le planisphère sert de texture pour colorer la sphère de droite. Vous remarquez que le pôle sud et le pôle nord sont très agrandis sur la texture d'origine par rapport à leur taille réelle sur la planète. Mais l'image finale est parfaite. En fait, lorsqu'un graphiste colorie un objet 3D, il peint dans l'image de texture 2D et le logiciel affiche le résultat sur la surface 3D. C'est à lui de déformer son dessin de texture pour que les couleurs arrivent au bon endroit sur l'objet 3D.

En production, les volumes sont bien plus complexes que nos sphères et ils sont composés de facettes triangulaires. Le principe reste le même. Cependant, une étape supplémentaire est présente, elle consiste à déplier le maillage 3D (unfolding / unwrapping) pour le mettre à plat dans une image 2D. Cette image est ensuite colorée par le graphiste. Pour que l'algorithme puisse fonctionner, on associe

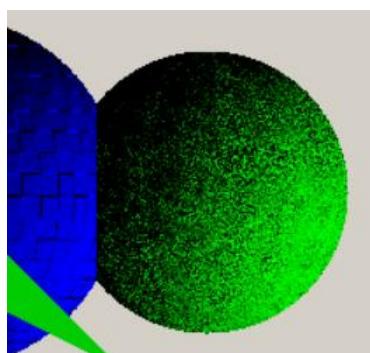
à chaque point du maillage 3D une coordonnées 2D (u, v) qui permet de retrouver sa correspondance dans l'image de texture. Il reste au GPU à dessiner une facette 3D et à la colorier à partir de la zone de couleur définie par un triangle 2D dans l'image de texture.



A partir des fichiers joints au projet, appliquez des textures sur vos sphères tout en laissant activer les modèles d'illumination précédents. Voici le résultat obtenu à partir des textures OR et METAL. Attention la texture doit recouvrir une fois et une unique fois la totalité de l'objet. Vous disposez d'une texture « uvtest.jpg » vous permettant de débugger votre application de texture.



Remarque utile

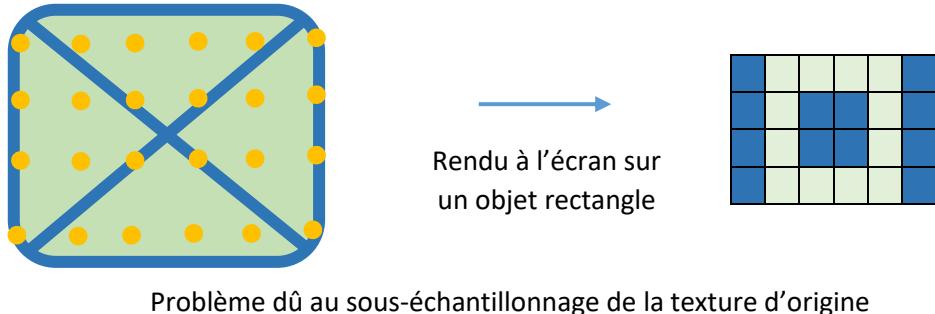


Voici un exemple de « texture à problème ». Il y a deux situations à savoir détecter :

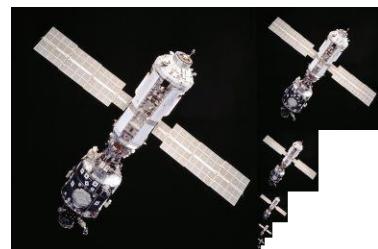
Problème 1 : la résolution de la texture est trop petite par rapport à la taille à l'écran, elle va être étirée, le résultat sera soit flou soit plein de gros carrés.

Problème 2 : c'est généralement celui-là qui vous malmène et qui produit le résultat de gauche. Dans cette configuration, la résolution de texture est bien plus importante que la taille de l'objet à l'écran. Si vous appliquez les textures fournies pour le projet en 512x512 sur une sphère ou un rectangle de 100 pixels à l'écran, cela veut dire que seulement 1 pixel sur 25 de l'image de texture d'origine va être utilisé dans l'image finale. Cet appauvrissement de la résolution d'origine va produire des effets parfois

indésirables à l'écran. La recommandation est d'avoir un facteur 2 au maximum entre résolution de texture et taille de l'objet à l'écran. Ainsi, afin de préserver la qualité graphique de vos images, nous vous conseillons d'utiliser les textures en pleine résolution sur les grands objets de votre scène et de réduire les textures utilisées sur les objets de petite taille à l'écran. Pour cela, l'outil Paint.exe permet de facilement redimensionner les images à la résolution souhaitée.



En production, on utilise la technique du MipMapping. Elle consiste à stocker en mémoire plusieurs résolutions de la même texture avec un facteur d'échelle égal à 2. Cela a pour effet de corriger ce problème.



En fait l'algorithme d'application de texture est très simple et il ne permet pas d'obtenir un bon résultat dans le cas général. Le fait de redimensionner la texture dans un logiciel de dessin permet de construire une nouvelle image avec des algorithmes qui préservent les détails de l'image d'origine.



10. Bump Mapping

Historique

Quels facteurs ont joué dans l'amélioration du rendu des jeux vidéo ?



Moteur Half Life 1 – 1998



Moteur Half Life 2 - 2004

On sent une montée en gamme. Pourtant, on compare ici des surfaces de bâtiments, c'est-à-dire des géométries relativement pauvres car assez plates. La différence se fait principalement dans l'augmentation de la densité des textures. A gauche, nous utilisons des textures 256x256 (une texture par habit, mur, visage, sol, drapeau, ciel...). Comptez environ 80 textures en RGB pour cette scène, cela fait $256 \times 256 \times 80 \times 3 = 16\text{Mo}$ le maximum des GPU de l'époque. En 2004, on passe au Giga de mémoire sur les cartes graphiques, ce qui permet d'embarquer des textures 1024x1024 ou plus. A une époque où la résolution d'écran standard était 1024x768, nous arrivions donc à avoir une texture aussi fine que la matrice de l'écran. Ainsi, regardez un mur de face, revenez à regarder une photo de ce mur. On avait atteint la densité nécessaire pour l'aspect « réaliste » et nous allions enfin pouvoir oublier les gros carrés de DOOM 1 :



Attention, texture 32x32 en approche !! Préparez-vous à griller du gros pixel !

Nous allons maintenant aborder un nouvel effet qui a permis d'effectuer une montée en gamme dans la qualité des images. Examinez les images ci-dessous :

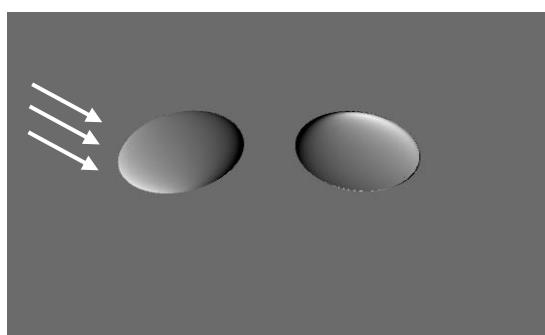


Nous avons à gauche et à droite la même texture et la même géométrie. Et pourtant, la qualité n'a rien à voir. Les pierres sur la droite ont l'air bien plus réelles et bien plus rondes !! Il s'agit ici d'un effet de BumpMapping. Vous avez peut-être croisé ce terme au fin fond du menu des options de rendus de vos jeux. Nous allons voir à quoi correspond cet effet et comment le mettre en place, puisque cela fait partie de votre projet !

Le Bump-Mapping est une technique permettant de **simuler** la variation des normales de l'objet relativement à une faible déformation de la surface. Cet effet visuel est très impressionnant, il s'agit par exemple des crénelures, des cabossages, des rainures :

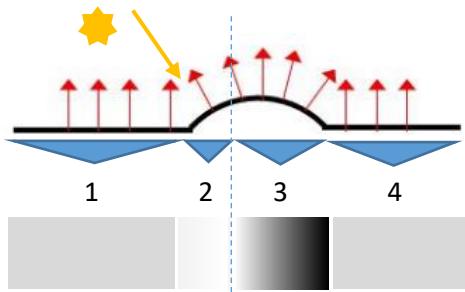


Principe du Bump-Mapping



Examinons ce qu'il se passe sur une surface plane qui comporte une bosse et un creux. Veuillez noter la direction des rayons lumineux. L'information géométrique est faible, le creux et la bosse sont relativement discrets. Le cerveau, pour interpréter cette scène, va se servir de la direction des rayons lumineux puis de la position des ombres et des éclairages apparaissant sur les surfaces.

Comment cela fonctionne-t-il ? Prenons un cas simple en 2D et faisons apparaître les normales à la surface. Si nous avons un éclairage par le haut sur la gauche, en appliquant le calcul de lumière difuse nous obtenons le rendu suivant. Sur la zone 1 et 4, la normale est constante et orientée vers le haut. Le cosinus du diffus est donc constant sur cette partie et la luminosité est homogène. Sur la gauche de la zone 2, la normale est presque colinéaire aux rayons du soleil, ainsi le cosinus du diffus est de l'ordre de 80%. Puis en allant sur la droite, la normale devient colinéaire aux rayons du soleil et le cosinus devient égal à 100%. La luminosité est maximale. Une fois que l'on entre dans la zone 3, l'angle entre la normale et le rayon du soleil augmente de 0° pour atteindre 90° . Ainsi, sur la droite de la zone 3, la zone est totalement sombre. En quittant la zone 3 et en entrant dans la 4, on a une cassure au niveau de la luminosité, en passant du noir au gris homogène de la surface. Ainsi avec cette direction des rayons lumineux, voici comment le cerveau interprète la scène :



Transition : gris/clair/sombre/gris → bosse

Transition : gris/sombre/clair/gris → creux

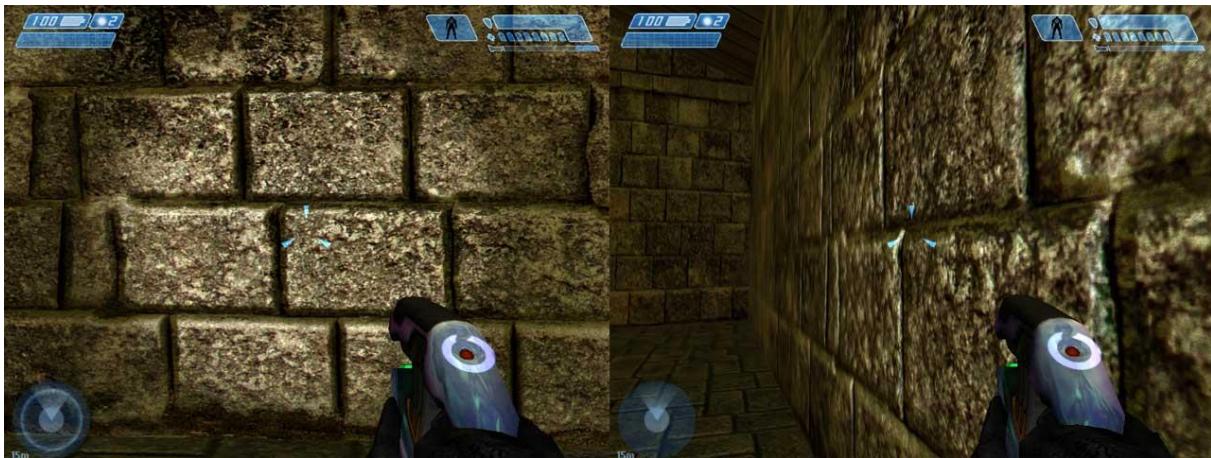
Ainsi l'information lumineuse porte l'ensemble de l'information géométrique dans ces configurations où les défauts de surface sont relativement fins. Nous allons donc mettre en place un système de « trompe l'œil » dont le principe va consister à garder une géométrie basique mais à faire « bumper » la normale pour qu'elle corresponde à la normale présente avec le défaut de surface :



Voici donc l'exemple précédent, la normale est bumpée mais la surface reste lisse.

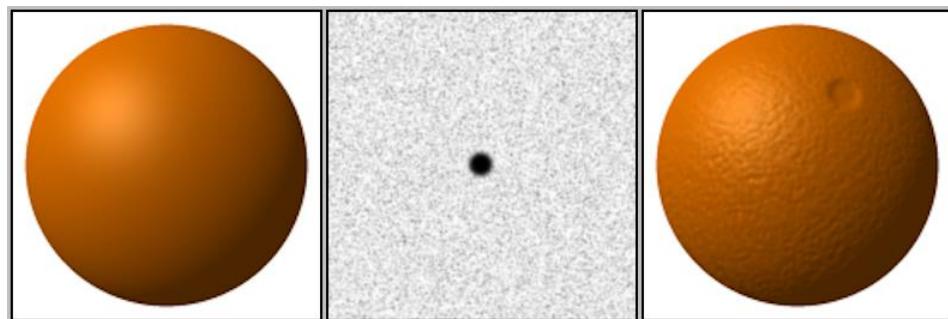
L'effet de bump est bien un effet de trompe l'œil car la géométrie des surfaces restent simples. L'effet s'applique uniquement au calcul de la lumière. Ainsi, si vous regardez cette sphère sur laquelle un effet de bump a été appliqué, vous avez l'impression que sa surface est rebondie. Pourtant, si vous examinez le pourtour de la sphère, vous constatez qu'il s'agit d'un cercle parfait. La géométrie de la surface n'est pas modifiée avec le bump. Le bump est juste un trompe l'œil basé sur la variation de luminosité.



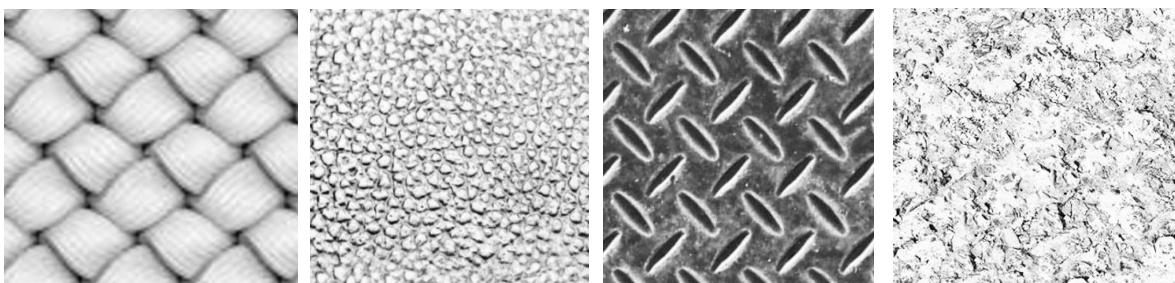


Voici un mur sorti tout droit de Halo. L'effet de bump fait apparaître sur la gauche des briques avec une surface granitée. Pourtant, si votre personnage se colle contre le mur comme sur l'image de droite et qu'il le regarde de manière rasante, les choses changent. L'effet granité est encore présent, mais l'effet -bords bisotés- des briques est aplati. Si vous mettiez la caméra contre le mur, vous véreriez une surface totalement plate avec une texture de brique.

La bumpmap correspond à une texture en niveaux de gris. En effet, nous allons stocker dans une texture 2D uniquement l'information sur la variation de hauteur :



Le bump et la notion de texture s'associe bien. En effet, la bumpmap décrit une sorte de texture de surface : bosselée, rainurée, métalisée, granitée, velours... Il semble naturel que l'on puisse utiliser la bumpmap sur tous les objets ayant la même texture de surface :



laine

cuir

métal

béton

Rappel

La **dérivée partielle** de $f(u,v)$ par rapport à u se calcule comme la dérivée de $f(u)$ en considérant que v est une constante. Ainsi, lorsque $f(u,v) = 2.u.v^3$, nous avons $\frac{\partial f(u,v)}{\partial u} = 2v^3$ et $\frac{\partial f(u,v)}{\partial v} = 6uv^2$.

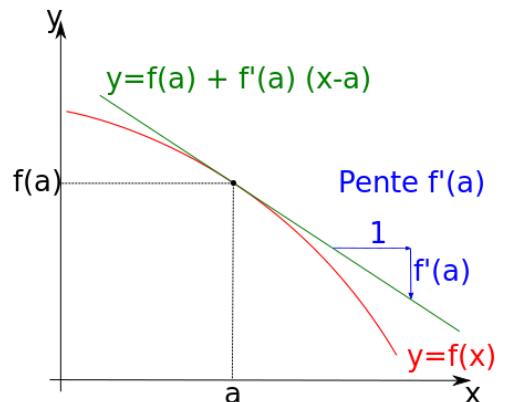
Le **produit vectoriel** entre deux vecteurs permet de construire un nouveau vecteur perpendiculaire aux deux premiers. Voici la formule de calcul :

$$\begin{vmatrix} a \\ b \\ c \end{vmatrix} \wedge \begin{vmatrix} e \\ f \\ g \end{vmatrix} = \begin{vmatrix} bg - fc \\ ce - ag \\ af - be \end{vmatrix} \quad \text{Ainsi, nous avons : } \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} \wedge \begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix} = \begin{vmatrix} 0 \\ 0 \\ 1 \end{vmatrix}$$

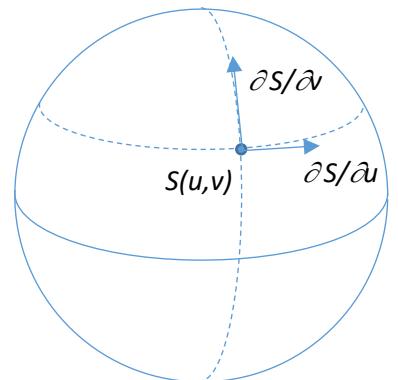
Dérivée partielle d'une surface paramétrique par rapport au paramètre u et au paramètre v :

Lorsque $M(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix}$ Nous avons $\frac{\partial M(u,v)}{\partial u} = \begin{bmatrix} \partial x(u,v)/\partial u \\ \partial y(u,v)/\partial u \\ \partial z(u,v)/\partial u \end{bmatrix}$

L'interprétation géométrique est proche de celle de la dérivée en 2D. Ainsi la valeur de la dérivée $f'(a)$ d'une fonction f nous donne la pente de la droite tangente au point $(a,f(a))$ de la courbe :



En 3D, le vecteur des dérivées partielles $\partial S/\partial u(u,v)$ correspond à la direction d'une droite tangente au point $S(u,v)$. Cette direction s'obtient en faisant varier u de manière infinitésimale. La construction de $\partial S/\partial v(u,v)$ est similaire.



Mise en place

La texture de bump nous fournit une information de variation de hauteur : $h(u,v)$. Les zones sombres correspondent aux creux et les zones claires aux bosses. Nous définissons le point M' de la nouvelle surface ainsi :

$$M'(u,v) = M(u,v) + h(u,v) \times N(u,v)$$

Ainsi, le point $M'(u,v)$ à la surface du nouvel objet s'obtient en prenant le point $M(u,v)$ et en le déplaçant dans la direction de sa normale $N(u,v)$ d'une distance $h(u,v)$. Par définition, nous rappelons que la normale d'une surface paramétrique est égale à :

$$N(u,v) = \pm \frac{\frac{\partial M(u,v)}{\partial u} \wedge \frac{\partial M(u,v)}{\partial v}}{\left\| \frac{\partial M(u,v)}{\partial u} \wedge \frac{\partial M(u,v)}{\partial v} \right\|}$$

L'interprétation de cette formule n'est pas si compliquée. Le \pm indique qu'il faut adapter le signe pour obtenir une normale orientée vers l'extérieur. Le dénominateur de la fraction sert uniquement à normaliser le vecteur final. Le numérateur correspond à un vecteur résultat du produit vectoriel \wedge entre deux vecteurs. Nous avons vu dans les rappels que chacune des dérivées partielles nous permet d'obtenir une direction tangente au point courant. Ainsi, en effectuant un produit vectoriel entre ces deux directions tangentes, nous obtenons un vecteur normal à la surface au point courant.

Nous devons maintenant déterminer $N'(u,v)$ la normale de la surface « bumpée », c'est cette information qui nous intéresse car à aucun moment nous utiliserons la valeur de M' dans notre programme. Par définition et application des règles de calcul nous obtenons :

$$\frac{\partial M'(u,v)}{\partial u} = \frac{\partial M(u,v)}{\partial u} + \frac{\partial h(u,v)}{\partial u} \times N(u,v) + h(u,v) \times \frac{\partial N(u,v)}{\partial u}$$

Attention, à ce niveau, la grandeur $\partial h / \partial u$ est un réel multiplié à un vecteur. Par contre, $\partial N / \partial u$ est un vecteur multiplié à un réel, ce qui donne aussi un vecteur, donc, tout va bien. Le chercheur inventeur du BumpMap (Blinn, J., "Simulation of Wrinkled Surfaces", Proceedings SIGGRAPH 1978) appliqua une approximation très intéressante. En effet, il remarqua que le Bump correspondant à des effets d'hauteur très faibles par rapport à la taille des objets. Ainsi, le 3^{ème} terme dans l'équation ci-dessus peut être ignoré car le paramètre $h(u,v)$ est une quantité très faible par rapport aux autres vecteurs. Ainsi nous avons :

$$\frac{\partial M'(u,v)}{\partial u} \sim \frac{\partial M(u,v)}{\partial u} + \frac{\partial h(u,v)}{\partial u} \times N(u,v)$$

Vous devez effectuer un produit vectoriel entre $\partial M'/\partial u$ et $\partial M'/\partial v$:

$$\frac{\partial M'(u,v)}{\partial u} \wedge \frac{\partial M'(u,v)}{\partial v} \sim \frac{\partial M(u,v)}{\partial u} \wedge \frac{\partial M(u,v)}{\partial v} + T2 + T3 + \dots N(u,v) \wedge N(u,v)$$

Vous identifieriez les termes T2 et T3 correspondant que vous réinjecterez dans la formule finale du BUMP. Le 4^{ème} terme est égal au produit vectoriel de deux vecteurs colinéaires, il est donc nul et peut

être supprimé. Le premier terme, si vous l'examinez, doit vous rappeler la définition du vecteur normal de la surface d'origine. Il faut maintenant intégrer le facteur K qui est l'intensité du Bump. Ce facteur permet de régler la puissance de l'effet de Bump comme un potentiomètre. Ainsi, nous simplifions l'ensemble des équations comme suit et nous obtenons la **formule finale du BUMP** :

$$N'(\mathbf{u}, \mathbf{v}) = N(\mathbf{u}, \mathbf{v}) + K \cdot [T2 + T3]$$

Les grandeurs $\partial h/\partial u$ et $\partial h/\partial v$ sont calculées grâce à la fonction Bump de la classe Texture. Les grandeurs $\partial M/\partial u$ et $\partial M/\partial v$ sont à calculer à la main et à intégrer dans le programme. Le facteur K est à déterminer pour que l'effet soit saisissant ! Voici le résultat obtenu :



Attention, l'opérateur $^{\wedge}$ est moins prioritaire que les autres. Ainsi l'expression : $A^{\wedge}B+D^{\wedge}C$ ne produira pas le résultat que vous attendez. Pensez à parenthésier correctement.

11. Les rectangles

Pour votre projet, il faut construire une scène à partir des surfaces paramétriques dont nous disposons. Nous allons donc rajouter les objets rectangles dans le code. Un objet rectangle est un objet rectangulaire plat mais posé dans la scène 3D. Pour vous simplifier le travail, nous vous conseillons d'utiliser une formulation similaire aux surfaces paramétriques des sphères. Lorsque les paramètres u et v varient entre 0 et 1, $R(u,v)$ décrit le rectangle défini par les trois points de l'espace A, B et C :

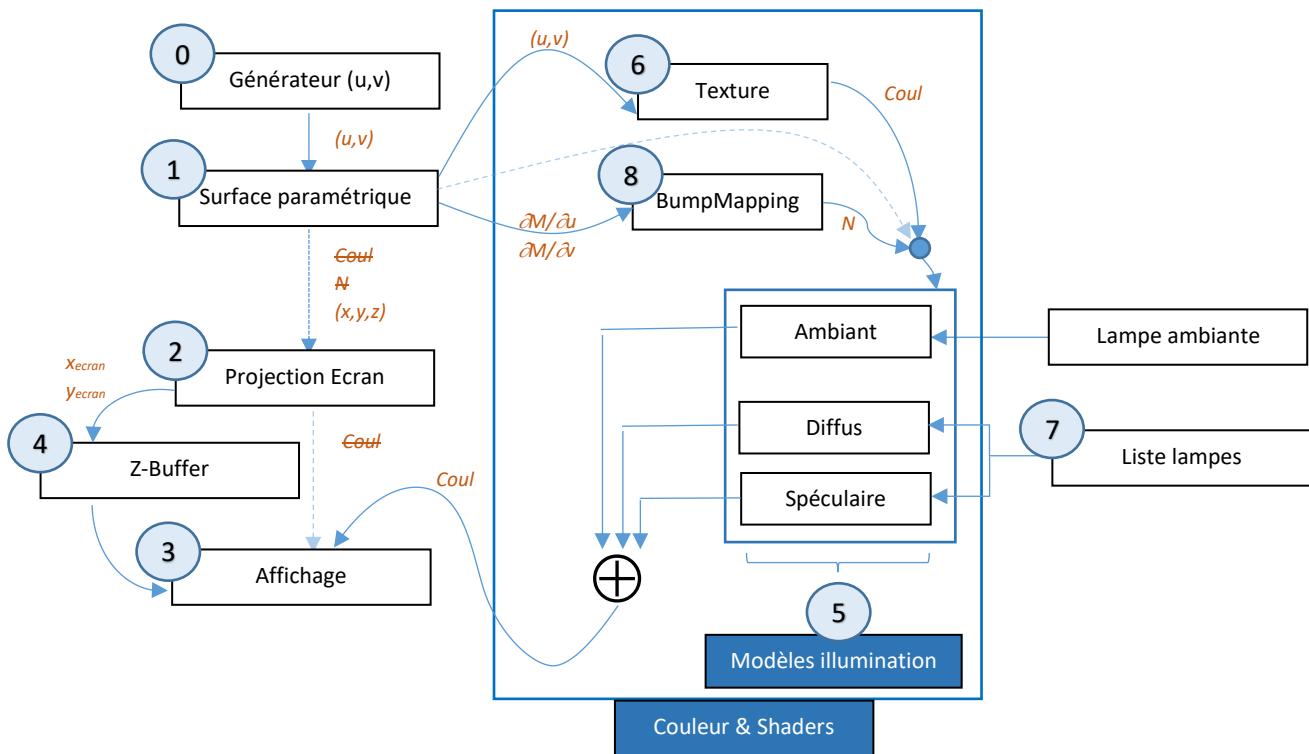


$$R(u, v) = A + u\overrightarrow{AB} + v\overrightarrow{AC}$$

Vous devez intégrer dans votre projet une classe représentant les rectangles. Comme vous devez gérer une liste d'objets 3D représentant la scène, vous devez regrouper les classes Rectangle et Sphere dans une même hiérarchie. Utiliser le polymorphisme peut vous aider à simplifier et structurer votre code.

12. Flux des traitements

Voici un résumé de la chaîne de traitement à mettre en place.



13. Evaluation

Félicitations ! En arrivant ici, vous avez obtenu la moyenne ! Si vous avez réussi à implémenter chaque partie, vous obtiendrez une note entre 10 et 12 suivant la qualité graphique de la scène.

Il reste de nombreuses options à implémenter. Je vous laisse découvrir la suite pour les plus téméraires.

14. RayCasting

Historique

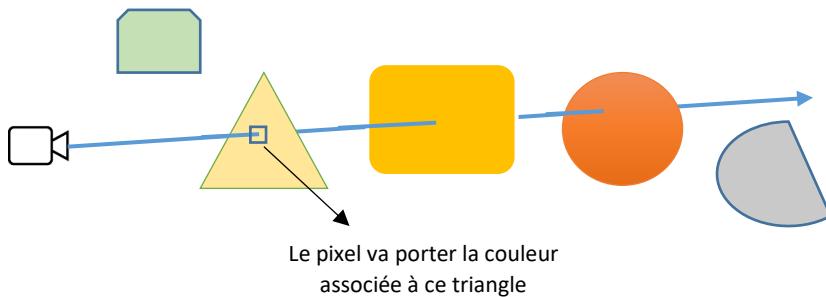
Le premier jeu ayant réussi à mettre en place cette technique fut Wolfenstein 3D grâce au moteur développé par ID Software dans les années 1990. Ensuite, DOOM I utilisa aussi ce moteur 3D assez révolutionnaire pour l'époque. Ces deux jeux lancèrent l'ère du FPS. Ces jeux devaient gérer des décors basés sur des murs et des salles dans lesquelles évoluaient divers ennemis. Il fallait donc gérer les problèmes d'occultation. L'algorithme du peintre devenait trop difficile à mettre en place et le RayCasting apportait une solution performante. Cette solution disparut dans la foulée car les GPU virent le jour et offrirent des solutions basées sur le Z-buffer temps réel. Cependant, les jeux actuels tirent quasiment le maximum de ce que peut apporter la technologie du Z-Buffer et cette approche va finir par être dépassée par des méthodes de lancée de rayons temps réel dont le RayCasting est le point de départ.

Avantages de la méthode du RayCasting :

- Gestion de la profondeur
 - o Supprime le ZBuffer
 - o Sans nécessité de calculer un ordonnancement pour l'affichage
- Gestion de la perspective
 - o Les formes plus éloignées apparaissent naturellement plus petites
- Gestion facilité de la transparence
- Pas forcément plus coûteuse qu'un Z-Buffer

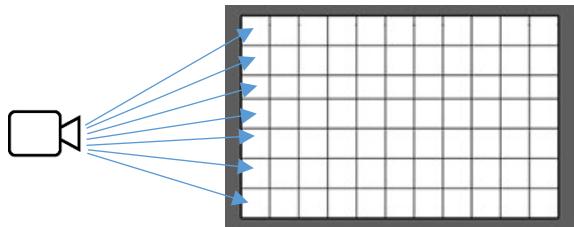
Principe

Le RayCasting signifie simplement « lancer un rayon » depuis la position de la caméra. Ainsi le premier objet traversé par ce rayon correspond à l'objet qui doit être affiché à l'écran. Il faut cependant tester l'intersection de ce rayon avec l'ensemble des objets de la scène, détecter s'il y a une intersection et dans ce cas conserver l'intersection la plus proche pour passage à l'affichage :



Pour cette opération, il est intéressant de tester l'intersection pour chaque objet de la scène avec le rayon. Une méthode polymorphe peut être une bonne approche.

Nous avons, lors de l'implémentation du spéculaire, donné une position à la caméra à l'intérieur de la scène. Pour dessiner les couleurs à l'écran, la méthode du RayCasting va lancer pour chaque pixel de l'écran un rayon qui part de la caméra et qui traverse ce pixel. Le résultat obtenu sert à colorer le pixel en question.

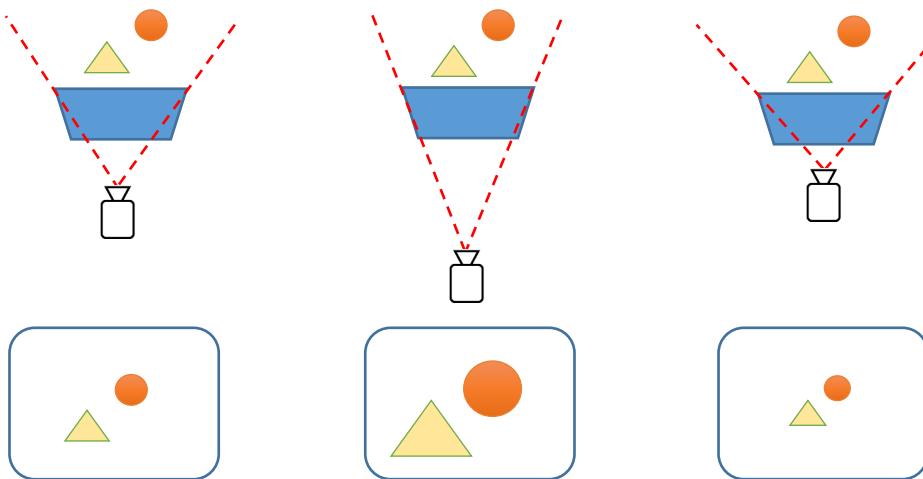


La mise en place réside dans une double boucle servant à parcourir l'ensemble des pixels de l'écran :

```
for (int x_ecran = 0 ; x_ecran <= largeur_ecran ; x_ecran++)
for (int y_ecran = 0 ; y_ecran <= hauteur_ecran ; y_ecran++)
{
    P3 PosPixScene = new P3(x_ecran,0,y_ecran);
    P3 DirRayon = PosPixScene - PosCamera;
    Couleur C = RayCast(PosCamera, DirRayon, ListObjetsScene);
    Affiche(x_ecran,y_ecran,C);
}
```

Un peu d'optique

S'il semble naturel d'aligner la caméra avec le milieu de l'écran, on peut hésiter sur la valeur de l'éloignement de la caméra par rapport au plan de l'écran. Le RayCast étant basé sur une méthode de lancer de rayon, l'angle de vision de l'image générée va dépendre de cette position, mais comment ?



Pour visualiser le cône de vision de la caméra, il suffit de lancer 4 rayons partant de la caméra et joignant chaque coin de l'écran. Si l'on se rapproche de l'écran, le cône s'ouvre et s'élargit. Si l'on recule, il se rétrécit. Cette situation se retrouve lorsque vous regardez à travers une fenêtre. Si vous êtes proche de la vitre, vous voyez à 160° l'extérieur. Si vous reculez à l'intérieur de la pièce, vous voyez juste l'immeuble ou l'arbre en face. On a cependant l'impression qu'en se rapprochant de la fenêtre, on va voir plus grand car l'angle de vision augmente ! Non. Car au niveau de l'ordinateur, nous travaillons à résolution fixe. Quelle que soit la position de la caméra dans la scène, le rendu final à l'écran a la même taille. Ainsi, en étant proche de la fenêtre, l'angle de vision augmente, mais les objets

rapetissent. En effet, comme le champ de vision s'agrandit et que la taille de l'image reste constante, les objets sont forcément plus petits. A l'inverse, en reculant, les objets s'agrandissent. En reculant, cela équivaut à augmenter la focale de votre appareil photo, c'est-à-dire à zoomer sur l'objet. L'angle de vision rétrécit et l'objet s'agrandit.

Intersection Rayon Sphere

Afin de mener à bien la détection de l'intersection entre un rayon et une sphère, nous vous guidons dans cette démarche. Descriptions des éléments :

Rayon : $R(t) = R_0 + t \cdot R_d$ avec $t > 0$

Sphère : Centre $C(x, y, z)$ et rayon r

Le rayon correspond à une demi-droite. Son origine R_0 est située sur la caméra. Le paramètre t est positif car le rayon est dirigé vers l'écran suivant la direction R_d . Nous cherchons l'intersection entre ce rayon et la sphère, nous devons donc satisfaire l'équation suivante :

$$(R_0 + tR_d - C)^2 = r^2$$

En développant cette égalité, nous obtenons une équation du second degré de la forme : $At^2 + Bt + D = 0$. Le déterminant $\Delta = B^2 - 4AD$ nous indique s'il existe une intersection. Lorsque $\Delta > 0$, deux intersections sont possibles :

$$t_1 = (-B - \sqrt{\Delta}) / 2A \quad \text{et} \quad t_2 = (-B + \sqrt{\Delta}) / 2A$$

A ce niveau, il ne faut pas oublier la contrainte $t > 0$, ainsi on ne s'intéresse qu'aux intersections présentes devant la caméra et s'il y en a plusieurs, à la plus proche :

- $t_1, t_2 > 0 \Rightarrow R(t_1)$ est l'intersection la plus proche
- $t_1 < 0, t_2 > 0 \Rightarrow R(t_2)$ est la seule intersection devant la caméra
- $t_1, t_2 < 0 \Rightarrow$ aucune intersection visible

Une fois le point d'intersection déterminé, il faut retrouver les paramètres (u, v) associés à ce point afin de retrouver les autres paramètres : la couleur de texture, la normale, les valeurs de bump... Pour cela, nous avons mis dans le projet la fonction InvertCoordSpherique qui vous permettra d'effectuer cette opération.

Intersection Rayon Rectangle

Afin d'éviter de lourds calculs, nous vous présentons une approche simple permettant de résoudre cette question. Nous allons d'abord résoudre l'intersection entre un rayon et le plan porté par ABC :

Rayon : $R(t) = R_0 + t \cdot R_d$ avec $t > 0$

Plan : $A + \alpha AB + \beta AC$

L'équation à résoudre est la suivante :

$$R_0 + t \cdot R_d = A + \alpha AB + \beta AC$$

Nous utilisons la normale au plan : $n = AB \wedge AC / \|AB \wedge AC\|$

En appliquant un produit scalaire sur l'équation précédente, on obtient :

$$R_0 \cdot n + t \cdot R_d \cdot n = A \cdot n + \alpha AB \cdot n + \beta AC \cdot n$$

Or AB et AC étant perpendiculaires à la normale n , nous avons finalement :

$$t = \frac{R_0 \cdot A \cdot n}{R_d \cdot n}$$

Ainsi, on déduit le point d'intersection $I = R_0 + tR_d$

Il nous reste maintenant à retrouver les paramètres α et β . Nous savons :

$$\alpha AB + \beta AC = AI$$

Comme nous travaillons avec des rectangles, AB et AC sont perpendiculaires. Ainsi, en utilisant un produit scalaire, nous simplifions cette équation :

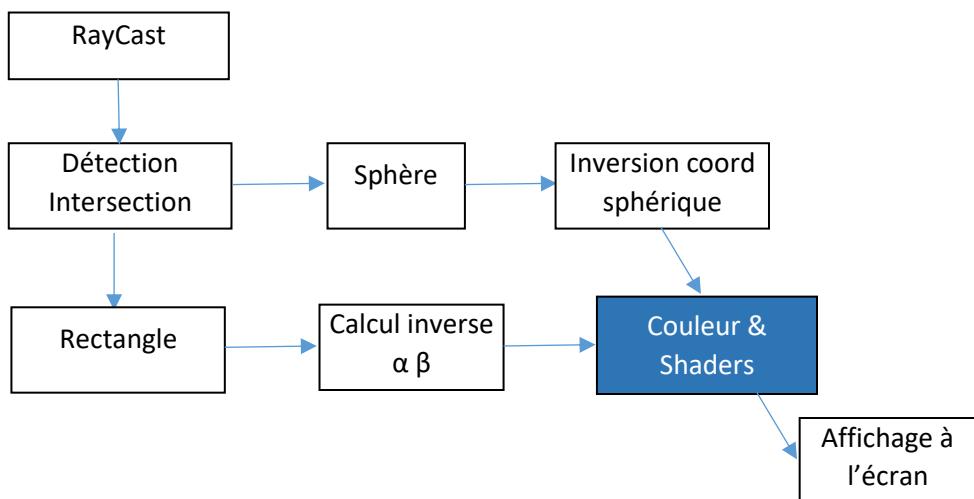
$$\alpha AB \cdot AB = AI \cdot AB$$

Finalement, nous obtenons pour valeur de α :

$$\alpha = \frac{AI \cdot AB}{AB \cdot AB}$$

En dernière étape, il nous reste à vérifier que le rayon traverse le rectangle. Il suffit pour cela de vérifier que $0 \leq \alpha \leq 1$ et que $0 \leq \beta \leq 1$.

Diagramme



15. Structuration

Ce projet de synthèse d'images est conséquent, il peut facilement atteindre plusieurs centaines de lignes pour ceux qui iront jusqu'au bout des défis. On n'aborde pas un projet important comme un

code d'une « centaine de lignes pour tester un truc ». On peut penser pouvoir le faire, mais, c'est un doux rêve.

Différents niveaux de structuration du logiciel

1. Noms de variables significatifs
2. Une chose = une variable
3. Duplication de code => écriture d'une fonction
4. Noms de fonctions significatifs
5. Une action = une fonction
6. Logique de traitement claire avec un minimum de détours
7. Gestion des dépendances
8. Commentaires / Documentation
9. Gestion des exceptions
10. Tests unitaires
11. Recette (VABF : vérification d'aptitude au bon fonctionnement)
12. Déploiement (VSR : vérification service régulier)
13. Maintenance (TMA : tierce maintenance applicative)

Voici les différents niveaux qui permettent de réaliser un « bon » logiciel. Pour la réalisation de ce projet, nous allons vous demander d'essayer d'atteindre le niveau 5, voir le 6 pour les meilleurs. Pour la plupart d'entre vous, cette mission s'avère anecdotique. En général, durant votre vie d'étudiant et/ou d'informaticien, vous avez principalement rédigé du code pour vous seul, sur des périodes courtes et pour des projets sans mise en exploitation.

La vie professionnelle s'éloigne de ce mode de production. En effet, vous travaillez en équipe, c'est-à-dire que vous utilisez le code des autres et que les autres vont aussi utiliser votre code. Vous avez tous déjà vu/fait/utiliser du mauvais code, vous savez ce que c'est et vous avez perdu suffisamment de temps à essayer de le faire fonctionner. Vous travaillerez sur des projets longs où il faudra reprendre le code un an après. Un code mal rédigé, pas structuré, où l'on ne sait pas ce qu'il se passe, fait perdre un temps conséquent pour faire le moindre ajout/modification. Votre logiciel est cette fois destiné à être exploité sur les plateformes de vos clients/utilisateurs. Les problèmes vont remonter, le code tortueux, mal structuré va se transformer en véritable labyrinthe et il viendra à bout de votre motivation et parfois à bout du projet.

Ne fantasmer pas le : « je structurerai quand j'aurais fini ». Il ne faut pas remettre l'organisation à plus tard car « plus tard » on n'aura toujours autre chose à faire, et plus tard signifiera jamais. Il en est de même de l'attitude, il faut tout jeter et réécrire le programme, si le premier essai s'est fait sans structuration, il en sera généralement de même pour le second.

Vitesse de codage

	TP	POC	Petit projet	Utilitaire	Petit logiciel
Lignes de code	100	500	1 500	4 000	10 000
Contexte	langage connu	structuration fonctionnelle	Utilisation de lib tierces	Structuration classes	Ecriture de tests unitaires
Vitesse	50 lignes / h	40 lignes / h	30 lignes / h	20 lignes / h	10 lignes / h
Temps requis	2h	12h	1 semaine	1 mois	6 mois

Voici un tableau qui décrit le ralentissement du codage en fonction du niveau d'exigence requis pour arriver à la fin du projet. On peut rêver à réaliser un petit logiciel à la vitesse de croisière d'un petit projet, on mettrait alors 1 mois au lieu des 6 nécessaires, simple non ! Cependant, ce scénario n'est pas possible. Se fixer un objectif sans choisir la qualité de code qui va avec, cela signifie se mentir à soi-même et rater le projet. C'est comme vouloir monter en haut de l'Everest en basket et les mains dans les poches... c'est dangereux et mortel pour le projet.

Chaque année, le niveau de structuration, de clarté et de lisibilité du projet permet aux étudiants d'atteindre les questions supérieures. La corrélation entre la note finale et la structuration est très forte, non pas que j'évalue directement la structure du projet rendu, mais parce que la non structuration du projet va entraîner tellement de soucis pour les étudiants qu'au bout d'un certains nombres de lignes, le projet devra s'arrêter de lui-même, le code devenant trop lourd à faire évoluer.

Pour aller plus loin : vous pouvez lire l'ouvrage « coder proprement » de Robert C. Martin qui constitue un recueil simple et efficace des principaux réflexes à adopter pour mener à bien de longs projets.

16. Ombres et occultations

Si le chapitre sur la structuration se trouve juste avant le chapitre sur les ombres, c'est loin d'être un hasard. En effet, la mise en place des occultations est normalement sans création de lignes de code supplémentaires car elle n'est qu'une réutilisation de la méthode de RayCast. Cependant... suivant l'état de structuration du projet à ce niveau, ce sera pour certains un dur choix à faire :

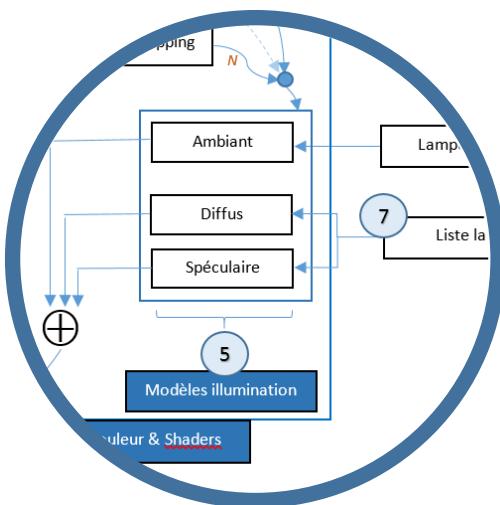
- laisser tomber car trop de travail pour restructurer
- impossible de continuer à coder à la vitesse lumière, on casse les murs et on refait l'entrée
- j'avais déjà commencé à structurer, j'ai quelques adaptations à faire

Principe

Lors du calcul des modèles d'illumination nous parcourions la liste des lampes et calculions la valeur de diffus et de spéculaire pour chaque lampe.

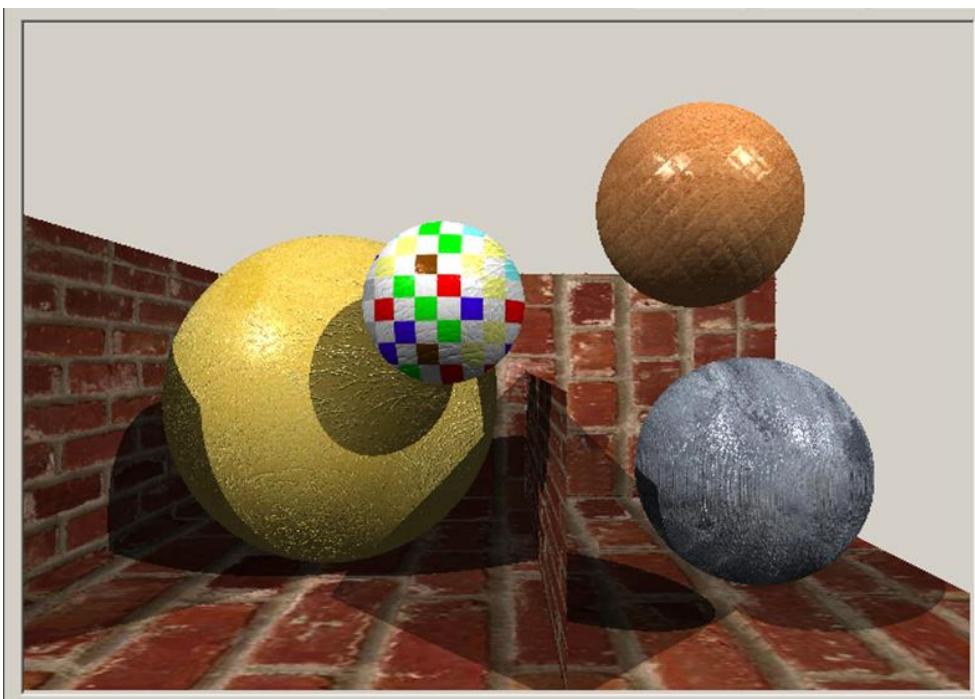
Pour gérer l'occultation, il faut se poser la question suivante : existe-t-il un objet entre le point courant et la source de lumière qui empêche le rayon lumineux de parvenir à la surface de l'objet ?

Ce problème peut être résolu facilement en réutilisant l'algorithme du RayCasting. En effet, il suffit pour répondre à cette question de lancer un rayon depuis le point courant dans la direction de la source lumineuse. Si une intersection est détectée avec un objet présent



le point courant et la source, alors, il y a occultation et la lampe ne peut contribuer au calcul du diffus et du spéculaire du point courant.

Résultat



Dès que les ombres sont présentes dans une scène l'effet de réalisme est plus impressionnant. Remarquez que les ombres ne sont pas noires à 100%, cela est dû à l'utilisation de la lampe ambiante.

17. Méthodes avancées

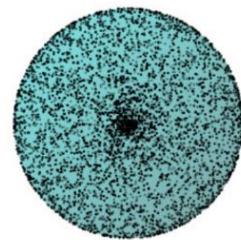
A ce niveau, vous avez parcouru l'essentiel du projet. Nous vous offrons l'opportunité d'intégrer **une** dernière option à choisir parmi plusieurs suivant vos affinités. Les deux premières options VPL et PathTracer doivent être considérées en premier choix car elles représentent la pointe des technologies actuelles et sont donc à la mode. Les autres options sont amusantes et sont données à titre indicatif.

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 :

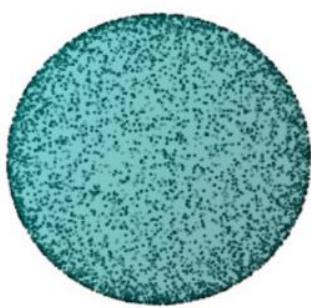


Vue de face

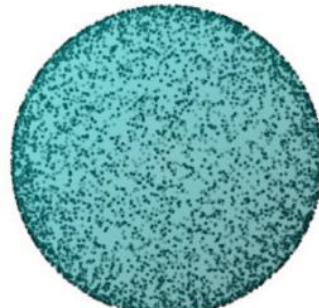


Vue de dessus

En effet, le problème se situe près des pôles. Si l'on prend la zone $[0^\circ, 360^\circ] \times [85^\circ, 90^\circ]$ (le pôle nord) comparée à la zone $[0^\circ, 360^\circ] \times [0^\circ, 5^\circ]$ (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 aimerais 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 :

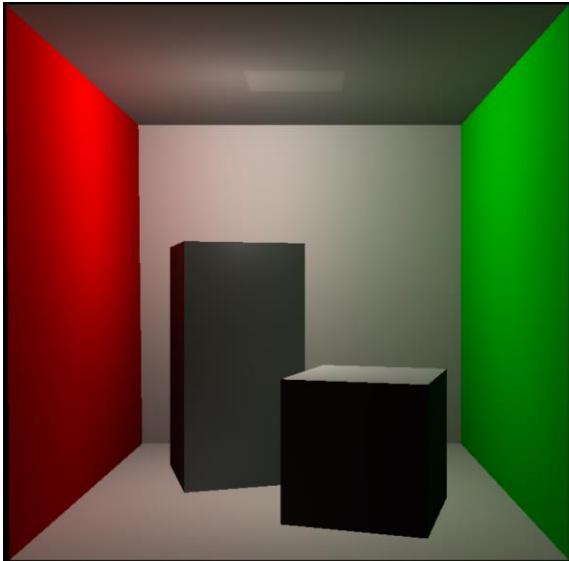
```
for(int i = 0 ; i < nb ; i++)
{
    double theta = 2*PI*rand(0.0,1.0);
    double phi = acos(2*rand(0.0,1.0)-1.0);
    x[i] = cos(theta)*sin(phi);
    y[i] = sin(theta)*sin(phi);
    z[i] = cos(phi);
}
```

VPL

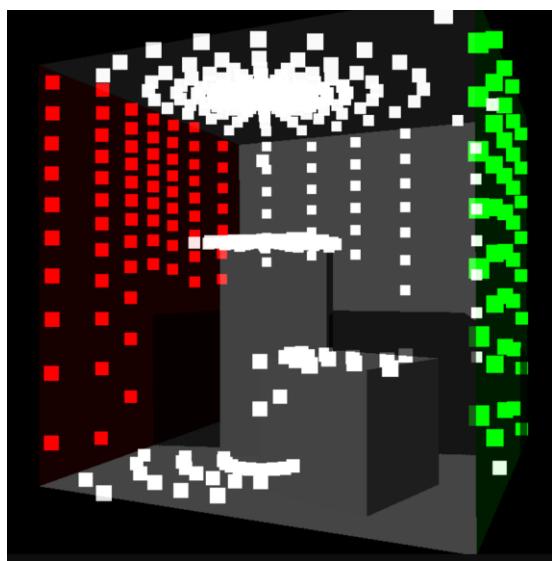
La méthode des Virtual Point Light permet de simuler des rayons lumineux de niveau 2 et ceci à moindre coût. Les modifications à apporter à votre code ne sont pas forcément très lourdes car il s'agit d'une légère variation du moteur actuel. Cependant, l'augmentation de la quantité de calcul va vous pousser à optimiser le code de manière drastique, ce sera l'occasion d'utiliser le profiler de Visual Studio.

L'idée est la suivante, nous allons prendre la source lumineuse principale et allons lancer 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. En production, on peut aller jusqu'à la création de 100 000 nouvelles lampes.

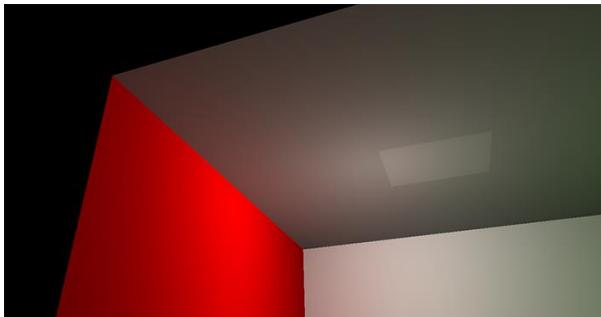
Dans une deuxième passe, nous reprenons le moteur précédent et nous lançons le calcul de la scène en utilisant les lampes principales et les VPL annexes. Nous utilisons uniquement les modèles d'illumination de niveau 1 habituels. Les VPS simulant des rayons de niveau 1, nous allons ainsi arriver à simuler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

Le modèle d'illumination diffus de niveau 1 est très simple. En effet, on ne considère que la contribution directe des rayons lumineux des lampes sur l'intersection alors que 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

CouleurTotale = Couleur(0,0,0)

Pour i = 1 à n

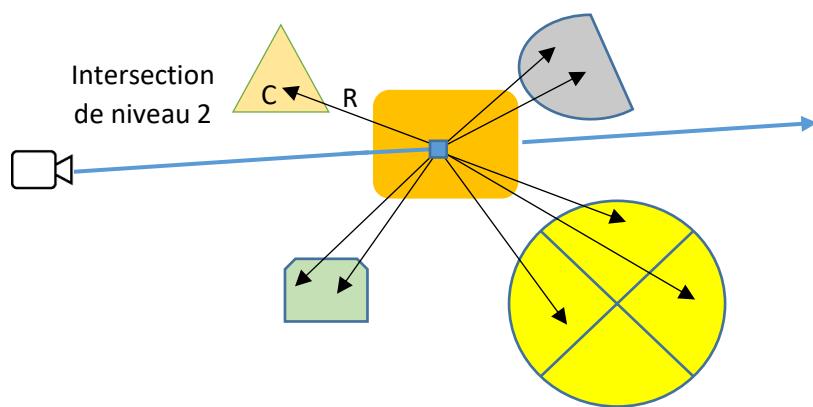
Lancer un rayon R dans une direction aléatoire

Calculer la couleur C de l'intersection avec un objet (diff/spec niv 1) ou avec une lampe

CouleurTotale += (N.R)*C

CouleurTotale /= n

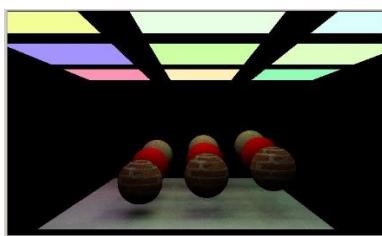
Afficher CouleurTotale



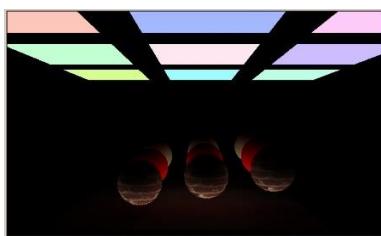
Dans le modèle du PathTracing, les lampes doivent être modélisées comme des objets de la scène (par une sphère ou un rectangle). L'idée ici est que les sources de lumière ont une surface et plus elles sont grandes, plus elles éclairent la scène.

Dans notre approche, nous calculons la couleur de l'intersection de niveau 2 à partir des modèles d'illumination de niveau 1. En production, la méthode du PathTracing est en fait récursive et les rebonds peuvent être multiples jusqu'à toucher une source. Cependant, ils utilisent la puissance de calcul de plusieurs GPU et peuvent donc se permettre cette approche.

Exemple de notre méthode :



Diffus Niv2 des lampes

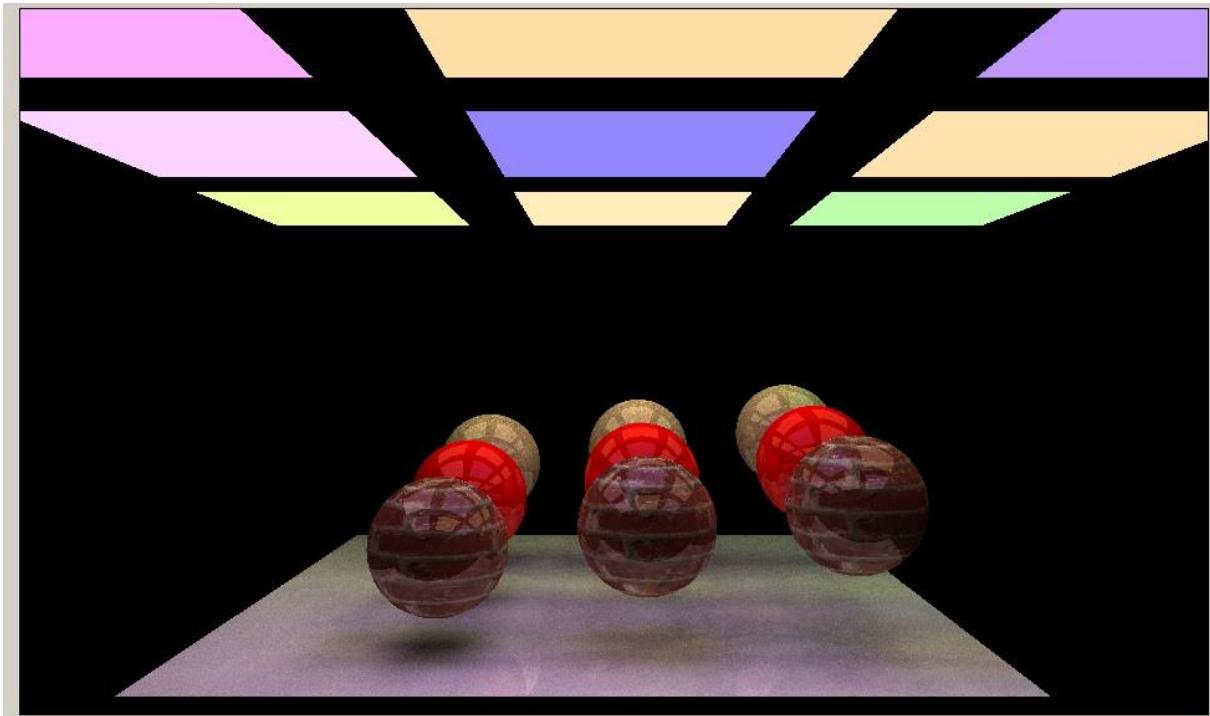


Diffus Niv2 des objets



Spéculaires lampes+objets

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.

Mesh renderer

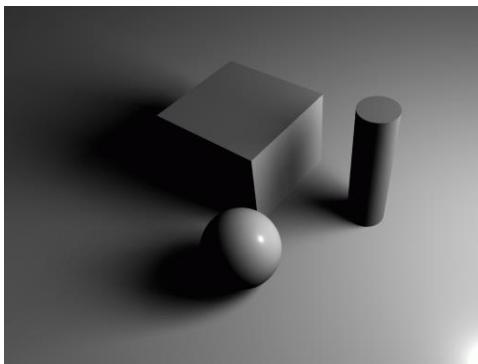
Vous pouvez intégrer dans votre projet un lecteur d'objets 3D. Nous vous conseillons fortement le format historique OBJ qui contient des géométries de triangles et de textures. Il est standardisé et relativement simple et bien documenté. Il est cependant difficile de trouver des objets complets (mesh+fichiers texture) et Low Poly, mais trouver un modèle 3D complet et fonctionnel est souvent compliqué.

Pour parvenir à intégrer des objets externes, il va falloir créer un nouveau type d'objets dans votre moteur : les triangles. Il s'agit ici d'une variation des rectangles :

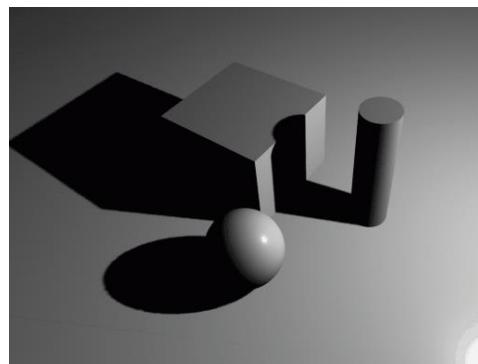
Triangle ABC : $T(u,v) = A + uAB + vAC$ tel que $0 \leq u+v \leq 1$

Il faudra faire attention à vos calculs inverses car cette fois les arêtes AB et AC ne sont plus forcément perpendiculaires.

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 complexes, il suffit lors du test d'occultation de faire tilt le vecteur de quelques degrés. En tirant plusieurs rayons, une dizaine, une centaine et en augmentant l'angle de tilt, on obtient des ombres et des transitions plus douces, c'est l'algorithme dit des soft shadows. Comptez 20° maximum pour produire l'effet escompté.

Surface de révolution

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

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

Nous pourrions travailler comme pour la sphère, mais l'étape d'inversion des coordonnées sera difficile à résoudre. Le plus simple consiste à échantillonner 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 rectangle à la surface du vase !

Raytracer

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

$$Coul_{Intersection} = Coul_{Diff/Spec\ Niv\ 1} + Coul_{Réflexion} + Coul_{Réfraction}$$

Ainsi un rayon qui heurte une surface nécessite un calcul de Diff/Spec de niveau 1 habituel puis génère 2 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. La direction du rayon réfracté se déduit à partir de la loi de Fresnel : $n_1 \sin i_1 = n_2 \sin i_2$.

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 met une borne sur la profondeur maximale de la récursivité égale à 5 ou 6 ce qui est largement suffisant.

18. Best-Of Images Raytracers

Pour rendre hommage aux générations d'ESIEEens s'étant illustrés dans le projet de synthèse d'image (ancienne version raytracer), voici quelques exemples des créations :

