Modélisation et Visualisation du solide

Vincent Feuvrier

Master M2IM
Année 2008—2009

Introduction

Mes coordonnées : vincent.feuvrier@normalesup.org, et l'adresse du site du cours : http://www.math.u-psud.fr/~feuvrier/enseignement/2008/M2. On pourra y trouver entre autres les sujets des travaux pratiques.

Le cours consiste en une introduction à certaines techniques utilisées en milieu industriel pour la modélisation et la représentation planaire informatisée de scènes tridimensionnelles. Les sujets traités sont les suivants.

La première partie traite des images digitales, et des problèmes de représentation qui y sont liés. En particulier on étudie les phénomènes d'aliasing, leur interprétation spectrale et le théorème d'échantillonnage de Nyquist-Shannon.

La deuxième partie présente quelques bases de l'infographie. On aborde la notion de projection perspective et sa représentation efficace par les applications projectives avant d'introduire les deux principales méthodes de rendu qui seront abordées (OpenGL et le lancer de rayon).

La troisième partie présente plus en détails l'utilisation de la librairie OpenGL.

La quatrième partie traite des splines, B-splines et NURBS, utilisés pour modéliser des courbes et des surfaces lisses.

Enfin la cinquième et dernière partie constitue une compilation de diverses techniques générales pour la visualisation : les modèles d'éclairement, les diverses optimisations du rendu, l'utilisation des textures, etc...

Un grand merci à Pierre Pansu pour avoir rédigé certaines parties qui concernent les splines et la perspective.

Première partie

Images digitales

Contenu

1 Colorimétrie

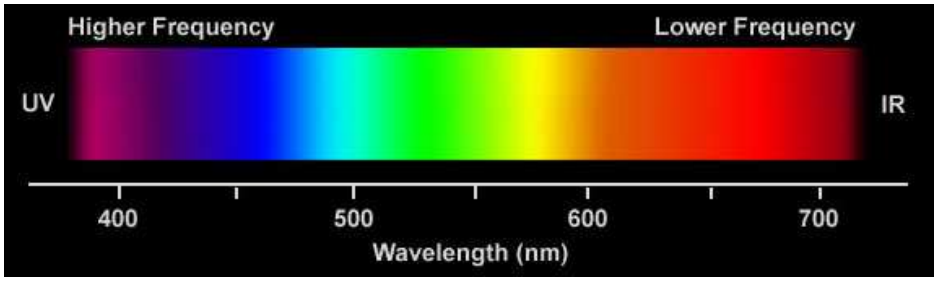
1.1 Perception de la couleur

La lumière est un phénomène électromagnétique ondulatoire, se propageant dans l'espace et le temps depuis une source matérielle émettrice. Toute onde lumineuse pure a une longueur d'onde associée (la distance qu'elle parcourt en une seconde) ; dans la nature le rayonnement lumineux est en général un mélange de ces longueurs d'ondes.

On peut distinguer essentiellement deux types de sources matérielles :
- les sources primaires, qui produisent elles-mêmes l'énergie nécessaire à l'émission du rayonnement, souvent par agitation moléculaire (soleil, flammes, lampe à incandescence, etc...) ;
- les sources secondaires, qui émettent de l'énergie en réaction à un rayonnement incident, en général par diffusion (plastique mat, peau, poussière en suspension), ou encore réfraction (eau, verre), réflection (métal, miroir), ou une combinaison des trois phénomènes.

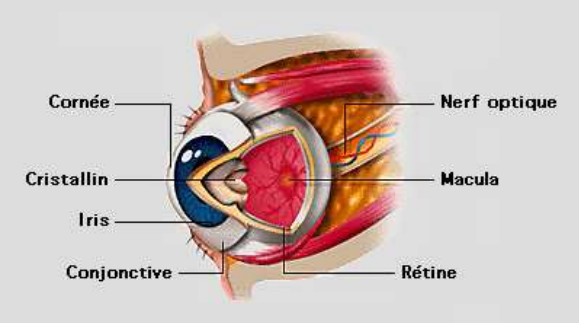
Ce qui nous intéresse c'est la perception humaine de cette diffusion, c'est à dire pas directement la source du rayonnement mais son flux, visible par l'œil.

Tout d'abord notons que le système visuel humain ne peut détecter, dans le spectre de la lumière, que des longueurs d'ondes comprises environ entre 400 `nm` (après les ultra-violets) et 700 `nm` (avant les infra-rouges). Le système visuel perçoit cet intervalle de fréquences d'ondes lumineuses comme un arc-en-ciel de couleurs variant continûment : le spectre visible.



Le spectre visible

Lorsqu'un rayon lumineux pénètre dans l'œil, il est d'abord réfracté par des milieux transparents d'indices différents, la cornée et le cristallin assurant l'accommodation de l'image en formant une lentille convergente de focale variable. C'est au moment de traverser la rétine qu'il sera absorbé par des photorécepteurs, et interprété dans le cortex cérébral dédié à la vison.

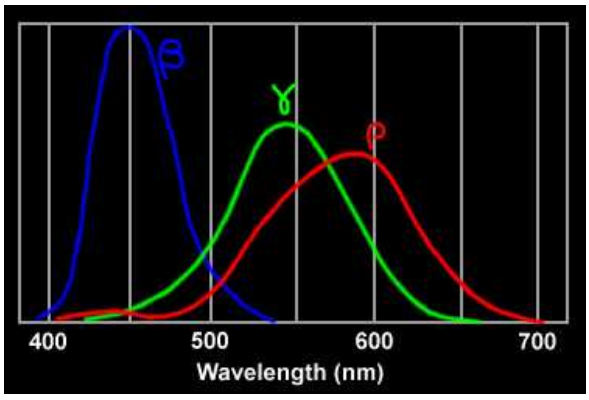


Une coupe de l'œil

La rétine est constituée d'une fine couche de cellules nerveuses, pour la plupart sensibles à la lumière, qui forment deux familles :

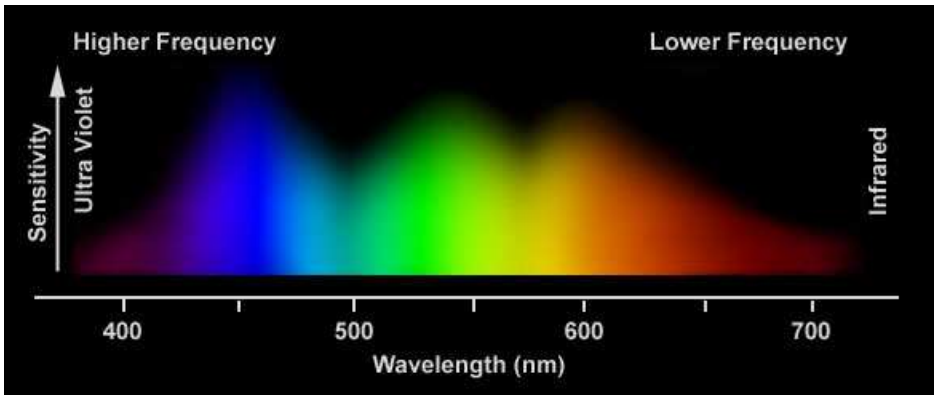
- les bâtonnets (ou *rods* en anglais), qui atteignent leur maximum de sensibilité vers 500 nm et permettent la vision nocturne (vision *scotopique*), une perception achromatique du rayonnement lumineux et de ses variations. Leur fonctionnement est assuré par un pigment protéique, la rhodopsine (ou *pourpre rétinienne*) qui blanchit à la lumière du jour et les rend insensibles pendant la journée ;
- les cônes, qui permettent une perception à la fois photométrique et chromatique de jour (vision *photopique*) grâce a des pigments protéiques (opsines) sensibles à différentes longueurs d'onde, comprises entre 400 et 700 nm.

Les cônes peuvent eux-mêmes être répartis en trois familles appelées β, γ et ρ (*Blue*, *Green* et *Red*) ou encore L , M et S (long, medium et short) dont les sensibilités spectrales maximales se situent respectivement dans le bleu (450 nm), le vert (540 nm) ou le rouge (580 nm). La réponse photométrique de ces cellules est approximativement fonction logarithmique de l'énergie lumineuse absorbée.



Sensibilité spectrale des cônes

Tout rayonnement de longueur d'onde comprise entre 400 et 700 nm va provoquer lors de son absorbtion par la rétine des réponses relatives différentes par ces trois types de cellules, ce qui est à la base de la vision des couleurs et de son aspect trichromatique.



Spectre perçu et sensibilité de l'œil

1.2 Synthèse des couleurs

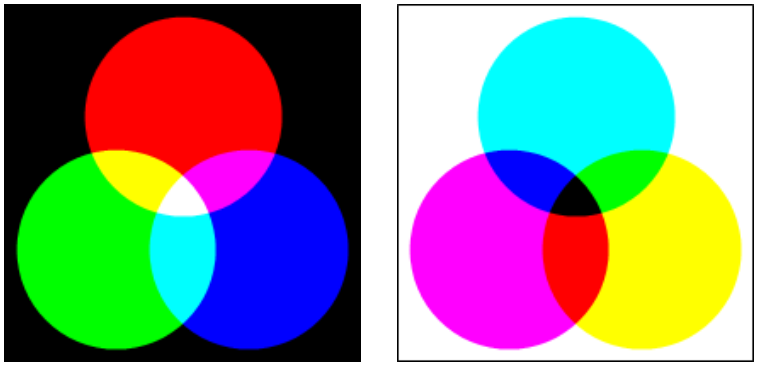
On a vu que les mécanismes de la perception des couleurs par l'œil mettaient en jeu trois types de récepteurs, chacun avec sa longueur d'onde privilégiée. En combinant les intensités de trois ondes pures de fréquences fixées et convenablement choisies, il est possible de reproduire toutes les couleurs du spectre perçu par l'œil, y compris les mélanges de longueurs d'ondes : il suffit de recomposer les intensités relatives perçues par les trois types de cônes, en adaptant l'intensité des trois composantes monochromatique choisies.

Les longueurs d'onde communément utilisées correspondent à du rouge, du vert et du bleu, proches des sensibilités maximales des cônes. Les couleurs de cette base de décomposition sont appelées les *couleurs primaires*.

Lorsqu'on veut représenter une couleur pour l'œil, on distingue deux modèles de synthèse qui dépendent du support physique d'affichage :

- la synthèse additive, utilisée dans les supports actifs émettant leur propre lumière (tubes cathodiques, projecteurs vidéos, écrans à plasma) consiste à additionner les longueurs d'ondes nécessaire à la recombinaison de la couleur originale. Par exemple du jaune sera obtenu en mélangeant du vert et du rouge, et du blanc en mélangeant les trois couleurs primaires ;
- la synthèse soustractive, utilisée dans les supports passifs qui émettent de la lumière par diffusion lorsqu'ils sont éclairés (peinture, imprimerie) consiste à utiliser une base de couleurs primaires qui émettent dans deux longueurs d'onde privilégiées. On utilise par exemple du cyan (bleu et vert), du magenta (bleu et rouge), du jaune (vert et rouge) et souvent aussi du noir pour faire varier facilement la luminosité (quadrichromie).

  Lorsqu'on mélange deux couleurs en synthèse soustractive (par exemple en mélangeant deux peintures) le spectre de diffusion résultant vaut pour chaque longueur d'onde le minimum des valeurs des spectres des deux couleurs en ce point. Ainsi, le rouge sera théoriquement obtenu en mélangeant du jaune et du magenta et le noir en mélangeant les trois couleurs primaires (le blanc est obtenu en n'utilisant aucun pigment, par défaut c'est la couleur du support de peinture).



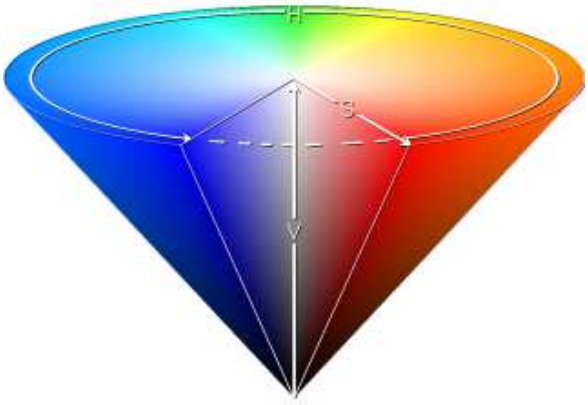
Synthèse des couleurs : additive (à gauche) et soustractive (à droite)

1.3 Espaces de couleur

Les espaces de couleurs sont généralement liés à des périphériques d'affichage (écrans, imprimantes) ou de numérisation (scanner, appareil photo). L'ICC (International Color Consortium) a été constitué en 1993 dans le but de créer un système universel de gestion des couleurs : sa création a débouché sur la mise au point de la norme ICC, qui permet de garantir la fidélité des couleurs lors du passage d'un espace de couleur à un autre.

En particulier cette norme a permis d'uniformiser le rendu des couleurs par les périphériques d'affichage. Ainsi, par exemple le format RGB (Red, Green, Blue) consiste à coder la couleur sur un triplet de trois valeurs d'intensités de rouge, de vert et de bleu comprises entre 0 et 1. Ces nombres sont appelés *canaux*. La normalisation des supports d'affichage garantit par exemple que (0,0,0) sera perçu comme le noir le plus sombre que le support peut afficher, et (1,1,1) comme le blanc le plus clair.

On peut citer plusieurs espaces de couleurs répandus :
- l'espace RVB ou RGB, qui peut être considéré comme un espace vectoriel (si on fait abstraction des bornes sur les valeurs des canaux). En effet, le fait de multiplier une couleur par 2 rend celle-ci deux fois plus intense, sans changer la teinte perçue ;
- l'espace des niveaux de gris, qui peut être lui-aussi considéré comme un espace vectoriel, possède un seul canal. Il est simplement défini à partir de l'espace RGB en attribuant les mêmes valeurs aux trois canaux ;
- l'espace CMJN (Cyan, Magenta, Jaune, Noir) ou CMYB (Cyan, Magenta, Yellow, Black) en anglais, utilisé pour l'impression quadrichromique. Puisqu'il correspond à une synthèse soustractive des couleurs, on ne peut pas le considérer comme un espace vectoriel ;
- l'espace TSV (Teinte, Saturation, Valeur) ou HSV (Hue, Saturation, Value) en anglais est une transformation non-linéaire de l'espace RGB.



Une représentation conique de l'espace TSV

- Le canal T représente une teinte comprise entre 0 et 360° ;
- le canal S de saturation est compris entre 0 et 100%, et correspond à la pureté de la couleur : plus la saturation d'une couleur est faible, plus elle apparaitra fade et proche du gris ;
- le canal V est compris entre 0 et 100%, et correspond à la brillance de la couleur : plus la valeur est faible et plus la couleur est proche du noir.

1.4 Représentation informatique de la couleur

L'espace de couleur RGB est généralement le plus couramment utilisé, tant pour le stockage de données graphiques que pour l'affichage à l'écran. Sur la plupart des systèmes, une couleur est stockée sur trois octets, correspondant respectivement à une valeur de rouge, de vert et de bleu, comprise entre 0 et 255. Certains systèmes rajoutent un quatrième octet parfois inutilisé, ou parfois utilisé pour définir une information de transparence.

En général une couleur occupe donc 24 ou 32 bits, et une image numérique est souvent stockée en mémoire comme un tableau bidimensionnel de couleurs. Sachant que les normes actuelles de résolution des écrans varient entre 1024×768 et 1280×1024, la mémoire simplement dédiée à l'affichage pour un système moderne peut donc varier entre 2 et 6 Mo.

Certains systèmes anciens d'affichage ainsi que des format d'images compressées utilisent une palette pour représenter les couleurs : ainsi, une couleur est stockée comme un indice parmi les couleurs de la palette (qui sont elles-mêmes stockées sur 24 ou 32 bits). Cet indice occupe en général entre 1 et 16 bits, et permet de diminuer la taille occupée en mémoire par un tableau de pixels (mais aussi le nombre maximum de couleurs différentes qu'il peut contenir).

2 Éléments de mathématiques et de théorie du signal

Considérons pour commencer qu'une image «réelle» est un signal bidimensionnel continu f défini sur le plan $\mathbb{R}^2$, d'énergie finie (c'est à dire de carré sommable). On verra ultérieurement comment l'étude du rayonnement lumineux (qui est un phénomène spatial) est réduite à celle de sa projection sur un plan.

Une image digitale est un échantillonnage g sur une grille finie d'un signal bidimensionnel réel $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^m$ où m dépend de l'espace vectoriel colorimétrique choisi ($\mathbb{R}$ pour une image en niveaux de gris, ou $\mathbb{R}^3$ pour une image en RVB).

2.1 Transformée de Fourier

La transformée de Fourier est un outil d'analyse puissant, analogue aux séries de Fourier pour les fonctions non périodiques, et qui permet de leur associer un spectre en fréquences continues (c'est à dire non discrètes). On cherche à obtenir l'expression d'une fonction comme sommation des fonctions trigonométriques de toutes les fréquences qui forment son spectre : une telle sommation se présentera donc sous forme d'intégrale.

2.1.1 Définitions

La transformée de Fourier $\mathcal{F}$ est une opération qui transforme une fonction intégrable f en une autre fonction $\hat{f}$, décrivant le spectre en fréquences de f. Si $f \in \mathcal{L}^1(\mathbb{R})$, sa transformée de Fourier est la fonction $\hat{f} = \mathcal{F}(f)$ donnée par la formule suivante (pour $s \in \mathbb{R}$) :

$$\mathcal{F}(f) : \ s \longmapsto \hat{f}(s) = \int_{-\infty}^{+\infty} e^{-2i\pi sx} f(x) dx \ .$$

L'ensemble de départ est l'ensemble des fonctions intégrables de $\mathbb{R}$ dans $\mathbb{R}^m$, l'ensemble d'arrivée est l'ensemble des fonctions de $\mathbb{R}$ dans $\mathbb{C}^m$.

Lorsque f est une fonction intégrable de deux variables réelles x et y la transformée de f s'écrit comme une intégrale double :

$$\hat{f}(s,t) = \iint_{\mathbb{R}^2} e^{-2i\pi(sx+ty)} f(x,y) dxdy = \int_{-\infty}^{+\infty} e^{-2i\pi ty} \left(\int_{-\infty}^{+\infty} e^{-2i\pi sx} f(x,y) dx \right) dy \ .$$

Cette écriture peut être généralisée pour les fonctions d'une variable $x \in \mathbb{R}^n$ en utilisant un produit scalaire (dans ce cas $s \in \mathbb{R}^n$) :

$$\hat{f}(s) = \int_{x \in \mathbb{R}^n} e^{-2i\pi(s \cdot x)} f(x,y) dx \ .$$

Concrètement lorsque cette transformation est utilisée en traitement d'image, on dit que :

- x est une variable spatiale ;
- f est à valeurs dans le domaine colorimétrique $\mathbb{R}^m$ choisi ;

– s est dans le domaine fréquentiel ;
– $\hat{f}$ est à valeurs dans le domaine spectral $\mathbb{C}^m$.

La formule dite de transformation de Fourier inverse (notée $\mathcal{F}^{-1}$) est celle qui permet (sous conditions) de retrouver f à partir de son spectre :

$$\mathcal{F}^{-1}(\hat{f}) : \; x \longmapsto f(x) = \int_{s\in\mathbb{R}^n} \hat{f}(s) e^{2i\pi s.x} ds \; .$$

Cette formule permet de reconstruire f si $\hat{f}$ est elle-même intégrable.

2.1.2 Propriétés et interprétation

Sur l'espace des fonctions de carré sommable ($\mathcal{L}^2(\mathbb{R}^n)$) l'opérateur $\mathcal{F}$ peut aussi être défini, et vérifie les propriétés suivantes d'après le théorème de Plancherel :

$$\forall f \in \mathcal{L}^2(E) : \begin{cases} \hat{f} \in \mathcal{L}^2(\mathbb{R}^n) \\ \mathcal{F}^{-1}\left(\hat{f}\right) = f \\ \|\hat{f}\|_2 = \|f\|_2 \end{cases}$$

c'est à dire que $\mathcal{F}$ est une isométrie de l'espace $\mathcal{L}^2$. Le terme $\|\hat{f}\|_2$ est interprété dans la pratique comme l'énergie du signal f, liée de façon quadratique à son intensité.

Soit f une application telle que $f, \hat{f} \in \mathcal{L}^1(\mathbb{R}^n)$ (ou plus simplement telle que $f \in \mathcal{L}^2(\mathbb{R}^n)$ comme on vient de le voir). On a alors les propriétés suivantes :
– d'abord $\hat{f}$ est une fonction continue qui vérifie

$$\lim_{|s|\to+\infty} \hat{f}(s) = 0 \quad \text{et} \quad \widehat{\hat{f}}\,(x) = -f(-x) \, ;$$

– si f est paire alors $\hat{f}$ est paire, à valeurs réelles et

$$\hat{f}(s) = 2\int_0^{+\infty} f(x)\cos(2\pi s x)dx \quad \text{et} \quad \widehat{\hat{f}}\,(x) = -f(x) \, ;$$

– si f est impaire alors $\hat{f}$ est impaire, à valeurs imaginaires pures et

$$\hat{f}(s) = -2i\int_0^{+\infty} f(x)\sin(2\pi s x)dx \quad \text{et} \quad \widehat{\hat{f}} = f \, ;$$

Considérons à présent la fonction «*porte*» Π définie de $\mathbb{R}$ dans $\mathbb{R}$ par :

$$\Pi(x) = \begin{cases} 1 & \text{si } |x| \leq \dfrac{1}{2} \\ \\ 0 & \text{si } |x| > \dfrac{1}{2} \; . \end{cases}$$

Π est une fonction de carré intégrable, et puisqu'elle est paire on a :

– si $s \neq 0$ alors :

$$\widehat{\Pi}(s) = 2\int_0^{1/2} \cos(2\pi s x)dx = \frac{\sin(\pi s)}{\pi s} ;$$

– si $s = 0$ alors :

$$\widehat{\Pi}(s) = \int_{-\infty}^{+\infty} \Pi(x)dx = 1 .$$

Cette fonction $\widehat{\Pi}$ est donc prolongeable par continuité en zéro, on note parfois son prolongement sinc (sinus cardinal) :

$$\widehat{\Pi} = \text{sinc} : \ s \longmapsto \begin{cases} 1 & \text{si } s \neq 0 \\ \dfrac{\sin \pi s}{\pi s} & \text{sinon.} \end{cases}$$

2.1.3 Masse de Dirac

Reprenons la fonction *porte* Π vue précédemment et contractons son support tout en dilatant ses valeurs de façon à garder une surface (intensité totale) constante sous la courbe : pour $X > 0$ on définit la fonction d'*impulsion* Π_X

$$\Pi_X(x) = \begin{cases} \dfrac{1}{X} & \text{si } |x| \leq \dfrac{X}{2} \\ 0 & \text{si } |x| > \dfrac{X}{2} . \end{cases}$$

Par un changement de variable simple ($u = x/X$) on obtient facilement :

$$\widehat{\Pi_X}(s) = \widehat{\Pi}(sX) = \text{sinc}(sX) .$$

Lorsque X tend vers zéro, on admettra que Π_X tend vers une limite qui n'est pas une fonction, appelée *distribution* ou *masse de Dirac* et notée δ.

Cette distribution vérifie les propriétés suivantes, qu'on admettra :
– $\hat{\delta}(s) = 1$ pour toute fréquence s ;
– pour toute fonction continue f de $\mathbb{R}$ on a :

$$\int f(t)\delta(t)dt = f(0) .$$

Cette identité se généralise aussi aux produits de convolution :

$$(f * \delta)(x) = \int f(t)\delta(x-t)dt = f(x) .$$

Pour simplifier les formules, on notera par la suite δ_y (pour $y \in \mathbb{R}$) le translaté de δ au point y :

$$\delta_y(x) = \delta(x-y) .$$

Ainsi, δ_y vérifie pour toute fonction continue f :

$$\int f(t)\delta_y(t)dt = \int f(t)\delta(t-y)dt = f(y) \ .$$

On définit aussi la masse de Dirac bidimensionnelle au point $(x, y) \in \mathbb{R}^2$:

$$\delta_{x,y} = \delta_x \delta_y \ .$$

Ainsi, pour toute fonction continue f de deux variables réelles :

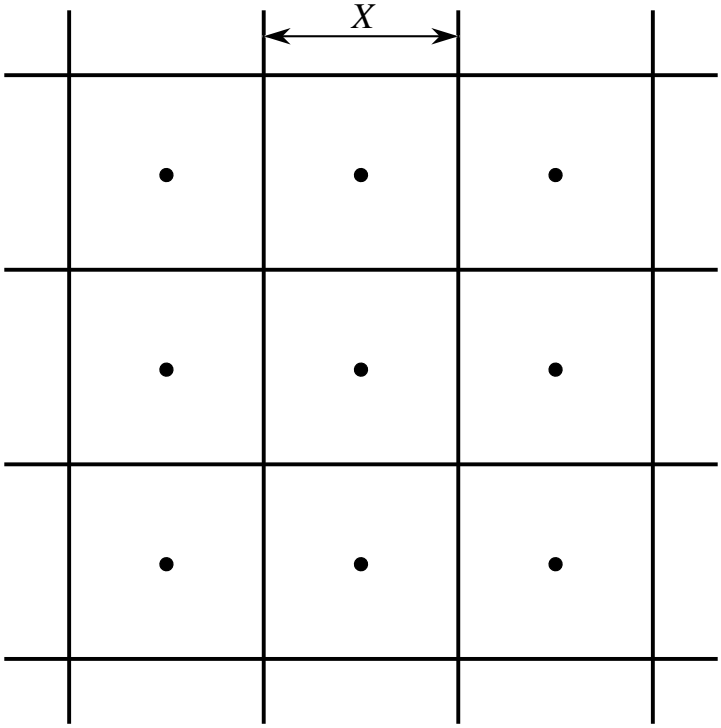
$$\iint f(u,v)\delta_{x,y}(u,v)dudv = \int \left(\int f(u,v)\delta_x(u)du \right) \delta_y(v)dv = f(x,y) \ .$$

2.2 Échantillonnage

Les limitations physiques de la capacité des stockages numériques imposent, lors de la numérisation d'un signal analogique, de ne garder qu'une partie discrète des valeurs du signal original. On va donner une interprétation spectrale de l'effet de cette simplification sur le signal échantillonné.

2.2.1 Échantillonnage par peigne de Dirac

On va ici se limiter au cas d'un échantillonnage régulier du plan de l'image, pavé par des points régulièrement répartis sur une grille carrée de pas $X > 0$: les *pixels* (de l'anglais *picture element*) de l'image. La fréquence de l'échantillonnage spatial sera alors $\frac{1}{X}$.



Échantillonnage sur une grille carrée de pas X

On peut donner deux interprétations de la représentation échantillonnée :
- dans le cadre de la théorie de l'échantillonnage on considère l'image échantillonnée g comme un ensemble d'impulsions de Dirac où chaque impulsion est au centre d'un pixel et a pour amplitude l'intensité de ce pixel, tous les autres points de la surface du pixel étant d'intensité nulle ;
- pour simplifier, dans la pratique on considère que l'image échantillonnée est une fonction continue par morceaux g', constante sur chacun des carrés des pixels et ayant pour valeur celle de l'intensité associée au pixel.

La seconde représentation peut être obtenue en faisant le filtrage (produit de convolution) de la première par un filtre passe-bas (i.e. : qui ne laisse passer que les basses fréquences du spectre) ϕ égal à un sur le support carré d'un pixel centré à l'origine et zéro partout ailleurs :

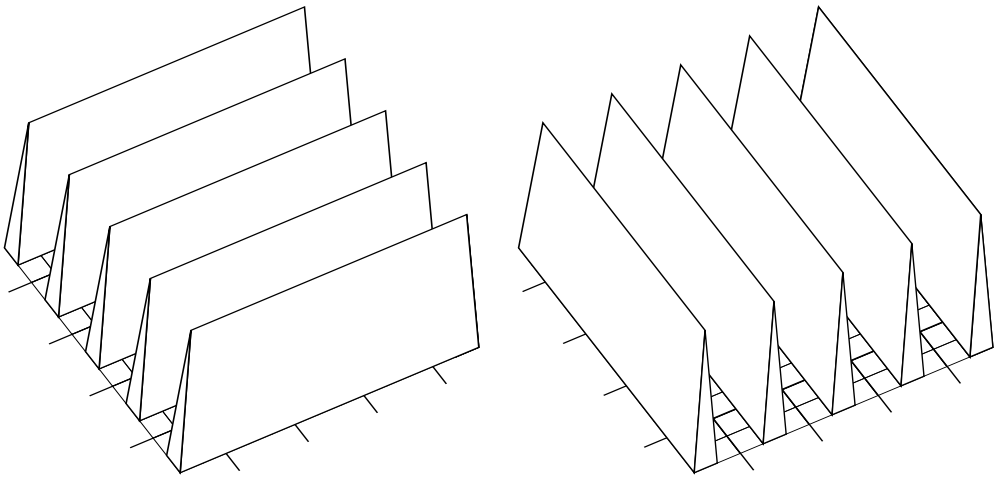
$$\phi(x,y) = \Pi\left(\frac{x}{X}\right)\Pi\left(\frac{y}{X}\right) = \begin{cases} 1 & \text{si } \max(|x|,|y|) \leq X \\ 0 & \text{si } \max(|x|,|y|) > X \end{cases}$$
$$g' = g * \phi \ .$$

On considère que l'échantillonnage g est obtenu en faisant le produit du signal analogique initial f par une somme $\beta(x,y)$ de masses de Dirac centrées en chacun des pixels de l'image :

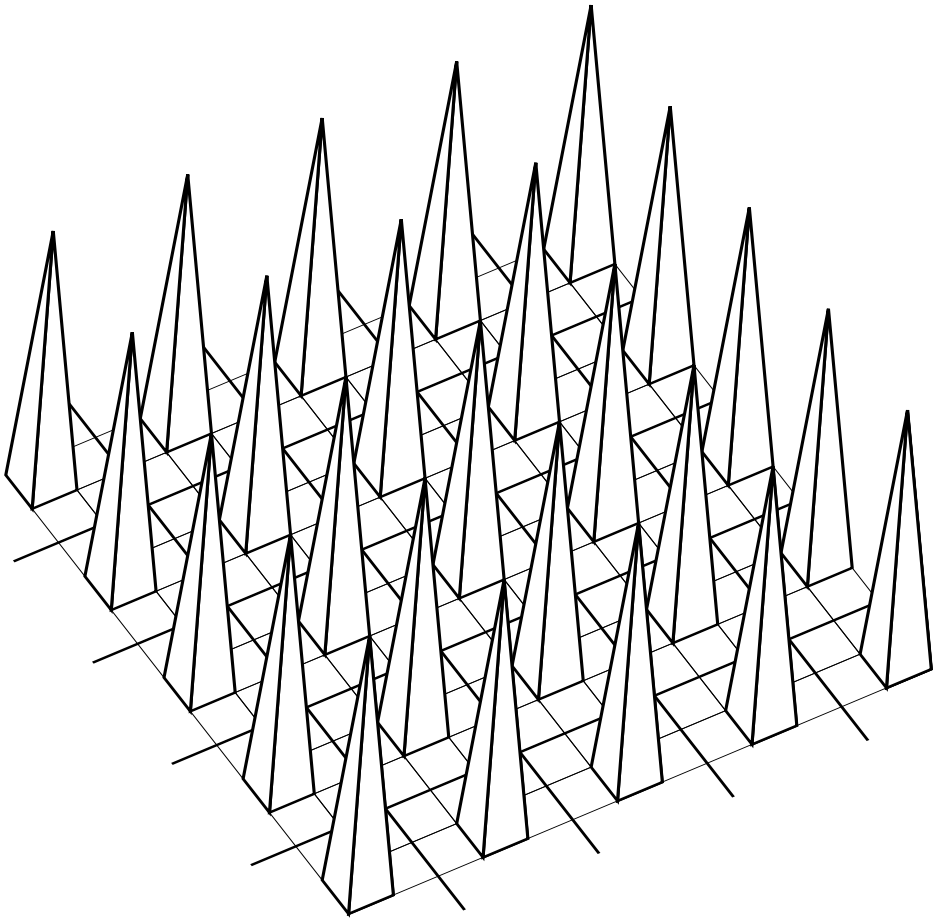
$$\begin{aligned} g(x,y) = f(x,y)\beta(x,y) &= f(x,y) \sum_{(m,n)\in\mathbb{Z}^2} \delta_{mX,nX}(x,y) \\ &= f(x,y) \sum_{n=-\infty}^{+\infty} \left(\sum_{m=-\infty}^{+\infty} \delta_{mX}(x) \right) \delta_{nX}(y) \\ &= f(x,y) \left(\sum_{m=-\infty}^{+\infty} \delta_{mX}(x) \right) \left(\sum_{n=-\infty}^{+\infty} \delta_{nX}(y) \right) \ . \end{aligned}$$

La somme $\beta(x,y)$ utilisé ici est appelé *brosse de Dirac* ou *peigne de Dirac bidimensionnel*, elle est égale au produit de deux peignes monodimensionnels orthogonaux étendus p_1 et p_2 :

$$p_1(x,y) = \sum_{m\in\mathbb{Z}} \delta_{mX}(x) \quad \text{et} \quad p_2(x,y) = \sum_{n\in\mathbb{Z}} \delta_{nX}(y) \ .$$



Les deux peignes étendus p_1 et p_2



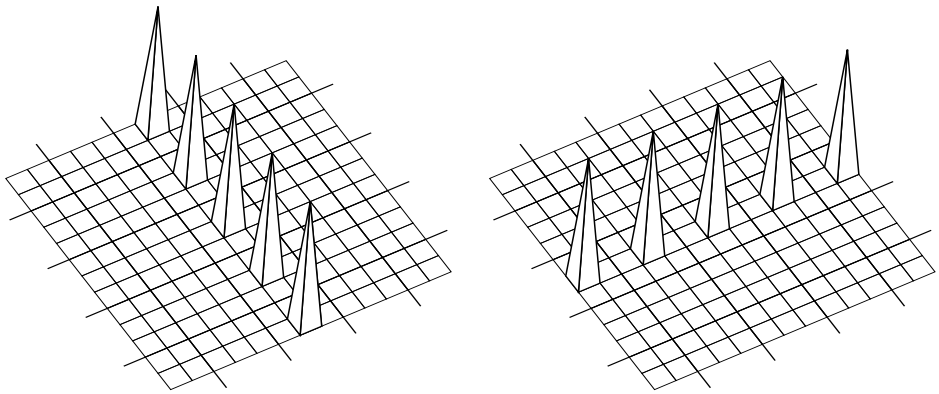
Brosse de Dirac bidimensionnelle

2.2.2 Repliement spectral

La transformée de Fourier $\hat{g}$ de la fonction échantillonnée g est la convolution dans le domaine des fréquences de la brosse $\widehat{\beta}$ (transformée de la fonction d'échantillonnage) et de $\hat{f}$ (transformée de l'image avant échantillonnage).

La transformée de la brosse est elle même la convolution des transformées $\widehat{p_1}$ et $\widehat{p_2}$ des deux peignes étendus orthogonaux, qui sont deux peignes monodimensionnels non étendus, respectivement sur l'axe des abcisses et celui des ordonnées :

$$\widehat{p_1}(u,v) = \delta(v) \sum_{k\in\mathbb{Z}} \delta_{k/X}(u)$$
$$\widehat{p_2}(u,v) = \delta(u) \sum_{l\in\mathbb{Z}} \delta_{l/X}(v) \ .$$



Les deux peignes non étendus $\widehat{p_1}$ et $\widehat{p_2}$

On trouve donc, après calcul des produits de convolution :

$$\widehat{\beta}(u,v) = \sum_{(k,l)\in\mathbb{Z}^2} \delta_{\frac{k}{X},\frac{l}{X}}(u,v)$$

c'est à dire que la transformée de la fonction d'échantillonnage est elle-même un peigne de Dirac bidimensionnel, de période $\frac{1}{X}$ (dans le domaine des fréquences). La transformée du signal échantillonné s'écrit donc, après calcul du produit de convolution :

$$\hat{g}(u,v) = \sum_{(k,l)\in\mathbb{Z}^2} \hat{f}\left(u - \frac{k}{X}, v - \frac{l}{X}\right) \ .$$

On peut tout de suite remarquer qu'en remplaçant u par $u + \frac{1}{X}$ (respectivement v par $v + \frac{1}{X}$) la sommation précédente reste égale (il suffit de faire le changement de variable $k' = k + 1$, respectivement $l' = l + 1$). Ceci amène deux remarques importantes :

1. la transformée du signal échantillonné est périodique ;
2. elle est la somme des translatés de la transformée du signal avant échantillonnage par tous les vecteurs $\left(\frac{k}{X}, \frac{l}{Y}\right)$ (pour $(k,l) \in \mathbb{Z}^2$).

2.3 Phénomène d'*aliasing*

Lors de l'analyse ou l'affichage du signal échantillonné d'une image, on est amené à effectuer la reconstruction du signal continu original (c'est à dire l'inverse de l'échantillonnage). Pour que cette opération puisse se faire sans perte de données, il faut de manière équivalente pouvoir recomposer le spectre du signal original.

2.3.1 Théorème de Nyquist-Shannon

On a vu que pour une fonction intégrable $f \in \mathcal{L}^1(\mathbb{R}^2)$ on a :

$$\lim_{|u|+|v|\to+\infty} \hat{f}(u,v) = 0 \ .$$

Notons $K(\hat{f})$ le support de $\hat{f}$ (l'adhérence des points où $\hat{f}$ n'est pas nulle) :

$$K(\hat{f}) = \overline{\hat{f}^{-1}(\mathbb{C} \setminus 0)}$$

et supposons que f est continue, et que $K(\hat{f})$ est compact, c'est à dire inclus dans un carré de côté $2\mu \geq 0$ centré en zéro (on choisira la valeur minimale de μ parmi celles qui peuvent convenir) :

$$\forall (u,v) \notin]-\mu,\mu[^2 : \ \hat{f}(u,v) = 0 \ .$$

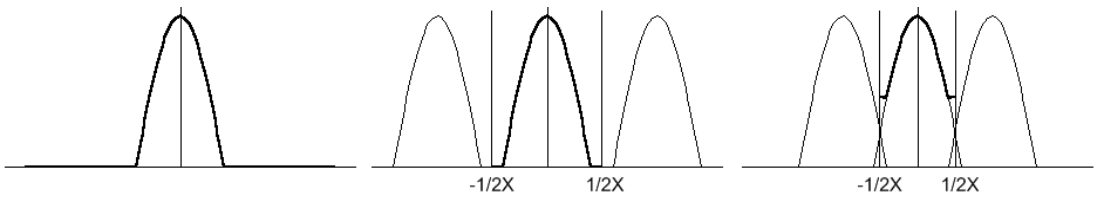
En d'autres termes, on impose ici que le signal f à échantillonner ne contient pas de fréquences spatiales supérieures à μ.

On peut distinguer deux cas selon les valeurs relatives de μ et X :

- si $\mu \leq \frac{1}{2X}$ alors dans la somme formant $\hat{g}$, les translatés de $\hat{f}$ sont de supports disjoints deux à deux. Par conséquent le spectre original $\hat{f}$ peut être recomposé facilement : il suffit de le multiplier par le filtre passe-bas ϕ. En d'autres termes :

$$\hat{f}(u,v) = \hat{g}(u,v)\phi(u,v) \, ;$$

- si $\mu > \frac{1}{2X}$ alors dans la somme formant $\hat{g}$, les translatés de $\hat{f}$ ne sont plus de supports disjoints deux à deux. Par conséquent le spectre original $\hat{f}$ ne peut plus être fidèlement recomposé à partir des données échantillonnées, puisque dans le spectre $\hat{g}$ restreint à $[-\mu,\mu]^2$ des amplitudes parasites correspondant à d'autres harmoniques du spectre de f viennent se superposer : on appelle ce phénomène le *recouvrement de spectre.*



À gauche le spectre du signal original, au milieu le spectre du signal échantillonné lorsque $\mu \leq \frac{1}{2X}$ et à droite lorsque $\mu > \frac{1}{2X}$

Le principe qu'on vient d'énoncer ici est en fait à peu de choses près le théorème d'échantillonnage de Nyquist-Shannon, qui est à la base de la conversion numérique des signaux :

> *La fréquence d'échantillonnage d'un signal doit être au moins le double de la fréquence maximale du signal, afin de le convertir sans perte d'information sous forme numérique.*

Dans le cas des images digitales, les phénomènes de recouvrement de spectre sont à l'origine d'imperfections et d'*artefacts* divers :

- crênelage le long des frontières des objets, courbes lisses rendues sous forme d'escaliers ;
- moirés, lorsque l'image contient de vastes zones de hautes fréquences ;
- aberrations chromatiques liées à la perte de synchronisation naturelle entre les canaux.

Il faut noter que ces imperfections sont en général **irréversibles**, car il est très difficile après échantillonnage de distinguer les fréquences originales de celles qui ont été superposées.

2.3.2 Méthodes d'anti-aliasing

On va considérer trois méthodes couramment utilisées pour réduire l'aliasing d'une image.

Filtrage avant échantillonnage – cette méthode est de loin la meilleure, mais elle nécessite une connaissance a priori du signal original complet (ce qui n'est pas toujours une condition réalisable ni souhaitable en synthèse d'image). Elle consiste simplement à faire la convolution de l'image avant échantillonnage par un filtre passe-bas qui élimine les fréquences supérieures à la moitié de la fréquence d'échantillonnage. Idéalement, le filtre de convolution utilisé devrait être la transformée du filtre ϕ :

$$\widehat{\phi}(x, y) = \text{sinc}\left(\frac{x}{X}\right) \text{sinc}\left(\frac{y}{X}\right)$$

mais en général on utilise pour simplifier les calculs la fonction ϕ elle-même.

Sur-échantillonnage ou *oversampling* – cette méthode est la plus simple, mais est aussi très coûteuse en performances de calcul. Il s'agit d'échantillonner sur une image 2^k fois plus grande (pour $k \geq 1$) que la résolution souhaitée, puis de la réduire à la taille originale en moyennant successivement les pixels par groupe de quatre sur un voisinage carré de taille deux par deux. Cette méthode revient à faire l'échantillonnage sans filtrage à une fréquence 2^k fois plus grande.

Sur-échantillonnage adaptif (*adaptive oversampling*) – cette méthode est en fait une combinaison des deux premières : elle consiste à sur-échantillonner uniquement les zones de l'image supposées contenir des hautes fréquences (bord des lignes, frontière des polygones, etc...).

Deuxième partie

Bases de l'infographie

Contenu

3 Perspective

3.1 Vues

Une *vue* d'une scène 3D est déterminée par les données suivantes.

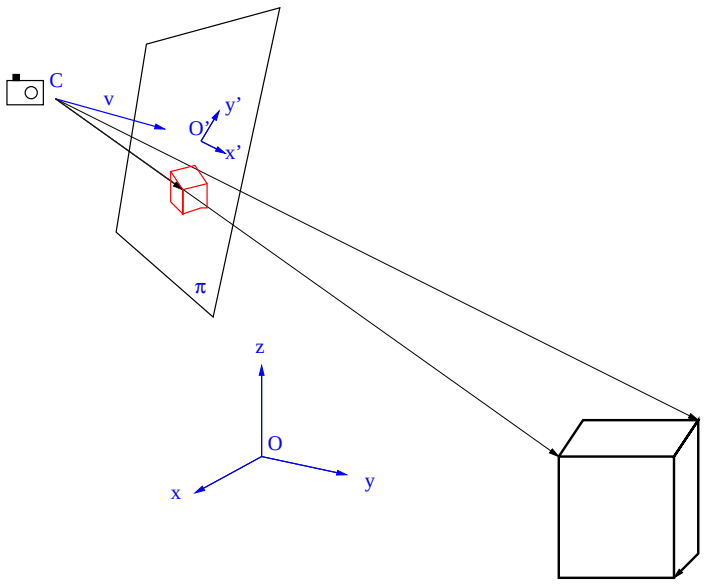
- La position de la caméra : un point C de coordonnées $\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$;
- une direction de visée : un vecteur unitaire $\vec{v}$ de composantes $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$;
- un *écran*, i.e. un plan Π perpendiculaire à la direction de visée : il est déterminé par sa distance à C, un réel positif d ;
- un repère orthonormé $(O', \vec{e_1'}, \vec{e_2'})$ du plan Π.

Voici un moyen systématique de construire le repère (O', e_1', e_2'). Il suffit de choisir une fois pour toute un vecteur unitaire $\vec{\nu}$ non colinéaire à $\vec{v}$. On prend pour O' la projection orthogonale de C sur Π, soit $O' = C + d\vec{v}$. On prend

$$\vec{e_1'} = \frac{\vec{v} \wedge \vec{\nu}}{|\vec{v} \wedge \vec{\nu}|} \qquad \text{et} \qquad \vec{e_2'} = \vec{v} \wedge \vec{e_1'} \ .$$

Une *vue* de la scène est une application $W : \mathbf{R}^3 \to \mathbf{R}^2$, qui aux coordonnées d'un point dans le *repère du monde* $(O, \vec{e_1}, \vec{e_2}, \vec{e_3})$ associe les coordonnées de sa projection sur l'écran Π, dans le *repère de l'image* $(O', \vec{e_1'}, \vec{e_2'})$.

Définition 3.1 *La* vue en perspective *depuis C sur l'écran Π consiste à projeter un point p sur p', point d'intersection de la droite Cp avec Π.*



Vue en perspective

Remarque 3.2 *La projection n'est pas définie si p est dans le plan passant par C et parallèle à Π. C'est normal : on n'arrive pas à voir dans les directions situées à 90° de sa direction de vision.*

Définition 3.3 Prise de vue à distance infinie. *Soit D une droite, appelée* axe de visée*, et Π un plan orthogonal à D. La prise de vue à distance infinie dans la direction D consiste à projeter orthogonalement sur Π.*

C'est ce qu'on obtient à la limite, lorsque, Π et O' étant fixés, C tend vers l'infini le long de la droite $D = (O', \vec{u})$.

Proposition 3.4 *On choisit $\nu = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. Les coordonnées de p' sont données, en fonction des coordonnées $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ de p, par les formules*

$$x' = \frac{-cd(x - x_0) + ad(z - z_0)}{\sqrt{a^2 + c^2}(a(x - x_0) + b(y - y_0) + c(z - z_0))},$$

$$y' = \frac{abd(x - x_0) - d(a^2 + c^2)(y - y_0) + bcd(z - z_0)}{\sqrt{a^2 + c^2}(a(x - x_0) + b(y - y_0) + c(z - z_0))}.$$

Preuve. Voir exercice 1. ∎

Exercice 1 *On prend une photo depuis le point $C = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$, dans la direction $\vec{v} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$, sur un écran situé à distance d. On choisit $\nu = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ comme vecteur de référence. Calculer les coordonnées $\begin{pmatrix} x' \\ y' \end{pmatrix}$ de l'image p' sur l'écran d'un point $p = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ de l'espace.*

Solution de l'exercice 1. *La projection perspective.*

On travaille d'abord dans le repère du monde, supposé orthonormé.

Un point $M = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$ est dans Π si et seulement si $\vec{O'M} \cdot \vec{v} = 0$, i.e. si $a(X - x_0 - da) + b(Y - y_0 - db) + c(Z - z_0 - dc) = 0$, autrement dit, $a(X - x_0) + b(Y - y_0) + c(Z - z_0) = d$.

On paramètre la droite Cp par

$$t \mapsto c(t) = (1-t)C + tp = \begin{pmatrix} (1-t)x_0 + tx \\ (1-t)y_0 + ty \\ (1-t)z_0 + tz \end{pmatrix}.$$

Le point $c(t)$ est dans Π si et seulement si

$$t = \frac{d}{a(x-x_0) + b(y-y_0) + c(z-z_0)}.$$

Par conséquent, les coordonnées de la projection p' dans le repère du monde sont

$$\begin{pmatrix} x_0 + \frac{d(x-x_0)}{a(x-x_0)+b(y-y_0)+c(z-z_0)} \\ y_0 + \frac{d(y-y_0)}{a(x-x_0)+b(y-y_0)+c(z-z_0)} \\ z_0 + \frac{d(z-z_0)}{a(x-x_0)+b(y-y_0)+c(z-z_0)} \end{pmatrix}.$$

On calcule les composantes des vecteurs du repère de l'image.

$$\vec{v} \wedge \vec{\nu} = \begin{pmatrix} -c \\ 0 \\ a \end{pmatrix},$$

d'où

$$\vec{e_1'} = \begin{pmatrix} -\frac{c}{\sqrt{a^2+c^2}} \\ 0 \\ \frac{a}{\sqrt{a^2+c^2}} \end{pmatrix}$$

et

$$\vec{e_2'} = \frac{\vec{v} \wedge \vec{e_1'}}{|\vec{v} \wedge \vec{e_1'}|} = \begin{pmatrix} \frac{ab}{\sqrt{a^2+c^2}} \\ -\sqrt{a^2+c^2} \\ \frac{bc}{\sqrt{a^2+c^2}} \end{pmatrix}.$$

Les coordonnées de p' dans le repère de l'image s'obtiennent par

$$x' = \vec{Op'} \cdot \vec{e_1'} = \frac{-cd(x-x_0) + ad(z-z_0)}{\sqrt{a^2+c^2}(a(x-x_0) + b(y-y_0) + c(z-z_0))},$$

$$y' = \vec{Op'} \cdot \vec{e_2'} = \frac{abd(x-x_0) - d(a^2+c^2)(y-y_0) + bcd(z-z_0)}{\sqrt{a^2+c^2}(a(x-x_0) + b(y-y_0) + c(z-z_0))}.$$

Remarque. Lorsque C, d et v varient, la méthode ci-dessus ne fournit pas tous les angles de prise de vue de la scène possibles. Pour les avoir tous, il faut encore faire tourner la caméra autour de la direction de visée, i.e. remplacer $(\vec{e_1'}, \vec{e_2'})$ par

$$(\cos\psi \vec{e_1'} + \sin\psi \vec{e_2'}, -\sin\psi \vec{e_1'} + \cos\psi \vec{e_2'})$$

pour $\psi \in [0, 2\pi]$.

3.2 Coordonnées homogènes

On constate (proposition 3.4), que la vue en perspective s'exprime par des fractions rationnelles

$$x' = \frac{-cd(x-x_0) + ad(z-z_0)}{\sqrt{a^2+c^2}(a(x-x_0)+b(y-y_0)+c(z-z_0))},$$
$$y' = \frac{abd(x-x_0) - d(a^2+c^2)(y-y_0) + bcd(z-z_0)}{\sqrt{a^2+c^2}(a(x-x_0)+b(y-y_0)+c(z-z_0))}.$$

Complétons le repère de l'image en un repère orthonormé de $\mathbf{R}^3$ en posant $\vec{e_3'} = \vec{v}$. La troisième coordonnée de p' est évidemment $z' = 0$.

On ramène ces expressions rationnelles à des expressions linéaires en adoptant la convention suivante : Un point p de $\mathbf{R}^3$ peut être représenté, non seulement par $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$, mais par n'importe quel vecteur de $\mathbf{R}^4$ qui lui est proportionnel. En particulier, un point p' du plan Π peut être représenté par n'importe quel vecteur de $\mathbf{R}^3$ proportionnel à $\begin{pmatrix} x' \\ y' \\ 0 \\ 1 \end{pmatrix}$.

Avec cette convention, on peut choisir pour représentant de la vue en perspective p' d'un point p le vecteur

$$\begin{pmatrix} -cd(x-x_0) + ad(z-z_0) \\ abd(x-x_0) - d(a^2+c^2)(y-y_0) + bcd(z-z_0) \\ 0 \\ \sqrt{a^2+c^2}(a(x-x_0)+b(y-y_0)+c(z-z_0)) \end{pmatrix}.$$

Ce vecteur est l'image du vecteur $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$ par l'endomorphisme de $\mathbf{R}^4$ de matrice

$$\begin{pmatrix} -cd & 0 & ad & cdx_0 - adz_0 \\ abd & -d(a^2+c^2) & bcd & -abdx_0) + d(a^2+c^2)y_0 - bcdz_0 \\ 0 & 0 & 0 & 0 \\ a\sqrt{a^2+c^2} & b\sqrt{a^2+c^2} & c\sqrt{a^2+c^2} & \sqrt{a^2+c^2}(-ax_0 - by_0 - cz_0) \end{pmatrix}$$

C'est quand même plus sympathique (et commode à implémenter informatiquement) de passer par une matrice pour représenter la vue en perspective.

Définition 3.5 *Soit* $p = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ *un point de* $\mathbf{R}^3$. *On appelle* coordonnées homogènes *de* p *tout quadruplet* (u, v, w, t) *proportionnel à* $(x, y, z, 1)$. *On* note $p = [x; y; z; t]$.

Une application $f : \mathbf{R}^3 \to \mathbf{R}^3$ (éventuellement définie seulement en dehors d'un hyperplan affine) est dite projective *s'il existe une application linéaire $L : \mathbf{R}^4 \to \mathbf{R}^4$ telle que, pour tout $p = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbf{R}^3$, d'image $f(p) = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$,*

$$L \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \text{est proportionnel à} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

Autrement dit, en coordonnées homogènes, f est donnée par une matrice 4×4.

Remarque. La définition précédente s'étend à toutes les dimensions.

Exemple 3.6 *Toute application affine est en particulier projective.*

Exemple 3.7 *La vue en perspective est une application projective, définie en dehors de l'hyperplan parallèle à l'écran passant par la caméra, non surjective (son image est contenue dans l'écran).*

Exemple 3.8 *Soient Π_1 et Π_2 deux plans affines dans $\mathbf{R}^3$, C un point qui n'est pas dans Π_2. Considérons la restriction f à Π_1 de la vue en perspective sur Π_2. Après choix de coordonnées cartésiennes sur Π_2 et Π_1, cette application devient une application projective de $\mathbf{R}^2$ dans $\mathbf{R}^2$.*

Elle est définie en dehors d'une droite D_1, l'intersection de Π_1 avec le plan parallèle à Π_2 passant par C. Son image est le complémentaire d'une droite D_2, l'intersection de Π_2 avec le plan passant par C parallèle à Π_1. Si C n'est pas sur Π_1, f est une bijection de $\Pi_1 \setminus D_1$ sur $\Pi_2 \setminus D_2$. Dans ce cas, la bijection réciproque est la vue en perspective de Π_2 sur l'écran Π_1 depuis C. f est affine si et seulement si Π_1 et Π_2 sont parallèles.

Exercice 2 *Soit Π_1 le plan passant par $O'_1 = O$ et de vecteurs directeurs $\vec{u_1} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ et $\vec{v_1} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. Soit Π_2 le plan passant par $O'_2 = O$ et de vecteurs directeurs $\vec{u_2} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$ et $\vec{v_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. Soit $C = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}$. Calculer la matrice 3×3 qui représente la vue en perspective de Π_1 sur Π_2 depuis C. Vérifier qu'elle est inversible.*

Solution de l'exercice 2. *Vue en perspective d'un plan.*

Soit $p_1 \in \Pi_1$, de coordonnées (x_1, y_1) dans le repère $(O_1, \vec{u_1}, \vec{v_1})$. Alors $p_1 = O + x_1\vec{v_1} + y_1\vec{v_1}$ a pour coordonnées $\begin{pmatrix} x_1 \\ y_1 \\ x_1 \end{pmatrix}$ dans le repère du monde. De même, un point $p_2 \in \Pi_2$, de coordonnées (x_2, y_2) dans le repère $(O_2, \vec{u_2}, \vec{v_2})$ a pour coordonnées $\begin{pmatrix} x_2 \\ y_2 \\ -x_2 \end{pmatrix}$ dans le repère du monde. L'équation de Π_2 est $X + Z = 0$. Un point courant

$$(1-t)C + tp_1 = \begin{pmatrix} tx_1 \\ ty_1 \\ tx_1 + 2 - 2t \end{pmatrix}$$

sur la droite Cp_1 est dans Π_2 si et seulement si $2tx_1 + 2 - 2t = 0$, d'où $t = 1/1 - x_1$ et les coordonnées de $p_2) = f(p_1)$ sont

$$x_2 = \frac{x_1}{1-x_1} \quad \text{et} \quad y_2 = \frac{y_1}{1-x_1}.$$

On constate que $\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}$ est proportionnel à $\begin{pmatrix} x_1 \\ y_1 \\ 1 - x_1 \end{pmatrix}$ qui est l'image de $\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$ par la matrice $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}$. On conclut que f est projective. Sa matrice a pour déterminant 1 donc elle est inversible.

3.3 Transformations projectives

Une *transformation projective* de $\mathbf{R}^n$, c'est une application projective dont la matrice est inversible. On montre aisément que deux matrices inversibles de taille $n+1$ définissent la même transformation de $\mathbf{R}^n$ si et seulement si elles sont proportionnelles. D'autre part, si L et L' sont des matrices associées à des transformations f et f', l'application projective associée à LL' est $f \circ f'$. Par conséquent, les transformations projectives de $\mathbf{R}^n$ forment un groupe isomorphe à $Gl(n+1, \mathbf{R})/\sim$ où $L \sim L'$ s'il existe un réel non nul λ tel que $L' = \lambda L$. Ce groupe quotient, *noté* $PGl(n+1, \mathbf{R})$, a pour dimension $(n+1)^2 - 1$. Par exemple, si $n = 3$, une transformation projective dépend de 15 paramètres, alors qu'une transformation affine dépend de seulement 12 paramètres.

3.4 Vues en perspective d'objets

Exercice 3 *Vérifier que, vue en perspective, une droite reste en général une droite. Quelles sont les exceptions ? Montrer que, vues en perspectives, les droites parallèles à une droite D sont en général concourantes en un point appelé* point de fuite *de D. Quelles sont les exceptions ? Où se trouvent les points de fuites des droites contenues dans un même plan ?*

Solution de l'exercice 3. *Vue en perspective de droites.*

Si D est une droite qui ne passe pas par la caméra C, sa vue en perspective est l'intersection du plan contenant D et C avec l'écran Π. C'est donc une droite. Si D passe par C, elle est vue comme un point. Si D passe par C et est parallèle à Π, elle n'est pas vue du tout.

Si D et D' sont des droites parallèles ne passant pas par C et non parallèles à Π, les plans Q et Q' qu'elles déterminent avec C se coupent suivant une troisième droite D'' parallèle à D et à D', qui passe par C. Comme D'' n'est pas parallèle à Π, la vue en perspective de D'' est un point p par lequel passent les vues en perspective de D et D'. Vues en perspective, toutes les droites parallèles à D passent donc par p. Si D est parallèle à Π, D'' ne coupe pas Π, donc les vues en perspective de D et D' sont parallèles.

Si D est contenue dans un plan Π' non parallèle à Π, la droite parallèle à D passant par C est contenue dans le plan Π'' parallèle à Π' passant par C, donc le point de fuite p de D est contenu dans la droite intersection de Π'' et de Π. Les points de fuites des droites de Π' sont donc tous alignés.

Remarque 3.9 *Dans une ville, les immeubles sont souvent des parallélépipèdes aux arêtes parallèles à trois directions fixées. Vues en perspective, ces trois familles d'arêtes sont portées par des droites concourantes en trois points. Si la direction de visée est horizontale, les arêtes verticales restent parallèles, et il ne reste plus que deux points de concours nettement visibles sur l'image : on parle de* perspective à deux points de fuite.

Exercice 4 *Soit D_+ une demi-droite, S un segment de droite. Qu'est ce qu'ils donnent, vus en perspective ? Attention, il y a de multiples cas de figure.*

Solution de l'exercice 4. *Vue en perspective d'une demi-droite ou d'un segment de droite.*

Soit D la droite qui porte S ou D_+.

Si D est parallèle à Π, la projection sur Π est affine, donc envoie une demi-droite (resp. un segment) porté par D sur une demi-droite (resp. un segment).

Supposons que D n'est pas parallèle à Π et ne passe pas par C. Alors elle coupe Π en un point G, et la partie visible de D (l'intersection de D avec le demi-espace délimité par Π qui est devant la caméra) est une demi-droite Δ issue de G. La droite D possède aussi un point de fuite F, l'intersection de Π avec la droite passant par C parallèle à D.

Un segment S de D est ou bien entièrement invisible (s'il ne rencontre pas Δ), ou bien se projette en un seul point (si D passe par C ou si S part de G dans la direction opposée à Δ), ou bien un segment d'origine G (si S contient G) ou bien un segment dont les extrémités sont les projections des extrémités de S.

Une demi-droite D_+ de D est ou bien entièrement invisible (si elle ne rencontre pas Δ), ou bien se projette en un seul point (si D passe par C ou si D_+ part de G dans la direction opposée à Δ), ou bien un segment d'origine G (si D_+ contient Δ) ou bien

un intervalle $[G, F[$ (si D_+ contient la demi-droite opposée à Δ), ou bien un intervalle $[pr(A), F[$ où A est l'extrémité de D_+ (si D_+ est disjointe de Δ).

Remarque 3.10 *On a décrit une vue comme la projection sur l'écran de tout l'espace. En réalité, une vue ne montre que ce qui se trouve devant l'écran. La vue en perspective réelle d'une droite est donc la projection d'une demi-droite ne coupant pas le plan parallèle à l'écran passant par la caméra : on voit un segment dont les extrémités sont le point de fuite et l'itersection de la droite avec l'écran.*

Exercice 5 *A quoi ressemble une sphère vue en perspective ?*

Solution de l'exercice 5. *Vue en perspective d'une sphère.*

La réunion des droites passant par C et coupant la sphère S est un cône de révolution K dont l'axe passe par C et le centre C' de S. Si Π est orthogonal à l'axe (i.e. si le centre est sur l'axe de visée), $K \cap \Pi$ est un disque. Sinon, c'est une ellipse dont le grand axe est dans le plan contenant l'axe de visée et le centre de S. On va montrer que l'ellipse est d'autant plus allongée que S est loin de l'axe de visée, cette distance étant rapportée à la distance de S à la caméra.

La figure est symétrique par rapport au plan Π' contenant la droite de visée et le centre C' de S. On peut donc choisir le repère du monde de sorte que l'origine soit en $O = C$, que la droite de visée soit Oz, que l'équation de Π soit $\{z = 1\}$ et que l'équation du plan Π' soit $\{y = 0\}$. Soient $(x_0, 0, z_0)$ les coordonnées de C', et R le rayon de la sphère S. L'équation de S s'écrit

$$(x - x_0)^2 + y^2 + (z - z_0)^2 - R^2 = 0.$$

Soit $P' = (x', y', 1)$ un point de Π. Pour t réel, le point point $M = O + t\vec{OP}$ appartient à S si et seulement si

$$(x'^2 + 1 + y'^2)t^2 + (-2x_0x' - 2z_0)t + x_0^2 + z_0^2 - R^2 = 0.$$

La droite OP est tangente à S si et seulement si cette équation du second degré possède une racine double, i.e. si et seulement si son discriminant

$$(R^2 - z_0^2)x'^2 + 2z_0x_0x' + (R^2 - x_0^2 - z_0^2)y'^2 + R^2 - x_0^2$$

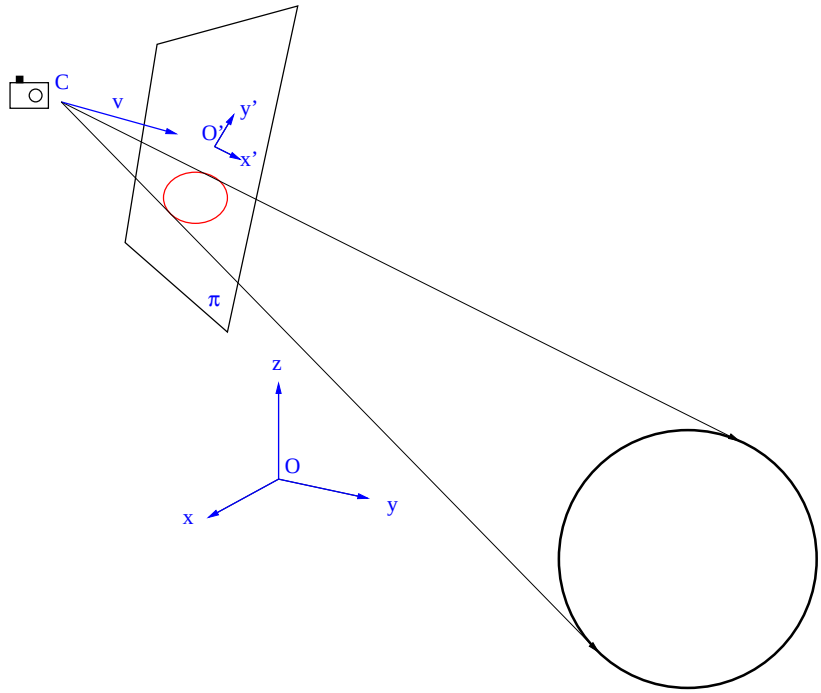
s'annule. Posons $X = x' + \dfrac{x_0z_0}{R^2 - z_0^2}$, $Y = y'$. On obtient l'équation normalisée de l'ellipse $\dfrac{1}{a^2}X^2 + \dfrac{1}{b^2}Y^2 - 1 = 0$, avec

$$a^2 = \frac{R^2(x_0^2 + z_0^2 - R^2)}{(z_0^2 - R^2)^2} \quad \text{et} \quad b^2 = \frac{R^2}{z_0^2 - R^2}.$$

Le rapport

$$\frac{a^2}{b^2} = \frac{x_0^2 + z_0^2 - R^2}{z_0^2 - R^2}$$

est toujours plus grand que 1. Autrement dit, l'ellipse s'étale le long de l'axe Ox', i.e. le long du plan Π'.



Vue en perspective d'une sphère

Remarque. Lorsqu'on programme une vue en perspective d'une scène, attention aux unités de longueur. La caméra doit être placée loin de la scène, sinon les distorsions (sphères qui deviennent des ellipses) seront trop visibles.

Exercice 6 *On photographie (caméra à distance finie, direction de visée horizontale) un paysage comportant les ruines d'un temple grec. La scène est jonchée de tronçons de colonnes, certaines debout, d'autres couchées. Chaque colonne est, en première approximation, un cylindre à base circulaire coupé par des plans orthogonaux à son axe.*

a. *Vue en perspective, une section de colonne verticale est représentée par une ellipse. Il y a t'il des cas où on peut aisément déterminer la direction de son grand axe ? L'excentricité de cette ellipse dépend-elle de la hauteur de la colonne ?*

b. *On considère une rangée de colonnes de même hauteur, alignées le long de la droite de visée. Les sections des colonnes, vues en perspective, sont-elles homothétiques ?*

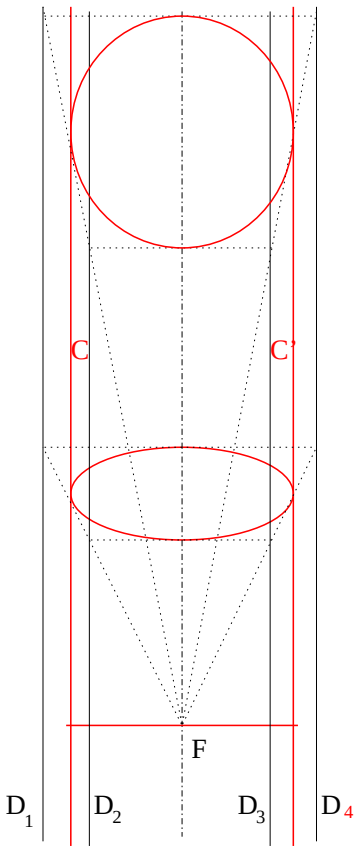
c. *Qu'en est-il si les colonnes sont alignées le long d'une droite parallèle à mais distincte de la droite de visée ? Les ellipses sont elles asymptotiquement homothétiques ?*

d. *Est-il possible qu'une colonne couchée soit représentée par un cercle ?*

Solution de l'exercice 6. *Vue d'un temple grec.*

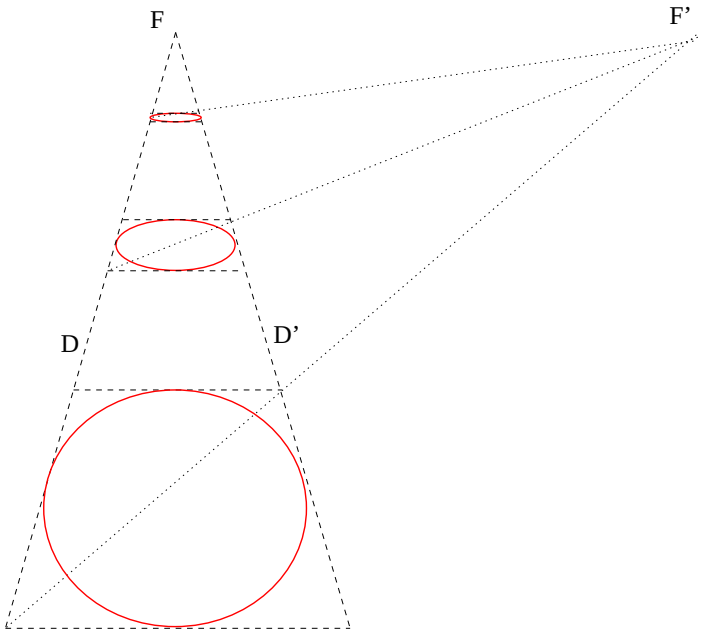
a. Lorsque l'axe d'une colonne verticale coupe la droite de visée, l'ensemble de la figure (colonne, axe de visée, écran) est symétrique par rapport à un plan vertical, donc la vue

de la section est symétrique par rapport à une droite verticale. L'ellipse est réduite à un segment si le plan de la section passe par la caméra. Par conséquent, l'excentricité de cette ellipse dépend de la hauteur de la colonne. La largeur de l'ellipse (dans la direction horizontale) est constante. La hauteur (dans la direction verticale) tend vers l'infini lorsque la hauteur de la colonne tend vers l'infini. En effet, la colonne est inscrite dans un parallélépidède à base carrée, dont les faces verticales sont parallèles (resp. orthogonales) à la droite de visée. Vu en perspective, les arêtes verticales du parallélépipède donnent 4 droites D_1, D_2, D_3 et D_4. La projection du cylindre est une bande délimités par deux droites verticales C et C', traces sur Π des plans passant par la caméra et tangents au cylindre. Sur la figure, on a représenté la vue en perspective de 3 sections horizontales du cylindre et du parallélépipède. Celle qui se trouve dans le plan horizontal contenant la droite de visée donne un segment contenant le point de fuite F de la famille des droites parallèles à la droite de visée. Toute section horizontale du parallélépipède est un carré dont deux côtés, parallèles au plan Π, se projettent en des segments parallèles reliant D_2 à D_3 (resp. D_1 à D_4) et les deux autres côtés, parallèles à la droite de visée, se projettent sur des segments portés par des droites passant par F. On peut donc tracer exactement les trapèzes correspondant à ces carrés. On constate que la hauteur du trapèze tend vers l'infini avec la hauteur de la section. Les sections du cylindre se projettent suivant des ellipses tangentes à C, C' ainsi qu'aux côtés du trapèze correspondant. Leur hauteur tend donc aussi vers l'infini.



b. Non. Chaque section de colonne est inscrite dans un carré dont deux cotés sont parallèles à la direction de visée. Sa vue en perspective $\mathcal{E}$ est inscrite dans un trapèze $\mathcal{T}$, projection du carré. Montrons que si deux ellipses $\mathcal{E}$ et $\mathcal{E}'$ de la famille sont homothétiques, alors les trapèzes correspondants sont homothétiques. Une homothétie préserve les rapports de distances. Elle envoie le centre de $\mathcal{E}$ sur le centre de $\mathcal{E}'$, car le centre est l'unique point intérieur à $\mathcal{E}$ où la distance au bord est maximale. La distance du centre à un point de $\mathcal{E}$ atteint son minimum aux extrémités du petit axe et son maximum aux extrémités du grand axe. L'homothétie envoie donc les extrémités du petit (resp. grand) axe de $\mathcal{E}$ sur les extrémités du petit (resp. grand) axe de $\mathcal{E}'$. Par conséquent, l'homothétie envoie les deux bases du trapèze $\mathcal{T}$ circonscrit à $\mathcal{E}$, tangentes aux extrémités du petit (resp. grand) axe de $\mathcal{E}$, sur les bases du trapèze $\mathcal{T}'$ circonscrit à $\mathcal{E}'$. Orientons $\mathcal{E}$. Une homothétie de rapport positif préserve cette orientation. Pour chaque vecteur unitaire v, $\mathcal{E}$ possède exactement une tangente orientée de vecteur directeur v. L'homothétie envoie toute droite sur une droite parallèle. Par conséquent, elle envoie chacune des droites D et D', tangentes à $\mathcal{E}$ et à $\mathcal{E}'$, sur elle-même. On conclut que les trapèzes $\mathcal{T}$ et $\mathcal{T}'$ sont homothétiques.

Si les trapèzes étaient homothétiques, les diagonales de ces trapèzes formeraient deux familles de droites parallèles. Or ces diagonales sont des projections de diagonales de carrés qui sont parallèles dans un plan horizontal. Les projections sont donc concourantes (elles ont le même point de fuite F'). Par conséquent, les projections de sections de colonnes alignées ne sont pas homothétiques.



c. Si les colonnes sont alignées le long d'une droite parallèle à mais distincte de la droite de visée, les projections des sections ne sont pas plus homothétiques, mais elles restent asymptotiquement homothétiques : l'excentricité converge. En effet, considérons des colonnes très éloignées. Leurs projections étant très petites, appliquons leur une homothétie pour grossir l'ensemble de la figure (cela revient à éloigner le plan de projection). Le

point de fuite F' des diagonales des trapèzes s'éloigne vers la droite. Ces diagonales deviennent asymptotiquement parallèles. Le théorème de Thalès montre que les trapèzes sont asymptotiquement homothétiques, donc les ellipses inscrites dans ces trapèzes sont asymptotiquement homothétiques.

d. Une colonne dont l'axe est la droite de visée se projette sur un cercle. D'autres colonnes partagent cette propriété, car déplacer la colonne vers la droite en gardant son axe tourné vers la caméra allonge l'ellipse dans la direction Ox' (voir exercice 5) tandis que faire tourner l'axe autour d'un axe vertical allonge l'ellipse le long de Oy'.

4 Visualisation en temps réel

La structure des systèmes utilisés en synthèse d'image résulte souvent d'un compromis entre rapidité d'exécution et qualité du résultat visuel. La visualisation en temps réel impose en premier lieu d'être capable de générer des images à une vitesse suffisante pour que la discrétisation temporelle du signal affiché ne soit pas (ou pas trop) perceptible par l'œil. On appelle *FPS* (Frames Per Second) la fréquence d'échantillonnage temporel qu'un système est capable d'atteindre à un moment donné, elle est en gros proportionnelle au rapport «puissance de calcul disponible sur complexité de la scène à représenter».

Lors de l'affichage sur un écran, l'œil humain est sensible aux fréquences d'échantillonnage temporel des images jusqu'à 50 `Hz` en vision centrale, et jusqu'à 75 `Hz` en version périphérique. À 85 `Hz` on considère que les effets de scintillement/saccades sont complètement imperceptibles. En dessous de 10 `Hz`, les scènes animées sont très saccadées et la perception du mouvement devient plus difficile. À l'heure actuelle, on considère en général qu'une scène animée en temps réel est assez fluide à partir de 20–25 *FPS* (plages de fréquences utilisées pour les films, historiquement échantillonnés à 24 `Hz` au cinéma), et de bonne qualité à partir de 40–60 *FPS*.

4.1 Architecture générale des systèmes d'exploitation modernes

Avant de rentrer dans les détails du fonctionnement pratique des systèmes de visualisation temps réel, quelques rappels et notions théorique concernant l'architecture des systèmes d'exploitation actuels — en particulier la répartition du temps-machine entre les différentes tâches — sont nécessaires afin d'en mieux comprendre les enjeux.

4.1.1 Systèmes d'exploitation à temps partagé

Le système d'exploitation (en anglais OS, *Operating System*) est un ensemble de programmes assurant la liaison entre les ressources matérielles d'une machine (claviers, souris, mémoire, processeurs, écrans, etc...) et les *applications* informatiques de l'utilisateur (traitement de texte, modélisations, jeux, etc...). Son but est de fournir une *couche d'abstraction matérielle* uniformisée rendant l'écriture des programmes (plus ou moins) indé-

pendante de la machine qui va les exécuter, au moyen de l'utilisation de bibliothèques standard offrant des points d'accès génériques aux différents périphériques.

La plupart des systèmes d'exploitation ont été écrits en C ou C^{++}, faisant de ce langage la référence en terme de *portabilité* entre systèmes (i.e. : la possibilité de compiler un même code source pour utilisation avec des systèmes différents).

Process

Les systèmes d'exploitation modernes implémentent en général un comportement multitâche : plusieurs programmes peuvent y être lancés en même temps, et s'exécuter — du moins en apparence — simultanément. Les tâches actives sur une machine à un instant donné constituent l'ensemble des processus (*process* en anglais) en cours d'exécution. Le système d'exploitation se charge de gérer la mémoire et quelques autres types de données (contexte graphique, fenêtres etc...) séparément pour chaque processus, ainsi que d'assurer son isolation par rapport aux autres.

En particulier lors de son lancement, tout processus se voit attribuer un espace de *mémoire virtuelle*. Par exemple sur une machine 32 bits, l'intervalle théorique de toutes les adresses mémoire possibles s'étend de 0 à $2^{32}-1$, soit un total de plus de 4×10^9 octets (en comparaison, c'est à peu près la capacité d'un DVD). Actuellement la quantité de mémoire (RAM) des machines grand public est très inférieure à ce total : dans la pratique, au fur et à mesure qu'un processus demande au système d'allouer de la mémoire, une correspondance est établie entre les intervalles valides de la mémoire virtuelle et des emplacements de la capacité de stockage physique du matériel. Ainsi, seuls les emplacements valides correspondent effectivement à un emplacement de la mémoire physique, et toute tentative d'accès à une adresse (virtuelle) non encore allouée déclenche une exception (`segmentation fault` par exemple sous Linux).

Threads

Les *fils d'exécution* (*threads* en anglais) sont semblables aux processus en cela qu'ils constituent des suites d'instructions en langage machine ; on les appelle parfois *processus légers* pour cette raison. Toutefois les threads ne possèdent pas d'espace de mémoire virtuelle indépendant en propre : ils dépendent du contexte de mémoire virtuel d'un processus père. Ainsi, la même adresse virtuelle correspond aux mêmes données physiques lorsqu'on se place dans le contexte de deux threads du même père.

Lors du lancement d'un processus le système crée automatiquement un premier thread (le *thread principal*) : par exemple en C, c'est la fonction `main` qui y est exécutée ; souvent, on identifie la notion de processus avec celle de son thread principal. Lorsque le thread principal se termine (en quittant la fonction `main`), on considère en général que le processus père est terminé, et les éventuels autres threads qui en dépendent sont interrompus.

Signaux

Lorsqu'un programme lance plusieurs threads, il est parfois nécessaire de synchroniser l'accès aux données. Le cas le plus fréquent est celui d'un thread qui récolte des données, et d'un autre qui les traite. On peut donner comme exemple un correcteur d'orthographe dans un traitement de texte moderne, qui permet de taper du texte en même temps : pendant qu'un thread vérifie l'orthographe, un autre se charge de récolter les lettres tapées sur le clavier. Plutôt que de patienter dans une boucle à l'intérieur du thread de traitement que des données arrivent — méthode qui demande de tester une condition répétitivement, et consomme beaucoup de puissance de calcul — il est recommandé d'utiliser un *signal*. Ce mécanisme, géré par le système, permet à un thread de patienter jusqu'à ce qu'un signal émis par un autre thread soit reçu. Son avantage est de ne consommer aucune ressource : le thread en attente est en quelque sorte dans un état de «veille», jusqu'à ce que le signal le «réveille».

Répartition du temps machine

Le code machine associé à un thread est une suite linéaire d'instructions, destinées à être exécutées les unes après les autres par un processeur. En théorie, un système multitâche complètement parallèle devrait disposer d'au moins autant de processeurs que de threads en cours. Or la plupart des machines ne disposent que d'un seul processeur — ou tout du moins d'un nombre limité. Le multitâche tel qu'il est actuellement implémenté repose sur un principe de partage du temps machine : un thread se voit attribuer une certaine durée de temps dans le processeur (on parle de *time slice*), puis c'est au tour d'un autre et ainsi de suite. Chaque changement de thread constitue une *commutation de contexte* (*context switch*). En théorie la répartition du temps de processeur entre les threads (et les processus) est équitable, mais certaines tâches critiques du système (par exemple l'affichage du pointeur de la souris) peuvent se voir attribuer une priorité supérieure aux autres.

La durée typique allouée à un thread (appelée *granularité*) est actuellement de l'ordre de quelques millisecondes (mais dépend aussi beaucoup du type de système d'exploitation). Ainsi, des commutations entre les différents threads en cours ont lieu jusqu'à plusieurs centaines de fois par secondes, donnant l'apparence d'un traitement des tâches complètement parallélisé.

4.1.2 L'interface utilisateur

Les systèmes graphiques modernes proposent des représentations 2D hiérarchisées pour faciliter les interactions avec l'utilisateur, reposant sur la notion de fenêtre. Chaque processus peut en créer un certain nombre, et être averti des différents évènements qui y sont liés (clic de souris, touches du clavier, etc...). L'organisation des différentes fenêtres d'un processus constitue son *interface graphique utilisateur* ou *GUI* (Graphical User Interface).

Fenêtres du système

Le système d'exploitation expose un certain nombre de bibliothèques permettant de gérer la création/destruction des fenêtres, ainsi que leur fonctionnement. En général les fenêtres sont de forme rectangulaire, et possèdent un certain nombre d'attributs, citons par exemple :

- coordonnées des coins dans le repère de l'écran ;
- état actif/inactif ;
- couleur ;
- texte ;
- etc...

Le système peut proposer en outre un certain nombre de classes de fenêtres natives spécialisées, par exemple :

- bouton cliquable (`Button`) ;
- case à cocher représentant un état booléen On/Off (`CheckBox`) ;
- boîte permettant de choisir entre plusieurs options dans une liste (`ListBox`) ;
- zone de saisie de texte (`Edit` ou `Memo`) ;
- système de menu (`Menu` et `MenuItem`) ;
- etc...

Le gestionnaire de fenêtres se charge de les organiser selon une double hiérarchie :

- arborescente, où les fenêtres peuvent être définies comme filles d'une fenêtre mère, par exemple un bouton déposé sur un panneau. On parle alors de sous-fenêtre ;
- de profondeur, où les sous-fenêtres d'un même parent se superposent selon un ordre défini, de celle qui est la plus «au fond» à celle qui est la plus «devant». Les occlusions qui peuvent se produire (lorsqu'une fenêtre est devant une autre et la recouvre au moins partiellement) sont respectées lors de l'affichage, et lors de la capture des évènements de la souris.

En général, c'est la fenêtre d'ordre le plus élevé dans cette double hiérarchie (c'est à dire la plus en avant plan parmi celles qui n'ont pas de parent) qui reçoit les évènements du clavier.

La gestion d'une fenêtre est à la charge du thread qui l'a créée, souvent c'est le thread principal. Lorsque celui-ci se termine, la fenêtre est automatiquement détruite si le thread ne l'a pas fait explicitement lui-même.

Pile d'événements

Les différents évènements (on parle parfois de *signaux*) associés aux fenêtres d'un thread sont accumulés dans une pile *FIFO* (*first in first out*). Cette structure de données correspond à une liste munie de trois fonctions :

- une fonction `PUSH` qui permet d'ajouter un élément en fin de liste ;
- une fonction `POP` qui renvoie l'élément situé au début, et le retire de la liste ;
- une fonction `WAIT` permettant au thread de se mettre dans un état d'attente tant que la liste est vide, jusqu'à ce qu'au moins un élément y soit stocké.

On donne une liste non exhaustive de quelques évènements courants associés à l'interface graphique :

- réaffichage d'une fenêtre (par exemple lorsqu'elle était en arrière-plan recouverte par d'autres et qu'elle passe en avant-plan) ;
- redimensionnement d'une fenêtre ;
- appui sur une touche du clavier ;
- mouvement de la souris ;
- clic de souris ;
- signal d'oisiveté (*idle*), lorsque le système a effectué toutes ses tâches et que le processeur est inutilisé.

Il faut noter que certains évènements peuvent être envoyés par le programme lui-même (par exemple pour indiquer qu'une fenêtre doit être réaffichée dès que possible).

Boucle d'événements

La gestion de l'interface graphique d'une application se fait habituellement dans le thread principal. Le programme attend que des évènements arrivent dans la pile, puis les traite séquentiellement. Ce comportement est implémenté à travers l'entrée du thread dans une *boucle d'événements* (parfois appelée *boucle principale*) qu'on peut représenter schématiquement :

```
void enterMainLoop(){
  /* Variable destinée à stocker un évènement, type
     dépendant du système d'exploitation */
  EVENT e;
  do {
    /* Attente de l'arrivée d'un évènement dans la
       pile, ne consomme pas de temps machine */
    WAIT();
    /* Extraction du premier évènement (ordre
       chronologique) */
    e=POP();
    /* Traitement de l'évènement */
    processEvent(e);
  } while (1);
}
```

Le comportement du programme est ensuite passé à une fonction de l'application qui s'occupe de traiter les évènements prélevés sur la pile. Dans la plupart des cas, l'exécution est passée à une fonction par défaut fournie par le système : par exemple, un bouton est en général affiché par le système (sauf si le programmeur désire un affichage personnalisé) et seuls les clics de la souris sur ce bouton sont effectivement associés à un traitement particulier.

La fonction de traitement de bas niveau de ces évènements prend schématiquement cette forme :

```
void processEvent(EVENT e){
  switch (e.TYPE){
    case MOUSE_MOTION:
      /* La souris a bougé */
      ...;
    case MOUSE_DOWN:
      /* Un bouton de la souris a été enfoncé */
      ...;
    case MOUSE_UP:
      /* Un bouton de la souris a été relâché */
      ...;
    case KEYBOARD:
      /* Une touche du clavier a été appuyée */
      ...;
    case WINDOW_RESHAPE:
      /* Une fenêtre a changé de taille */
      ...;
    case WINDOW_DISPLAY:
      /* Une fenêtre doit être repeinte */
      ...;
    /* Autres cas */
    ...
    case QUIT:
      /* L'utilisateur souhaite quitter le programme
         */
      exit(0);
  }
}
```

La plupart des bibliothèques d'interface utilisateur permettent toutefois une programmation plus simple et plus structurée du comportement de l'interface graphique par l'écriture de fonctions séparées pour chaque évènement (*callback*) qui sont appelées depuis la fonction générale, laquelle se charge de leur distribuer de manière appropriée les bons paramètres (*dispatching*) :

- soit par l'intermédiaire de fonctions préalablement inscrites (*registered*) dans une liste en fonction du type d'évènement qui va déclencher son appel, comme c'est le cas pour la bibliothèque `GLUT` ;
- soit par l'intermédiaire d'un système hiérarchisé de classes (*wrappers*) munies d'un certains nombre de méthodes virtuelles dont il suffit d'écrire des descendants pour changer le comportement par défaut. On parle alors de *bibliothèque de composants* : par exemple `GTK`, `Qt`, `wxWidgets`, etc...

4.1.3 Serveur graphique 3D

L'affichage de l'interface graphique est réalisé par un composant du système selon un modèle client/serveur : le serveur d'affichage. Lors de la création d'une fenêtre, une application (le *client*) envoie implicitement une requête au serveur pour allouer un *contexte* graphique associé à sa surface de dessin. Ce contexte permet de réaliser un certain nombre d'opérations et instructions graphiques de base :

- tracé de lignes ;
- coloriage de polygones ;
- rendu de texte ;
- remplissage par les couleurs des pixels d'une image ;
- etc...

Il est aussi possible de créer des contexte graphiques *offscreen* associés non pas à la surface d'une fenêtre visible, mais à un tableau de pixels stocké en mémoire représentant la surface d'une image (qui peut elle-même ensuite être dessinée sur une autre surface).

Le travail du serveur se fait dans un thread système muni d'une pile d'instructions graphiques. Le client accumule dans cette pile des instructions associées à un contexte (on parle de *mise en cache*), puis lorsqu'il en donne le signal (ou lorsque la taille de la pile dépasse une certaine valeur) le thread système vide la pile (*vide le cache*) et exécute les instructions demandées.

L'intérêt de cette architecture est multiple :

- d'abord le thread client n'est pas obligé d'attendre que les instructions graphiques qu'il envoie — qui peuvent parfois prendre du temps à s'exécuter — aient été totalement effectuées pour continuer son travail. Ainsi, si par exemple les instructions graphiques sont données dans la boucle d'évènements du thread principal (c'est le cas le plus fréquent, en réponse à un évènement signifiant qu'une fenêtre doit être repeinte) le fonctionnement des fenêtres n'est pas bloqué en attendant la fin de l'affichage : le thread peut continuer à traiter les évènements suivants de la pile pendant que le serveur d'affichage se charge d'exécuter les instructions graphiques indépendamment, et ainsi l'interface utilisateur reste réactive ;
- actuellement les machines sont souvent dotées d'un processeur supplémentaire spécialisé dans le traitement des tâches graphiques, le *GPU* (*graphic processing unit*). En général celui-ci n'est pas cadencé à la même fréquence que le processeur central (le *CPU*, *central processing unit*), la mise en cache est donc nécessaire à un fonctionnement parallèle optimal des deux processeurs ;
- certains serveurs graphiques (par exemple le serveur `X` sous Linux) permettent de «délocaliser» l'affichage sur une machine externe, munie seulement d'un processeur graphique. Les instructions peuvent alors transiter via un réseau, et ne sont effectivement traitées que sur la machine distante.

Le système graphique 3D est un sous-système spécialisé dans l'affichage de points, lignes et polygones tridimensionnels. Son fonctionnement est basé sur la même architecture client/serveur et mise en cache des instructions.

Fonctionnement du *GPU*

Le processeur graphique travaille sur une partie de la mémoire dédiée à l'affichage à l'écran : la mémoire vidéo ou *frame buffer*. Cette mémoire peut être prélevée sur une certaine partie de la mémoire vive (*RAM*) de la machine ou dans le cas de matériel haut de gamme, elle est allouée directement dans la carte graphique qui fournit elle-même la mémoire utilisée par son processeur. Des composants électroniques se chargent de convertir à tout instant cette mémoire en un signal analogique pour l'affichage à l'écran, à une fréquence dépendant des réglages de celui-ci.

Les cartes vidéo actuelles proposent un certain nombre de fonctions dédiées à la visualisation 3D (on parle parfois d'*accélérateur 3D*). En simplifiant un peu, on peut considérer que les instructions graphiques 3D (essentiellement constituées de coordonnées homogènes 3D de lignes et de polygones à tracer, ainsi que d'informations de couleur) sont d'abord passées dans le *vertex pipeline* afin d'être converties en un ensemble d'instructions 2D. Puis a lieu le tramage, opération qui consiste à déterminer les ensembles de pixels qui vont être effectivement recouverts et modifiés par ces lignes et polygones. Enfin, les coordonnées de ces pixels sont passées au *pixel pipeline* qui se charge entre autres d'interpoler les couleurs données aux sommets des polygones, faire les calculs de transparence si nécessaire et va finalement modifier effectivement la mémoire vidéo.

La présence d'un GPU n'est pas une condition obligatoire pour qu'un système dispose d'un serveur graphique 3D. En effet, dans le cas contraire les fonctionnalités 3D peuvent être implémentées dans du code exécuté sur le processeur central : on parle alors de rendu *logiciel*. Lorsque la machine est munie d'un GPU on parle inversement de rendu *matériel*, ou de rendu *accéléré* (matériellement).

Vertex pipeline ou *geometry engine*

La fonction de ce pipeline est de transformer les coordonnées homogènes 3D des *vertices* (sommets des polygones, pluriel de *vertex*) en coordonnées 2D dans le plan de la caméra, selon une matrice 4×4 de transformation projective. C'est également à cette étape que sont effectués les calculs d'éclairement, en fonction de la position et de la normale de chaque vertex. Tous les calculs associés aux différents vertices étant indépendants, les GPU modernes font un usage intensif du calcul en parallèle en utilisant plusieurs pipelines simultanés.

Tramage

C'est l'opération qui consiste à discrétiser les primitives en un ensemble de pixels. Les polyèdres sont d'abord découpés selon les bords de la surface d'affichage (éventuellement en excluant les régions recouvertes par d'autres fenêtres), puis découpés en polyèdres plus simples (généralement des triangles).

Pixel pipeline* ou *raster engine

Cette opération consiste à réaliser une interpolation linéaire des différents attributs des vertices calculés par le vertex pipeline (couleur, éclairement, transparence, coordonnées de texture) pour chaque pixel composant les primitives simples (segments et triangles). Là encore, le calcul des différents pixels pouvant se faire indépendamment, les GPU modernes utilisent plusieurs pipelines travaillant en parallèle (jusqu'à 32 pour les dernières cartes grand public).

4.2 Introduction à OpenGL

On va donner ici les principes généraux du fonctionnement d'OpenGL. Plus d'information pourra être trouvée dans la dernière partie, ainsi que sur Internet. On pourra consulter en particulier — en complément du cours et des exemples et TP disponibles sur son site – les adresses suivantes :

- `http://nehe.gamedev.net` — ou encore sa traduction en français `http://nehe.developpez.com` — contient un large ensemble de tutoriels et d'exemples d'utilisation d'OpenGL ;
- `http://www.opengl.org` est la page officielle de développement d'OpenGL. On pourra y retrouver entre autre le manuel complet des spécification de l'API, et la description des fonctions.

4.2.1 Définition

OpenGL (*open graphics library*) est un ensemble de spécifications définissant une API (*application programming interface*) multi-plateforme destinée à l'écriture d'applications utilisant des graphismes 2D et 3D. Ces spécifications introduisent environ 250 fonctions de base, une vingtaine de types simples, des constantes, ainsi que la possibilité d'inclure de nouvelles fonctionnalités par le biais d'*extensions* spécifiques à certains systèmes.

Les conventions de nommage des constantes sont les suivants :

- les trois premières lettres forment le préfixe «`GL_`» ;
- les lettres suivantes forment une suite de mots en majuscule séparés par le symbole *underscore* «`_`».

Citons comme exemples :

```
GL_COLOR_BUFFER_BIT
GL_MODELVIEW
GL_ONE
GL_TRUE
```

Les conventions de nommage des types sont les suivants :

- les deux premières lettres forment le préfixe «`GL`» si le type n'est pas pointé, sinon les trois premières lettres forment le préfixe «`PGL`» ;

– éventuellement ce préfixe est suivi de «u» lorsque le type est non signé (à l'exception de `GLenum` et `GLboolean`) ;
– les lettres suivantes forment un mot en minuscule correspondant à un type standard du C :

Types entiers :

OpenGL	C standard
(P)boolean	unsigned char (*)
(P)GL(u)byte	(unsigned) char (*)
(P)GL(u)short	(unsigned) short (*)
(P)GL(u)int	(unsigned) int (*)
(P)GLenum	unsigned int (*)
(P)GL(u)sizei	(unsigned) long int (*)

Types flottants :

OpenGL	C standard
(P)GL(u)single	(unsigned) single (*)
(P)GL(u)float	(unsigned) float (*)
(P)GL(u)double	(unsigned) double (*)

Les conventions de nommage des fonctions sont les suivantes :
– les deux premières lettres forment le préfixe «gl» ;
– les lettres suivantes forment une suite de mots juxtaposés commençant chacun par une majuscule ;
– éventuellement, les dernières lettres forment un suffixe indiquant le nombre et/ou le type des arguments de la fonction (par exemple «f» pour `float`, «3ui» pour trois `unsigned int` ou encore «4dv» pour un tableau de quatre `double` juxtaposés).

Citons comme exemples :

```
void glVertex2fv(const GLfloat *v);
void glVertex4i(const GLint x,const GLint y,const GLint z,
   const GLint w);
void glColor3ub(const GLubyte r,const GLubyte g,const
   GLubyte b);
void glMatrixMode(GLenum mode);
```

4.2.2 La librairie utilitaire GLU

GLU (*OpenGL Utility Library*) est une bibliothèque associée à OpenGL : elle vient compléter cette dernière en introduisant quelques fonctions supplémentaires, faisant appel à des commandes OpenGL standard de bas niveau. GLU apporte les fonctionnalités suivantes :
– gestion de la matrice de projection pour mettre en place une vue orthogonale ou en perspective ;
– gestion de la matrice de visualisation avec une routine pour placer la caméra ;
– redimensionnement d'images, construction automatique de mipmaps pour les textures ;
– affichage d'objets quadriques (sphère, cylindre, disque), de courbes et de surfaces de Bézier ou encore de surfaces NURBS (Non Uniform Rational B-Spline) ;
– etc...

Les conventions de nommage des fonctions et constantes de l'API GLUT sont les mêmes que pour OpenGL, à ceci près qu'il faut replacer le préfixe «`GL`» par «`GLU`» pour les constantes, et «`gl`» par «`glu`» pour les fonctions. Citons comme exemples :

```
void gluPerspective(GLdouble fovy,GLdouble aspect,GLdouble
    zNear,GLdouble zFar);

void gluOrtho2D(GLdouble left,GLdouble right,GLdouble
   bottom,GLdouble top);

GLboolean gluProject(GLdouble objx,GLdouble objy,GLdouble
   objz,const GLdouble modelMatrix[16],const GLdouble
   projMatrix[16],const GLint viewport[4],GLdouble *winx,
   GLdouble *winy,GLdouble *winz);

GLboolean gluUnProject(GLdouble winx,GLdouble winy,
   GLdouble winz,const GLdouble modelMatrix[16],const
   GLdouble projMatrix[16],const GLint viewport[4],
   GLdouble *objx,GLdouble *objy,GLdouble *objz);
```

4.2.3 Utilisation avec GLUT

La gestion des fenêtres (ou encore des surfaces de rendu offscreen, appelées *pixel buffers*) ne fait pas partie de la norme définie par OpenGl. On utilisera donc une API supplémentaire pour l'interface graphique : GLUT (*OpenGL Utility Toolkit*).

GLUT permet de réaliser de nombreuses tâches d'une manière transparente ne dépendant pas du système d'exploitation concerné :

- création de fenêtre multiples avec un contexte OpenGL automatiquement associé ;
- gestion évènementielle de l'interface graphique par l'intermédiaire de *callbacks* ;
- prise en charge des périphériques d'entrée utilisateurs standards ou non (clavier, souris, joystick, etc...) ;
- quelques fonctions utilitaires pour afficher certaines formes 3D standard (sphère, tore, cône, cube, etc...) ;
- affichage de texte avec plusieurs polices possibles ;
- etc...

Les conventions de nommage des fonctions et constantes de l'API GLUT sont les mêmes que pour OpenGL, à ceci près qu'il faut replacer le préfixe «`GL`» par «`GLUT`» pour les constantes, et «`gl`» par «`glut`» pour les fonctions.

5 Visualisation photo-réaliste

On va cette fois-ci décrire différentes méthodes de visualisation de scènes 3D réalistes. Les temps de calcul correspondant sont beaucoup plus élevés, et peuvent atteindre plusieurs heures dans le cas de scènes complexes de bonne qualité. En contrepartie, certaines

images obtenues sont parfois difficilement différentiables d'une photographie pour un œil non exercé.

5.1 Lancer de rayon

Le lancer de rayon s'appuie sur un modèle corpusculaire des phénomènes lumineux. Son algorithme repose sur le principe du *chemin inverse* : selon celui-ci, on peut reconstruire le chemin parcouru par un *rayon lumineux* incident — entendre par là la trajectoire suivie par un photon jusqu'à son absorption par un photo-récepteur — en calculant le chemin parcouru par un photon émis parallèlement depuis le point d'incidence et dans la direction opposée.

Ainsi, étant donnée la position d'une caméra et de son écran, on peut calculer l'intensité perçue en chaque pixel de l'écran par retour inverse en suivant le trajet d'un rayon lumineux «virtuel» émis depuis le pixel, dans la direction opposée de la position de la caméra.

5.1.1 Optique géométrique

Il a été mis en évidence que la vitesse v de la lumière varie en fonction du milieu qu'elle traverse. Elle est maximale et vaut c (environ 300 000 $\texttt{km.sec}^{-1}$) dans le vide. Le rapport $\frac{c}{v}$, appelé *indice optique* ou *indice de réfraction* caractérise un milieu optique ; il est toujours au moins égal à 1, et dépend en général de la longueur d'onde du rayonnement lumineux considéré, on considèrera en outre qu'il ne dépend pas de la direction du rayonnement.

Supposons qu'on dispose d'une application continue $f : [0,1] \to \mathbb{R}^3$, de classe $\mathcal{C}^1$ par morceaux telle que $f(0) = A$ et $f(1) = B$. Le graphe de cette fonction définit un chemin $\mathcal{C}$ dans l'espace, d'extrémités A et B. La dérivée de f étant continue par morceaux, on peut définir l'*abcisse curviligne* $s(t)$:

$$s : \begin{cases} [0,1] \longrightarrow \mathbb{R}^+ \\ t \longmapsto s(t) = \displaystyle\int_0^t \|f'(t)\| dt \ . \end{cases}$$

Il est facile de vérifier que s est croissante et que $s(t)$ est égale à la longueur du chemin parcouru depuis $t = 0$; sa valeur en $t = 1$ est donc égale à la longueur totale L du chemin. Si l'on suppose en outre que f' ne s'annule jamais, s est alors strictement croissante et réalise donc une bijection de l'intervalle $[0,1]$ sur $[0,L]$. Ainsi, par un changement de variable on peut paramétrer le chemin $\mathcal{C}$ en utilisant s elle-même. Dans ces conditions, l'équation $g(s)$ (pour $s \in [0,L]$) de l'arc $\mathcal{C}$ obtenue après changement de variable ne dépend pas de l'application f initiale, et vérifie en plus que $\|g'(s)\| = 1$ pour tout $s \in [0,L]$ — c'est à dire qu'il permet de parcourir le chemin à une vitesse constante égale à 1. On parle alors de paramétrage normal du chemin $\mathcal{C}$.

L'intérêt du paramétrage normal est qu'il permet de calculer facilement le temps de trajet T d'un photon allant de A à B en suivant le chemin $\mathcal{C}$, à une vitesse $v(s)$ qui dépend

de la position :

$$T = \int_{s\in[0,L]} \frac{ds}{v(s)} = \frac{1}{c}\int_{s\in[0,L]} n(s)ds \ .$$

Le *chemin optique* $\mathcal{L}(\mathcal{C})$ est défini comme la distance qu'aurait parcourue la lumière dans le vide pendant la même durée, c'est à dire :

$$\mathcal{L}(\mathcal{C}) = cT = \int_0^L n(s)ds \ .$$

Le *principe de Fermat* permet de déterminer, parmi l'ensemble $\mathfrak{C}(A,B)$ de tous les chemins possibles reliant deux points A et B, ceux qui peuvent effectivement être suivis par un photon. Il s'énonce très simplement :

> *La lumière se propage d'un point à un autre sur des trajectoires telles que la durée de parcours soit stationnaire.*

Par *stationnaire* on entend ici qu'une variation infinitésimale de la trajectoire du premier ordre entraine seulement une variation infinitésimale du second ordre ou plus faible encore.

Pour formaliser plus rigoureusement la notion de variation infinitésimale de trajectoire, pour deux chemins $\mathcal{C}_1$ et $\mathcal{C}_2$, respectivement de longueurs L_1 et L_2 et de paramétrages normaux g_1 et g_2, on définit la distance moyenne qui les sépare :

$$\mathbf{d}(\mathcal{C}_1,\mathcal{C}_2) = \int_0^1 \|g_1(tL_1) - g_2(tL_2)\|dt \ .$$

On vérifiera facilement que $\mathbf{d}$ définit une distance sur l'ensemble des chemins. Avec cette définition, un chemin $\mathcal{C}$ reliant deux points A et B est un chemin optique stationnaire s'il vérifie :

$$\limsup_{\substack{\mathcal{D}\in\mathfrak{C}(A,B)\\ \mathbf{d}(\mathcal{C},\mathcal{D})\to 0}} \frac{|\mathcal{L}(\mathcal{C}) - \mathcal{L}(\mathcal{D})|}{\mathbf{d}(\mathcal{C},\mathcal{D})} = 0 \ .$$

Intuitivement, on peut interpréter cette condition comme une annulation de la différentielle $d\mathcal{L}$ du chemin optique.

Pour un chemin $\mathcal{C}$, notons $\overline{\mathcal{C}}$ le même chemin parcouru dans le sens inverse. On peut vérifier, en faisant les changements de variables $s' = L - s$ et $t' = 1 - t$ dans les deux intégrales ci-dessus que :

$$\mathcal{L}(\overline{\mathcal{C}}) = \mathcal{L}(\mathcal{C}) \qquad \text{et} \qquad \mathbf{d}(\overline{\mathcal{C}},\overline{\mathcal{D}}) = \mathbf{d}(\mathcal{C},\mathcal{D}) \ .$$

Cette remarque montre qu'un chemin stationnaire inversé est encore stationnaire, et justifie donc le *principe de retour inverse* :

> *Le trajet suivi par la lumière pour aller d'un point à un autre ne dépend pas du sens de propagation de la lumière.*

Cas d'un milieu homogène

Lorsqu'on considère un milieu d'indice constant n, le segment $[A, B]$ est un chemin optique stationnaire entre les points A et B.

Démonstration : cette propriété est facile à vérifier lorsqu'on se restreint à des chemins formés de l'union de deux segments ; soit M un point quelconque distinct de A et B, on notera :
- $[AMB]$ le chemin formé par l'union des segments $[AM]$ et $[MB]$;
- h la distance de M au segment $[AB]$.

On peut alors remarquer que :

$$\mathbf{d}([AB],[AMB]) \geq \frac{h}{2} \qquad \text{et} \qquad \mathcal{L}([AMB]) = n(AM+MB) \leq n\sqrt{AB^2+h^2}\ .$$

Ainsi :

$$\frac{|\mathcal{L}([AB]) - \mathcal{L}([AMB])|}{\mathbf{d}([AB],[AMB])} \leq \frac{n|AB - \sqrt{AB^2+h^2}|}{h/2} = 2nAB\frac{\sqrt{1+\frac{h^2}{AB^2}}-1}{h}\ .$$

Il suffit alors de remarquer que $h \to 0$ lorsque $\mathbf{d}([AB],[AMB]) \to 0$, de plus en faisant un développement limité de $x \mapsto \sqrt{1+x}$ à l'ordre 1 on trouve

$$\sqrt{1+\frac{h^2}{AB^2}} - 1 = \frac{h^2}{2AB^2} + \mathcal{O}(h^4) = \mathcal{O}(h^2)$$

ce qui donne effectivement :

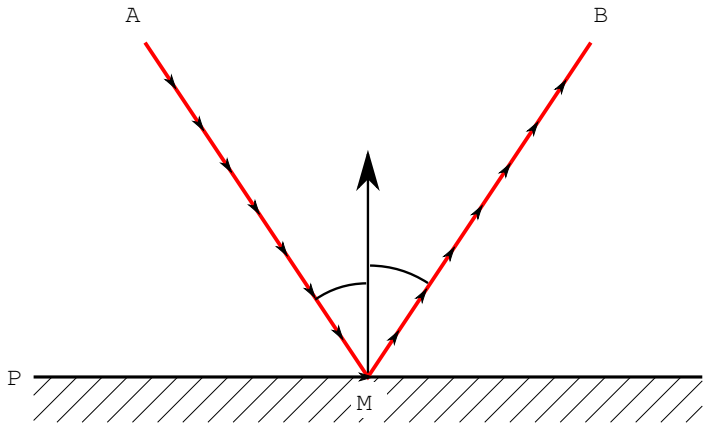
$$\limsup_{\mathbf{d}([AB],[AMB])\to 0} \frac{|\mathcal{L}([AB]) - \mathcal{L}([AMB])|}{\mathbf{d}([AB],[AMB])} \leq \limsup_{h\to 0} \frac{\mathcal{O}(h^2)}{h} = 0\ .$$

Ainsi, le segment $[AB]$ est stationnaire parmi l'ensemble des chemins $[AMB]$ ($M \in \mathbb{R}^3$). On pourrait démontrer cette propriété dans le cas général, ce qu'on ne fera pas ici.

Il est en outre possible de montrer que dans un milieu d'indice constant, le chemin rectiligne est le seul chemin stationnaire reliant les points A et B.

Cas d'une réflection

On suppose qu'on dispose d'un plan affine P de vecteur normal $\vec{n}$, non perméable à la lumière et de deux points A et B situés à égale distance et du même côté de P. L'ensemble des plans qui contiennent les points A et B engendre $\mathbb{R}^3$ tout entier, donc il est possible d'en trouver un qui est parallèle au vecteur $\vec{n}$; on l'appellera Π, et on notera M un point de $P \cap \Pi$. Dans ces conditions on peut montrer en appliquant le principe de Fermat que le seul chemin stationnaire qui passe par P est celui formé de l'union des segments $[AM]$ et $[MB]$, obtenu lorsque M est la projection orthogonale du milieu du segment $[AB]$ sur P. En d'autres termes, il faut que les angles i_1 et i_2 que forment respectivement les droites (AM) et (MB) avec $\vec{n}$ soient égaux. Cette relation est appelée la *deuxième loi de Descartes*.



Deuxième loi de Descartes, dans le cas d'une réflection : $i_1 = i_2$

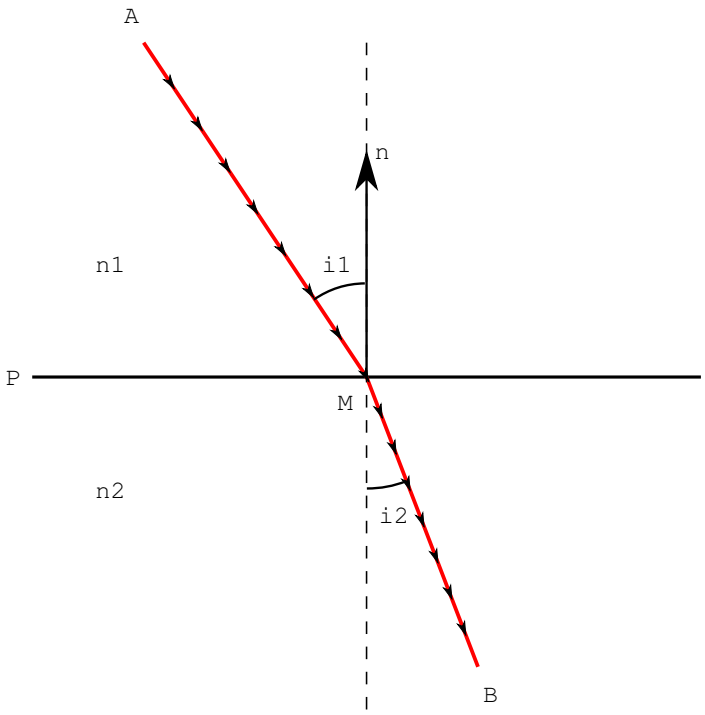
Ce principe peut se généraliser à des surfaces quelconques, à condition qu'on puisse définir un vecteur normal en tout point.

Cas d'une réfraction

On suppose qu'on dispose d'un plan affine P de vecteur normal $\vec{n}$, séparant $\mathbb{R}^3$ en deux milieux d'indices optiques respectifs n_1 et n_2 (on parle alors de *dioptre* optique). Soient aussi A et B deux points situés respectivement dans les milieux d'indice n_1 et n_2. L'ensemble des plans qui contiennent les points A et B engendre $\mathbb{R}^3$ tout entier, donc il est possible d'en trouver un qui est parallèle au vecteur $\vec{n}$; on l'appellera Π, et on notera M un point de $P \cap \Pi$. Dans ces conditions on peut montrer que le seul chemin stationnaire qui passe par P est celui formé de l'union des segments $[AM]$ et $[MB]$ obtenu lorsque les angles i_1 et i_2 que forment respectivement les droites (AM) et (MB) avec $\vec{n}$ vérifient la relation :

$$n_1 \sin i_1 = n_2 \sin i_2 \ .$$

Cette relation est appelée la *troisième loi de Descartes*.



Troisième loi de Descartes, dans le cas d'une réfraction : $n_1 \sin i_1 = n_2 \sin i_2$

Ce principe peut se généraliser à des surfaces quelconques, à condition qu'on puisse définir un vecteur normal en tout point. On peut remarquer que dans le cas où $n_1 > n_2$, si l'angle d'incidence i_1 est trop grand (en fait dès que $i_1 > \arcsin \frac{n_2}{n_1}$) il n'existe aucun angle i_2 vérifiant la relation, puisque :

$$\sin i_2 = \frac{n_1}{n_2} \sin i_1 > 1 \ .$$

On parle alors de *réflection totale* : les seuls chemins optiques stationnaires qui passent par M sont ceux correspondant à une réflection, le rayon lumineux ne peut pas passer à travers le dioptre P.

Limites de l'optique géométrique

Les seules lois de l'optique géométrique ne suffisent pas à décrire toutes les interactions observables entre le rayonnement lumineux et la matière. On peut citer comme exemple la diffusion : dans ce cadre, la lumière est réémise dans toutes les directions par la surface éclairée, ce qui contredit le principe de stationnarité. C'est pour cette raison qu'on ne suit pas la direction réfléchie pour calculer l'éclairement direct dans l'algorithme de lancer de rayon, mais qu'on cherche les rayons issus d'une source lumineuse sans se préoccuper de leur direction.

5.1.2 Algorithme

Un algorithme «naïf» de lancer de rayon peut être défini ainsi. Pour un point P et un vecteur non nul $\vec{u}$ de $\mathbb{R}^3$ on définit la fonction $I(P, \vec{u})$ par :

$I(P, \vec{u})$:

(a) pour chaque objet de la scène, déterminer ses intersections avec la demi-droite d'origine P et de direction $\vec{u}$;

(b) s'il n'y en a aucune, aller en (e), sinon aller en (c)

(c) soit Q le point d'intersection le plus proche de P avec un objet de la scène. On pose initialement $I_{\text{éclairement}} = I_{\text{réfraction}} = I_{\text{réflection}} = 0$.
- pour chaque lampe de la scène située en un point L, déterminer si le segment $[L, Q]$ rencontre un objet. Si ce n'est pas le cas, ajouter à $I_{\text{éclairement}}$ la contribution de la lampe (par exemple selon le modèle de Phong) ;
- si l'objet est réfractant, on pose $I_{\text{réfraction}} = I(Q, \vec{v})$ où $\vec{v}$ est un vecteur parallèle et de même sens à la direction de réfraction du rayon PQ (calculée à partir de la normale $\vec{n}$ à l'objet en Q) ;
- si l'objet est réfléchissant, on pose $I_{\text{réflection}} = I(Q, \vec{w})$ où $\vec{w}$ est le symétrique de $-\vec{u}$ par rapport au vecteur normal $\vec{n}$ à l'objet en Q (c'est à dire la direction de réflection du rayon PQ) ;

(d) renvoyer $I_{\text{éclairement}} + I_{\text{réfraction}} + I_{\text{réflection}}$;

(e) renvoyer 0.

Pour chaque pixel de centre P de l'écran, son intensité sera donnée par $I(C, \overrightarrow{CP})$ où C est la caméra.

5.1.3 Description des objets

Contrairement à la visualisation en temps réel où les objets sont généralement discrétisés en un ensemble de facettes polygonales (voire triangulaires) le rendu par lancer de rayon permet des représentations volumiques, sans imposer de discrétisation autre que celle de l'image en pixels.

En général, on utilise des représentations faisant intervenir l'équation implicite d'un ensemble fermé :

$$f(x, y, z) \leq 0 \ .$$

Lorsque f est continue, la frontière de l'objet est incluse dans l'ensemble des points vérifiant :

$$f(x, y, z) = 0 \ .$$

En général, on s'assurera d'utiliser des équations implicites qui ne s'annulent que sur la frontière de l'objet.

Un avantage important est que lorsque f est dérivable, on peut calculer très facilement un vecteur normal $\vec{n}$ en un point (x_0, y_0, z_0) où f s'annule. Il suffit d'utiliser le gradient :

$$\vec{n}(x_0, y_0, z_0) = \nabla f(x_0, y_0, z_0) = \left(\frac{\partial f}{\partial x}(x_0), \frac{\partial f}{\partial y}(y_0), \frac{\partial f}{\partial z}(z_0) \right) .$$

Pour une demi droite d'origine (x_0, y_0, z_0) et de vecteur directeur (u, v, w), le calcul des intersections avec la frontière d'un objet se ramène à résoudre l'équation à une inconnue (ici λ) :

$$f(x_0 + \lambda u, y_0 + \lambda v, z_0 + \lambda w) = 0 .$$

L'existence de la dérivée de f permet alors l'utilisation de méthodes numériques à convergence rapide, telle la méthode de Newton.

Représentation d'un demi-espace

Étant donné un plan P d'équation $ax + by + cz - d = 0$, les équations implicites des deux demi-espaces dont il est la frontière sont respectivement :

$$ax + by + cz - d \leq 0 \qquad \text{et} \qquad ax + by + cz - d \geq 0 .$$

Représentation d'un cube

Équation du cube de centre O dont les arêtes mesurent 1 :

$$\max(|x|, |y|, |z|) - \frac{1}{2} \leq 0$$

Représentation d'une sphère

Équation de la sphère de centre $(0, 0, 0)$ et de rayon 1 :

$$x^2 + y^2 + z^2 - 1 \leq 0 .$$

Représentation d'un cylindre

Équation du cylindre d'axe Oz à base circulaire de rayon 1 :

$$x^2 + y^2 - 1 \leq 0 .$$

De façon générale, si dans le plan Oxy l'équation implicite de la base du cylindre est $g(x, y) \leq 0$, l'équation du cylindre complet est aussi :

$$g(x, y) \leq 0 .$$

Représentation d'un cône

Équation du cône à base circulaire d'axe Oz, d'angle $\frac{\pi}{4}$ et de centre O :

$$x^2 + y^2 - z^2 \leq 0 \ .$$

De façon générale, si $g(x, y) \leq 0$ est l'équation implicite de la base du cône dans le plan Oxy, l'équation du cylindre complet est :

$$g\left(\frac{x}{|z|}, \frac{y}{|z|}\right) \leq 0 \ .$$

Représentation des quadriques

Certains moteurs de lancer de rayon s'appuient sur une description faisant intervenir uniquement des fonctions polynômes implicites de degré 2, de forme générale :

$$P(x, y, z) = Ax^2 + By^2 + Cz^2 + Dxy + Eyz + Fxz + Gx + Hy + Iz + J \leq 0$$

avec les coefficients $A, B \ldots I$ non tous nuls. Les surfaces obtenues ($P(x, y, z) = 0$) sont alors des *quadriques*, et servent de base à la construction d'objet complexes par opérations d'algèbre ensembliste. Ainsi, la sphère, les cônes et les cylindres à base circulaire sont des exemples de quadriques. L'intérêt se mesure en terme de performance : la résolution de l'équation d'intersection avec une droite se ramène à trouver les racines d'un trinôme, problème parfaitement connu et optimisable.

Image par un endomorphisme affine inversible de $\mathbb{R}^3$

On suppose que ϕ est une application affine injective (donc inversible) de $\mathbb{R}^3$ dans $\mathbb{R}^3$. Dans ces conditions, si le fermé A a pour équation :

$$f(x, y, z) \leq 0$$

alors $\phi(A)$ a pour équation :

$$f \circ \phi^{-1}(x, y, z) = f\left(\phi^{-1}(x, y, z)\right) \leq 0 \ .$$

Algèbre ensembliste

Bien que pour des raisons en autre numériques, de dérivabilité, et d'optimisation les moteurs de lancer de rayon ne les utilisent pas telles quelles, il est possible de donner facilement la formule implicite h de certaines combinaisons de deux volumes A et B dont on connait respectivement deux formules implicites f et g.

Intersection : $A \cap B$ est l'ensemble des points tels que $f(x,y,z) \leq 0$ et $g(x,y,z) \leq 0$, en d'autres termes tels que :

$$h(x,y,z) = \max(f(x,y,z), g(x,y,z)) \leq 0 \ .$$

Union : $A \cup B$ est l'ensemble des points tels que $f(x,y,z) \leq 0$ ou $g(x,y,z) \leq 0$, en d'autres termes tels que :

$$h(x,y,z) = \min(f(x,y,z), g(x,y,z)) \leq 0 \ .$$

Différence : $A \setminus \overset{\circ}{B}$ est l'ensemble des points tels que $f(x,y,z) \leq 0$ et $g(x,y,z) \geq 0$, en d'autres termes tels que :

$$h(x,y,z) = \max(f(x,y,z), -g(x,y,z)) \leq 0 \ .$$

5.1.4 Améliorations

L'algorithme naïf peut être amélioré de nombreuses façons, améliorant la qualité et le réalisme des images obtenues ou diminuant le temps de calcul.

Quantification des intensités réfléchies et réfractées

En général, les objets transparents sont à la fois réfléchissants et réfractants. Les formules de Fresnel permettent de quantifier la répartition de l'intensité réfléchie et de l'intensité réfractée en fonction de l'angle d'incidence et de l'indice du milieu. Si on note :

- R la proportion d'intensité lumineuse réfléchie,
- T la proportion d'intensité lumineuse réfractée,
- i l'angle d'incidence avec la normale au dioptre,
- r l'angle de la réfraction avec la normale au dioptre,
- n le rapport entre les indices optiques du milieu respectivement avant et après le dioptre

alors la formule de Fresnel suivante qui traduit la conservation de l'énergie de rayonnement permet de calculer T et R :

$$(1 - R^2)\cos i = nT^2 \cos r \ .$$

Degré de récursivité maximal

L'algorithme naïf tel qu'il a été énoncé plus haut peut potentiellement entrer dans une boucle infinie (prendre l'exemple d'une salle fermée recouverte de miroirs). On définit en général un degré de récursivité maximal au-delà duquel la fonction $I(P, \vec{u})$ renvoie zéro. En outre, au fur et à mesure des réflections/réfractions du rayon lumineux suivi, il est multiplié par des coefficients compris entre 0 et 1 dépendant de la nature des matériaux rencontrés et des angles d'incidence avec les surfaces. Ce produit de coefficients peut tendre assez rapidement vers zéro, en général on se donne une valeur limite au-delà de laquelle l'intensité peut être considérée comme négligeable et on arrête alors l'algorithme dès que ce seuil est atteint.

Anti-aliasing

L'algorithme de lancer de rayon peut être vu comme un échantillonnage du signal lumineux théorique en chaque pixel de l'écran. Il est donc préférable d'appliquer un filtre anti-repliement lors de la génération de l'image, mais cette opération peut s'avérer désastreuse en termes de complexité algorithmique. C'est pourquoi on utilise parfois une version approximative du filtre de sur-échantillonnage consistant à sur-échantillonner uniquement les zones de l'image présentant des variations de contraste brusques (ou en d'autres termes des hautes fréquences spatiales). On parle alors de sur-échantillonnage adaptatif (*adaptive oversampling*).

Le principe est le suivant : pour un carré $\delta \subset \mathbb{R}^2$ on définit la fonction $J(\delta)$ qui réalise une approximation de la moyenne du signal I sur ce carré.

$$J(\delta) : \begin{cases} \text{(a) calculer les intensité } I_k = I(C, \overrightarrow{CP_k}) \text{ (pour } 1 \leq k \leq 4) \text{ par la méthode du lancer de rayon où } P_1, P_2, P_3 \text{ et } P_4 \text{ sont les 4 coins du carré } \delta\,; \\ \text{(b) poser } I_0 = \dfrac{I_1 + I_2 + I_3 + I_4}{4}\,; \\ \text{(c) s'il existe } k \text{ tel que } |I_k - I_0| > \rho \text{ alors aller en (d) sinon aller en (e)}\,; \\ \text{(d) en notant } \delta_1, \delta_2, \delta_3 \text{ et } \delta_4 \text{ les 4 sous-carrés obtenus en découpant } \delta \text{ selon ses médianes, renvoyer } \dfrac{J(\delta_1) + J(\delta_2) + J(\delta_3) + J(\delta_4)}{4}\,; \\ \text{(e) renvoyer } I_0. \end{cases}$$

Pour chaque pixel (u, v) de l'image, on lui affectera alors l'intensité $J(\delta_{u,v})$ où $\delta_{u,v}$ est son support. La constante ρ utilisée est un seuil arbitraire pour détecter les variations brusques de contraste : plus elle est petite plus l'algorithme va être précis. On remarquera que sur une image, le support de chaque pixel a 3 sommets en commun avec les pixels suivants (à part ceux de la dernière colonne ou de la dernière ligne). Pour une image de couleur uniforme de taille $m \times n$ il suffira donc de lancer seulement $(m + 1) \times (n + 1)$ rayons (en réutilisant ceux des pixels déjà calculés) ce qui revient asymptotiquement au même que pour l'algorithme sans filtrage où il fallait en lancer $m \times n$.

5.2 Radiosité

La radiosité est une méthode de calcul d'éclairement utilisant un *modèle d'illumination global*, par opposition aux modèles locaux (Gouraud ou Phong). Une fois ce calcul effectué, il est possible d'afficher la scène avec une méthode de rendu quelconque (OpenGL ou lancer de rayon) en utilisant les valeurs d'éclairement données par le calcul de radiosité, par exemple en les substituant au terme d'éclairement constant dans le modèle de Gouraud.

Principe

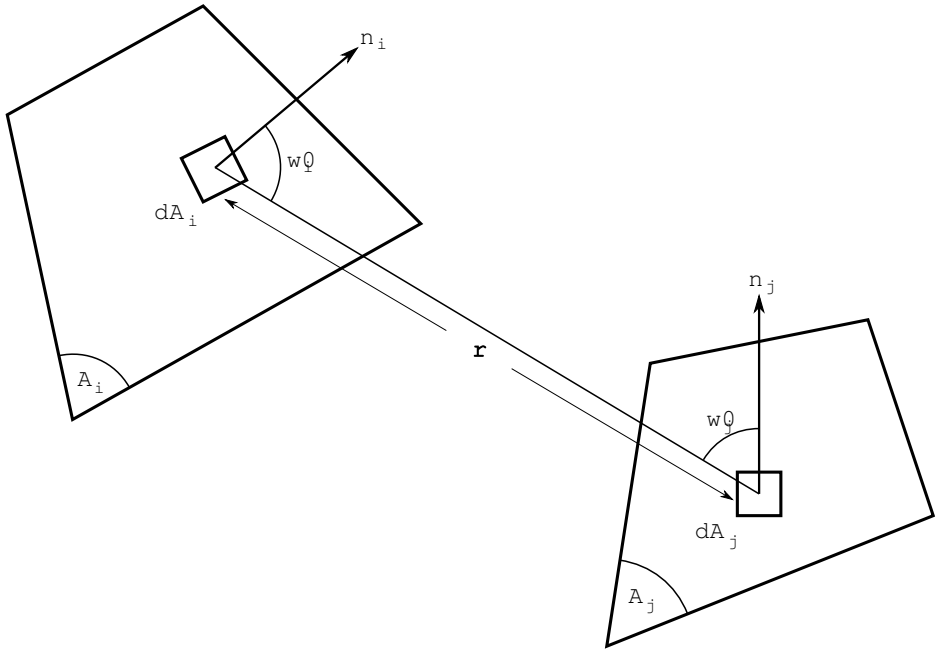
Chaque objet d'une scène émet un rayonnement lumineux uniforme dans toutes les directions, qui est la somme de deux composantes :

- un rayonnement intrinsèque propre (source lumineuse primaire) ;
- une énergie réémise par diffusion atomique, proportionnelle à l'énergie reçue en provenance des autres objets de la scène.

L'étude de la *radiosité* d'une scène consiste à calculer l'énergie par unité de temps — c'est à dire une puissance, exprimée en Watts — émise par chaque objet. Pour cela, il suffit de résoudre une équation liée à l'équilibre énergétique de la scène.

Facteur de forme

On va supposer ici que les objets de la scène ont été discrétisés en une famille $A_1, \ldots A_N$ de facettes suffisamment petites. On définit le facteur de forme $F_{i,j}$ comme le rapport entre l'énergie reçue par la facette A_j et l'énergie émise par la facette A_i, en d'autres termes comme la fraction du rayonnement émis par A_j — toujours comprise entre 0 et 1 — qui est reçue par A_i.



Calcul du facteur de forme $F_{i,j}$

En notant $H_{i,j}$ le facteur de visibilité entre deux points situés respectivement sur A_i et A_j, valant 1 si aucun objet ne les sépare, 0 sinon, on définit le facteur de forme $dF_{di,dj}$ entre deux éléments de surface différentielle dA_i et dA_j des facettes A_i et A_j :

$$dF_{di,dj} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{i,j} dA_i dA_j \ .$$

Ici θ_i et θ_j sont les angles du segment reliant les deux surfaces avec respectivement la normale à A_i et A_j, et r est la longueur de ce segment.

Le facteur de forme entre A_i et A_j s'exprime comme une double intégrale surfacique (sur A_i et A_j) du facteur de forme différentiel $dF_{di,dj}$:

$$F_{i,j} = \int_{A_i}\int_{A_j} dF_{di,dj} dA_i dA_j = \int_{A_i}\int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{i,j} dA_j dA_i = \int_{A_i} F_{di,j} dA_i \ .$$

Ici, $F_{di,j}$ est le facteur de forme entre la surface différentielle dA_i et la facette A_j :

$$F_{di,j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{i,j} dA_j \ .$$

Le calcul de l'intégrale double qui définit $F_{i,j}$ est compliqué, même dans des cas simples, on va faire l'approximation suivante, qui se justifie si les facettes sont suffisamment petites :

$$F_{i,j} \approx F_{di,j} \ .$$

On se ramène donc au problème plus simple de calculer une seule intégrale surfacique. Une façon de réaliser ce calcul est de faire la projection des parties visibles de A_j depuis dA_i sur une sphère unitaire centrée sur dA_i, puis de diviser la surface de la projection sur dA_i par celle du cercle unité.

Certains systèmes utilisent des facettes qui ne sont pas planaires, mais courbes : dans ce cas le facteur de forme $F_{i,i}$ peut être non nul (c'est-à-dire qu'une facette peut s'éclairer elle-même).

Algorithme

Une fois les facteurs de forme calculés, il ne reste plus qu'à écrire une équation d'équilibre énergétique pour chaque facette A_i. En notant B_i la radiosité de la surface (exprimée par exemple en Watts par unité de surface) et E_i son rayonnement intrinsèque propre il vient :

$$B_i = E_i + \rho_i \sum_j B_j F_{j,i} \frac{A_j}{A_i} \ .$$

Ici, le coefficient ρ_i (compris entre 0 et 1) est la réflectivité de la facette. En remarquant que l'approximation qu'on a faite pour les facteurs de forme entraine que $A_i F_{i,j} = A_j F_{j,i}$ on trouve alors :

$$B_i = E_i + \rho_i \sum_j B_j F_{i,j} \ .$$

Cette équation peut s'écrire pour chacune des facettes A_i, on obtient finalement un système linéaire de N équations à N inconnues :

$$\begin{pmatrix} 1-\rho_1 F_{1,1} & -\rho_1 F_{1,2} & \dots & -\rho_N F_{1,N} \\ -\rho_2 F_{2,1} & 1-\rho_2 F_{2,2} & \dots & -\rho_N F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{N,1} & -\rho_n F_{N,2} & \dots & 1-\rho_N F_{N,N} \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{pmatrix}$$

Il suffit alors de résoudre ce système par une méthode itérative efficace (par exemple Gauss-Siedel) pour trouver les coefficients B_i recherchés avec la précision désirée.

Troisième partie

Utilisation d'OpenGL

Contenu

6 Gestion d'une interface graphique utilisateur

On va énoncer ici les fonctionnalités de base de l'API GLUT. On pourra trouver un manuel détaillé du fonctionnement de l'API, en particulier la liste des fonctions existantes à cette adresse : `http://www.opengl.org/documentation/specs/glut/spec3/node1.html`.

6.1 Initialisation

Avant toute chose il faut commencer par initialiser la librairie GLUT, ainsi que le mode d'affichage.

```
void glutInit(int* argc,char** argv);
void glutInitDisplayMode(unsigned int mode);
```

`glutInit` — Il est **obligatoire** d'appeler cette fonction avant toute utilisation des autres fonctions de GLUT. Les paramètres `argv` et `argc` correspondent aux paramètres passés à la fonction `main` d'un programme standard en C. De cette façon, il est possible à l'utilisateur d'initialiser certaines propriétés de la librairie GLUT. Par exemple, le paramètre `-gldebug`, lorsqu'il est spécifié sur la ligne de commande, force la vérification automatique des erreurs OpenGL potentielles, afin de déboguer facilement les programmes.

`glutInitDisplayMode` — Définit le mode d'affichage. Le paramètre `mode` peut être une combinaison bit-à-bit (avec l'opérateur `|`) des constantes suivantes :

- `GLUT_RGB` — affichage utilisant le modèle de couleurs RGB. Il est recommandé de l'utiliser ;
- `GLUT_DOUBLE` — utiliser un double buffer. Cette option permet de supprimer des effets de clignotement lors de l'affichage de scènes animées. Il faudra alors utiliser la fonction `glutSwapBuffers` à la fin de la fonction d'affichage, sinon rien ne sera visible ;
- `GLUT_DEPTH` — utiliser un tampon de profondeur (*depth buffer*).

6.2 Création d'une fenêtre

La création d'une fenêtre est très simple avec GLUT : il suffit d'appeler la fonction `glutCreateWindow` (après avoir initialisé la position et la taille désirées).

```
void glutInitWindowSize(int width,int height);
void glutInitWindowPosition(int x,int y);
int glutCreateWindow(char* name);
```

`glutInitWindowSize` — Définit la taille par défaut des fenêtres qui vont être créées.

`glutInitWindowPosition` — Définit la position par défaut des fenêtres qui vont être créées. Les coordonnées `x` et `y` définissent les coordonnées du coin supérieur gauche des fenêtres en unité de pixels, relativement au coin supérieur gauche de l'écran.

`glutCreateWindow` — Crée une fenêtre selon les dimensions et le mode d'affichage courant. Le paramètre `name` sera le titre de la fenêtre (généralement affiché sur la bordure du dessus par le système). Le résultat de la fonction est un entier identifiant de manière unique la fenêtre créée.

6.3 Définition des callbacks

Une fois une fenêtre créée il faut définir le comportement du programme en fonction des divers évènements que le système va transmettre.

```
void glutReshapeFunc(void (*func)(int width,int height));
void glutDisplayFunc(void (*func)(void));
void glutKeyboardFunc(void (*func)(unsigned char key,int x
    ,int y));
void glutMouseFunc(void (*func)(int button, int state,int
   x,int y));
void glutMotionFunc(void (*func)(int x,int y));
void glutPassiveMotionFunc(void (*func)(int x,int y));
void glutSpecialFunc(void (*func)(int key,int x,int y));
void glutIdleFunc(void (*func)(void));
```

`glutReshapeFunc` — Définit la fonction de callback personnalisée qui sera appelée dès que la fenêtre change de taille. Les paramètres `width` et `height` contiendront la nouvelle taille. Par défaut, GLUT se contente de définir le viewport (`glViewport`) lorsqu'aucun callback n'est spécifié. Une implémentation typique de cette fonction consiste à définir le viewport et la matrice de projection.

`glutDisplayFunc` — Définit la fonction de réaffichage qui sera appelée lorsque nécessaire. Il est possible à tout moment de forcer le système à réafficher la fenêtre dès que possible en utilisant `glutPostRedisplay()`. Il est recommandé de n'utiliser des fonctions OpenGL qu'à cet endroit (à l'exception du viewport et de la matrice de projection qui peuvent être modifiés dans le callback de redimensionnement).

`glutKeyboardFunc` — Définit la fonction appelée lorsque l'utilisateur appuie sur une touche. Le paramètre `key` contiendra le numéro ASCII de la touche appuée, `x` et `y` les coordonnées de la souris au moment où la touche a été appuyée (relativement au coin supérieur gauche de la fenêtre).

`glutMouseFunc` — Définit la fonction appelée lorsque l'utilisateur clique à l'intérieur de la fenêtre. Le paramètre `button` pourra prendre les valeurs suivantes :
- `GLUT_LEFT_BUTTON` lorsqu'un clic avec le bouton gauche se produit ;
- `GLUT_MIDDLE_BUTTON` pour un clic avec le bouton du milieu ;
- `GLUT_RIGHT_BUTTON` pour le bouton droit.

Le paramètre `state` pourra prendre les valeurs suivantes :
- `GLUT_DOWN` lorsque le bouton est enfoncé ;

– `GLUT_UP` lorsqu'il est relevé.

Les paramètres `x` et `y` contiendront les coordonnées de la souris au moment du clic (relativement au coin supérieur gauche de la fenêtre).

`glutMotionFunc` — Définit la fonction appelée lorsque l'utilisateur bougera la souris en maintenant un bouton enfoncé. Les paramètres `x` et `y` contiendront les coordonnées de la souris au moment du clic (relativement au coin supérieur gauche de la fenêtre).

`glutPassiveMotionFunc` — Définit la fonction appelée lorsque l'utilisateur bougera la souris au-dessus de la fenêtre (sans maintenir de bouton enfoncé). Les paramètres `x` et `y` contiendront les coordonnées de la souris (relativement au coin supérieur gauche de la fenêtre).

`glutSpecialFunc` — Définit la fonction appelée lorsque l'utilisateur appuiera sur une touche non-ASCII du clavier. Le paramètre `key` pourra prendre les valeurs suivantes :
– `GLUT_KEY_F1` ... `GLUT_KEY_F12` pour les touches F1 à F12 ;
– `GLUT_KEY_LEFT`, `GLUT_KEY_UP`, `GLUT_KEY_RIGHT` et `GLUT_KEY_DOWN` pour les 4 flèches de direction ;
– `GLUT_KEY_PAGE_UP`, `GLUT_KEY_PAGE_DOWN`, `GLUT_KEY_HOME`, `GLUT_KEY_END`, ou encore `GLUT_KEY_INSERT` pour les autres cas.

Les paramètres `x` et `y` contiendront les coordonnées de la souris (relativement au coin supérieur gauche de la fenêtre) au moment où la touche a été appuyée.

`glutIdleFunc` — Définit la fonction d'oisiveté qui sera appelée dès que le système dispose de ressources inutilisées. Cette fonction est utilisée pour représenter des scènes animées : il suffit d'appeler `glutPostRedisplay()` dans la fonction d'oisiveté pour que l'affichage de la fenêtre soit rafraichi aussi vite que le permet le système. L'intérêt est qu'ainsi, les autres évènements de l'interface graphique pourront être traités en priorité, et le programme va rester réactif aux actions de l'utilisateur.

6.4 Entrée dans la boucle principale

Une fois les callbacks définis, on peut entrer dans la boucle principale. Cette fonction (`glutMainLoop`) est bloquante, le programme se terminera dès qu'une des fenêtres sera fermée par l'utilisateur (ou par l'appel de la fonction `exit` dans un callback, par exemple). Le code situé après `glutMainLoop` ne sera dans tous les cas jamais exécuté.

Il est en outre obligatoire d'avoir créé au moins une fenêtre avant de lancer la boucle principale, sinon on déclenche une erreur.

```
void glutMainLoop(void);
```

Pour résumer, on peut donner un schéma global de la structure d'un programme GLUT :

```
/* Callback de redimensionnement */
void reshapeFunc(int width,int height){
  /* Définition de la zone de dessin aux dimensions de
     la fenêtre */
```

```
  glViewport(0,0,width,height);
  /* Définition de la matrice de projection */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  /* Vue en perspective de même ratio (width/height)
     que la fenêtre */
  gluPerspective(70,(float) width/height,1,20);
}

/* Callback de réaffichage */
void displayFunc(void){
  /* Tracé avec OpenGL (voir plus bas) */

  /* ... */

  /* Fin du tracé */
  glFlush();
  /* Mise à jour du buffer visible */
  glutSwapBuffers();
}

/* Callback pour les touches ASCII du clavier */
void keyboardFunc(unsigned char key,int x,int y){
  switch(key){
    case 'q':
      exit(0);
      break;
    /* ... autres cas ... */
  }
}

/* Callback d'oisiveté */
void idleFunc(void){
  /* Réafficher la fenêtre dès que possible (pour une
     animation par exemple) */
  glutPostRedisplay();
}

/* Programme principal */
int main(int argc,char* argv[]){
  /* Initialisation du système */
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH |
    GLUT_DOUBLE);
  /* Création d'une fenêtre */
  glutInitWindowPosition(300,200);
```

```
    glutInitWindowSize(800,600);
    glutCreateWindow("Programme de démonstration");
    /* Définition des callbacks */
    glutReshapeFunc(reshapeFunc);
    glutDisplayFunc(displayFunc);
    glutKeyboardFunc(keyboardFunc);
    glutIdleFunc(idleFunc);
    /* Entrée dans la boucle principale */
    glutMainLoop();
}
```

7 Affichage avec OpenGL

On va donner de manière générale, et sans trop rentrer dans les détails, le plan d'un programme qui réalise un affichage avec OpenGL.

7.1 Remise à zéro de la mémoire graphique

Avant de commencer à afficher des objets, il est d'usage d'effacer l'image stockée dans la mémoire graphique.

```
void glViewport(GLint x,GLint y,GLsizei width,GLsizei
    height);
void glClearColor(GLfloat red,GLfloat green,GLfloat blue,
    GLfloat alpha);
void glClear(GLenum mask);
```

`glViewport` — Définit une zone d'affichage rectangulaire relativement à la surface de la fenêtre. Les coordonnées entières `x` et `y` correspondent à la position du coin inférieur gauche du rectangle par rapport au coin inférieur gauche de la fenêtre. Cette fonction doit être appelée en premier (par exemple dans le callback GLUT de redimensionnement de la fenêtre). Les coordonnées normalisées (x, y, z) après multiplication par les matrices de modélisation et de projection sont transformées en coordonnées (u, v) à l'écran en utilisant la formule :

$$u = x_{\text{viewport}} + \frac{x+1}{2\text{width}} \qquad \text{et} \qquad v = y_{\text{viewport}} + \frac{y+1}{2\text{height}} \ .$$

Rappel : par défaut GLUT définit un callback de redimensionnement qui définit un viewport recouvrant la totalité de la surface de la fenêtre :

```
glViewport(0,0,width,height);
```

Il est donc inutile de le redéfinir si on ne souhaite pas effectuer d'opération supplémentaire en cas de redimensionnement.

`glClearColor` — Définit la couleur (RGB) utilisée pour effacer le tampon d'affichage. Les paramètres `red`, `green` et `blue` doivent être trois valeurs flottantes comprises entre 0 et 1. Le paramètre `alpha` correspond à une valeur de transparence qu'on n'utilisera pas.

`glClear` — Efface certains tampons de la mémoire graphique. Le paramètre `mask` doit être une combinaison bit-à-bit (avec l'opérateur `|`) des constantes suivantes :
- `GL_COLOR_BUFFER_BIT` indique d'effacer les couleurs de la mémoire vidéo, en les remplaçant par celle spécifiée par `glClearColor` (noir par défaut) ;
- `GL_DEPTH_BUFFER_BIT` indique d'effacer les valeurs du tampon de profondeur (si disponible). Indispensable si celui-ci est activé (sinon on risque de ne rien voir s'afficher).

7.2 Définition des matrices de projection/visualisation

Avant d'être transformées en coordonnées de pixels dans le repère de l'écran, les coordonnées homogènes des sommets des primitives d'OpenGL sont préalablement multipliées (à gauche) par une première matrice (la matrice de modélisation), puis par une seconde matrice (la matrice de projection). Ces deux matrices sont de taille 4×4 et correspondent à deux applications projectives travaillant sur des coordonnées homogènes. En général on stocke une application projective correspondant à la caméra choisie dans la matrice de projection, et une application affine dans la matrice de modélisation.

```
void glMatrixMode(GLenum mode);
void glPushMatrix(void);
void glPopMatrix(void);
void glLoadMatrixf(const GLfloat *m);
void glMultMatrixf(const GLfloat *m);
```

`glMatrixMode` — Définit quelle matrice va être modifiée par les fonctions de manipulation matricielle. Le paramètre `mode` peut prendre les valeurs suivantes :
- `GL_PROJECTION` pour la matrice de projection. Généralement celle-ci est définie dans le callback GLUT de redimensionnement de la fenêtre ;
- `GL_MODELVIEW` pour la matrice de modélisation ;
- `GL_TEXTURE` pour la matrice des coordonnées de texture.

`glPushMatrix` — Sauvegarde la matrice courante dans la pile de matrices.

`glPopMatrix` — Remet la matrice courante à sa valeur précédemment sauvegardée dans la pile. Les appels à `glPushMatrix` et `glPopMatrix` doivent former des paires ordonnées, éventuellement imbriquées (c'est à dire que tout appel à `glPushMatrix` doit être suivi d'un appel à `glPopMatrix`).

`glLoadMatrixf` — Remplace les 16 coefficients de la matrice homogène 4×4 courante par ceux spécifiés dans le tableau `m`. Celui-ci doit contenir 16 coefficients flottants

contigus, la nouvelle matrice est alors :

$$\begin{bmatrix} \texttt{m[0]} & \texttt{m[4]} & \texttt{m[8]} & \texttt{m[12]} \\ \texttt{m[1]} & \texttt{m[5]} & \texttt{m[9]} & \texttt{m[13]} \\ \texttt{m[2]} & \texttt{m[6]} & \texttt{m[10]} & \texttt{m[14]} \\ \texttt{m[3]} & \texttt{m[7]} & \texttt{m[11]} & \texttt{m[15]} \end{bmatrix}$$

`glMultMatrixf` — Multiplie la matrice courante (à droite) par celle dont les 16 coefficients sont stockés dans `m`. Ainsi, si on note `c` la matrice courante avant l'appel de la fonction, la nouvelle matrice courante s'écrit :

$$\begin{bmatrix} \texttt{c[0]} & \texttt{c[4]} & \texttt{c[8]} & \texttt{c[12]} \\ \texttt{c[1]} & \texttt{c[5]} & \texttt{c[9]} & \texttt{c[13]} \\ \texttt{c[2]} & \texttt{c[6]} & \texttt{c[10]} & \texttt{c[14]} \\ \texttt{c[3]} & \texttt{c[7]} & \texttt{c[11]} & \texttt{c[15]} \end{bmatrix} \times \begin{bmatrix} \texttt{m[0]} & \texttt{m[4]} & \texttt{m[8]} & \texttt{m[12]} \\ \texttt{m[1]} & \texttt{m[5]} & \texttt{m[9]} & \texttt{m[13]} \\ \texttt{m[2]} & \texttt{m[6]} & \texttt{m[10]} & \texttt{m[14]} \\ \texttt{m[3]} & \texttt{m[7]} & \texttt{m[11]} & \texttt{m[15]} \end{bmatrix}$$

Il est très rare d'avoir à spécifier manuellement une matrice avec OpenGL. On préfèrera utiliser des fonctions prédéfinies multipliant la matrice courante avec celle de certaines applications affines simples.

```
void glLoadIdentity(void);
void glTranslatef(GLfloat x,GLfloat y,GLfloat z);
void glScalef(GLfloat x,GLfloat y,GLfloat z);
void glRotatef(GLfloat angle,GLfloat x,GLfloat y,GLfloat z
    );
```

`glLoadIdentity` — Stocke la matrice homogène de l'identité dans la matrice courante :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`glTranslatef` — Multiplie la matrice courante (à droite) par celle de la translation de vecteur $(\texttt{x}, \texttt{y}, \texttt{z})$:

$$\begin{bmatrix} 1 & 0 & 0 & \texttt{x} \\ 0 & 1 & 0 & \texttt{y} \\ 0 & 0 & 1 & \texttt{z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`glScalef` — Multiplie la matrice courante (à droite) par celle de la dilatation de facteurs `x`, `y` et `z` selon les trois axes (Ox), (Oy) et (Oz) :

$$\begin{bmatrix} \texttt{x} & 0 & 0 & 0 \\ 0 & \texttt{y} & 0 & 0 \\ 0 & 0 & \texttt{z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`glRotatef` — Multiplie la matrice courante (à droite) par celle de la rotation d'angle `angle` dans le plan perpendiculaire au vecteur (`x`, `y`, `z`) ;

$$\begin{bmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ zx(1-c)-ys & zy(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

où $c = \cos(\texttt{angle})$, $s = \sin(\texttt{angle})$ et x, y, z sont les coordonnées du vecteur fourni, après l'avoir normalisé.

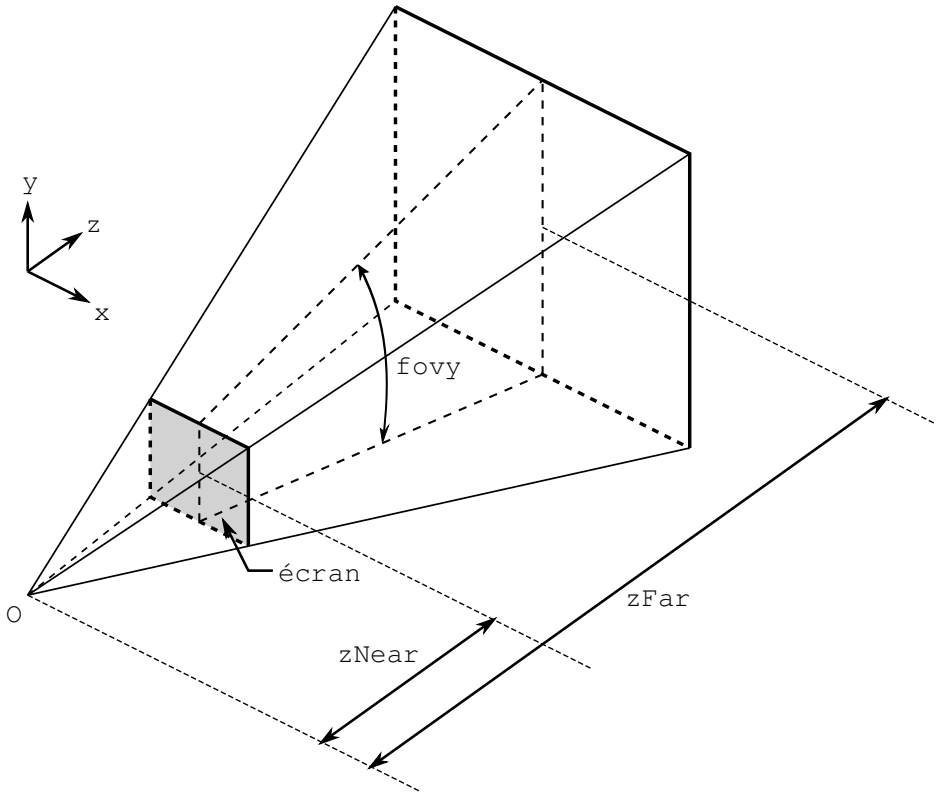
Afin de définir la caméra, on dispose encore de quelques fonctions pratiques introduites par la librairie `glu`. Elles permettent de définir le volume de la pyramide de vision ainsi que sa position.

```
void gluPerspective(GLdouble fovy,GLdouble aspect,GLdouble
    zNear,GLdouble zFar);
void gluOrtho2D(GLdouble left,GLdouble right,GLdouble
    bottom,GLdouble top);
void gluLookAt(GLdouble eyeX,GLdouble eyeY,GLdouble eyeZ,
    GLdouble centerX,GLdouble centerY,GLdouble centerZ,
    GLdouble upX,GLdouble upY,GLdouble upZ)
```

`gluPerspective` — Multiplie la matrice courante par celle de la projection perspective sur un écran, en définissant une pyramide de vision ayant pour sommet l'origine. L'écran est un rectangle de rapport *largeur*/*hauteur* égal à `aspect`, perpendiculaire à l'axe Oz et situé à distance `znear` du sommet. Le paramètre `fovy` est l'angle d'ouverture vertical de la pyramide, `zfar` la distance maximale de vision par rapport à l'origine. La matrice utilisée est la suivante (ici $\cot x = \frac{1}{\tan x}$) :

$$\begin{bmatrix} \frac{1}{\texttt{aspect}} \cot\left(\frac{\texttt{fovy}}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\texttt{fovy}}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{\texttt{zFar}+\texttt{zNear}}{\texttt{zNear}-\texttt{zFar}} & \frac{2\times\texttt{zFar}\times\texttt{zNear}}{\texttt{zNear}-\texttt{zFar}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Cette fonction s'utilise généralement avec la matrice de projection, juste après y avoir stocké l'identité.



La pyramide de vision telle que définie par `gluPerspective`

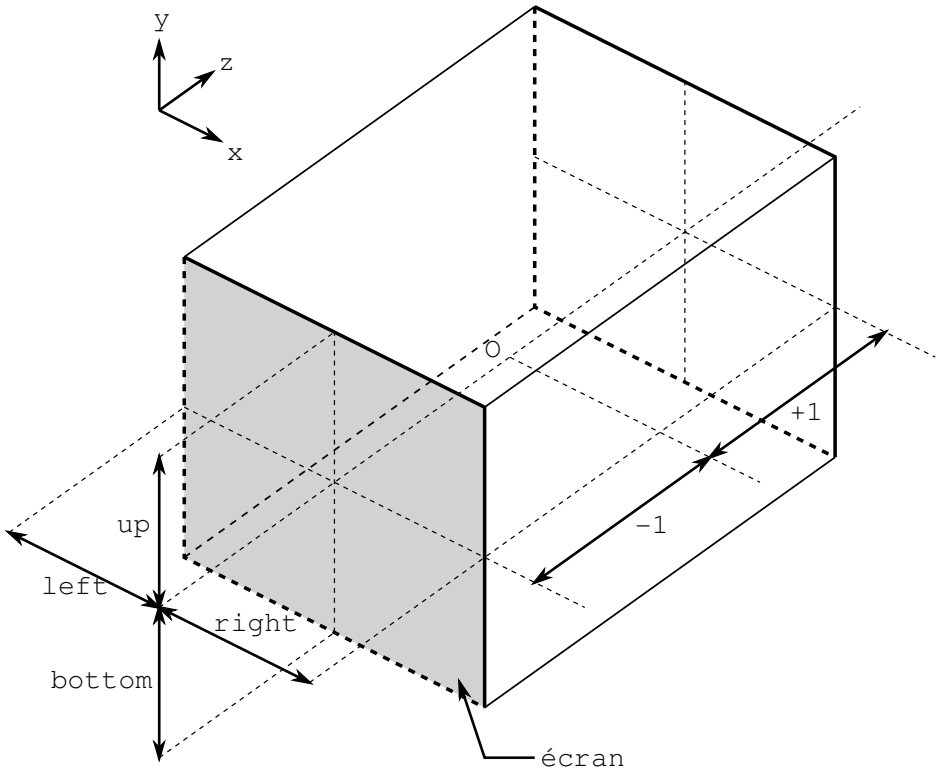
`gluOrtho2D` — Multiplie la matrice courante par celle de la projection orthogonale sur un écran. Le volume de vision est le pavé

$$[\texttt{left}, \texttt{right}] \times [\texttt{bottom}, \texttt{up}] \times [-1, 1] \ .$$

ce qui correspond à la projection à travers un point situé à l'infini. La matrice utilisée est la suivante :

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cette fonction s'utilise généralement avec la matrice de projection, juste après y avoir stocké l'identité.



La pyramide de vision telle que définie par `gluOrtho2D`

`gluLookAt` — Déplace la caméra au point `eye`, regardant dans la direction du point `center`, avec la verticale de l'écran orientée selon la direction `up`. Cette fonction s'utilise généralement avec la matrice de modélisation, juste après y avoir stocké l'identité.

7.3 Gestion des sources lumineuses

OpenGL permet de gérer simultanément jusqu'à 8 lampes, numérotées `GL_LIGHT0`, ..., `GL_LIGHT7`. Pour activer les calculs d'éclairage ainsi que la première lampe il suffit d'utiliser :

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Par défaut la première lampe est située à l'infini sur l'axe Oz négatif, de couleur blanche. Cette configuration est suffisante pour des cas simples, puisque l'éclairement est appliqué après transformations des coordonnées par la matrice de projection. Ainsi, la scène apparait comme éclairée par une lampe torche située derrière la caméra.

Le modèle complet est assez complexe à mettre en œuvre car il fait intervenir une vingtaine de paramètres. On peut toutefois utiliser une version simplifiée :

```
glEnable(GL_COLOR_MATERIAL);
```

Une fois que ce modèle simplifié est activé, on peut faire varier facilement les principaux coefficients du modèle de Phong en utilisant simplement les fonctions `glColor3X(r,g,b)` avec les couleurs désirées.

La direction de la normale intervient dans les calculs : si celle-ci est situé de l'autre côté de la source lumineuse, le point ne sera pas éclairé. Pour que les deux côtés soient éclairés on peut utiliser :

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,1);
```

Pour plus d'information sur les paramètres du modèle d'éclairement on pourra se reporter aux programmes interactifs de démonstration du cours.

7.4 Affichage des primitives graphiques

L'affichage des primitives OpenGL se fait selon un ordre précis. Seules les fonctions modifiant les paramètres de vertices sont autorisées entre les fonctions `glBegin` et `glEnd`. Par exemple, tenter de changer la matrice courante provoquera une erreur.

A – D'abord on spécifie quel type de primitive on veut dessiner avec la fonction `glBegin`.

```
void glBegin(GLenum mode);
```

On explicitera par la suite différentes valeurs possibles du paramètre `mode`.

B – On donne ensuite un certains nombres de vertices définissant les sommets de la primitive. Chaque vertex possède des attributs optionnels et un attribut obligatoire :

- on peut spécifier une couleur (ici le suffixe '`X`' correspond au type d'argument de la fonction utilisée, par exemple '`f`' pour des `float`)

  ```
  void glColor3X(r,g,b);
  ```

- on peut spécifier une normale

  ```
  void glNormal3X(x,y,z);
  ```

- on peut spécifier des coordonnées de texture

  ```
  void glTexCoord2X(u,v);
  ```

- enfin dans tous les cas et à la fin on doit spécifier les coordonnées du vertex

  ```
  void glVertex2X(x,y);
  void glVertex3X(x,y,z);
  void glVertex4X(x,y,z,t);
  ```

  La version `glVertex2X` est identique à `glVertex3X` avec `z` $= 0$. Le paramètre `t` de `glVertex4X` permet de donner la $4^{\text{ème}}$ composante des coordonnées homogènes (elle vaut implicitement 1 pour les autres versions à 2 ou 3 coordonnées).

C – Pour finir on doit obligatoirement appeler la fonction `glEnd`.

```
void glEnd(void);
```

C'est à ce moment seulement qu'OpenGL peut commencer à tracer la primitive.

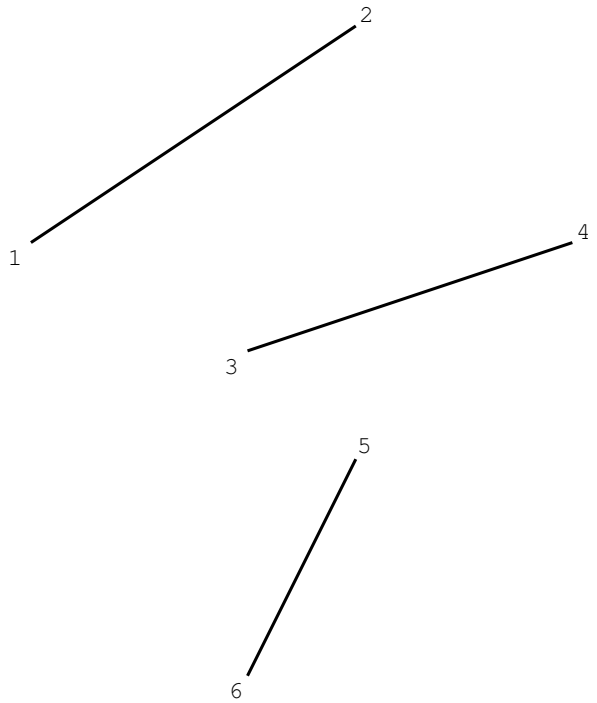
À présent on va donner la liste des primitives qu'il est possible de tracer, en fonction du `mode` spécifié pour `glBegin`.

7.4.1 Points

`GL_POINTS` — Trace une série de points. Le nombre de vertices doit être non nul.

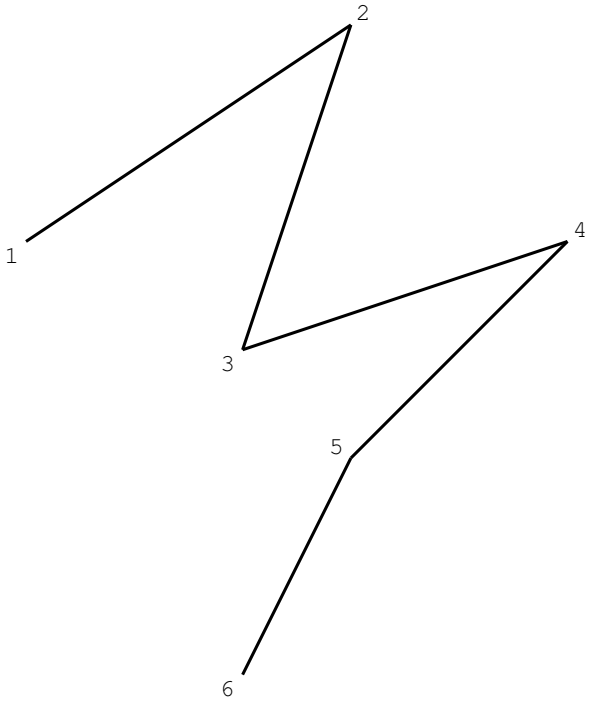
7.4.2 Lignes

`GL_LINES` — Trace une série de segments de droite. Le nombre de vertices doit être pair et non nul, chaque paire définit les deux extrémités d'un segment.



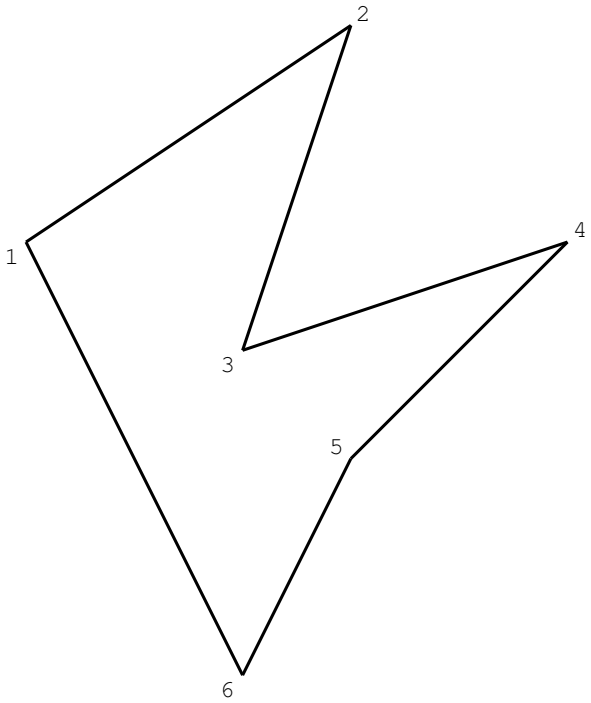
Tracé de segments avec `GL_LINES`

`GL_LINE_STRIP` — Trace une série de segments formant une ligne continue. Le nombre de vertices doit être au moins égal à 2, chaque nouveau sommet rajoute un segment au bout de la ligne en partant du précédent.



Tracé d'une ligne avec `GL_LINE_STRIP`

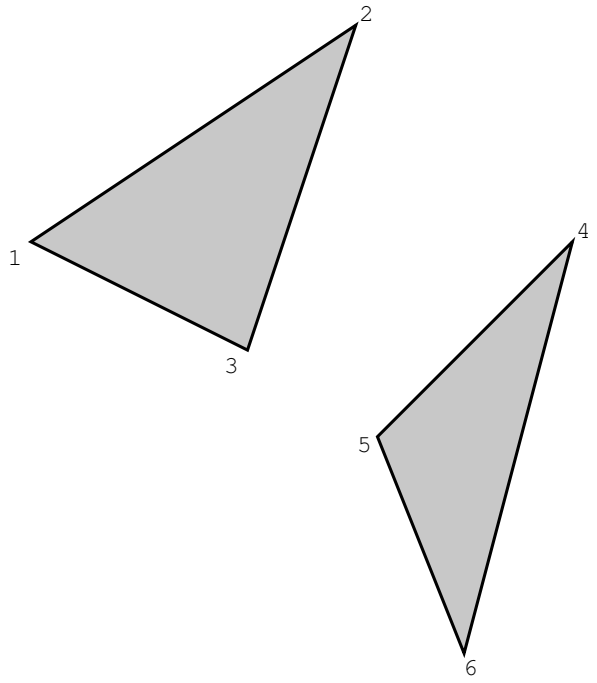
`GL_LINE_LOOP` — Trace une série de segments formant une courbe fermée. Le nombre de vertices doit être au moins égal à 3, chaque nouveau sommet rajoute un segment en partant du précédent, la ligne est fermée en reliant le dernier sommet au premier.



Tracé d'une courbe fermée avec `GL_LINE_LOOP`

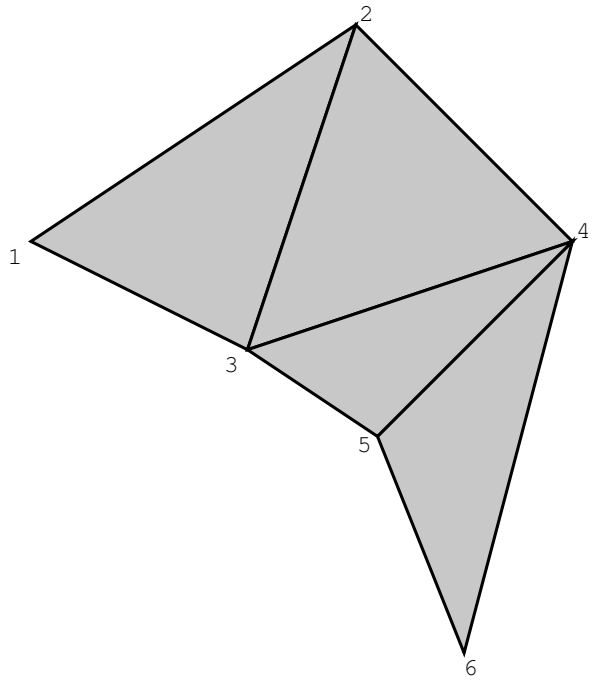
7.4.3 Triangles

`GL_TRIANGLES` — Trace une série de triangles. Le nombre de vertices doit être multiple de 3 et non nul, chaque triplet définit les trois sommets d'un nouveau triangle.



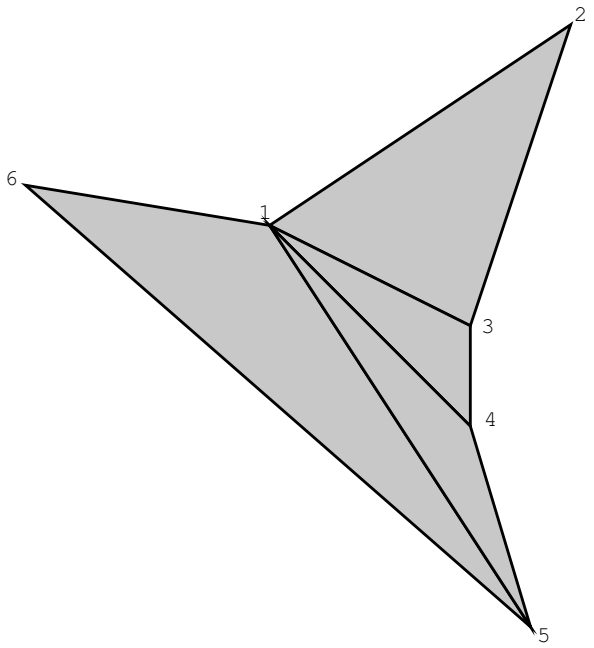
Tracé de triangles avec `GL_TRIANGLES`

`GL_TRIANGLE_STRIP` — Trace une série de triangles reliés entre eux par un côté. Le nombre de vertices doit être au moins égal à 3, chaque nouveau sommet définit le sommet d'un nouveau triangle avec les deux derniers sommets utilisés.



Tracé de triangles avec
`GL_TRIANGLE_STRIP`

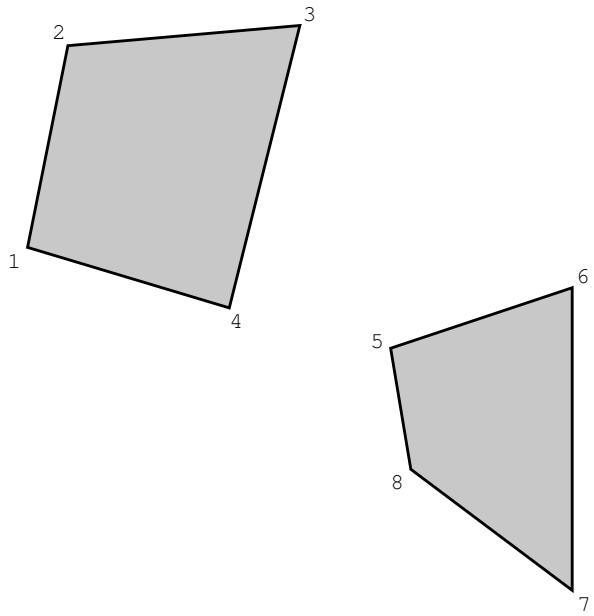
`GL_TRIANGLE_FAN` — Trace une série de triangles reliés entre eux par un côté et ayant tous en commun un même sommet. Le nombre de vertices doit être au moins égal à 3, chaque nouveau sommet définit le sommet d'un nouveau triangle avec le premier et le dernier sommet utilisés.



Tracé de triangles avec `GL_TRIANGLE_FAN`

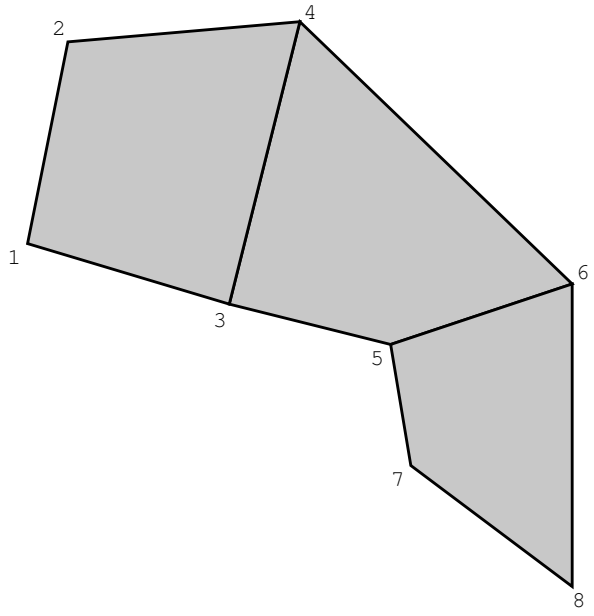
7.4.4 Polygones

`GL_QUADS` — Trace une série de quadrilatères. Le nombre de vertices doit être multiple de 4 et non nul, chaque quadruplet définit les quatre sommets d'un nouveau quadrilatère, en tournant autour du centre. Les quadruplets de sommets doivent être coplanaires, et former un quadrilatère convexe (sinon le dessin risque d'être faux).



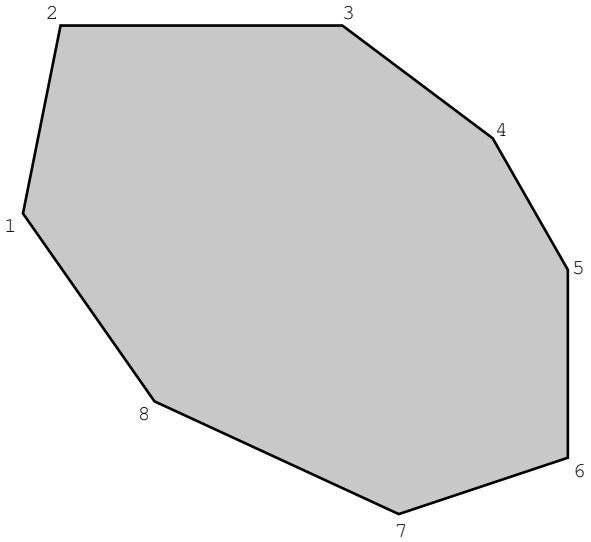
Tracé de quadrilatères avec `GL_QUADS`

`GL_QUAD_STRIP` — Trace une série de quadrilatères reliés entre eux par un côté. Le nombre de vertices doit être pair et au moins égal à 4, chaque nouveau couple de sommets définit un nouveau quadrilatère en utilisant les deux derniers sommets utilisés dans l'ordre inverse. Les quadruplets de sommets doivent être coplanaires, et former un quadrilatère convexe.



Tracé de quadrilatères avec
`GL_QUAD_STRIP`

`GL_POLYGON` — Trace un polygone. Le nombre de vertices doit être au moins égal à 3, les sommets doivent être coplanaires et définir un polygone convexe.



Tracé de polygone avec `GL_POLYGON`

7.4.5 Formes prédéfinies

La librairie GLUT introduit certaines fonctions permettant d'afficher des formes prédéfinies simples. On va en citer quelques unes (les paramètres `slices` et `stacks` correspondent à la fréquence de discrétisation en facettes des surfaces considérées, il suffit de les augmenter pour obtenir des surfaces d'aspect plus lisse) :

```
void glutSolidSphere(GLdouble radius,GLint slices,GLint
    stacks);
void glutSolidCube(GLdouble size);
void glutSolidCone(GLdouble base,GLdouble height,GLint
    slices,GLint stacks);
void glutSolidTorus(GLdouble innerRadius,GLdouble
    outerRadius,GLint nsides,GLint rings);
void glutSolidTeapot(GLdouble size);
```

En remplaçant `"Solid"` par `"Wire"` dans le nom des fonctions ci-dessus on obtient le même objet avec une structure en «fil de fer», c'est à dire seulement avec le contour des facettes.

`glutSolidSphere` — Affiche une sphère centrée à l'origine de rayon `radius`.

`glutSolidCube` — Affiche un cube centré à l'origine de côté `size`.

`glutSolidCone` — Affiche un cône circulaire de hauteur `height` et d'axe Oz. La base est le cercle de rayon `base` dans le plan d'équation $z = 0$ et le sommet a pour coordonnées $(0, 0,$ `height`$)$.

`glutSolidTorus` — Affiche un tore centré à l'origine, perpendiculaire à l'axe Oz, de grand rayon `outerRadius` et de petit rayon `innerRadius`.

`glutSolidTeapot` — Affiche une théière de taille `size`.

7.5 Fin du rendu

Une fois que toute la scène a été décrite, il est usage d'indiquer à OpenGL qu'il peut vider le cache d'instructions graphiques et procéder au rendu. En général, on utilise :

```
glFlush();
glutSwapBuffers();
```

On peut donner la structure simplifiée d'un programme OpenGL utilisant GLUT :

```
void reshapeFunc(int width,int height){
  /* Gestion du viewport */
  glViewport(0,0,width,height);
  /* Gestion de la matrice de projection */
  glMatrixMode(GL_PROJECTION);
  /* Remise à zéro de la matrice */
  glLoadIdentity();
  /* Caméra perspective d'angle vertical 70°, avec un
     écran de mêmes proportions que la fenêtre, l'
     écran est situé à 0.1 de l'oeil et la distance
     maximale de vision vaut 100 */
  gluPerspective(70,(float) width/height,0.1,100);
}

void displayFunc(void){
  /* Remise à zéro de la mémoire graphique et du
     tampon de profondeur */
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  /* Gestion de la matrice de modélisation */
  glMatrixMode(GL_MODELVIEW);
  /* Remise à zéro de la matrice */
  glLoadIdentity();
  /* Caméra située en (-2,1,4) regardant dans la
     direction de l'origine, la verticale de l'écran
     est dirigée selon le vecteur (0,1,0) */
  gluLookAt(-2,1,4,0,0,0,0,1,0);

  /* Affichage en rouge */
  glColor3f(1,0,0);

  /* Primitives graphiques: lignes, polygones, etc...
     */
  glBegin(...);
  glNormal3(...);
  glVertex(...);
  glVertex(...);
  /* ... */
```

```
    glEnd();

    /* Fin du rendu */
    glFlush();
    glutSwapBuffers();
}

int main(int argc,char* argv[]){
    /* Initialisation de GLUT */
    glutInit(&argc,argv);
    /* Initialisation du format d'affichage: ici RGB
       avec un double buffer et un tampon de profondeur
       */
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE |
       GLUT_DEPTH);
    /* Création d'une fenêtre */
    glutCreateWindow("Démonstration");
    /* Spécification des callbacks */
    glutReshapeFunc(reshapeFunc);
    glutDisplayFunc(displayFunc);

    /* Initialisation des paramètres OpenGL qui ne
       changeront pas, à faire après la création de la
       fenêtre */
    /* Utilisation du Z-Buffer */
    glEnable(GL_DEPTH_TEST);
    /* Utilisation d'une lampe */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    /* Modèle d'éclairement simplifié */
    glEnable(GL_COLOR_MATERIAL);
    /* Modèle d'éclairement double face */
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,1);

    /* Entrée dans la boucle principale */
    glutMainLoop();
}
```

Quatrième partie

Splines

Contenu

8 Motivation : les besoins du `design`

Les objets fabriqués par l'industrie comportent des lignes courbes. Les logiciels de CAO proposent un catalogue de formes simples (segments de droites, arcs de cercles, de côniques...) mais elles ne suffisent pas. Le `designer` a besoin d'une famille plus riche de courbes, dépendant de paramètres. Il souhaite

- disposer de suffisamment de paramètres pour pouvoir spécifier des conditions aux limites et autres contraintes ;
- deviner l'effet de chaque paramètre, pour trouver rapidement en les ajustant une courbe qui correspond à celle qu'il a imaginée.

D'autre part, le calcul de la courbe en fonction des paramètres doit être rapide.

Les B-splines, utilisées en analyse numérique depuis les années 30, possèdent les propriétés voulues. Comme il est impossible de paramétrer exactement un cercle au moyen de B-splines, on introduit une famille un peu plus large, les *B-splines rationnelles* (`NURBS`).

L'objet de ce chapitre est de définir les B-splines et leurs variantes. On commence par une sous-famille, celle des courbes de Bézier.

9 Courbes de Bézier

Etant donné un entier k, et $j = 0, \ldots, k$, on note

$$b_{j,k}(t) = C_k^j t^j (1-t)^{k-j}$$

le j-ème terme du développement par la formule du binôme de $(t + (1-t))^k$.

Définition 9.1 *Soit k un entier. Soient $P_0, \ldots, P_k$ des points de R^n. La courbe paramétrée par*

$$X(t) = \sum_{j=0}^{k} b_{j,k}(t) P_j, \quad t \in [0,1],$$

s'appelle la courbe de Bézier *de degré k associée au polygone de contrôle $P_0, \ldots, P_k$.*

Exemple 9.2 *Une courbe de Bézier de degré 1 est un segment de droite parcouru à vitesse constante. Une courbe de Bézier de degré 2 est un arc de parabole.*

Remarque 9.3 *Comme $b_{k-i,k}(t) = b_{i,k}(1-t)$, si on renverse l'ordre des points de contrôle, on obtient la même courbe, parcourue dans l'autre sens.*

Remarque 9.4 *Une courbe de Bézier est toujours contenue dans l'enveloppe convexe de son polygone de contrôle.*

De plus, lorsque t augmente, les poids de la représentation de $X(t)$ comme barycentre des P_i se répartissent différemment : au début (t petit), la masse est groupée sur les premiers poids, au milieu (t proche de $\frac{1}{2}$), sur ceux du milieu, à la fin (t proche de 1) sur ceux de la fin. La courbe épouse donc le polygone de contrôle. Il est utile de manipuler un logiciel (e.g. `DesignMentor` de Shene, d'accès libre) pour s'en rendre compte. Voici un énoncé plus précis.

Proposition 9.5 *Soit X la courbe de Bézier de degré k associée au polygone de contrôle $P_0, \dots, P_k$. La courbe des vecteurs vitesse, $t \mapsto X'(t)$, est la courbe de Bézier de degré $k-1$ associée au polygone de contrôle $k(P_1-P_0), \dots, k(P_k-P_{k-1})$. En particulier, $c'(0) = k(P_1 - P_0)$. Si $P_0 \neq P_1$, la courbe X est donc tangente au point P_0 au premier côté P_0P_1 du polygone de contrôle.*

Ce résultat est utile pour garantir la différentiabilité de courbes obtenues en raccordant des courbes de Bézier.

Le résultat suivant est un outil de calcul, implémenté dans tous les dispositifs utilisateurs de courbes de Bézier (pilotes d'imprimantes, logiciels de dessin).

Proposition 9.6 (Algorithme de de Casteljau). *Soit X la courbe de Bézier de degré k associée au polygone de contrôle $P_0, \dots, P_k$, soit $t \in [0,1]$.*

Pour $j = 1, \dots, k$, soit $Q_j^1 = (1-t)P_{j-1} + tP_j$ le point qui divise le segment $[P_{j-1}, P_j]$ dans les proportions t et $1-t$.

Pour $j = 2, \dots, k$, soit $Q_j^2 = (1-t)Q_{j-1}^1 + tQ_j^1$ le point qui divise le segment $[Q_{j-1}^1, Q_j^1]$ dans les proportions t et $1-t$.

Etc... Alors $X(t) = Q_k^k$. De plus, la courbe X est tangente au segment $[Q_{k-2}^{k-1}, Q_{k-1}^{k-1}]$ en Q_k^k.

Cet algorithme s'apparente au schéma de Horner pour l'evaluation d'un polynôme. Il minimise les opérations arithmétiques, et est donc très rapide.

9.1 Carreaux de Bézier

Il s'agit de surfaces paramétrées par le carré $[0,1]\times[0,1]$, dont les lignes de coordonnées sont des courbes de Bézier.

Définition 9.7 *Soient k, k' des entiers. Soient $P_{i,j}$, $i = 0, \dots, k$, $j = 0, \dots, k'$, des points de R^n. La surface paramétrée par*

$$X(u,v) = \sum_{i=0}^{k} \sum_{j=0}^{k'} b_{i,k}(u) b_{j,k}(v) P_{i,j}, \quad (u,v) \in [0,1] \times [0,1],$$

s'appelle le carreau de Bézier *(`Bézier rectangular patch`) de bidegré k associée au polygone de contrôle $P_{i,j}$.*

A v fixé, $u \mapsto X(u,v)$ est la courbe de Bézier de degré k associée au polygone de contrôle

$$Q_i(v) = \sum_{j=0}^{k'} b_{j,k}(v) P_{i,j}, \quad i = 0, \ldots, k.$$

Noter que $v \mapsto Q_j(v)$ est la courbe de Bézier de degré k' associée au polygone de contrôle $P_{i,j}$, $j = 0, \ldots, k'$. Autrement dit, prenons le réseau de contrôle colonne par colonne. Construisons les courbes de Bézier associées aux colonnes. Pour chaque v, cela donne $k+1$ points, i.e. un polygone de contrôle pour une courbe de Bézier de degré k. Le carreau est la réunion de ces courbes.

10 B-splines

10.1 Motivation

Le degré d'une courbe de Bézier est égal au nombre de cotés du polygone de contrôle. Une courbe de forme compliquée coûte donc très cher. Pour limiter le degré, on a recours à des courbes polynômiales par morceaux, mais garantissant une haute différentiabilité. Voici comment on procède.

On se donne une suite de points $t_0 \leq \ldots \leq t_m$ de la droite réelle, appelés *noeuds* (`knots`). Le vecteur $(t_0, \ldots, t_m)$ s'appelle le *vecteur des noeuds* (`knot vector`). Certains noeuds peuvent être confondus. Si r noeuds sont égaux à un réel τ, on dit que τ est de *multiplicité* r. Les B-splines seront des courbes paramétrées, sur chaque intervalle non vide $]t_i, t_{i+1}[$, par des polynômes.

On se donne d'autre part un *polygone de contrôle* (`control polygon`) $P_0, \ldots, P_m$ dans $\mathbf{R}^n$, appelés. On l'imagine comme une courbe $t \mapsto X_0(t)$ qui saute d'un point à l'autre aux temps t_i, i.e.

$$X_0(t) = P_i \quad \text{pour} \quad t \in [t_i, t_{i+1}[.$$

(Si le noeud $t_i = t_{i+1}$, le sommet P_i est simplement ignoré).

On cherche à approcher cette courbe discontinue par une courbe plus régulière. Le première étape consiste à faire passer une ligne polygonale par les points P_i, i.e. lorsque t varie entre deux noeuds t_i et t_{i+1}, $X_1(t)$ décrit le segment $[P_{i-1}, P_i]$ à vitesse constante. On trouve la formule

$$X_1(t) = (1 - \frac{t - t_i}{t_{i+1} - t_i}) P_{i-1} + \frac{t - t_i}{t_{i+1} - t_i} P_i.$$

(Si $t_i = t_{i+1}$, la courbe saute de P_{i-1} à P_i en t_i). Si les noeuds sont tous distincts, la courbe obtenue est continue mais non dérivable en général. Ses composantes sont des fonctions linéaires par morceaux.

L'étape suivante conduit (si les noeuds sont tous distincts) à une courbe X_2 de classe C^1 (mais non C^2 en général), au prix d'augmenter le degré : elle est quadratique par

morceaux. Elle ne passe plus par les sommets P_i, mais conserve une proximité au polygone en un sens différent : si t est compris entre les noeuds t_i et t_{i+1}, $X_2(t)$ est dans l'enveloppe convexe des sommets P_{i-2}, P_{i-1} et P_i.

Comment trouver X_2, et plus généralement, X_k pour $k \geq 2$? Supposons les points P_i affinement indépendants. Alors la courbe X_{k-1} s'écrit de manière unique comme combinaison

$$X_{k-1}(t) = \sum_i B_{i,k-1}(t) P_i$$

où les fonctions $B_{i,k}$ sont positives ou nulles, et leur somme vaut 1. Pour gagner un degré de différentiabilité, l'idée est de remplacer dans cette formule la suite de points fixés P_i par des points $P_i(t)$ mobiles le long du polygone de contrôle, où $P_i(t)$ se déplace de P_{i-1} à P_i pendant l'intervalle de temps $[t_i, t_{i+k}]$. Autrement dit, on pose

$$X_k(t) = \sum_i B_{i,k-1}(t)((1 - \frac{t - t_i}{t_{i+k} - t_i})P_{i-1} + \frac{t - t_i}{t_{i+k} - t_i} P_i)$$

Cela donne pour les fonctions $B_{i,k}$ la relation de récurrence

$$B_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i} B_{i,k-1}(t) + (1 - \frac{t - t_{i+1}}{t_{i+k+1} - t_{i+1}}) B_{i+1,k-1}(t)$$

qui les détermine uniquement.

Cela donne une famille de courbes plus vaste que la famille des courbes de Bézier. Le paramètre principal est le polygone de contrôle, dont la courbe épouse les formes. La complexité du calcul de la courbe dépend avant tout du degré de différentiabilité (on s'arrête souvent à $k = 3$). Le paramètre secondaire est le vecteur des noeuds. On s'en sert avant tout pour s'assurer que la courbe passe par des points prescrits avec des tangentes prescrites, i.e. pour contrôler les raccords.

10.2 Définition des fonctions B-splines

On se donne une suite de noeuds $t_0 \leq \cdots \leq t_m$ sur la droite réelle.

Notation 10.1 *Soit $j = 1, \ldots, m + 1 - i$. Si $t_i < t_{i+1}$, on* note

$$\omega_{i,j}(t) = \frac{t - t_i}{t_{i+j} - t_i}.$$

Si $t_i = t_{i+1}$, on pose $\omega_{i,j} = 0$.

Convention 10.2 *Cette notation illustre une convention qu'on va utiliser partout :* Chaque fois qu'on écrit une fraction dont le dénominateur est nul, il faut l'interprêter par 0.

Définition 10.3 *On définit par récurrence sur k les fonctions* B-splines $B_{i,k}$ *pour* $i = 0, \ldots, m-k-1$, *par les relations suivantes.*

$$B_{i,0}(t) = 1 \quad pour \quad t \in [t_i, t_{i+1}[, \quad = 0 \quad sinon,$$

et pour $k \geq 1$,

$$B_{i,k}(t) = \omega_{i,k}(t) B_{i,k-1}(t) + (1 - \omega_{i+1,k}(t)) B_{i+1,k-1}(t).$$

Remarque 10.4 *Les fonctions B-splines constituent une base (parmi d'autres) de l'espace vectoriel des fonctions définies sur l'intervalle* $[t_0, t_{m-k}]$, *polynômiales de degré inférieur ou égal à* k *sur chaque intervalle* $[t_i, t_{i+1}[$, *de classe* C^{k-r} *au voisinage de chaque noeud de multiplicité* r *(voir [R], paragraphe 1.4). Les mérites particuliers à cette base sont énumérés dans le théorème 1.*

10.3 B-splines uniformes

Proposition 10.5 *On pose, pour* $i \in \mathbf{Z}$, $t_i = i$. *Alors pour tout* $i \in \mathbf{Z}$, $k \geq 1$ *et* $t \in \mathbf{R}$,

$$B_{i,k}(t+1) = B_{i-1,k}(t) \quad et \quad B_{0,k}(k+1-t) = B_{0,k}(t).$$

Soient $i \in \mathbf{Z}$ et $t \in \mathbf{R}$. A l'aide des identités $B_{i,0}(t+1) = B_{i-1,0}(t)$ et $\omega_{i,k}(t+1) = \omega_{i-1,k}(t)$, on montre aisément par récurrence sur k que $B_{i,k}(t+1) = B_{i-1,k}(t)$ pour tout k.

Montrons par récurrence sur k que pour $k \geq 1$, $B_{0,k}(k+1-t) = B_{0,k}(t)$. On calcule $B_{0,1}(t) = t\,\mathbf{1}_{[0,1[} + (2-t)\mathbf{1}_{[1,2[}$. Cette fonction satisfait $B_{0,1}(2-t) = B_{0,1}(t)$. Supposons l'identité démontrée pour la B-spline uniforme de degré $k-1$. On remarque que $\omega_{0,k}(k+1-t) = \frac{k+1-t}{k} = 1 - \omega_{1,k}(t)$. Par l'hypothèse de récurrence et l'invariance par translation,

$$B_{0,k-1}(k+1-t) = B_{0,k-1}(t-1) = B_{1,k-1}(t).$$

Il vient

$$\begin{aligned} B_{0,k}(k+1-t) &= \omega_{0,k}(k+1-t) B_{0,k-1}(k+1-t) + (1 - \omega_{1,k}(k+1-t)) B_{1,k-1}(k+1-t) \\ &= (1 - \omega_{1,k}(t)) B_{1,k-1}(t) + \omega_{0,k}(t) B_{0,k-1}(t) \\ &= B_{0,k}(t). \end{aligned}$$

On ne calcule que $B_{0,k}$, les autres fonctions B-splines uniformes s'en déduisent par translation.

La relation de récurrence qui définit les B-splines devient

$$B_{0,k}(t) = \frac{t}{k} B_{0,k-1}(t) + \frac{k+1-t}{k} B_{0,k-1}(t-1).$$

On calcule $B_{0,0} = \mathbf{1}_{[0,1[}$, $B_{0,1} = t\mathbf{1}_{[0,1[} + (2-t)\mathbf{1}_{[1,2[}$, $B_{0,2} = \frac{t^2}{2}\mathbf{1}_{[0,1[} + \frac{-2t^2+6t-3}{2}\mathbf{1}_{[1,2[} + \frac{(3-t)^2}{2}\mathbf{1}_{[2,3[}$, $B_{0,3} = \frac{t^3}{6}\mathbf{1}_{[0,1[} + \frac{-3t^3+12t^2-12t+4}{6}\mathbf{1}_{[1,2[} + \frac{3t^3-24t^2+60t-44}{6}\mathbf{1}_{[2,3[} + \frac{(4-t)^3}{6}\mathbf{1}_{[3,4[}$.

10.4 Bézier comme cas particulier de B-splines

Proposition 10.6 *On pose* $t_0 = t_1 = \cdots = t_k = 0$, $t_{k+1} = \cdots = t_{2k+1} = 1$. *Alors, pour* $i = 0, \ldots, k$,

$$B_{i,k}(t) = C_k^i t^i (1-t)^{k-i} = b_{i,k}(t).$$

pour $t \in [0, 1[$, $B_{i,k}(t) = 0$ *sinon. Les autres fonctions B-splines sont nulles.*

10.5 Propriétés élémentaires

Théorème 1 *La* k*-ième B-spline* $t \mapsto X_k(t)$ *a les propriétés suivantes.*

1. *Les composantes de* $X_k(t)$ *sont sur chaque intervalle* $[t_i, t_{i+1}[$ *des polynômes de degré* k *;*
2. *en un noeud de multiplicité* r*, la courbe est de classe* C^{k-r} *;*
3. *si* $t \in [t_i, t_{i+1}[$, $X_k(t)$ *ne dépend que des points de contrôle* $P_{i-k}, \cdots, P_i$ *et se trouve dans l'enveloppe convexe de ces points ;*
4. *si* t_i *est un noeud simple et* $k \geq 1$, $X_k(t_i)$ *ne dépend que des points de contrôle* $P_{i-k}, \cdots, P_{i-1}$ *et se trouve dans l'enveloppe convexe de ces points ;*
5. *si* $t_i = \cdots = t_{i+k} < t_{i+k+1}$ *est un noeud de multiplicité* $k+1$*, alors* $X_k(t_i) = P_i$ *et* $X_k'(t_i) = \dfrac{k}{t_{i+k+1} - t_i}(P_{i+1} - P_i)$ *;*
6. *la construction de la courbe* X_k *à partir des points de contrôle* P_i *est invariante par application affine.*

Notation 10.7 *On dit qu'une courbe B-spline est* vissée aux extrémités *(*`clamped`*) si elle est de degré* k *et si les noeuds extrêmes* t_0 *et* t_m *sont de multiplicité* $k+1$*, i.e.* $t_0 = t_1 = \cdots = t_k = 0 < t_{k+1}$ *et* $t_{m-k} = \cdots = t_m$*. Dans ce cas, le nombre de points de contrôle effectivement utiles est* $m - k - 1$.

Il résulte du théorème 1 qu'une courbe vissée aux extrémités est tangente à son polygone de contrôle aux extrémités. Plus précisément

Proposition 10.8 *Soit* X_k *une courbe B-spline de degré* k *vissée aux extrémités. Alors* $X_k(t_0) = P_0$ *et* $X_k(t_m) = P_{m-k-1}$*. Si de plus* $P_1 \neq P_0$ *(resp.* $P_{m-k-2} \neq P_{m-k-1}$*), alors la courbe est tangente en* P_0 *au segment* P_0P_1 *(resp* $P_{m-k-2}P_{m-k-1}$*).*

10.6 Symétries

Toute isométrie du polygone de contrôle compatible avec une symétrie du vecteur de noeuds (toujours satisfaite pour les vecteurs de noeuds utilisés par DesignMentor par exemple) se traduit par une symétrie de la courbe.

Proposition 10.9 *Soit σ une isométrie du polygone de contrôle qui préserve l'ordre des sommets, i.e. une transformation affine telle que $\sigma(P_i) = P_{i+I}$. Si le vecteur de noeuds est lui aussi périodique, i.e $t_{i+I} = t_i + T$, alors la courbe B-spline de degré k associée l'est aussi,*

$$\sigma(X_k(t)) = X_k(t+T).$$

Proposition 10.10 *Soit σ une isométrie du polygone de contrôle qui renverse l'ordre des sommets, i.e. une transformation affine telle que $\sigma(P_i) = P_{I-i}$. Si le vecteur de noeuds est lui aussi périodique, i.e $t_{m-i} = 2a - t_i$ pour $i = 0, \ldots, m$, où $m = I + k + 1$, alors la courbe B-spline de degré k associée l'est aussi,*

$$\sigma(X_k(t)) = X_k(2a - t)$$

pour tout $t \in [t_k, t_I[$.

Proposition 10.11 *Si on se donne un vecteur de noeuds et un polygone de contrôle périodiques, i.e. tels qu'il existe $I \in \mathbf{N}$ et $T \in \mathbf{R}$ tels que $t_{i+I} = t_i + T$ et $P_{i+I} = P_i$, alors les courbes B-splines correspondantes sont périodiques, i.e. pour tout entier k et tout $t \in \mathbf{R}$, $X_k(t+T) = X_k(t)$.*

10.7 Algorithme de de Casteljau

Pour calculer et tracer une courbe B-spline, plutôt que de calculer symboliquement les fonctions B-splines, de les évaluer et calculer les combinaisons convexes de points de contrôle, il est plus efficace de partir des points de contrôle et d'effectuer des combinaisons convexes successives avec des poids linéaires en t, comme dans le schéma de Horner.

On se souvient que $X_k(t)$ coïncide avec la valeur en t de la courbe B-spline de degré $k-1$ associée au polygone de contrôle $P_i(t) = \omega_{i,k}(t)P_{i-1} + (1 - \omega_{i,k}(t))P_i$. C'est le point de départ de l'algorithme attribué à de Casteljau.

Proposition 10.12 *On fixe un vecteur de noeuds $\mathbf{t}$ et un polygone de contrôle $\mathbf{P}$. On cherche à calculer la courbe B-spline X_k de degré k correspondante.*

Soit $t \in [t_i, t_{i+1}[$. On pose $P_j^0 = P_j$ pour $i - k \leq j \leq i$. Puis, pour $r = 0, \ldots, k-1$, on pose

$$P_j^{r+1} = \omega_{j,k-r}(t)P_j^r + (1 - \omega_{j,k-r}(t))P_{j-1}^r$$

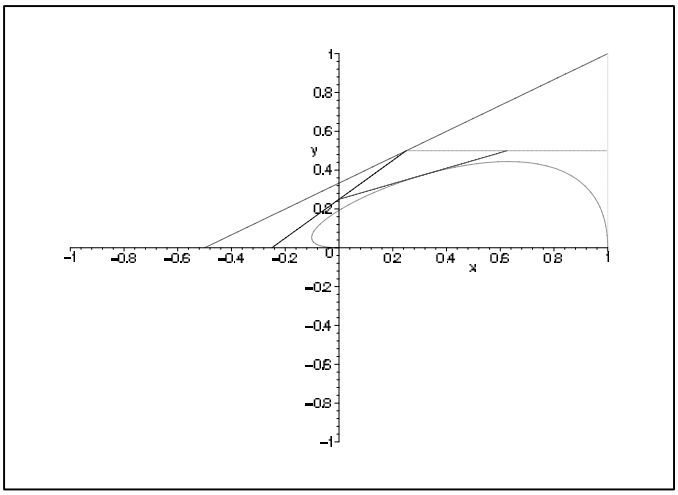
pour $i - k + r + 1 \leq j \leq i$. Alors $P_i^k = X_k(t)$.

Pratiquement, on peut disposer les points P_j^r en triangle. Par exemple, si $k = 3$ et $t \in [t_4, t_5[$, on écrit

$$\begin{array}{ccccccc}
P_1 & & P_2 & & P_3 & & P_4 \\
& P_2^1 & & P_3^1 & & P_4^1 & \\
& & P_3^2 & & P_4^2 & & \\
& & & P_4^3 & & &
\end{array}$$

et chaque valeur de la $r+1$-ième ligne s'obtient en faisant une combinaison convexe des deux valeurs situées juste au-dessus affectées des poids respectifs $1-\omega_{j,k-r}(t)$ (pour celui de gauche) et $\omega_{j,k-r}(t)$ (pour celui de droite).

Géométriquement, on place les points P_j^1 sur les côtés du polygone de contrôle. Cela donne un nouveau polygone. On place les points P_j^2 sur les côtés de ce polygone. Cela donne un troisième polygone, sur un côté duquel on trouve $P_i^3 = X_3(t)$.



10.8 Convexité des B-splines planes

Proposition 10.13 *Si le polygone de contrôle est plan et convexe, alors la courbe B-spline associée est convexe, i.e. contenue dans le bord d'un convexe.*

11 NURBS

11.1 Courbes rationnelles

Un cercle ne peut pas être paramétré par des polynômes. En effet, si P et Q sont des polynômes non constants, P^2+Q^2 est un polynôme non constant.

En revanche, le cercle unité admet une paramétrisation rationnelle

$$t \mapsto (\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}).$$

Exercice 7 *Une* cônique *est une courbe plane définie par une équation du second degré (i.e. le lieu des zéros d'un polynôme en deux variables de degré total 2). Soit C une cônique non vide et non dégénérée (i.e. non réduite à une ou 2 droites) et P un point de C. En coupant C par les droites passant par P, montrer que C admet une paramétrisation rationnelle.*

Définition 11.1 *Une courbe paramétrée est dite* rationnelle par morceaux *si sur chaque intervalle d'une subdivision, chaque coordonnée est donnée par une fraction rationnelle (i.e. le quotient de deux polynômes) du paramètre.*

11.2 Projection centrale

Réalisons l'espace affine $\mathbf{R}^n$ comme l'hyperplan affine d'équation $\{x_0 = 1\}$ dans l'espace vectoriel $\mathbf{R}^{n+1}$. La *projection centrale* de $\mathbf{R}^{n+1}$ privé de l'hyperplan vectoriel $\{x_0 = 0\}$ vers $\mathbf{R}^n$ est définie comme suit : la projection d'un vecteur v non nul est l'intersection de la droite engendrée par v avec l'hyperplan $\mathbf{R}^n$. Analytiquement, elle s'écrit

$$(v_0, \ldots, v_n) \mapsto (\frac{v_1}{v_0}, \ldots, \frac{v_n}{v_0}).$$

La projection centrale envoie les segments de droites ne rencontrant pas l'hyperplan $\{x_0 = 0\}$ sur des segments de droites. En particulier, sa restriction au demi-espace $\{x_0 > 0\}$ envoie segment sur segment, donc préserve la convexité.

Toute courbe rationnelle dans $\mathbf{R}^n$ s'obtient par projection centrale d'une courbe polynômiale tracée dans $\mathbf{R}^{n+1}$. En effet, quitte à réduire au même dénominateur, une courbe paramétrée rationnelle s'écrit

$$t \mapsto X(t) = (\frac{Q_1(t)}{Q_0(t)}, \ldots, \frac{Q_n(t)}{Q_0(t)})$$

où les Q_i sont des polynômes en t. Alors X est la projection centrale de la courbe polynômiale

$$t \mapsto X(t) = (Q_0(t), \ldots, Q_n(t)) \in \mathbf{R}^{n+1}.$$

11.3 B-splines rationnelles

Définition 11.2 *Soit* $\mathbf{t}$ *un vecteur de noeuds, soit* $\mathbf{P}$ *un polygone de contrôle dans* $\mathbf{R}^n$ *et* w_i *des poids* `(weights)` *attachés à chaque point de contrôle* P_i*. On suppose que les poids ne sont pas tous nuls. La* courbe B-spline rationnelle `(NURBS)` *de degré* k *associée à ces données est la courbe paramétrée par*

$$t \mapsto X(t) = \frac{\sum_i w_i B_{i,k}(t) P_i}{\sum_i w_i B_{i,k}(t)}.$$

Lorsque le vecteur de noeuds prend la forme spéciale $(0, 0, \ldots, 0, 1, 1, \ldots, 1)$*, on parle de* courbe de Bézier rationnelle*.*

Autrement dit, la courbe B-spline rationnelle est la projection centrale de la courbe B-spline dans $\mathbf{R}^{n+1}$ associée au vecteur de noeuds $\mathbf{t}$ et aux points de contrôle $R_i = (w_i, w_i P_i) \in \mathbf{R}^{n+1}$. Noter que le polygone de contrôle $\mathbf{P}$ est la projection centrale du polygone $\mathbf{R}$. D'autre part, multiplier tous les poids par une même constante non nulle ne change rien.

Exemple 11.3 *Le quart de cercle unité comme courbe de Bézier rationnelle quadratique.*

On pose $P_0 = (1,0)$, $P_1 = (1,1)$ et $P_2 = (0,1)$, $w_0 = w_1 = 1$, $w_2 = 2$. On choisit comme vecteur de noeuds $(0,0,0,1,1,1)$. Alors la courbe de Bézier rationnelle de degré 2 obtenue est donnée pour $t \in [0,1[$ par

$$t \mapsto c(t) = (\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}).$$

Son image est exactement un quart du cercle unité.

11.4 Propriétés des courbes B-splines rationnelles

Elles se déduisent immédiatement des propriétés des courbes B-splines et de celles de la projection centrale.

Proposition 11.4 *Soit* $\mathbf{t}$ *un vecteur de noeuds,* $\mathbf{P}$ *un polygone de contrôle et* $\mathbf{w}$ *un vecteur de poids non nul. La courbe B-spline rationnelle* X_k *de degré* k *associée à ces données a les propriétés suivantes.*

1. *Sur chaque intervalle* $[t_i, t_{i+1}[$*, les coordonnées de* X *sont des fractions rationnelles de degré* k *(i.e. quotients de deux polynômes de degré* k*).*
2. *En un noeud de multiplicité* r*, la courbe* X *est de classe* C^{k-r}*.*
3. *Si le vecteur de noeuds, le polygone de contrôle et le vecteur de poids sont périodiques, alors la courbe est périodique.*
4. *Si* $t_0 = t_1 = \ldots = t_k < t_{k+1}$ *est un noeud de multiplicité* $k+1$*, et si les poids* w_0 *et* w_1 *sont non nuls, alors la courbe* X_k *est tangente en* P_0 *au polygone de contrôle. De plus,*

$$X_k'(t_k) = \frac{k}{t_{k+1} - t_k} \frac{w_1}{w_0}(P_1 - P_0).$$

5. *Supposons le vecteur de noeud simple. Si* $k = 2$*, la courbe* X_2 *est tangente à chaque côté du polygone de contrôle. Si* $k = 3$ *et si trois sommets successifs de* $\mathbf{P}$ *sont sur une même droite* D*, celle-ci est tangente à la courbe* X_3*.*
6. *Si* $t \in [t_i, t_{i+1}[$*, le point* $X_k(t)$ *ne dépend que les points de contrôle* $P_{i-k}, P_{i-k+1}, \ldots, P_i$ *et des poids* $w_{i-k}, w_{i-k+1}, \ldots, w_i$*. Si de plus les poids sont tous positifs ou nuls,* $X_k(t)$ *est dans l'enveloppe convexe des points* $P_{i-k}, P_{i-k+1}, \ldots, P_i$*.*
7. *Si* H *est un hyperplan affine de* $\mathbf{R}^n$*, alors*

$$\#(X_k([t_k, t_m]) \cap H \leq \#\mathbf{P} \cap H.$$

   En particulier, si le polygone $\mathbf{P}$ *est convexe, la courbe* $X_k([t_k, t_m])$ *est convexe.*
8. *Soit* f *une transformation projective. Alors* $f(X)$ *est une courbe B-spline rationnelle associée au vecteur de noeuds* $\mathbf{t}$*, au polygone de contrôle* $f(\mathbf{P})$ *et à un nouveau vecteur de poids* $\mathbf{w}'$*.*

11.5 Effet des poids

On se limite au cas où les poids sont positifs ou nuls. Clairement, si on décrit un point Q de $\mathbf{R}^n$ comme barycentre de points P_i avec poids positifs ou nuls w_i, lorsque le rapport $w_i / \sum w_i$ tend vers l'infini, le point Q se rapproche de P_i. Par conséquent, augmenter le poids d'un point de contrôle rapproche la courbe de ce point. On trouvera des figures illustrant ce phénomène dans le livre [HL].

12 Surfaces B-splines produits tensoriels

12.1 Définition

On étend naïvement à la dimension 2 l'idée des courbes B-splines. Le domaine de variation des paramètres est un rectangle, produit cartésien de deux intervalles. On se donne deux vecteurs de noeuds $u_0, \ldots, u_m$ et $v_0, \ldots, v_{m'}$. On se donne, à la place d'un polygone de contrôle, un *réseau de contrôle* (`control net`), i.e. des points P_{ij} indexés par les couples de noeuds $i = 0, \ldots, m$, $j = 0, \ldots, m'$.

Définition 12.1 *La* surface B-spline de bidegré (k, ℓ) *associée aux vecteurs de noeuds* $\mathbf{u}$ *et* $\mathbf{v}$ *et au réseau de contrôle* $\mathbf{P}$ *est la surface paramétrée*

$$(u, v) \mapsto \sum_{i=0}^{m} \sum_{j=0}^{m'} B_{i,k}(u) B_{j,\ell}(v) P_{ij}.$$

Dans le cas particulier où les deux vecteurs de noeuds sont de la forme $(0, \ldots, 0, 1, \ldots, 1)$, on retrouve les carreaux de Bézier.

Remarque 12.2 *Noter que la restriction d'une telle surface paramétrée à toute droite du plan (u, v) parallèle à l'un des axes de coordonnées est une courbe B-spline (resp. rationnelle). Ce n'est pas le cas pour les droites obliques.*

Il y a des généralisations moins naïves des courbes B-splines aux dimensions supérieures. En effet, il y a une théorie des fonctions B-splines en plusieurs variables, les splines polyédrales, voir par exemple [R] paragraphe 4.7.

13 Surfaces NURBS

13.1 Définition

Définition 13.1 *Une surface NURBS dans $\mathbf{R}^3$, c'est la projection centrale d'une surface B-spline produit tensoriel dans $\mathbf{R}^4$. Autrement dit, on se donne deux vecteurs de noeuds,*

un polyèdre de contrôle et un poids w_{ij} pour chaque point de contrôle P_{ij}, et la surface associée est paramétrée par

$$(u,v) \mapsto X(u,v) = \frac{\sum_i^m \sum_{j=0}^{m'} w_{ij} B_{i,k}(u) B_{j,\ell}(v) P_{ij}}{\sum_{i=0}^m \sum_{j=0}^{m'} w_{ij} B_{i,k}(u) B_{j,\ell}(v)}.$$

De la sorte, les courbes à u constant et à v constant sont des NURBS.

13.2 Surfaces de révolution

Proposition 13.2 *Soit γ une NURBS de degré k tracée dans un plan Π et soit D une droite de Π. La surface de révolution obtenue en faisant tourner γ autour de D est une NURBS de bidegré $(2,k)$.*

Preuve. Montrons le pour un quart de cette surface. On va utiliser la paramétrisation rationnelle d'un quart de cercle,

$$u \mapsto (\frac{1-u^2}{1+u^2}, \frac{2u}{1+u^2}).$$

On voit cette courbe comme la projection centrale de la courbe de Bézier quadratique σ de $\mathbf{R}^3$ donnée par

$$u \mapsto \sigma(u) = (1+u^2, 1-u^2, 2u).$$

On peut supposer que la droite D est l'axe Oz et Π le plan des x et z. Notons P_{0j} le polygone de contrôle de γ, w_{0j} les poids et $R_{0j} = (w_{0j}, w_{0j}P_{0j}) \in \mathbf{R}^4$. Notons v le paramètre sur γ. On veut qu'à v constant, $u \mapsto X(u,v)$ soit le quart de cercle du plan $\{z = z(v)\}$ d'origine $\gamma(v) = (x(v), 0, z(v))$ et d'extrémité $(0, y(v), z(v))$, paramétré rationnellement. On pose donc

$$X(u,v) = \begin{pmatrix} x(v)\frac{1-u^2}{1+u^2} \\ x(v)\frac{2u}{1+u^2} \\ z(v) \end{pmatrix} = \begin{pmatrix} \frac{1-u^2}{1+u^2} & -\frac{2u}{1+u^2} & 0 \\ \frac{2u}{1+u^2} & \frac{1-u^2}{1+u^2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \gamma(v).$$

Soit

$$c(v) = \sum_j B_{j,k}(v) R_{0j}$$

la courbe B-spline dans $\mathbf{R}^4$ dont la projection centrale est γ. Posons

$$Y(u,v) = \begin{pmatrix} 1+u^2 & 0 & 0 & 0 \\ 0 & 1-u^2 & -2u & 0 \\ 0 & 2u & 1-u^2 & 0 \\ 0 & 0 & 0 & 1+u^2 \end{pmatrix} c(v).$$

Alors $Y(u,v) = (\sigma_0(u)c_0(v), \sigma_0(u)c_0(v)X(u,v))$. Autrement dit, X est la projection centrale de Y.

Il reste à montrer que Y est une surface B-spline produit tensoriel. Décomposons la matrice

$$M(u) = \begin{pmatrix} 1+u^2 & 0 & 0 & 0 \\ 0 & 1-u^2 & -2u & 0 \\ 0 & 2u & 1-u^2 & 0 \\ 0 & 0 & 0 & 1+u^2 \end{pmatrix}$$

dans la base des polynômes de Bernstein de degré 2. Il vient

$$M(u) = (1-u)^2 M_0 + 2u(1-u)M_1 + u^2 M_2$$

où

$$M_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$M_2 = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}.$$

Il vient

$$\begin{aligned} Y(u,v) &= M(u)c(v) \\ &= (\sum_i B_{i,2}(u) M_i)(\sum_j B_{j,k}(v) R_{0j}) \\ &= \sum_{i,j} B_{i,2}(u) B_{j,k}(v) R_{ij} \end{aligned}$$

où $R_{ij} = M_i R_{0j}$. On conclut que Y est une surface B-spline produit tensoriel de bidegré $(2,k)$, donc X est une surface NURBS.

Le polyèdre de contrôle de X s'obtient à partir de celui de γ comme suit. Les P_{0j} sont les points de contrôle de γ. P_{2j} est l'image de P_{0j} par la rotation de 90^0 autour de l'axe Oz, et P_{1j} est l'image de P_{0j} par la rotation de 45^0 autour de l'axe Oz suivie d'une dilatation d'un facteur $\sqrt{2}$ dans les directions horizontales.

Les poids pour la surface X sont les $w_{ij} = w'_i w_j$ où les w'_i sont les poids 1, 1 et 2 du quart de cercle et les w_j sont les poids de γ.

Références

[F] G. FARIN, *Curves and surfaces for computer aided geometric design, a practical guide.* 2nd ed. Academic Press. Boston, Mass. (1990).

[N] G. FARIN, *NURBS. From projective geometry to practical use.* 2nd ed. A. K. Peters, Wellesley, Mass. (1999).

[HL] J. HOSCHEK, D. LASSER, *Grundlagen der geometrischen Datenverarbeitung.* Teubner, Stuttgart (1989). `English translation :` *Fundamentals of computer aided geometric design.* A. K. Peters, Wellesley, Mass. (1993).

[R] J.-J. RISLER, *Méthodes mathématiques pour la CAO.* Masson, Paris (1991).

Cinquième partie

Techniques usuelles pour la visualisation 3D

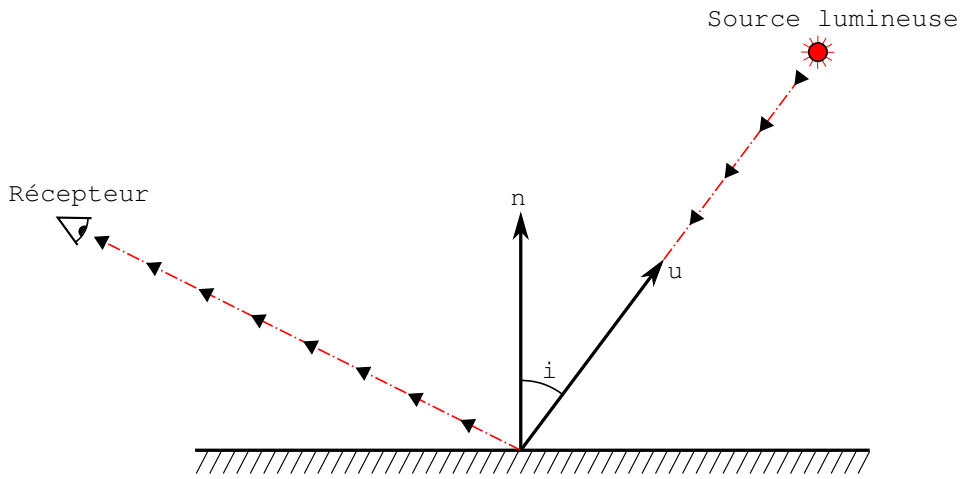
Contenu

14 Modèles d'éclairement

On va donner ici deux modèles de calcul d'éclairement local, c'est à dire qui ne prennent en compte que l'éclairement direct par une source lumineuse ponctuelle, sans s'occuper du rayonnement éventuel diffusé par d'autres objets de la scène.

14.1 Modèle et lissage de Gouraud

Ce modèle s'applique au calcul de l'intensité lumineuse réémise par diffusion d'un objet mat (papier, bois, etc...). L'intensité ne dépend que de l'angle d'incidence i avec le vecteur normal à l'objet, et est la même dans toutes les directions. Il faut noter que ce modèle est local, et ne permet pas — contrairement au calcul de radiosité par exemple — de prendre en compte les éclairement mutuels des objets entre eux. En outre, il ne permet que la modélisation de sources lumineuses ponctuelles.



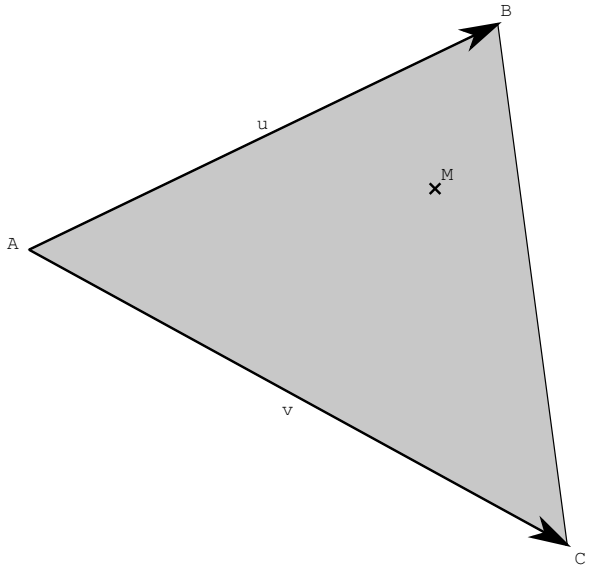
Modèle d'éclairement de Gouraud

En notant $\vec{u}$ un vecteur unitaire dans la direction de la source lumineuse et $\vec{n}$ un vecteur unitaire normal à la surface, l'intensité réémise I est donnée par la formule :

$$I = I_{ambiente} + (\cos i) C_{\text{diffuse}} I_{\text{incidente}} = I_{\text{ambiente}} + (\vec{u} \cdot \vec{n}) C_{\text{diffuse}} I_{\text{incidente}} \ .$$

Ici, le coefficient C_{diffuse} correspond à la réflectivité diffuse du matériau pour la longueur d'onde considérée (comprise entre 0 et 1) et I_{ambiente} correspond à une intensité lumineuse arbitraire, propre à l'objet en l'absence d'éclairement direct.

En OpenGL les objets sont généralement discrétisés en triangles par le système et la normale à la surface est seulement définie par le programme sur les sommets de ces triangles. Le lissage de Gouraud consiste, pour un triangle ABC, à considérer le repère local $(A, \overrightarrow{AB}, \overrightarrow{AC})$.



Le repère local associé à un triangle

Les coordonnées (u, v) de tout point M du triangle ABC peuvent s'exprimer dans cette base locale, et vérifient :

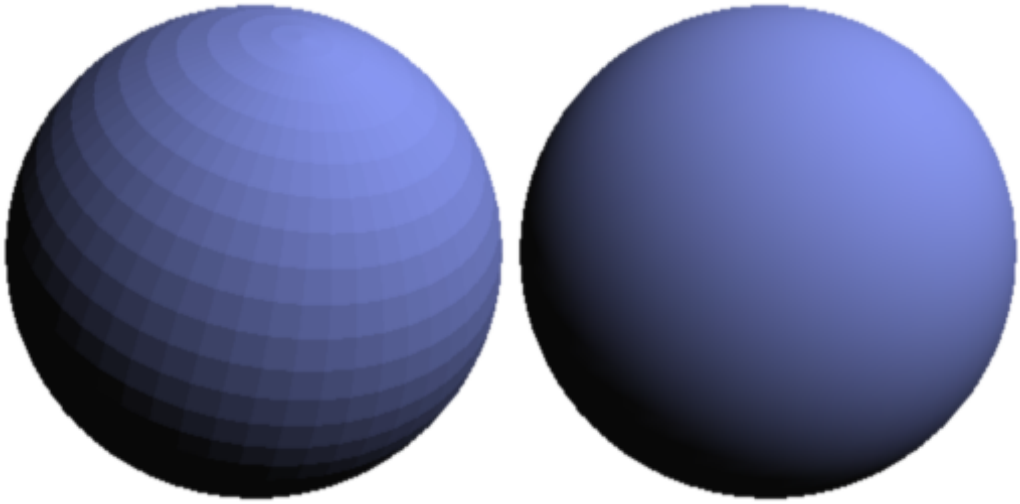
$$u \geq 0 \qquad v \geq 0 \qquad u + v \leq 1 \ .$$

Les sommets A, B et C du triangle peuvent par exemple s'obtenir respectivement pour les coordonnées $(0, 0)$, $(1, 0)$ et $(0, 1)$. Supposons à présent qu'on ait défini trois quantités λ_A, λ_B et λ_C (correspondant par exemple à une couleur ou un vecteur normal) définies en chaque sommet du triangle et considérons la forme affine :

$$\lambda(M) = \lambda(u, v) = \lambda_A + u(\lambda_B - \lambda_A) + v(\lambda_C - \lambda_A) \ .$$

On vérifie sans problème que $\lambda(A) = \lambda_A$, $\lambda(B) = \lambda_B$ et $\lambda(C) = \lambda_C$. Ainsi λ définit une fonction continue qui interpole les valeurs données aux sommets du triangle. Dans le cas où on l'utilise pour interpoler l'éclairement on parle de lissage (ou ombrage) de Gouraud.

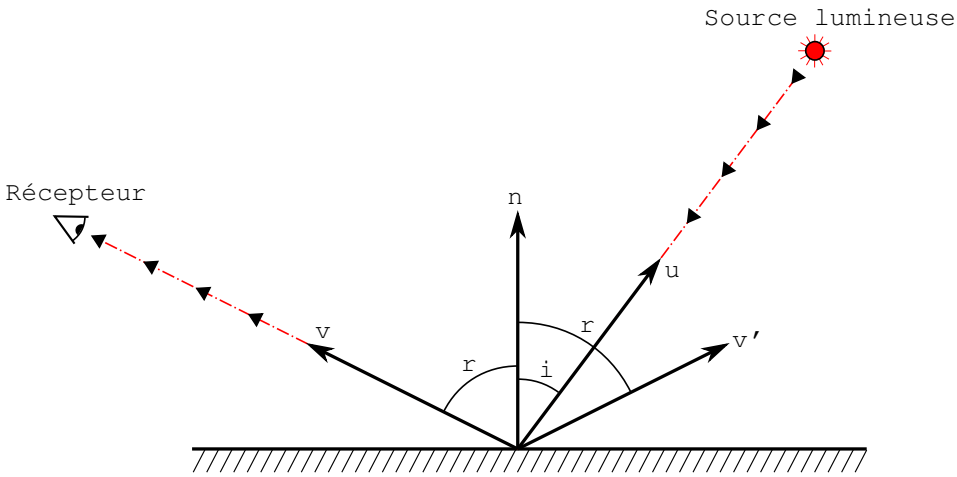
Dans le modèle utilisé par OpenGL, une couleur qui correspond à l'éclairement est ainsi calculé par la formule précédente aux sommets des triangles, et interpolée sur chaque pixel en utilisant λ.



Modèle de Gouraud, à droite avec lissage

14.2 Modèle de Phong

Ce modèle est une amélioration du modèle de Gouraud dans le cas de surfaces brillantes (plastique, métal, bois poli, etc...). Il prend en compte la direction $\vec{v}$ de réémission du rayonnement en ajoutant un terme supplémentaire. On notera $\vec{v}'$ le vecteur unitaire symétrique de $\vec{v}$ par rapport à la normale $\vec{n}$.



Modèle d'éclairement de Phong

En notant r l'angle entre la direction de réémission et le vecteur normal, l'intensité réémise I est donnée par la formule :

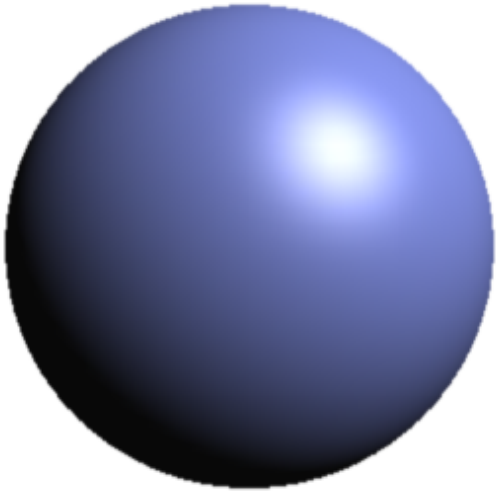
$$I = I_{ambiente} + (C_{\text{diffuse}} \cos i + C_{\text{spéculaire}} \cos^{\alpha} r)\, I_{\text{incidente}}$$

$$= I_{ambiente} + (C_{\text{diffuse}}(\vec{u} \cdot \vec{n}) + C_{\text{spéculaire}}(\vec{v}' \cdot \vec{u})^{\alpha})\, I_{\text{incidente}} \ .$$

Ce modèle est empirique, et n'est pas basé sur une modélisation physique rigoureuse des phénomènes électromagnétiques. Le coefficient $C_{\text{spéculaire}}$ est la réflectivité spéculaire

du matériau, α est l'exposant spéculaire. Le terme supplémentaire ajouté au modèle de Gouraud introduit des taches lumineuses brillantes lorsque le récepteur est quasiment dans la direction de réflection de la lumière incidente. Plus l'exposant α est grand, plus ces taches sont petites et perçues comme intenses. On peut donner quelques ordres de grandeur :

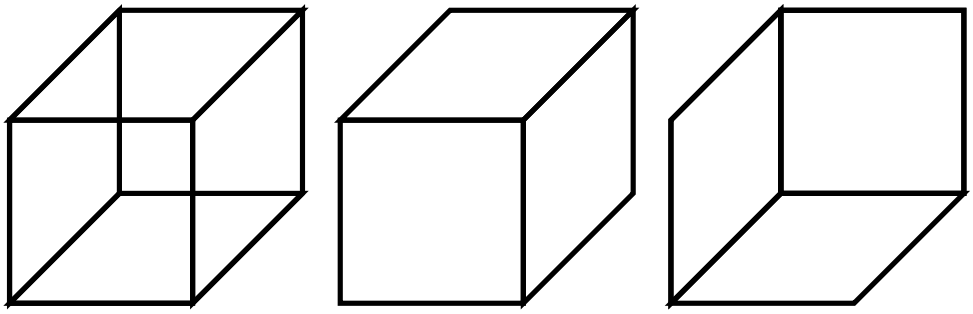
- $\alpha \approx 3$ pour du bois non verni ;
- $\alpha \approx 10$ pour du métal ;
- $\alpha \approx 30$ pour du plastique brillant.



Modèle de Phong avec $\alpha = 30$

15 Élimination des parties cachées

L'élimination des parties cachées permet de lever certaines ambiguïtés et aberrations de la visualisation 3D.

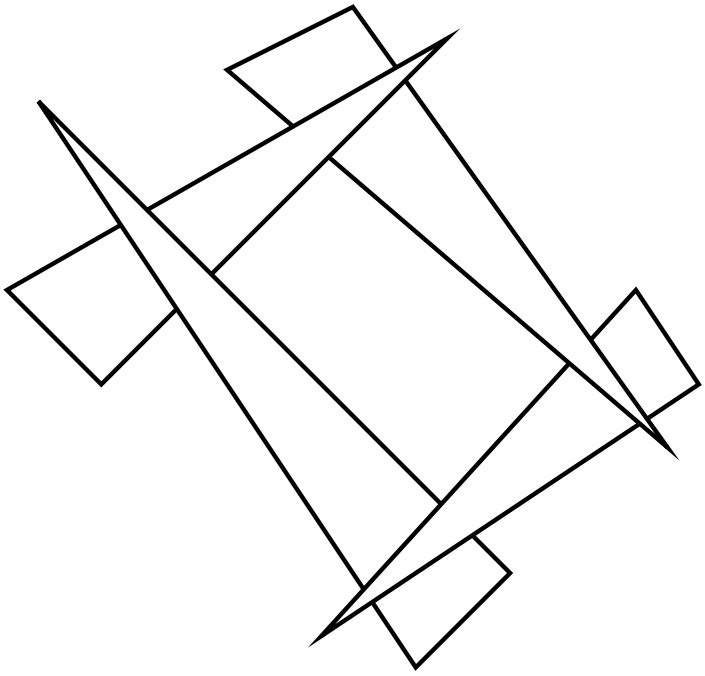


Exemple d'ambiguïté 3D : le cube de gauche peut être interprété par l'œil de deux façons différentes. L'ambiguïté est levée pour les deux autres cubes en enlevant les parties cachées.

Les méthodes suivantes sont utilisées exclusivement pour la visualisation en temps réel ; en effet l'algorithme de lancer de rayon élimine automatiquement les parties cachées puisqu'on considère toujours les intersections des rayons lancés avec l'objet le plus proche de la caméra

15.1 Le *depth buffer*

Une première idée pour cacher les faces non visibles consiste à les représenter de la plus éloignée à la plus proche. Ainsi, les faces situées «devant» vont cacher celles situées «derrière». Cette méthode, en plus d'être lourde algorithmiquement lorsqu'il y a beaucoup de faces, présente l'inconvénient de ne pas fonctionner dans tous les cas.



Exemple de 4 triangles pour lesquels la méthode de classement des faces ne fonctionne pas

C'est pourquoi on lui préfère la méthode du *tampon de profondeur* (*depth buffer*, ou encore *Z-buffer*). La mémoire graphique est munie d'un tampon supplémentaire de mêmes dimensions contenant la profondeur de chaque pixel par rapport à la caméra. Initialement chaque pixel a sa profondeur définie à $+\infty$. C'est lors de la discrétisation des objets en pixels (tramage) que cette information est utilisée. Pour un pixel recouvert lors de l'échantillonnage d'un objet, on note z' la distance du point correspondant sur l'objet à la caméra et z la valeur actuellement stockée dans le tampon de profondeur pour ce pixel.

- si $z' \geq z$ alors le pixel est situé plus loin de la caméra que le dernier objet échantillonnée à cet endroit. On l'ignore et on ne modifie rien ;
- si $z' < z$ alors le pixel est situé plus près de la caméra que le dernier objet échantillonné à cet endroit. On stocke alors z' dans le tampon de profondeur pour ce pixel, et on fait le calcul habituel pour déterminer sa couleur.

Ainsi, quel que soit l'ordre selon lequel les objets sont tracés, chaque pixel contiendra la couleur du point de la scène échantillonné le plus proche de la caméra, occultant automatiquement ceux qui sont situés derrière.

L'intérêt de cette méthode est qu'elle ne modifie pas la complexité algorithmique du rendu, au prix toutefois d'une consommation de mémoire plus grande.

Avec OpenGL, en supposant que le contexte d'affichage dispose d'un tampon de profondeur (voir `glutInitDisplayMode`), l'utilisation du tampon de profondeur est activée / désactivée en utilisant :

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_TEST);
```

Il ne faut pas oublier de le remettre à zéro préalablement à tout affichage de primitive :

```
glClear(GL_DEPTH_BUFFER_BIT);
```

15.2 Le *backface culling*

On suppose qu'on dispose d'un objet volumique de taille finie, dont on connait une représentation de la surface avec une normale sortante (c'est à dire dirigée vers l'extérieur). Soit P un point situé à l'extérieur de l'objet et $\vec{u}$ un vecteur non nul, alors la demi-droite d'origine P et de direction $\vec{u}$ rencontre la surface de l'objet en un nombre pair d'intersections (en comptant deux fois les endroits éventuels où la demi-droite est tangente à la surface). En groupant ces points d'intersection par paires de la plus proche à la plus éloignée de P, il est facile de constater qu'au premier point de chaque paire le vecteur normal $\vec{n}$ est de sens opposé à $\vec{u}$ ($\vec{u} \cdot \vec{n} \leq 0$, lorsque la demi-droite pénètre dans l'objet) et qu'au second point il est de même sens ($\vec{u} \cdot \vec{n} \geq 0$, lorsque la demi-droite ressort de l'objet).

Ainsi, quand on représente l'objet en projection sur l'écran d'une caméra située en P, tous les points de sa surface dont le vecteur normal est dans le sens de la caméra ne seront pas visibles, puisqu'ils seront recouverts par un point plus proche. Cette propriété permet automatiquement, une fois qu'on a discrétisé la surface de l'objet en polygones, d'en éliminer a priori environ la moitié (tous ceux qui sont «à l'arrière» de l'objet par rapport à la caméra). Cette méthode permet de gagner beaucoup de temps quand on fait de la visualisation en temps réel.

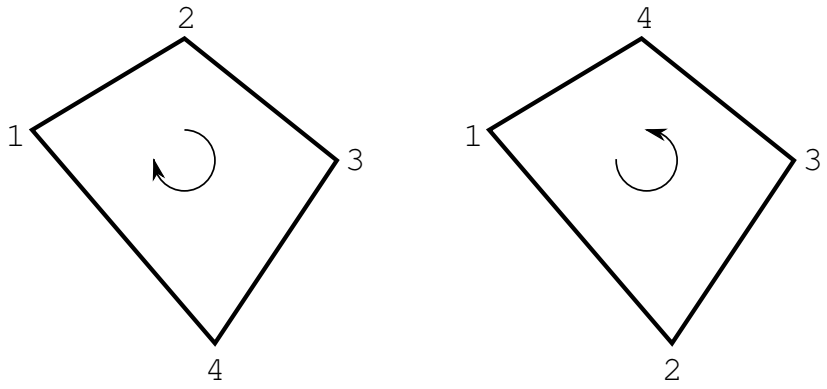
Avec OpenGL, l'élimination des polygones arrières (*backface culling*) est activée / désactivée en utilisant :

```
glEnable(GL_CULLFACE);
glDisable(GL_CULLFACE);
```

Puisque la normale à un polygone peut varier, le critère utilisé pour déterminer si un polygone est vu par l'avant ou par l'arrière est basé sur le sens de rotation de ses vertices après projection :

- si les vertices tournent dans le sens contraire des aiguilles d'une montre par rapport au centre du polygone, celui-ci est considéré comme vu par l'avant, et sera donc affiché ;
- si les vertices tournent dans le sens des aiguilles d'une montre par rapport au centre du polygone, celui-ci est considéré comme vu par l'arrière et ne sera donc pas affiché si le *backface culling* est activé.

Dans ces conditions, il faudra donc s'assurer de donner les vertices des polygones en tournant dans le sens direct par rapport au vecteur normal sortant de l'objet si l'on souhaite utiliser correctement cette fonctionnalité.



Le polygone de gauche est considéré comme vu par l'arrière (donc ne sera pas affiché si le backface culling est activé), celui de droite est considéré comme vu par l'avant

16 Textures

En synthèse d'images, utiliser uniquement des surfaces monochromes lisses ne suffit pas toujours à remplir certains objectifs de la visualisation 3D. La limitation n'est pas seulement d'ordre esthétique : il peut être utile lors de l'affichage d'un objet, de donner l'impression qu'il est fait en un matériau spécifique. Cette information supplémentaire permet en général une lisibilité plus grande des images de synthèse, en facilitant grandement la perception des volumes et l'identification des différents objets.

16.1 Principe général

La plupart des matériaux naturels présentent un aspect non uniforme à leur surface, qui varie en fonction du point considéré. Localement, cet aspect résulte de la combinaison de plusieurs paramètres :

- la normale au point considéré, qui peut varier beaucoup à petite échelle et varier très peu en moyenne à grande échelle (pierre rugueuse, métal rayé, etc...) ;
- le spectre d'absorption de la lumière incidente ou en d'autres termes la couleur selon laquelle le matériau est perçu localement (bois, tissu avec motif, etc...) ;
- de façon plus générale, tous les paramètres qui interviennent dans le modèle d'éclairement utilisé peuvent varier (ce qui peut faire jusqu'à 12 paramètres si on utilise le modèle de Phong, 4 pour chaque canal).

Dans la plupart des cas, la période spatiale des variations de ces attributs est très petite devant la taille moyenne de discrétisation des objets 3D — que ce soit des polygones en temps réel, ou des surfaces plus complexes en lancer de rayon. La discrétisation

de ces variations en objets de taille suffisamment petite avec des paramètres uniformes entrainerait un surcoût démesuré en terme de puissance de stockage et de calcul.

La méthode la plus utilisée pour implémenter ces variations locales des attributs de surface consiste à définir la géométrie simplement, sans découper les objets d'avantage, et à appliquer les variations uniquement au moment de la discrétisation en pixels. Par exemple, que ce soit en lancer de rayon ou en OpenGL, pour chaque surface 3D il est possible de définir une information de couleur $I(x, y, z)$ (correspondant généralement au coefficient de luminosité diffuse dans le modèle de Phong) qui dépend des coordonnées tridimensionnelles du point considéré. Lors de la visualisation de la scène, une correspondance est établie entre les centres (i, j) des pixels échantillonnés et des points $(x_{i,j}, y_{i,j}, z_{i,j})$ de la surface 3D. Pour calculer la couleur du pixel, on utilise alors tout simplement le coefficient de luminosité diffuse $I(x_{i,j}, y_{i,j}, z_{i,j})$ au lieu d'un coefficient constant.

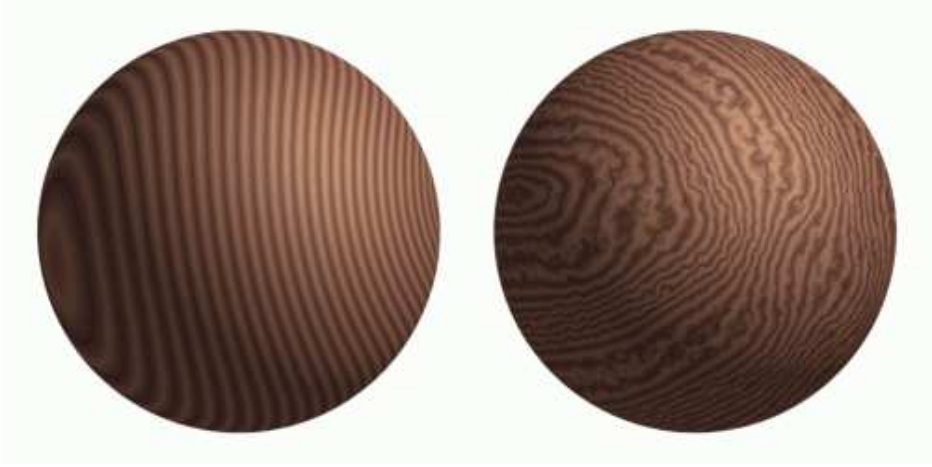
Le même procédé peut être mis en place pour d'autres attributs, par exemple il est possible d'ajouter une petite perturbation locale au vecteur usuel normal à la surface pour donner un aspect bosselé ou rugueux, sans augmenter la complexité géométrique des objets.

16.2 Textures procédurales

Les textures procédurales sont généralement utilisées pour le lancer de rayon, dans le cas où l'on connait une description mathématique des propriétés du matériau. Par exemple, on peut obtenir très simplement une texture ressemblant à du bois en simulant la maillure par des cylindres concentriques :

$$I(x, y, z) = \cos \sqrt{x^2 + y^2} \ .$$

On peut aussi ajouter une perturbation aléatoire locale à des textures de bases simples afin de donner une impression moins artificielle.



Une texture spatiale de bois, à gauche sans perturbation, à droite avec une perturbation aléatoire

Une grande variété de matériaux naturels peut ainsi être rendue, en utilisant des algorithmes génériques simples et des perturbations aléatoires pour gommer les structures trop périodiques.

Quelques exemples de textures procédurales, modélisées par le logiciel Pov-Ray

16.3 Textures précalculées

Les impératifs de performance exigée des logiciels de visualisation en temps réel interdisent l'emploi de textures procédurales $I(x, y, z)$ complexes. Toutefois rien n'empêche de les calculer a priori à une résolution suffisamment grande, et de stocker les valeurs obtenues pour les utiliser lorsqu'il faut échantillonner un attribut de texture en un point donné. Un autre problème technique apparait alors : la fonction I dépendant de trois paramètres, son échantillonnage spatial à une fréquence N va nécessiter une quantité de mémoire proportionnelle à N^3, ce qui peut vite devenir excessif. Si on prend par exemple $N = 1000$, le stockage d'une texture en mode RGB sur 24 bits va consommer 3.10^9 octets, c'est à dire plus de la moitié d'un DVD ! En outre, la surface des objets étant de dimension 2, seule une quantité infime de l'échantillonnage complet de la texture va effectivement être utilisée dans la pratique.

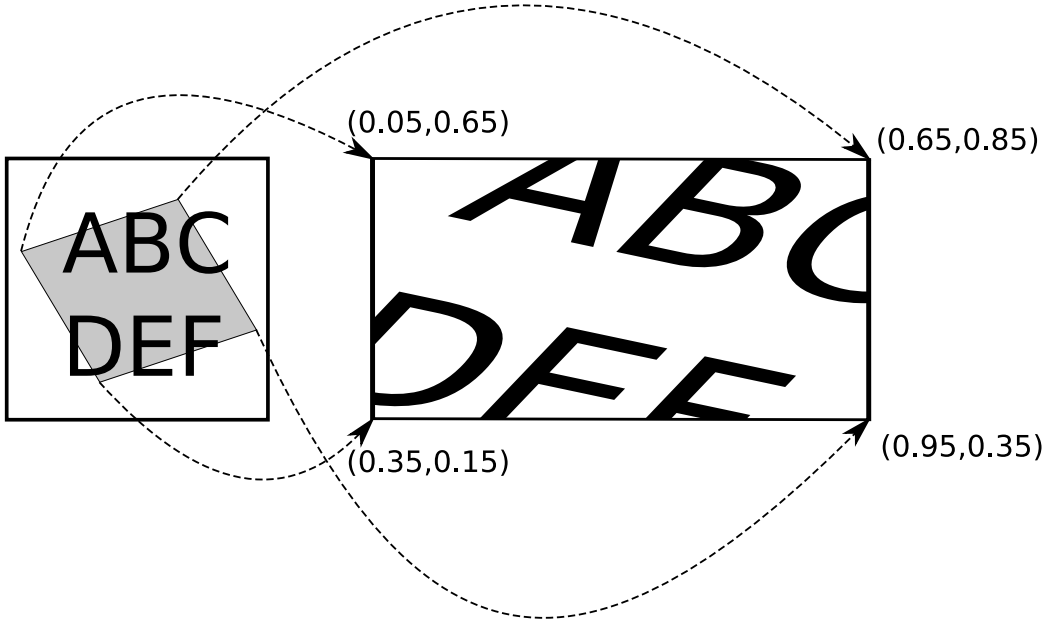
Pour ces raisons, les textures tridimensionnelles sont encore très peu utilisées dans les logiciels de temps réel, on préfère se ramener à des textures bidimensionnelles (voire monodimensionnelles dans certains cas). Pour cela, on suppose qu'on a défini deux fonctions $U(x, y, z)$ et $V(x, y, z)$ à valeurs dans l'intervalle $[0, 1]$. En considérant qu'une texture est un signal bidimensionnel $I(u, v)$ défini sur le pavé $[0, 1]^2$, on se ramène au cas tridimensionnel en prenant la valeur de $I(U(x, y, z), V(x, y, z))$ en chaque point de la surface. Ainsi, il suffit d'avoir préalablement stocké une fois pour toute un échantillonnage du signal I dans un tableau, l'information de texture en un point (x, y, z) pourra tout simplement

être obtenue en lisant la valeur du tableau dont les coordonnées sont les plus proches de $U(x, y, z)$ et $V(x, y, z)$.

Dans ces conditions une texture peut être représentée comme une image digitale. On emploiera le terme *texel* (équivalent de *pixel* pour les textures) pour désigner un point de l'échantillonnage.

16.3.1 Mappage

Avec OpenGL, lorsqu'on définit un polyèdre il est possible de définir des coordonnées de texture U et V en chaque sommet. Ces coordonnées sont interpolées linéairement sur les pixels à l'intérieur de la face grâce à la fonction d'interpolation `lambda` utilisée pour le lissage de Gouraud.



Mappage d'une texture (à gauche) sur une face rectangulaire. Les coordonnées (U, V) utilisées sont indiquées aux 4 coins

La difficulté principale de l'utilisation des textures 2D réside dans la définition des fonctions $U(x, y, z)$ et $V(x, y, z)$. Ainsi, si on veut recouvrir une sphère avec une texture

2D carrée, il est possible de montrer qu'il existe toujours au moins un point singulier où les coordonnées de textures se rejoignent (on parle alors de pôle). Certains systèmes récents utilisent des textures composées de 6 carrés qui forment le patron d'un cube pour remédier à ce problème : la forme peut alors être «enveloppée» plus facilement dans la texture.

16.3.2 Filtrage

L'utilisation de textures précalculées impose de reconstruire un signal continu à partir du signal échantillonné. Deux cas peuvent se produire, selon qu'un même texel recouvre plusieurs pixels (sous-échantillonnage) ou que plusieurs texels recouvrent un même pixel (sur-échantillonnage).

Sous-échantillonnage

À ce stade, il n'est plus possible de reconstruire l'information du signal original perdue lors de sa discrétisation. Le filtre théorique parfait nécessiterait de faire le produit de convolution avec la fonction $\text{sinc}(u)\text{sinc}(v)$ (voir la première partie), mais un tel calcul serait beaucoup trop long. On se contente en pratique de faire la convolution avec la transformée de Fourier du filtre parfait, qui est une fonction qui vaut 1 sur le support carré d'un pixel, 0 partout ailleurs et qui réalise une approximation du filtre parfait. Cette opération revient à pondérer la moyenne des couleurs aux 4 texels les plus proches par leur distance (horizontale et verticale) au point d'échantillonnage, on parle alors de filtrage linéaire.

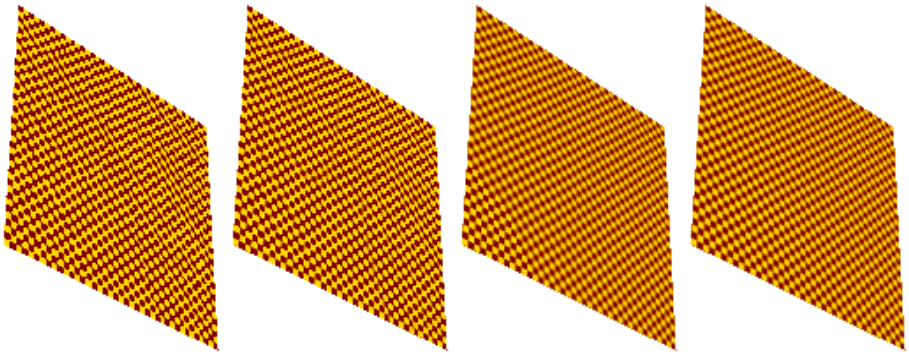
Sous-échantillonnage d'une texture, à gauche approximation par plus proche texel (pas de filtrage), à droite moyenne pondérée des 4 texels les plus proches

Sur-échantillonnage

Pour supprimer les effets de crènelage lors du sur-échantillonnage d'une texture, il suffit de moyenner les texels qui recouvrent le support du pixel concerné. Cette opération étant très coûteuse en terme de puissance de calcul, on va générer préalablement plusieurs copies de la texture, obtenues en divisant successivement ses dimensions par 2 et en moyennant les texels par groupe de 2×2 : ces copies réduites forment l'ensemble des *mipmaps*. Ainsi, il suffira de faire un sur-échantillonnage comme précédemment du mipmap de taille inférieure la plus proche de celle du pixel (on parle alors de filtrage bilinéaire).

On peut encore améliorer la technique en faisant une moyenne pondérée sur 8 texels :
- 4 pixels du mipmap de la taille immédiatement inférieure à celle du pixel ;
- 4 pixels du mipmap de la taille immédiatement supérieure à celle du pixel ;

On parle alors de filtrage trilinéaire.



Sur-échantillonnage d'une texture, de gauche à droite : approximation par plus proche texel, filtrage linéaire, filtrage linéaire du mipmap le plus proche (ou bilinéaire), filtrage trilinéaire

Échantillonnage anisotropique

Les techniques énoncées plus haut ne prennent pas en compte les cas où l'une des directions de la texture est sur-échantillonnée et l'autre direction sous-échantillonnée. Il existe encore des méthodes de filtrage dites *anisotropiques* qui permettent d'appliquer des filtres différents dans les deux directions.

À gauche filtrage bilinéaire, à droite filtrage anisotropique

Table des matières