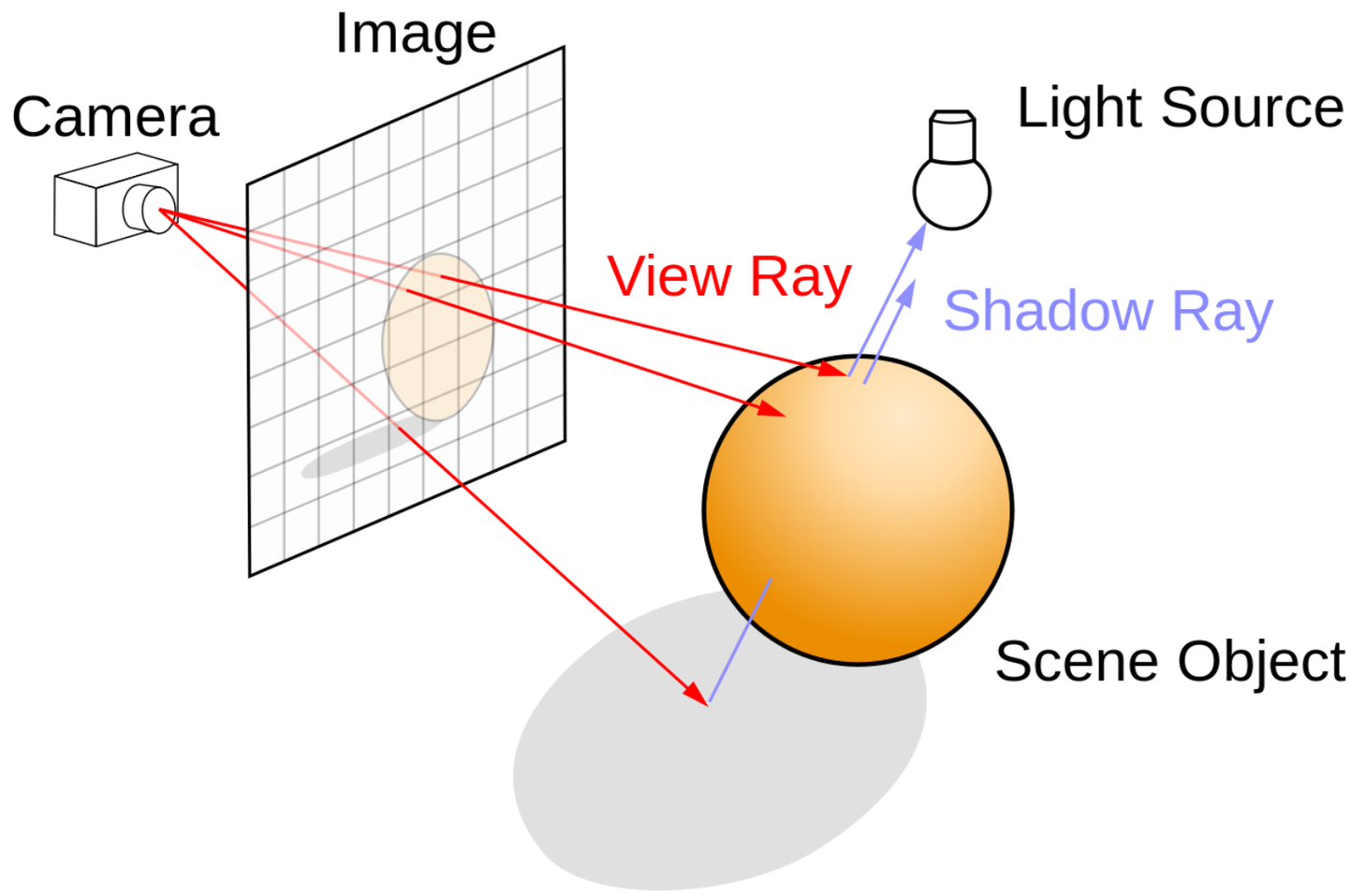
Rendu et éclairage

HAI605I

Rappel sur les vecteurs

Pourquoi des vecteurs ?



wikipedia

Mais aussi pour :

- Définir une scène virtuelle
- Définir une direction caméra
- Lancer une balle dans une scène
- Pour calculer une ligne de vue

→ Très utilisés dans la communauté du jeu (NVIDIA, Miscrosoft) :

- https://blogs.nvidia.com/blog/2018/03/21/ (epic-games-reflections-ray-tracing-offers-peek-gdc)
- https://www.youtube.com/watch?v=-zW3Ghz-WQw
- https://www.youtube.com/watch?v=81E9yVU-KB8

Scalaire

Quantité décrivant l'**amplitude** (i.e., un simple nombre)

- Poids
- Distance (Montpellier – Béziers = 73,6 km)
- Vitesse
- Taille d'un vecteur
- Nombres tels que $\pi = 3.14159$ ... etc

Sur l'ordinateur : int, float, double

Vecteurs

Quantité ayant une **direction en plus d'une amplitude**

Une représentation de vecteur B-M:

- Commencer en M et finir à B; le vecteur relie les 2
- Commencer en M : bouger de 73,6 km à 45 degrés au Sud ouest

Vecteurs

Quantité ayant une **direction en plus d'une amplitude**

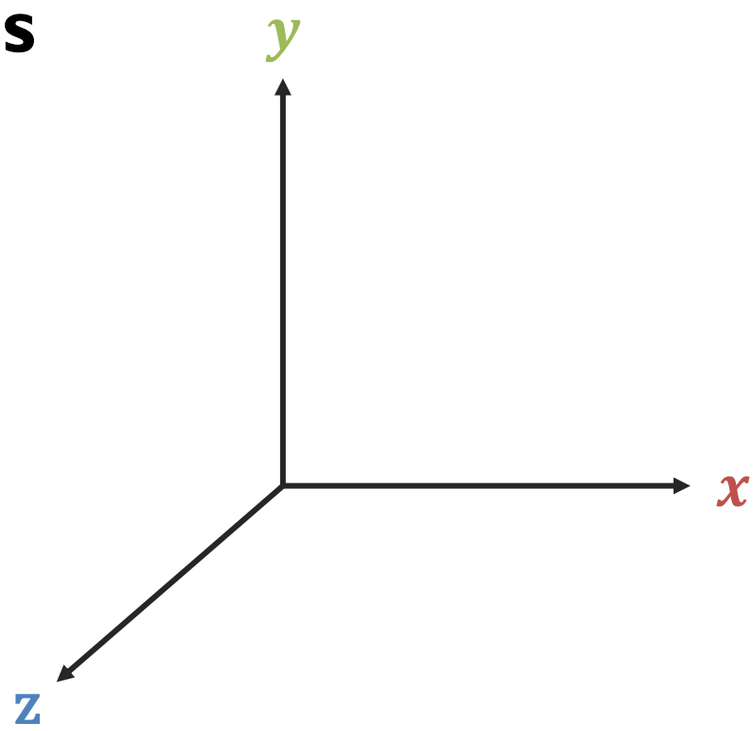
Une représentation **équivalente** du vecteur B-M

Directions de référence

→ **Système de coordonnées**

Nombre de directions de référence = dimension de l'espace
- Espace de dimension $d \Rightarrow \mathbb{R}^d$; 2D $\Rightarrow \mathbb{R}^2$; 3D $\Rightarrow \mathbb{R}^3$

Système de coordonnées **cartésiennes** en 3D :
- Direction de référence **orthogonales**

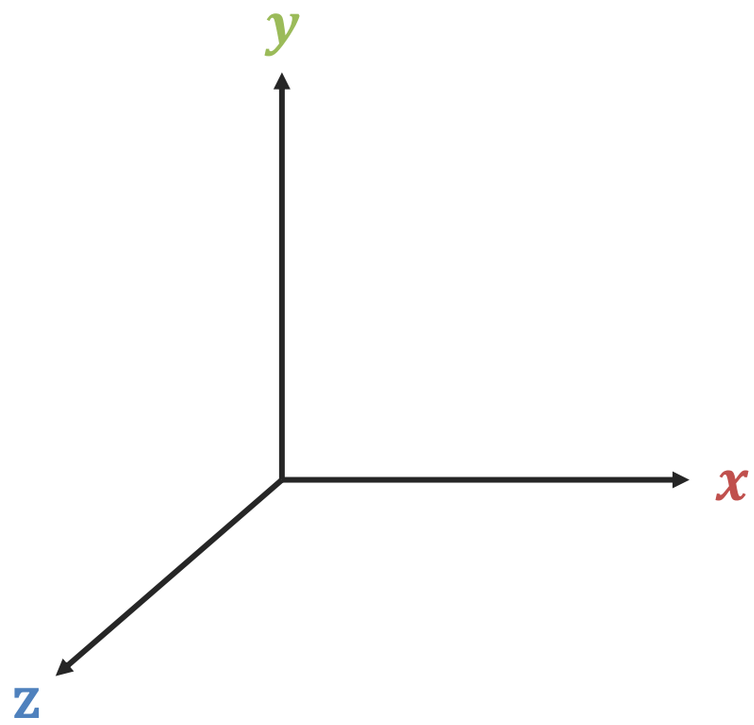


Système de coordonnées

Un point **P** est représenté par :

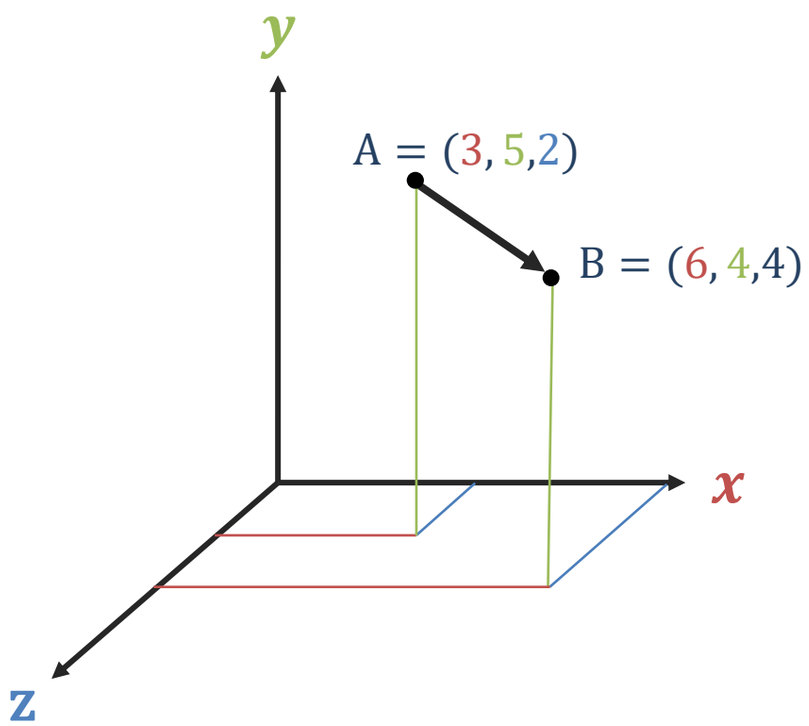
- Les coordonnées (x, y) en **deux** dimensions
- Les coordonnées (x, y, z) en **trois** dimensions
- Les coordonnées $(x_1, x_2, \ldots, x_d)$ en **d** dimensions

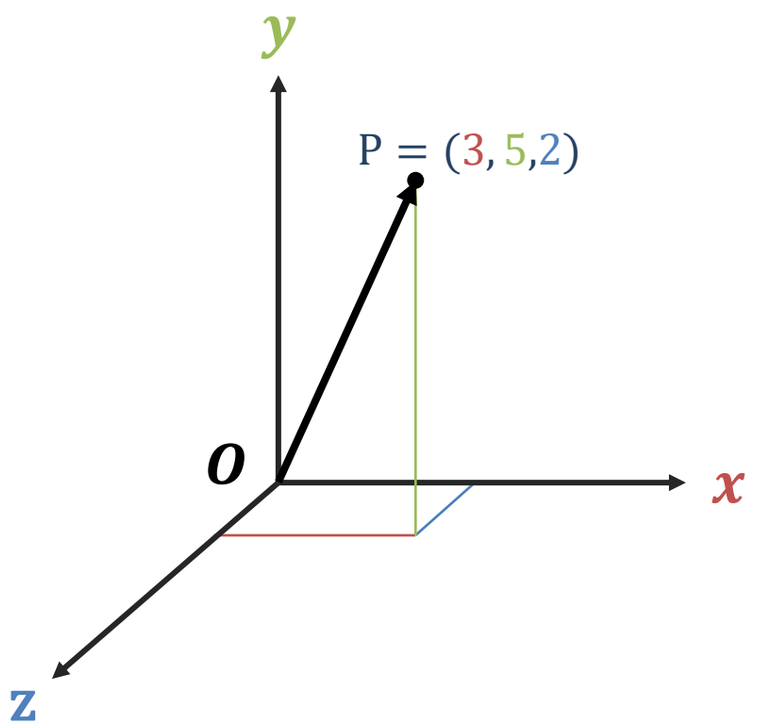
L'origine du système : toutes les entrées de P sont à zéro



Un **point** vs un **vecteur**

Un vecteur ne spécifie **pas de point de départ**





→ une **direction** et une **longueur** de la flèche

Représentation

- Un point P : $(x_1, x_2, \ldots, x_d)$ en d dimensions
  - par (x, y) en 2D et par $(\mathrm{x}, \mathrm{y}, \mathrm{z})$ en 3D

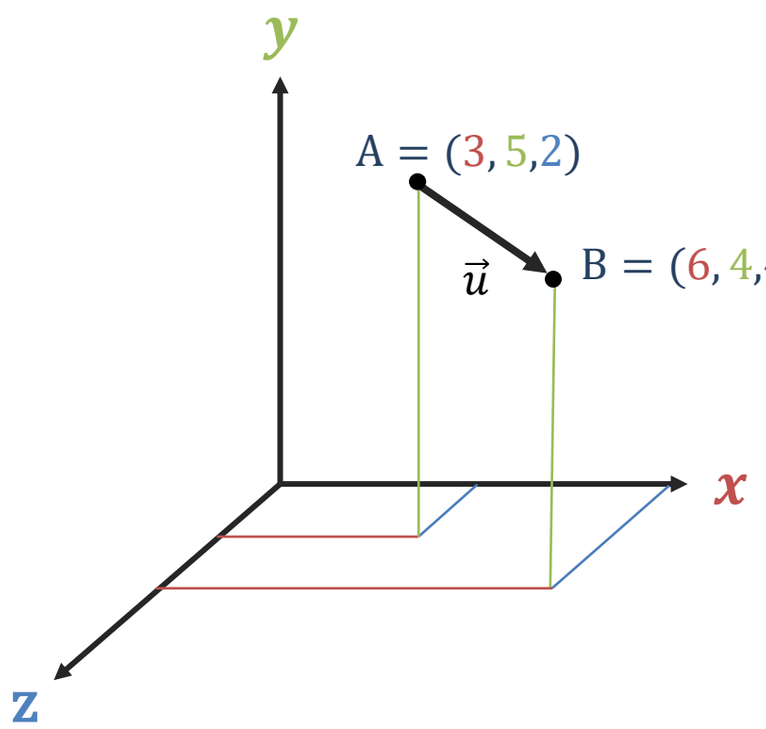
- Un vecteur $\vec{v}$: $\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{bmatrix}$ en dimension d
  - par $\begin{bmatrix} v_x \\ v_y \end{bmatrix}$ en 2D et par $\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$ en 3D

Représentation

Soit le vecteur $\vec{u}$ entre A et B :

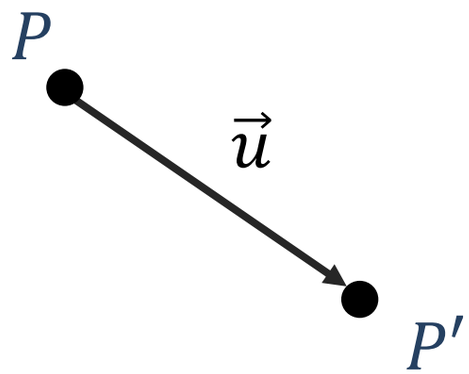
- définit une direction dans l'espace
- est définit par 3 coordonnées

$$u_x = B_x - A_x$$
$$u_y = B_y - A_y$$
$$u_z = B_z - A_z$$



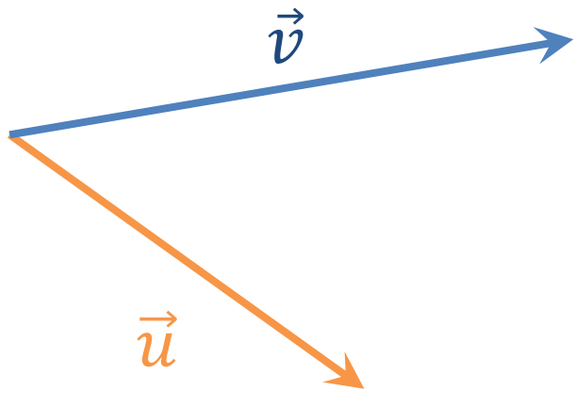
Permet de translater des objets :

$$P'_x = P_x + u_x$$
$$P'_y = P_y + u_y$$
$$P'_z = P_z + u_z$$



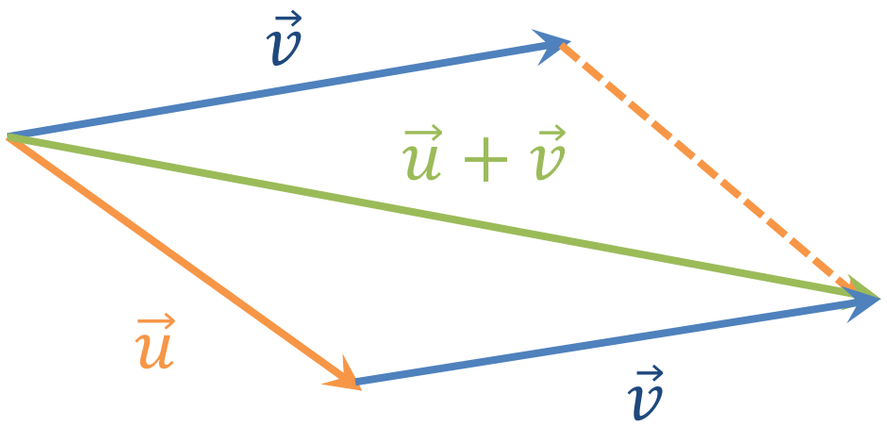
Opérations sur les vecteurs

Addition



Opérations sur les vecteurs

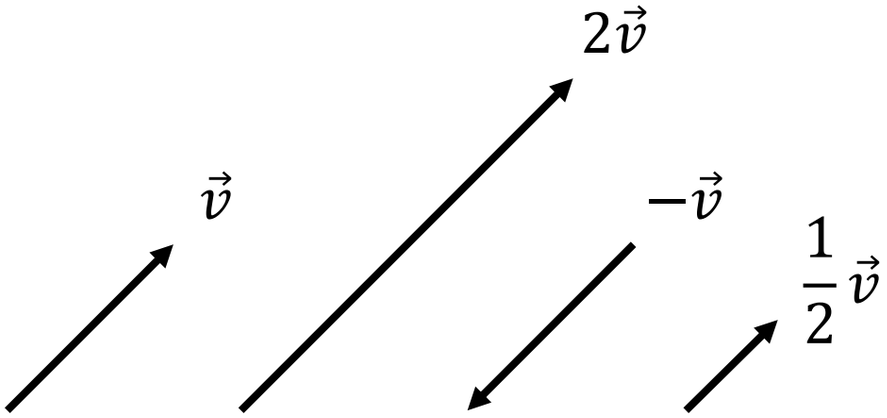
Addition



$$\vec{u} + \vec{v} = \begin{bmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \end{bmatrix}$$

Opérations sur les vecteurs

Multiplication avec un scalaire λ



$$\lambda\vec{v} = \begin{bmatrix} \lambda v_x \\ \lambda v_y \\ \lambda v_z \end{bmatrix}$$

Amplitude (longueur, norme)

La **norme** de $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$:

- Définie par $\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$
- Si $\vec{v} = \overrightarrow{AB}$, alors est la distance entre A et B
- Si $\|\vec{v}\| = 1$ alors $\vec{v}$ est **normalisé**
- Pour tout vecteur il existe un **vecteur normalisé unitaire** de même direction.
- Pour normaliser $\vec{v}$:

$$\hat{v} = \frac{1}{\|\vec{v}\|} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} v_x / \|\vec{v}\| \\ v_y / \|\vec{v}\| \\ v_z / \|\vec{v}\| \end{bmatrix}$$

Amplitude (longueur, norme)

La **norme** de $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$:

- Définie par $\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$
- Si $\vec{v} = \overrightarrow{AB}$, alors est la distance entre A et B
- Si $\|\vec{v}\| = 1$ alors $\vec{v}$ est **normalisé**
- Pour tout vecteur il existe un **vecteur normalisé unitaire** de même direction.
- Pour normaliser $\vec{v}$:

$$\hat{v} = \frac{1}{\|\vec{v}\|} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} v_x / \|\vec{v}\| \\ v_y / \|\vec{v}\| \\ v_z / \|\vec{v}\| \end{bmatrix}$$

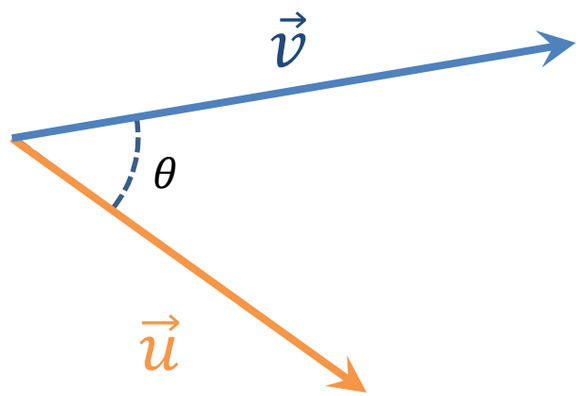
Les **vecteurs unitaires** des directions de référence forment les vecteurs de base ($\hat{x}$, $\hat{y}$, $\hat{z}$ en 3D).

Produit scalaire entre 2 vecteurs

« Dot product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$\vec{u}.\vec{v} = \vec{v}.\vec{u} = u_x v_x + u_y v_y + u_z v_z$ donne un **scalaire**

Géométriquement : $\vec{u}.\vec{v} = \|\vec{u}\|\|\vec{v}\| \cos\theta$



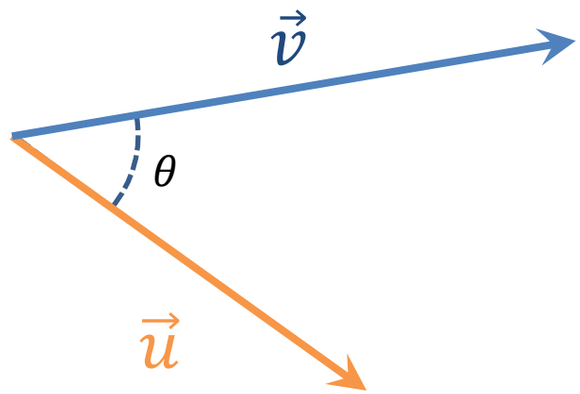
Produit scalaire entre 2 vecteurs

« Dot product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$\vec{u}.\vec{v} = \vec{v}.\vec{u} = u_x v_x + u_y v_y + u_z v_z$ donne un **scalaire**

Propriétés :

- Symétrie : $\vec{u}.\vec{v} = \vec{v}.\vec{u}$
- Distributivité : $\vec{u}.(\vec{v} + \vec{w}) = \vec{u}.\vec{v} + \vec{u}.\vec{w}$
- Homogénéité : $(\lambda\vec{u}).\vec{v} = \lambda(\vec{u}.\vec{v})$
- Lien avec la norme : $\vec{u}.\vec{u} = \|\vec{u}\|^2$



Remarque :

$$\vec{u}^T\vec{v} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

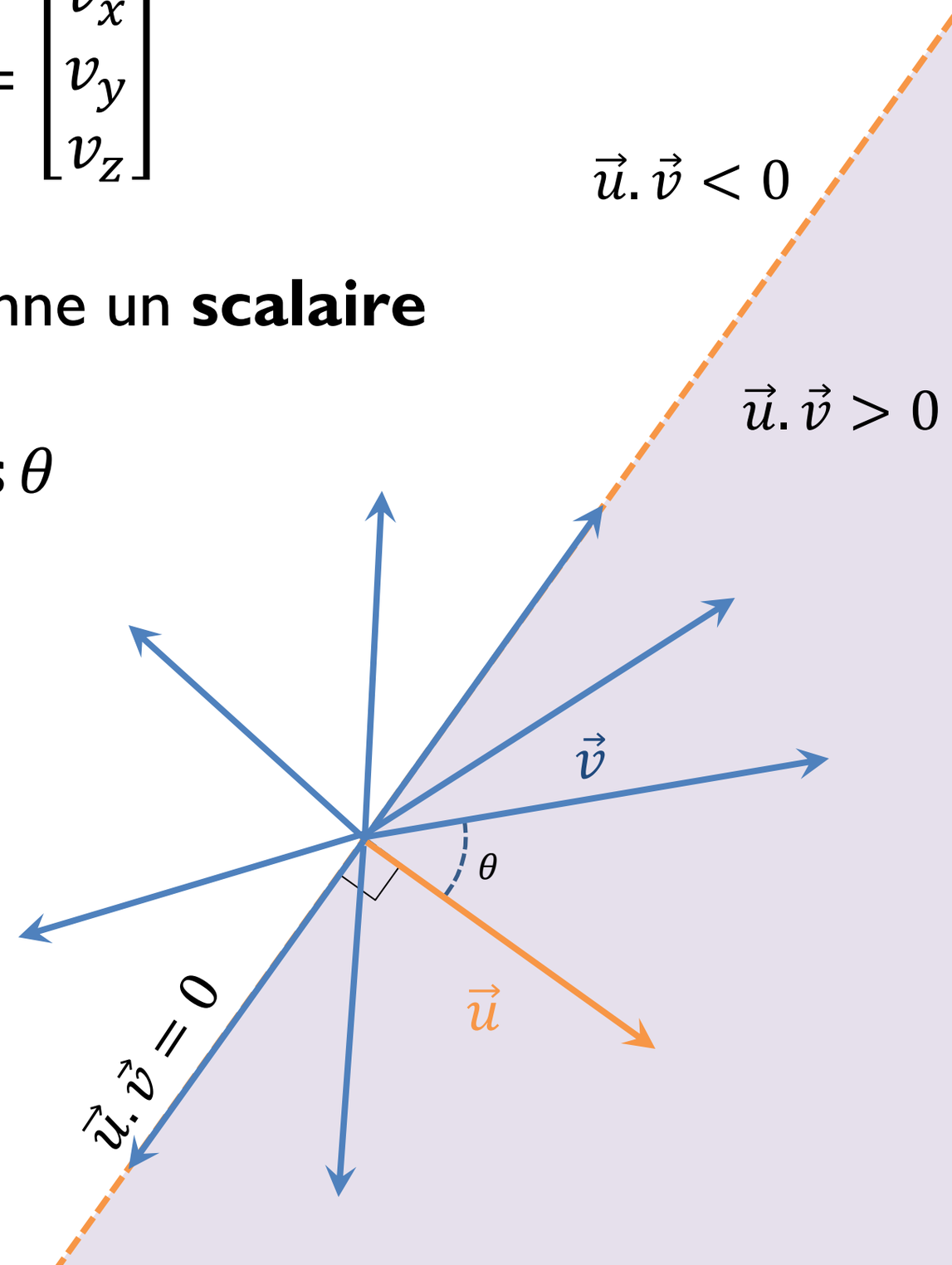
Produit scalaire entre 2 vecteurs

« Dot product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$\vec{u}.\vec{v} = \vec{v}.\vec{u} = u_x v_x + u_y v_y + u_z v_z$ donne un **scalaire**

Géométriquement : $\vec{u}.\vec{v} = \|\vec{u}\|\|\vec{v}\| \cos\theta$

$\vec{u}.\vec{v} = 0$ pour $\vec{u}$ et $\vec{v}$ **orthogonaux**
$(\theta = 90$ et $\theta = 270)$

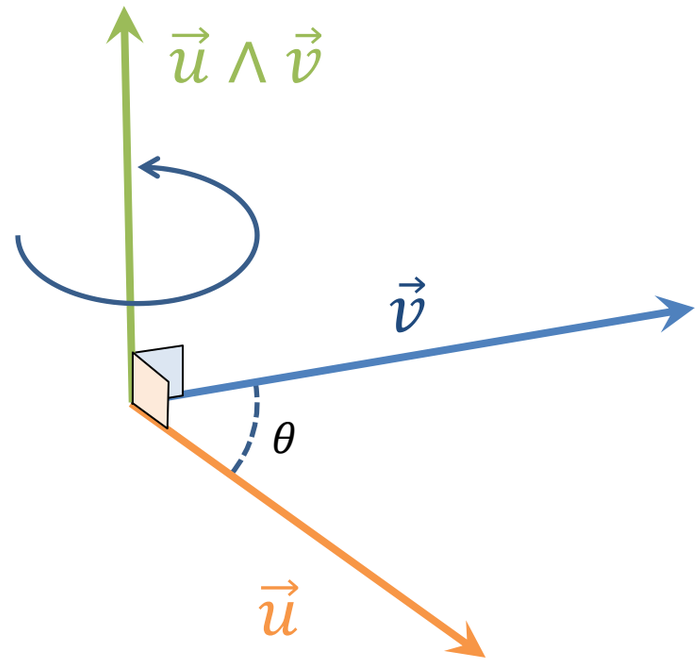


Produit vectoriel entre 2 vecteurs

« Cross product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$$\vec{u} \wedge \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

$\vec{v} \wedge \vec{v}$ est un vecteur orthogonal à $\vec{u}$ et à $\vec{v}$



Produit vectoriel entre 2 vecteurs

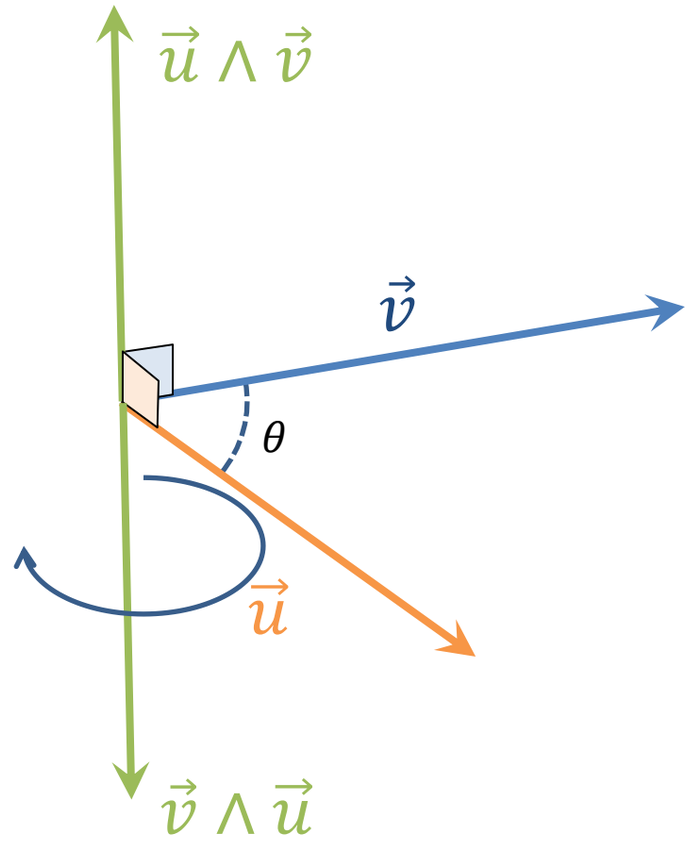
« Cross product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$$\vec{u} \wedge \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

$\vec{u} \wedge \vec{v}$ est un vecteur orthogonal à $\vec{u}$ et à $\vec{v}$

Propriétés :

- Antisymétrique : $\vec{u} \wedge \vec{v} = -\vec{v} \wedge \vec{u}$
- Distributivité : $\vec{u} \wedge (\vec{v} + \vec{w}) = \vec{u} \wedge \vec{v} + \vec{u} \wedge \vec{w}$
- Homogénéité : $(\lambda\vec{u}) \wedge \vec{v} = \lambda(\vec{u} \wedge \vec{v})$
- Non associatif : $\vec{u} \wedge (\vec{v} \wedge \vec{w}) = (\vec{u} \wedge \vec{v}) \wedge \vec{w}$



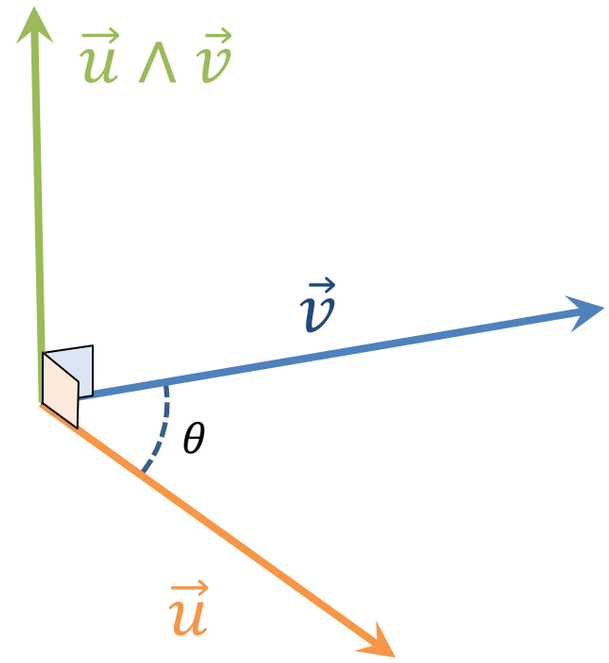
Produit vectoriel entre 2 vecteurs

« Cross product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$$\vec{u} \wedge \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

Géométriquement : $\|\vec{u} \wedge \vec{v}\| = \|\vec{u}\| \|\vec{v}\| \sin\theta$

$\vec{u} \wedge \vec{v} = 0$ pour $\vec{u}$ et $\vec{v}$ **colinéaires**
$(\theta = 0$ et $\theta = 180)$



Produit vectoriel entre 2 vecteurs

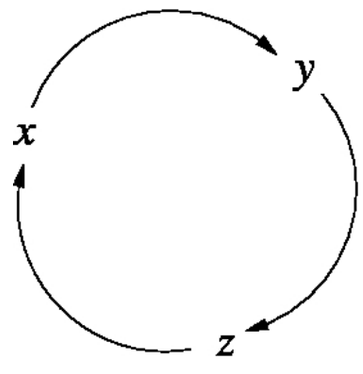
« Cross product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

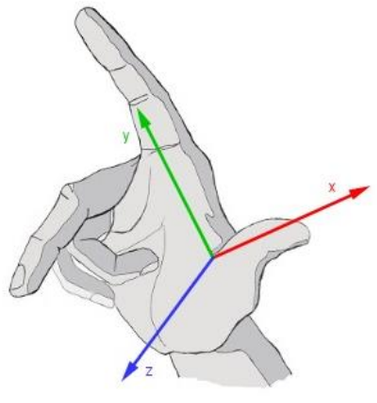
$$\vec{u} \wedge \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

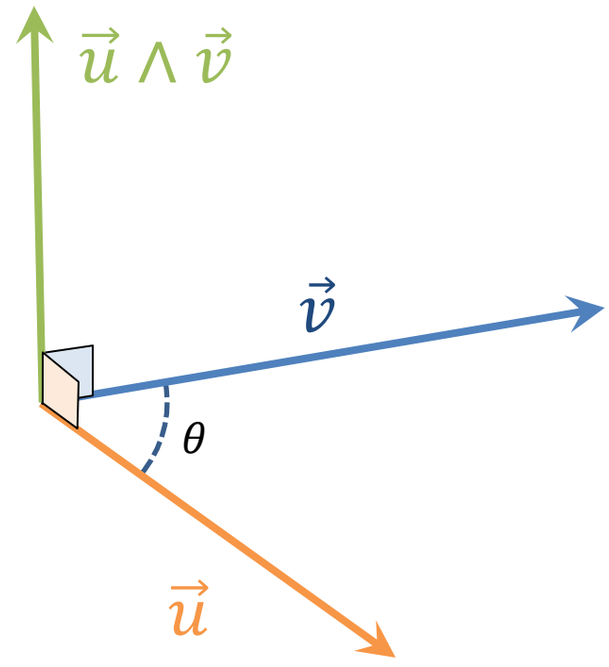
Système de coordonnées main droite :

$\hat{x} \times \hat{y} = \hat{z}, \hat{y} \times \hat{z} = \hat{x}, \hat{z} \times \hat{x} = \hat{y}$

i.e. $(\hat{x} \times \hat{y}).\hat{z} > 0$







Produit vectoriel entre 2 vecteurs

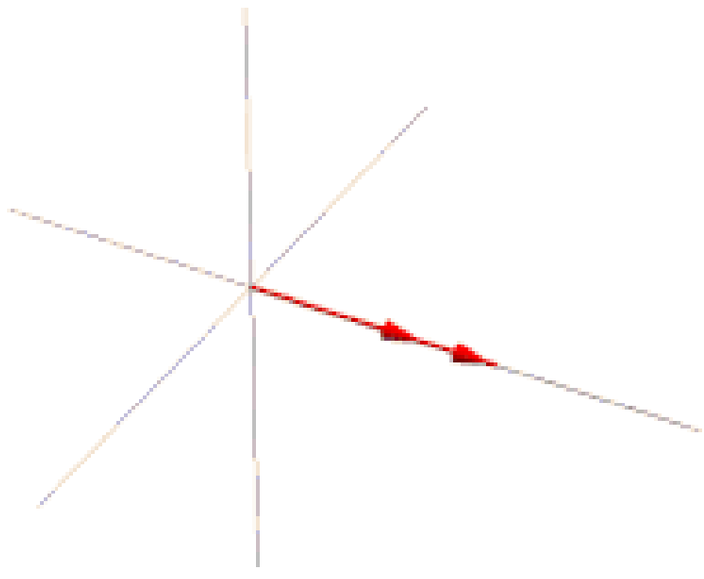
« Cross product » entre $\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$ et $\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

$$\vec{u} \wedge \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

$\vec{v} \wedge \vec{v}$ est un vecteur orthogonal à $\vec{u}$ et à $\vec{v}$

Utilisé pour calculer :

- les normales,
- Les repères orthonormés…



Projections

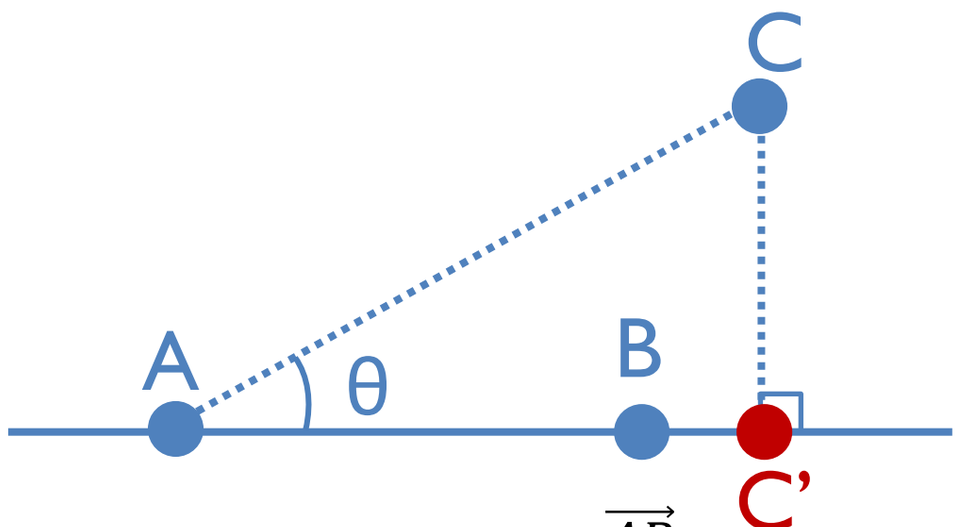
D’un point sur une droite

Soit la droit **d** définie par 2 points A et B. Le point C à projeter.

- Par Pythagore $\frac{\left\|\overrightarrow{AC'}\right\|}{\left\|\overrightarrow{AC}\right\|} = \cos\theta$
- On vient de voir que $\frac{\overrightarrow{AB}.\overrightarrow{AC}}{\left\|\overrightarrow{AB}\right\|\left\|\overrightarrow{AC}\right\|} = \cos\theta$

D’où $\left\|\overrightarrow{AC'}\right\| = \frac{\overrightarrow{AB}.\overrightarrow{AC}}{\left\|\overrightarrow{AC}\right\|}$

Finalement $C' = \begin{bmatrix} A_x + u_x\left\|\overrightarrow{AC'}\right\| \\ A_y + u_y\left\|\overrightarrow{AC'}\right\| \\ A_z + u_z\left\|\overrightarrow{AC'}\right\| \end{bmatrix}$



On note $\vec{u} = \frac{\overrightarrow{AB}}{\left\|\overrightarrow{AB}\right\|}$

Projections

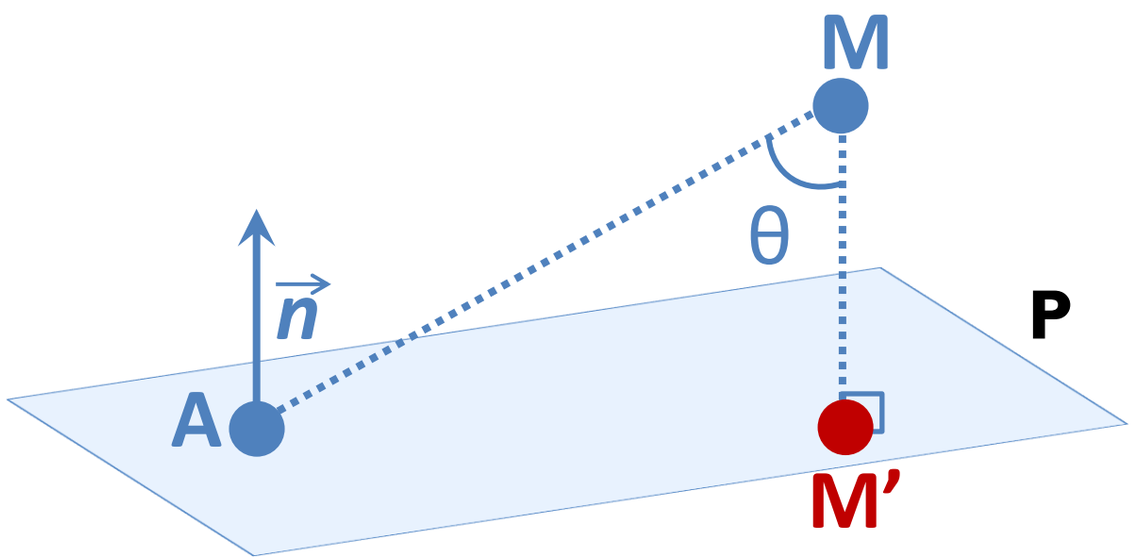
D’un point sur un plan

Soit le plan **P** défini par un point A et une normale $\vec{n}$. Le point M à projeter.

- Par Pythagore $\frac{\|\overrightarrow{MM'}\|}{\|\overrightarrow{MA}\|} = \cos\theta$
- On vient de voir que $\frac{\overrightarrow{MA}.\vec{n}}{\|\overrightarrow{MA}\|\|\vec{n}\|} = \cos\theta$

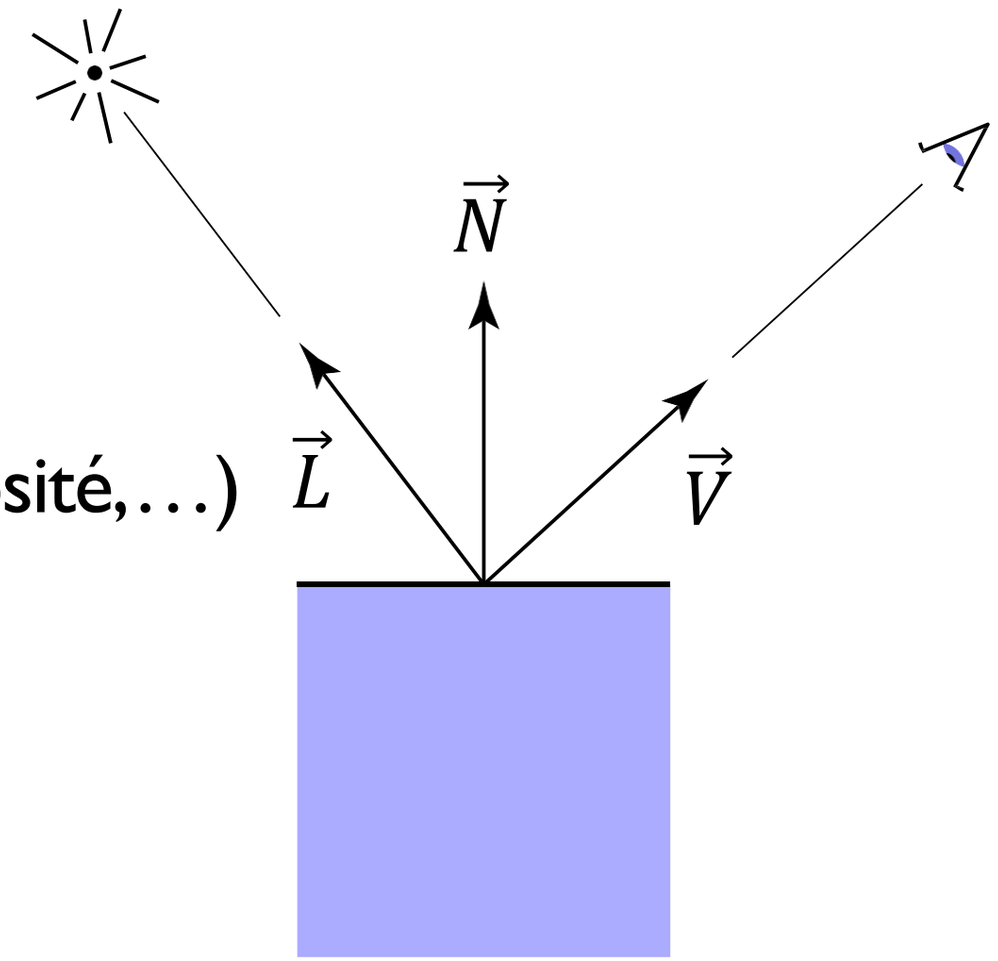
D’où $\|\overrightarrow{MM'}\| = \frac{\overrightarrow{MA}.\vec{n}}{\|\vec{n}\|}$

Finalement $M' = \begin{bmatrix} M_x - n_x\|\overrightarrow{MM'}\| \\ M_y - n_y\|\overrightarrow{MM'}\| \\ M_z - n_z\|\overrightarrow{MM'}\| \end{bmatrix}$



Shading

- **Calculer la lumière réfléchie vers la caméra**
- **Entrée :**
  - Direction de vue
  - Direction de lumière
    (pour chacune des lumières)
  - Normal à la surface
  - Paramètre de la surface (couleur, rugosité, …)



Lumières

- Réalisme dû à la perception par le système visuel humain de l'interaction entre la lumière avec les objets.

- Pas de couleurs ou de rendu 3D sans lumière.

- La manière dont l'objet « réfléchit» la lumière, fait que l'œil et le cerveau « reconstruisent la 3D ».

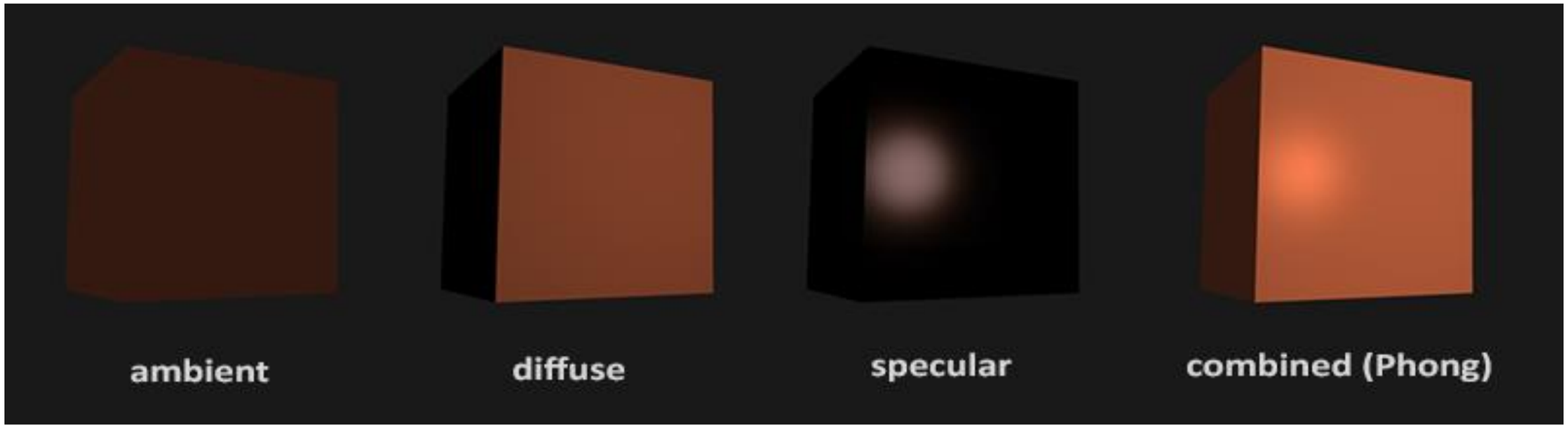
Lumières

- Les matériaux (propriétés physiques) changent les interactions avec la lumière

→ l'aspect visuel de l'objet

Illumination locale

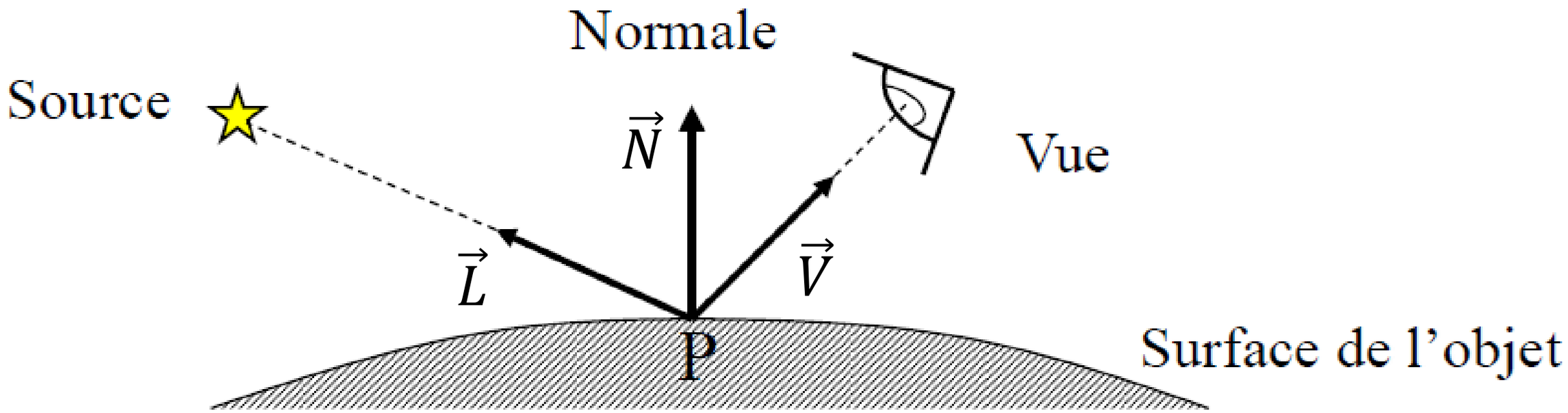
- Théorie : les objets sont vus parce qu'ils réfléchissent la lumière
  - réflexion ambiante
  - réflexion diffuse
  - réflexion spéculaire



En OpenGL : Phong = ambiante + diffuse + speculaire

Ingrédients géométriques

- Pour chaque point P de la surface :
  - Vecteur normal $\vec{N}$
  - Vecteur de direction de vue (camera) $\vec{V}$
  - Vecteur de direction de la source lumineuse $\vec{L}$



Réflexion ambiante

- Parasites provenant d'autre chose que la source considérée
  - lumière réfléchie par d'autres points
  - supposée égale en tout point de l'espace

$$I_a = I_{sa} * K_a$$

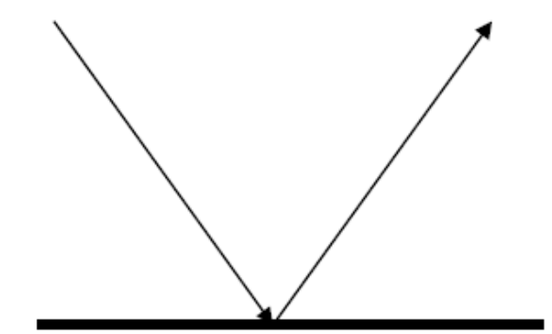
- I_a : intensité de la lumière ambiante réfléchie
- I_{sa} : intensité de la lumière ambiante
- $K_a \in [0,1]$: coeff. de réflexion ambiante de l'objet

Réflexion ambiante

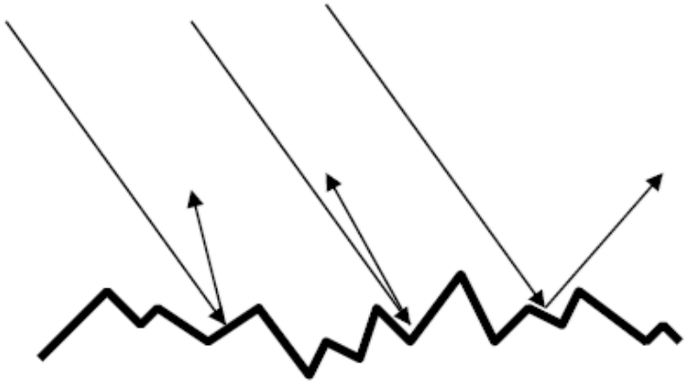
- Couleur ambiante d'un objet ne dépend que du coefficient de réflexion ambiante Ka de l'objet, pas de sa position par rapport à la lumière

Réflexion **diffuse** et **spéculaire**

- La lumière n'est pas réfléchie dans une direction unique mais dans un **ensemble de directions** dépendant des propriétés microscopiques de la surface



Miroir parfait théorique



Surface imparfaite réelle

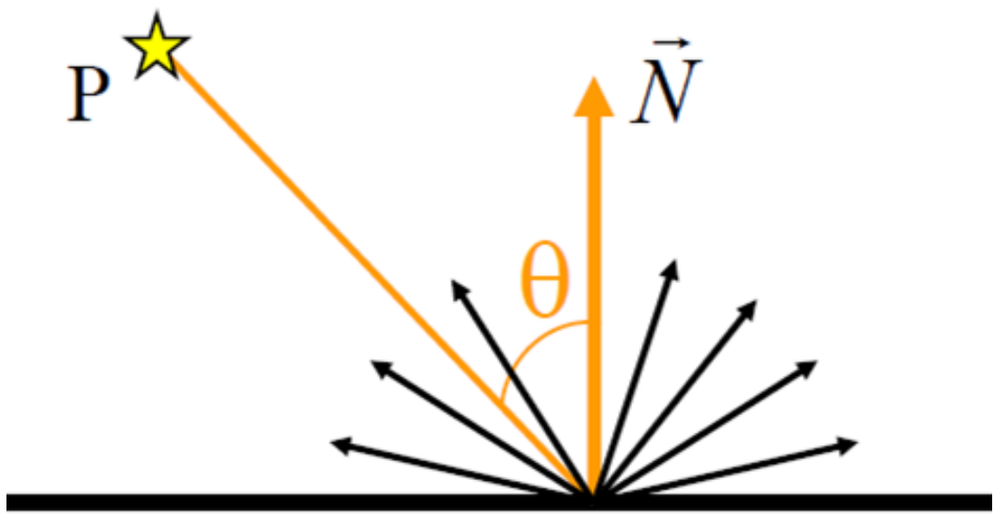
- Directions réparties selon une composante **diffuse** et une composante **spéculaire**, ajoutées à la composante ambiante pour donner plus de relief à l'objet.

Réflexion diffuse

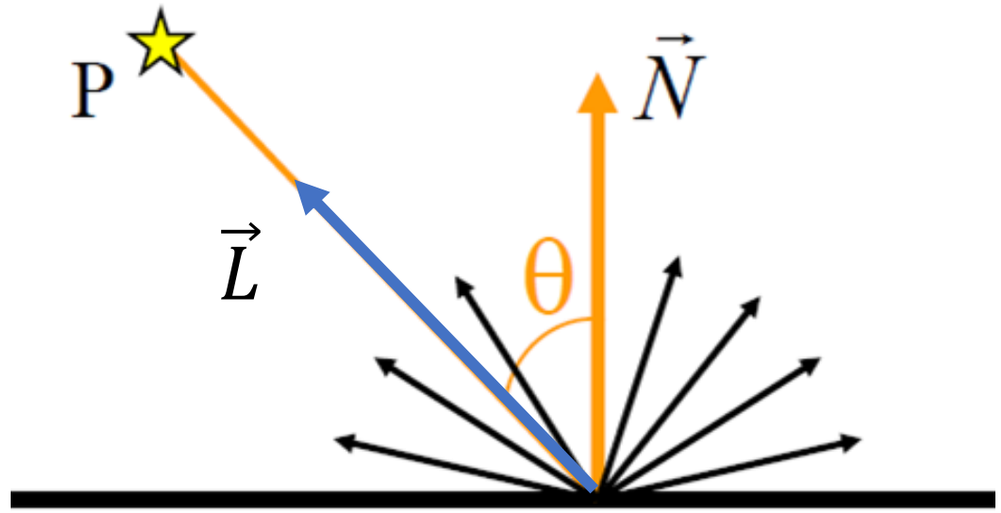
- Lumière réfléchie dans toutes les directions

→ indépendante de la position de l'observateur

- La couleur de l'objet dépend :
  - de l'angle θ entre la direction de la source et la normale
  - du coefficient de réflexion diffuse K_d de l'objet



Réflexion diffuse



- Loi de Lambert

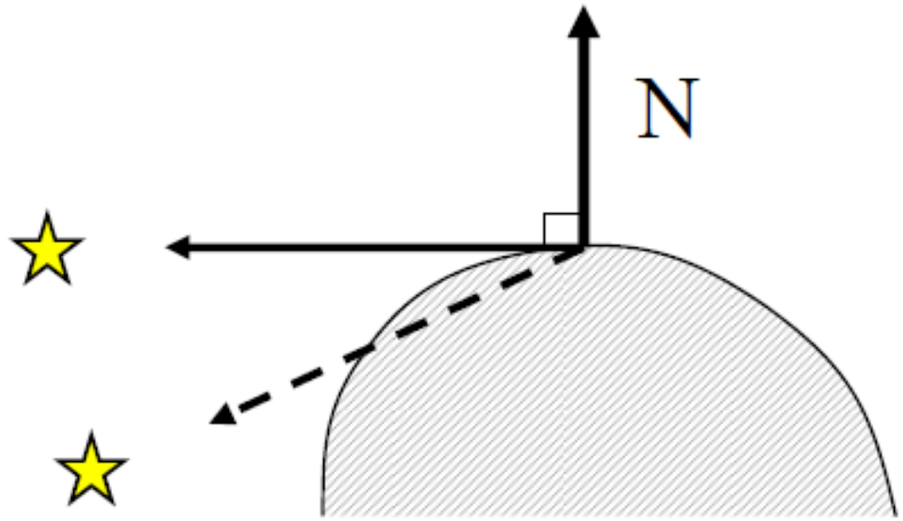
$$I_d = I_{sd} * K_d * \cos\theta$$

- I_d : intensité de la lumière diffuse réfléchie
- I_{sd} : intensité de la lumière diffuse
- $K_d \in [0,1]$: coeff. de réflexion diffuse du matériau
- θ : angle entre la source de lumière et la normale

- On peut également écrire :

$$I_d = I_{sd} * K_d * (\vec{L}.\vec{N})$$

Réflexion diffuse



- Loi de Lambert

$$I_d = I_{sd} * K_d * \cos\theta$$

- Maximale pour $\theta = 0°$ (source de lumière à la verticale de la surface, au zénith)
- Nulle pour un éclairage rasant $\theta = 90°$
- Si $\theta = 90°$ alors le point n'est pas visible par la source de lumière

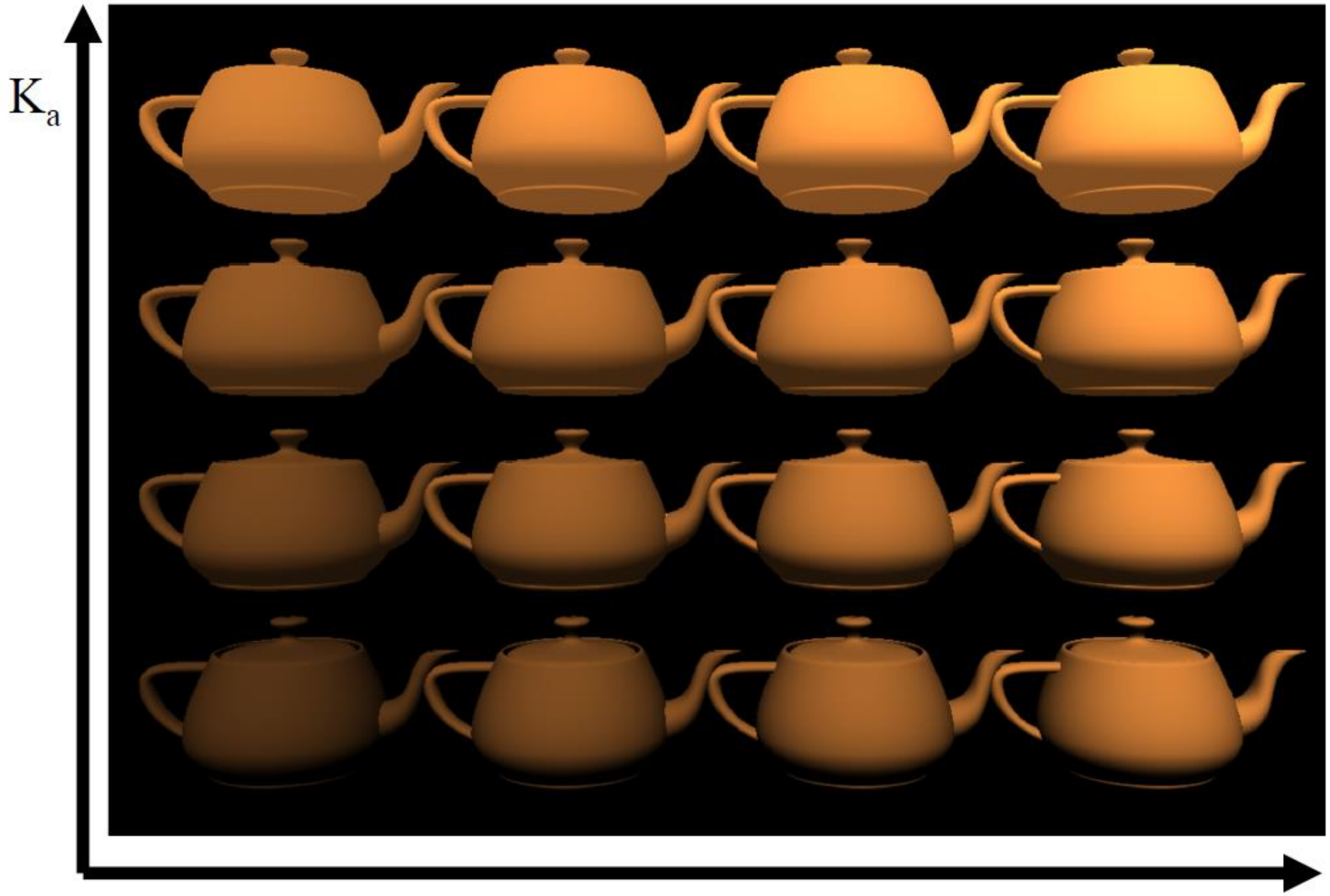
Réflexion diffuse

- Seule

On augmente K_d (avec $K_a = 0$)

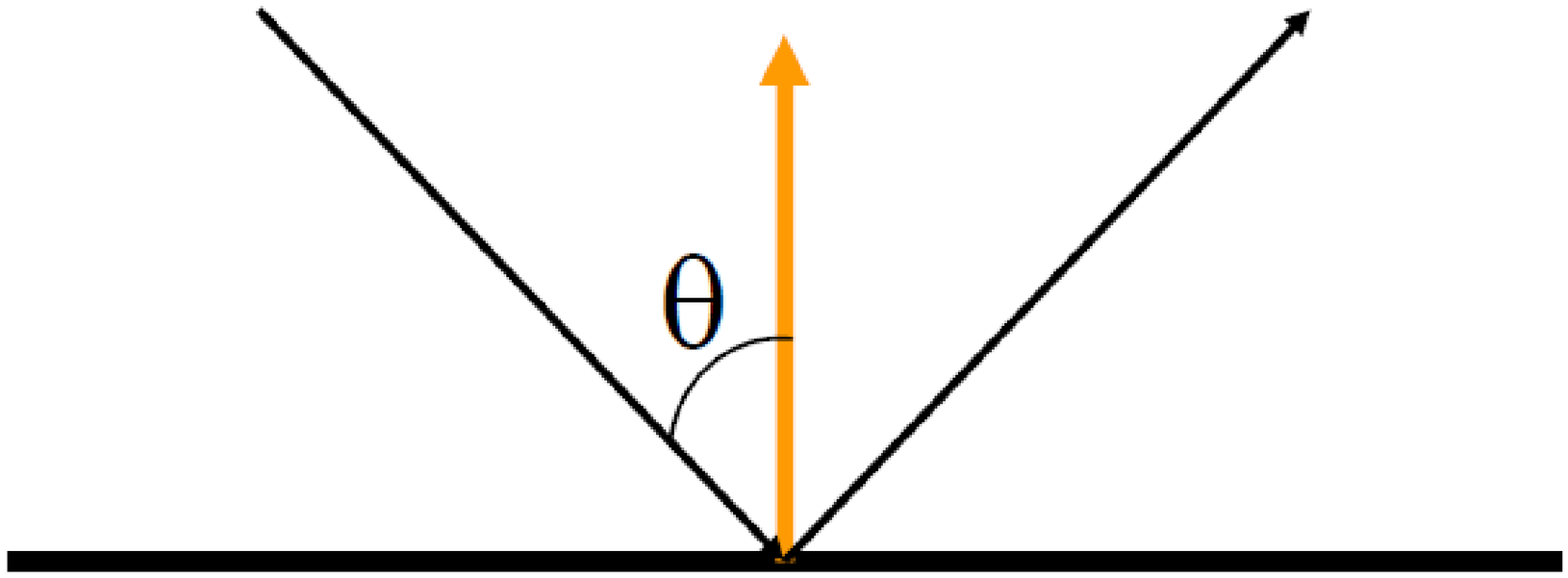
Réflexion diffuse

- Diffuse + ambiante



Réflexion spéculaire

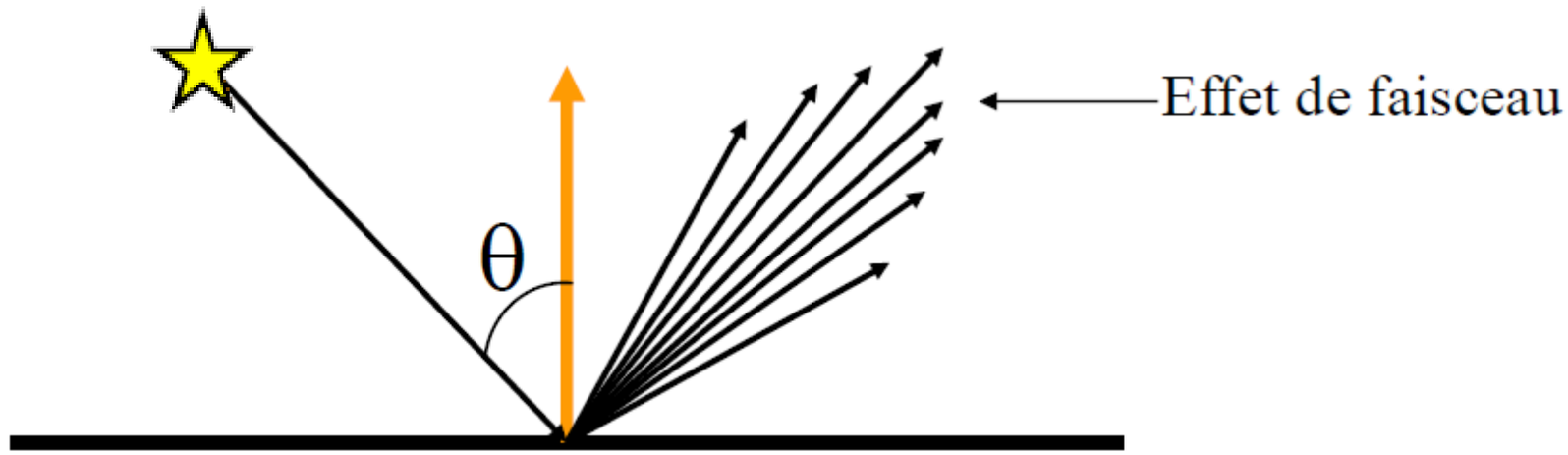
- Permet d'obtenir des reflets
- Miroir parfait → Loi de Descartes
- La lumière qui atteint un objet est réfléchie dans la direction faisant le même angle avec la normale



Réflexion spéculaire

En réalité, les surfaces ne sont jamais des miroirs parfaits

- réflexion spéculaire : miroir imparfait
- la lumière est réfléchie principalement dans la direction de réflexion miroir parfaite
- l'intensité de la lumière réfléchie diminue lorsqu'on s'éloigne de cette direction parfaite.

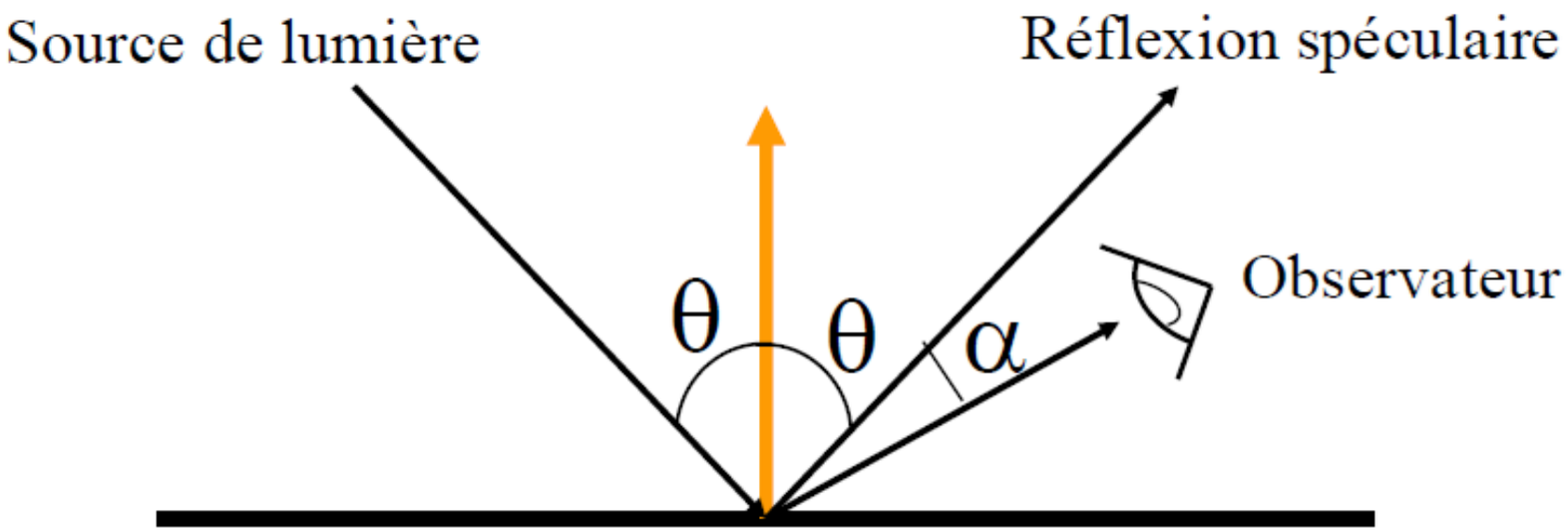


Réflexion spéculaire

- Modèle de Phong

$$I_s = I_{ss} * K_s * \cos \alpha^b$$

- I_s : intensité de la lumière spéculaire réfléchie
- I_{ss} : intensité de la lumière spéculaire de la source
- $K_s \in [0,1]$: coeff. de réflexion spéculaire du matériau
- α : angle entre les directions de réflexion et de la vue
- b : brillance ou shininess



Réflexion spéculaire

- On peut également écrire

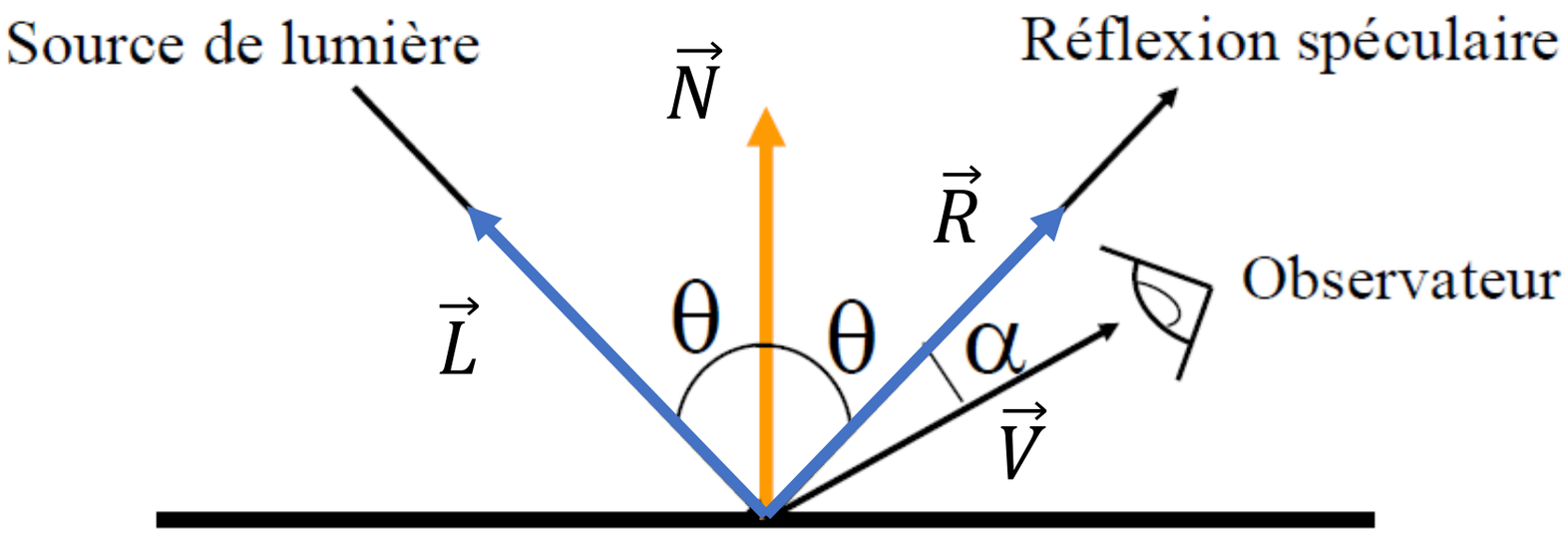
$$I_s = I_{ss} * K_s * \left(\vec{R}.\vec{V}\right)^n$$

avec

$\vec{R}$: direction de réflexion de la lumière sur un miroir

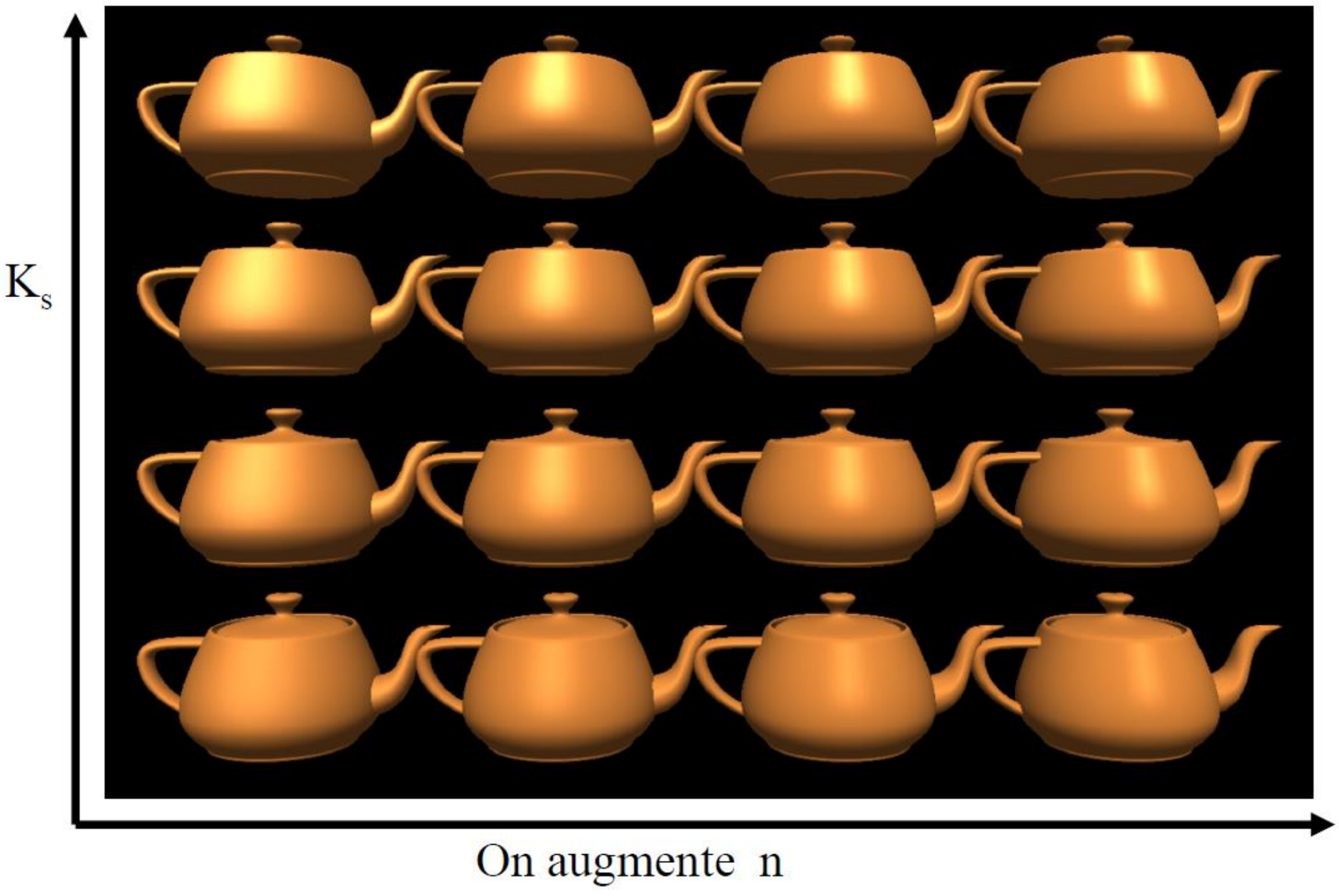
et

$$\vec{R} = 2\left(\vec{N}.\vec{L}\right)\vec{N} - \vec{L} = 2\cos\theta\,\vec{N} - \vec{L}$$



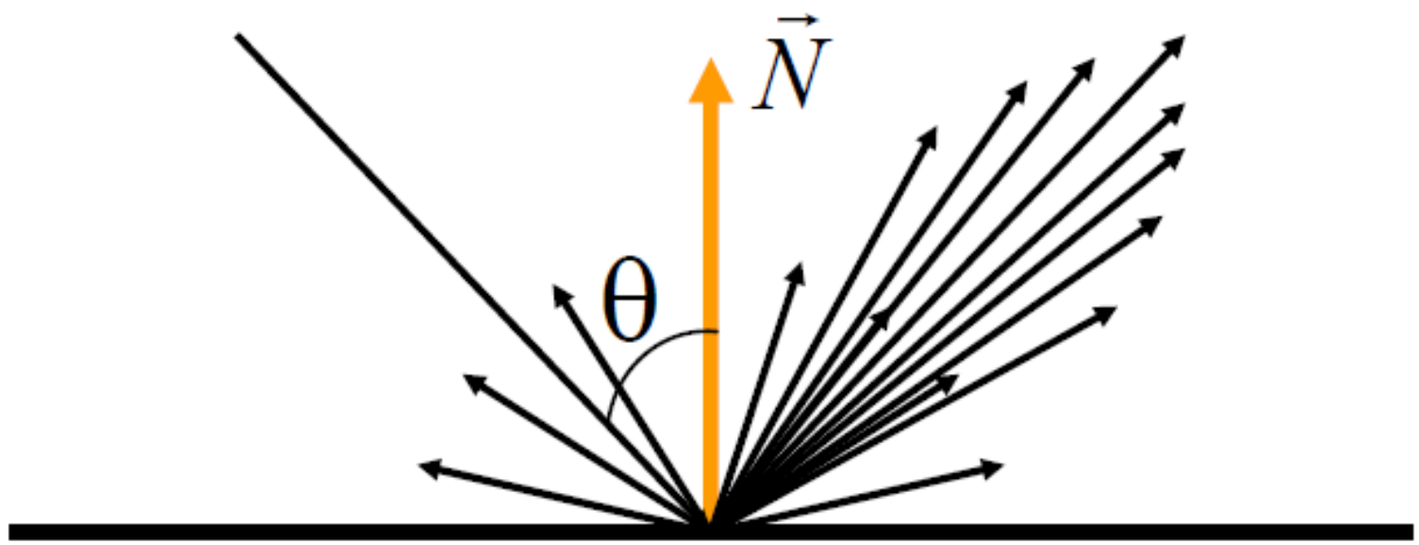
Réflexion spéculaire

$$I_s = I_{ss} * K_s * \cos \alpha^n$$

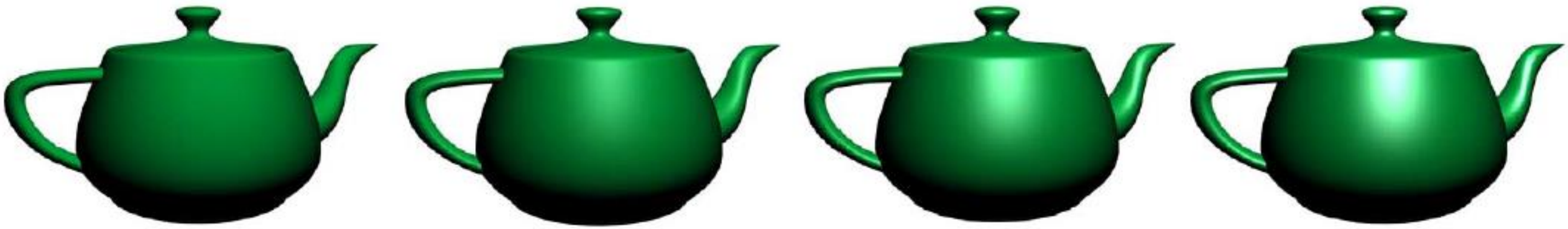


Modèle de Phong

- Dans la réalité, lumière réfléchie par une surface
  - réflexion diffuse + réflexion spéculaire
- Proportions de réflexion diffuse et spéculaire dépendent du matériau :
  - plus diffus (craie, papier …)
  - que spéculaires (métal, verre …)



Réflexion finale



Diffus

(« mat »)

Spéculaire

(« brillant »)

Equation de Phong

- Egale à la somme des réflexions ambiante, diffuse et spéculaire :

$$I = I_a + I_d + I_s$$

- Plusieurs sources lumineuses : somme des intensités

$$I = I_{sa}K_a + \sum_{l \in \{sources\}} I_{ld}K_d(\overrightarrow{L_l}.\vec{N}) + I_{ls}K_s(\overrightarrow{R_l}.\vec{V})^n$$

Calcul de la couleur

- Addition l'intensité lumineuse de chacune des composantes de la couleur.
- RVB : intensités rouge, verte et bleue

→On définit pour chacune de ces 3 composantes

- les caractéristiques des sources de lumière

$(I_{saR}, I_{saG}, I_{saB})$; $(I_{sdR}, I_{sdG}, I_{sdB})$; $(I_{ssR}, I_{ssG}, I_{ssB})$

- les caractéristiques des matériaux

(K_{aR}, K_{aG}, K_{aB}); (K_{dR}, K_{dG}, K_{dB}); (K_{sR}, K_{sG}, K_{sB})

Calcul de la couleur

- Les intensités lumineuses pour chacune des 3 composantes R, V, B s'obtiennent donc ainsi :

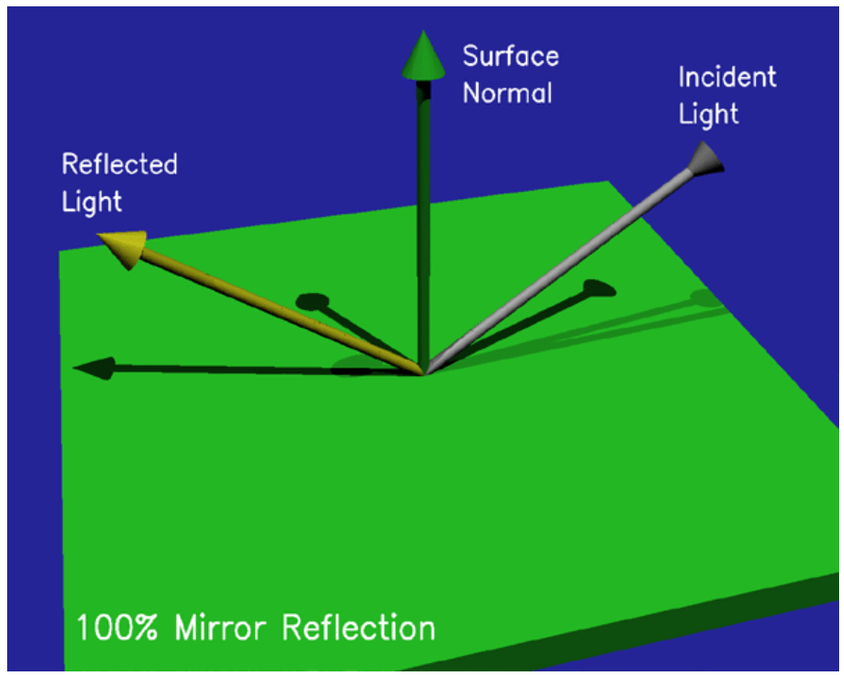
$$I_R = I_{saR}K_{aR} + I_{sdR}K_{dR}\cos\theta + I_{ssR}K_{sR}\cos\alpha^n$$
$$I_G = I_{saG}K_{aG} + I_{sdG}K_{dG}\cos\theta + I_{ssG}K_{sG}\cos\alpha^n$$
$$I_B = I_{saB}K_{aB} + I_{sdB}K_{dB}\cos\theta + I_{ssB}K_{sB}\cos\alpha^n$$

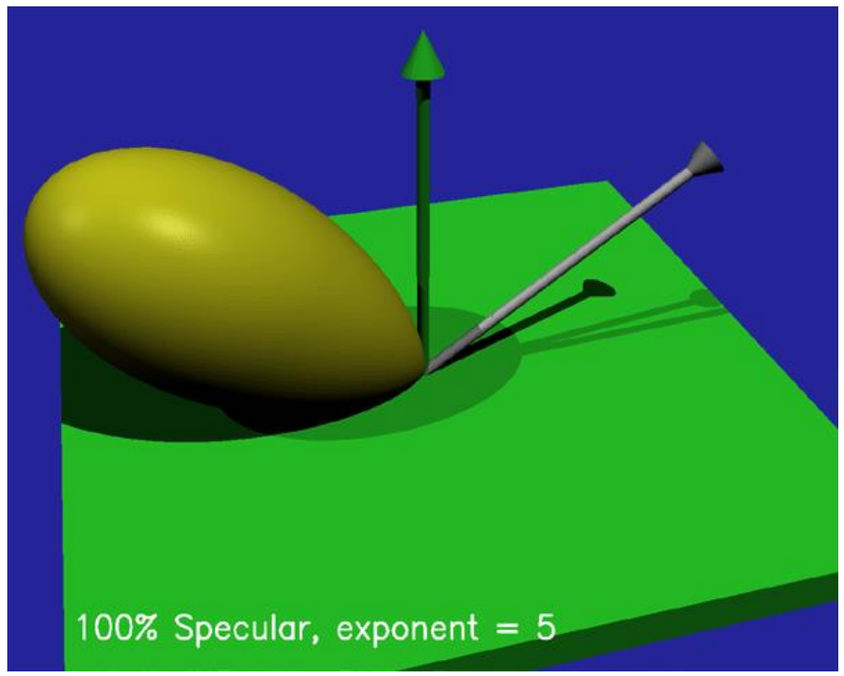
Ou

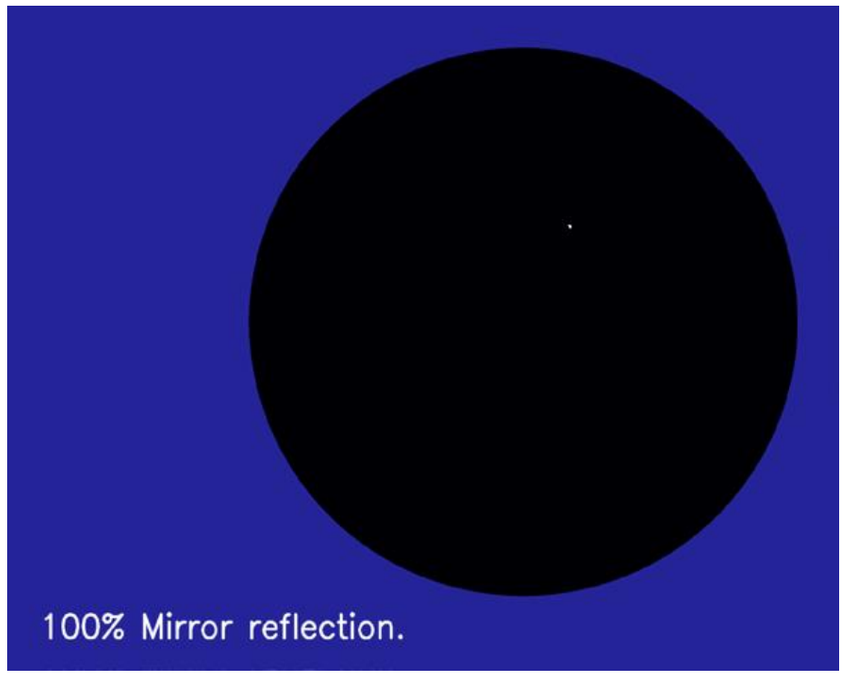
$$I_R = I_{saR}K_{aR} + I_{ldR}K_{dR}(\overrightarrow{L_l}.\vec{N}) + I_{lsR}K_{sR}(\overrightarrow{R_l}.\vec{V})^n$$
$$I_G = I_{saG}K_{aG} + I_{ldG}K_{dG}(\overrightarrow{L_l}.\vec{N}) + I_{lsG}K_{sG}(\overrightarrow{R_l}.\vec{V})^n$$
$$I_B = I_{saB}K_{aB} + I_{ldB}K_{dB}(\overrightarrow{L_l}.\vec{N}) + I_{lsB}K_{sB}(\overrightarrow{R_l}.\vec{V})^n$$

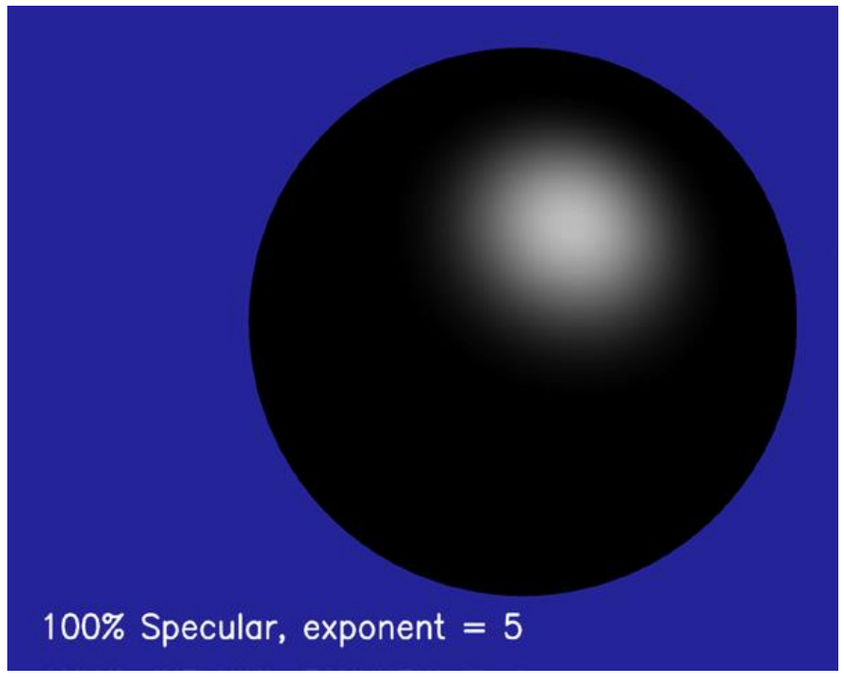
Illumination locale

Résultats



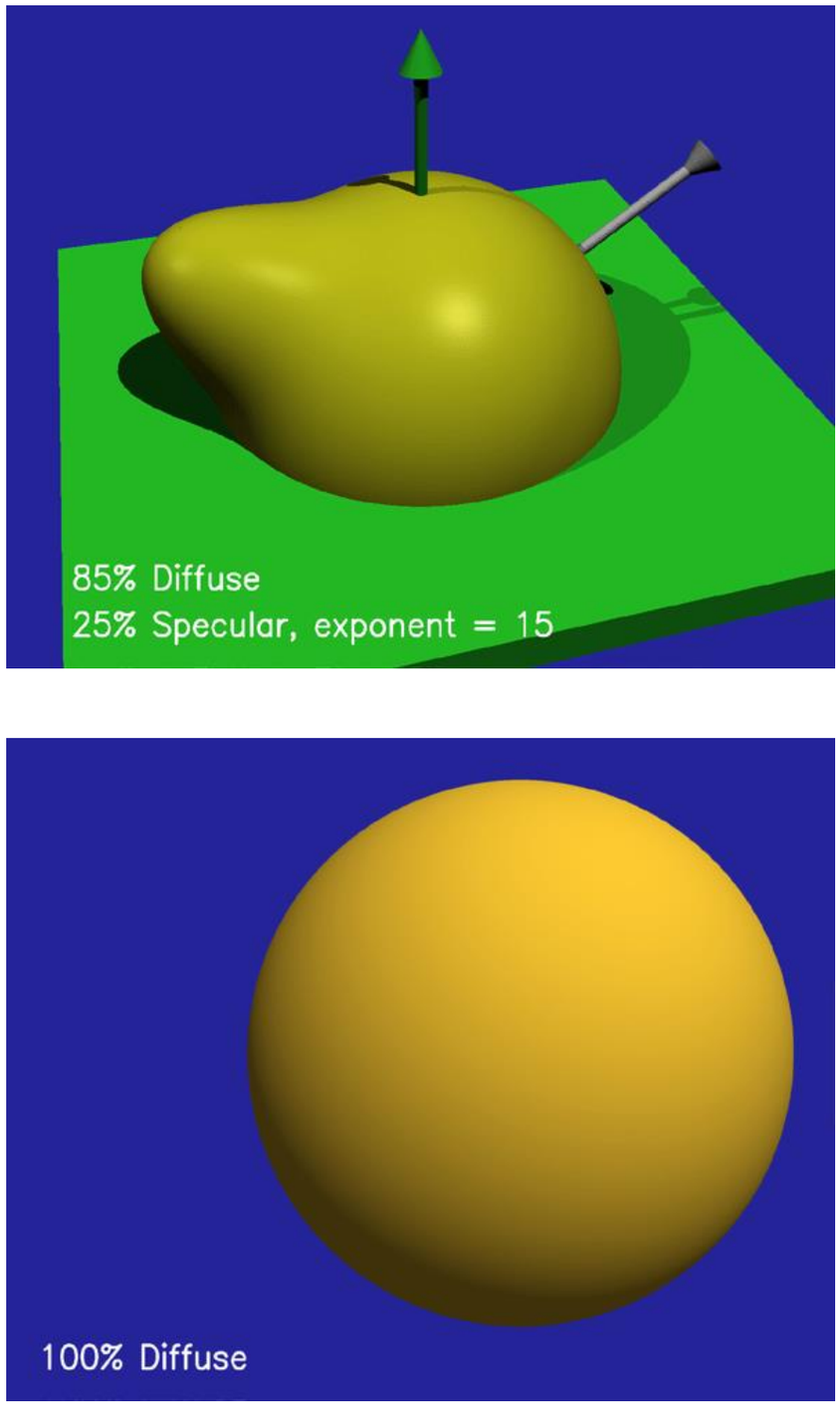






Illumination locale

Résultats



Modèle de Phong

- Avantages
  - très pratique (simple à utiliser, résultats intéressants)
  - rapide à calculer
- Désavantages
  - pas de sens physique
  - pas de lien avec les propriétés du matériau (rugosité …)

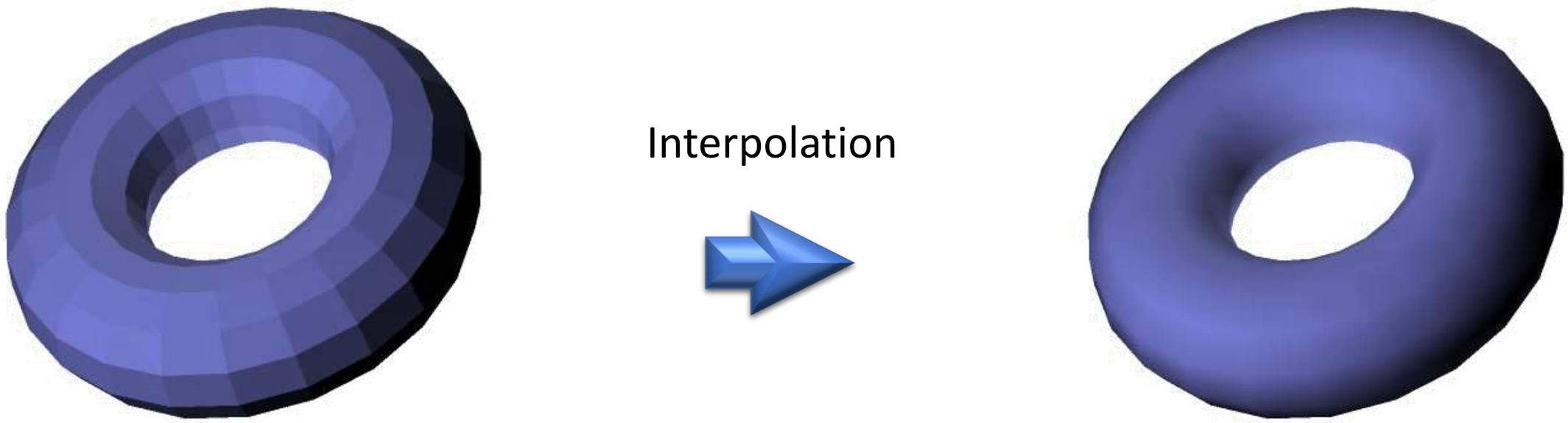
Autres modèles d'éclairages

- Plus efficaces ou réalistes
  - Cook-Torrance (1982)
  - Oren-nayar (1994)
  - Minnaert
  - Subsurface scattering (SSS)
  - BRDF
  - PBR modèle
  - Illumination globale

- Pipeline graphique programmable

Illumination d'un objet 3D

- Même illumination pour tout un polygone (élément du maillage) :
  - affichage « plat » (flat shading)



- **Solution** : calculer l'illumination pour chaque sommet des polygones, puis **interpoler** l'illumination d'un sommet à un autre.

Illumination d'un objet 3D

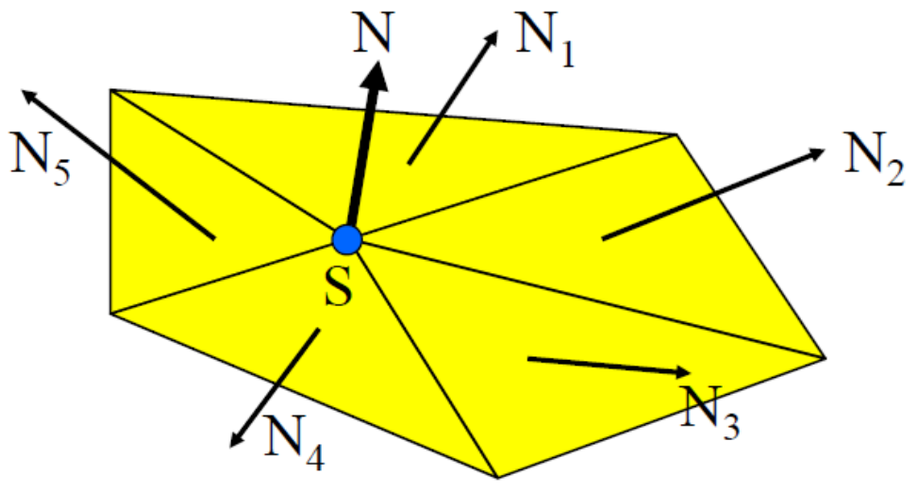
- But : calculer une couleur pour chaque point visible à l'écran de l'objet 3D qu'on affiche.
  - Lissage de **Gouraud** : interpolation de **couleurs**
  - Lissage de **Phong** : interpolation de **normales**

Interpolation de Gouraud

- Pour chaque polygone à afficher :
  - calculer pour chaque sommet du polygone une couleur au moyen d'un modèle d'illumination (Phong …)
  - interpoler les **couleurs** des sommets pour calculer la couleur de chaque pixel du polygone.

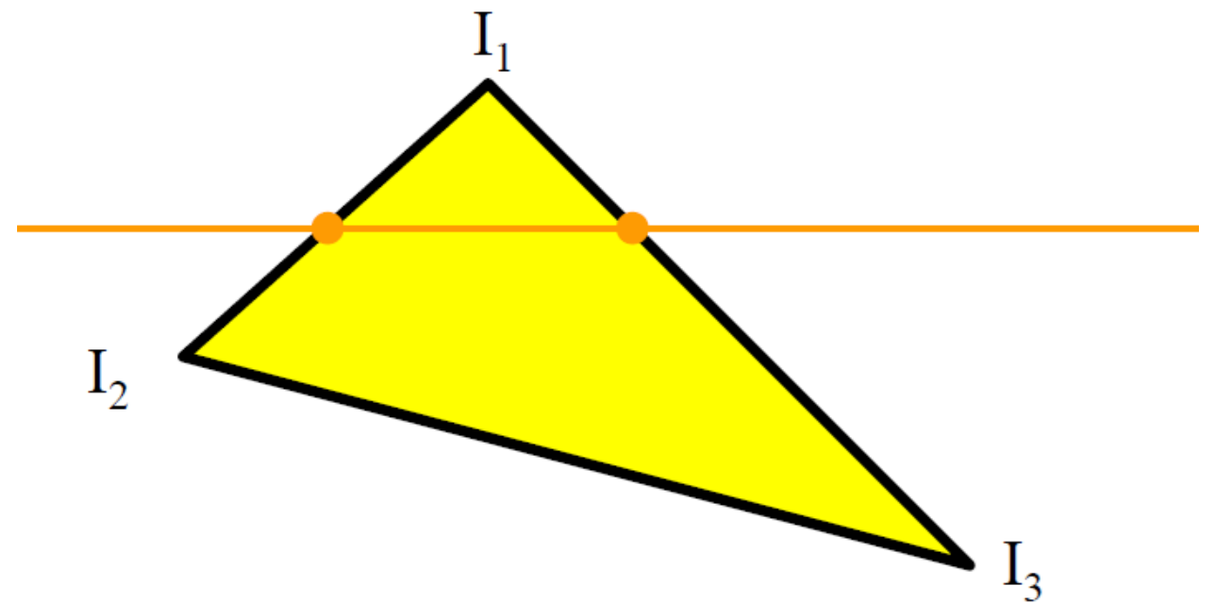
Interpolation de l'illumination

- Pour calculer la couleur en un sommet, on a besoin d'une normale en ce point :
  - surface analytiquement connue (ex : une sphère, un cylindre …) → calcul direct
  - surface de départ est un maillage polygonal, comment faire ?? → interpoler les normales de face



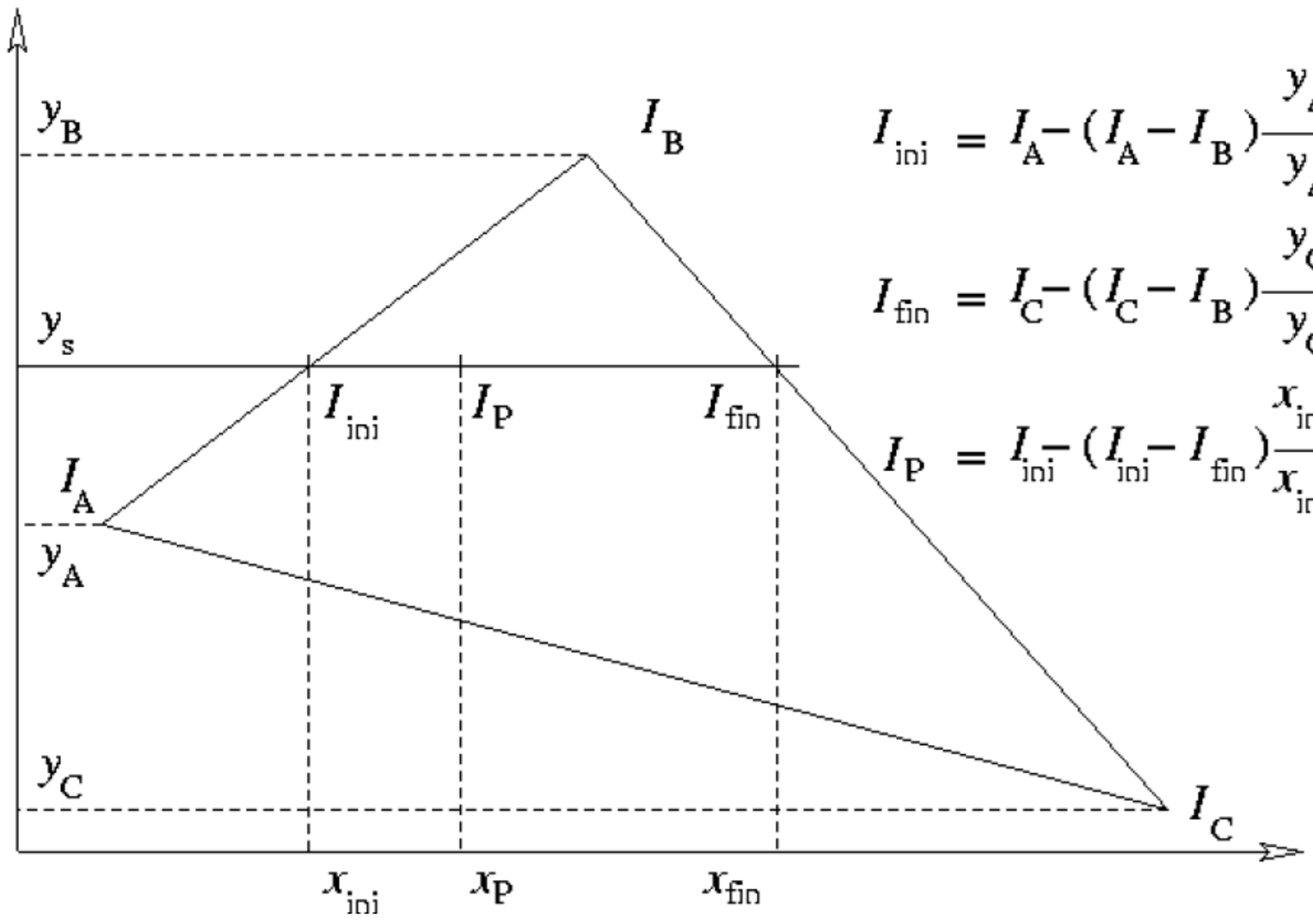
Interpolation de l'illumination

- Pour chaque sommet on calcule une couleur avec un modèle d'illumination
  - sur une arête, interpoler les couleurs entre les 2 sommets
  - sur une ligne de remplissage (scanline) du polygone, interpoler les couleurs entre 2 arêtes



Illumination d'un objet 3D

- Interpolation de l'illumination par **Gouraud** :



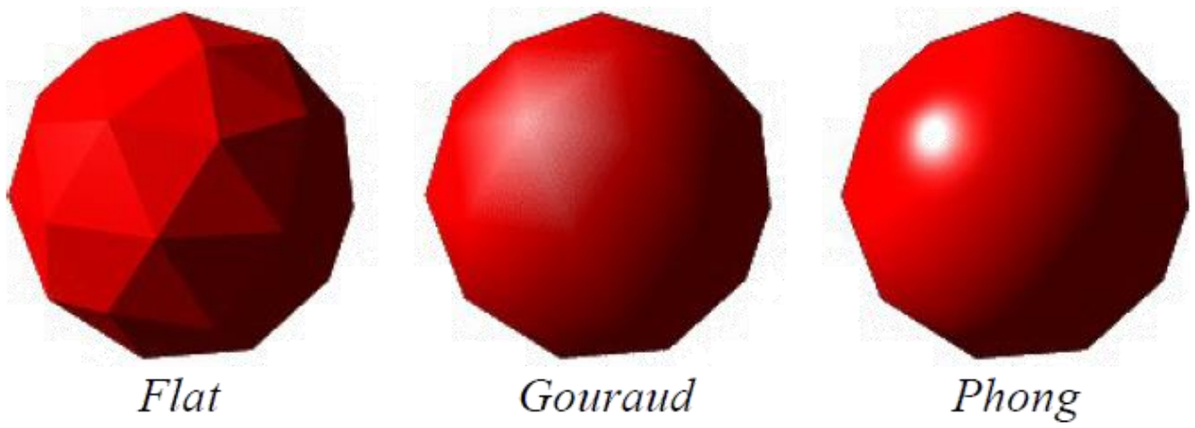
$$I_{ini} = I_A - (I_A - I_B)\frac{y_A - y_s}{y_A - y_B}$$

$$I_{fin} = I_C - (I_C - I_B)\frac{y_C - y_s}{y_C - y_B}$$

$$I_P = I_{ini} - (I_{ini} - I_{fin})\frac{x_{ini} - x_p}{x_{ini} - x_{fin}}$$

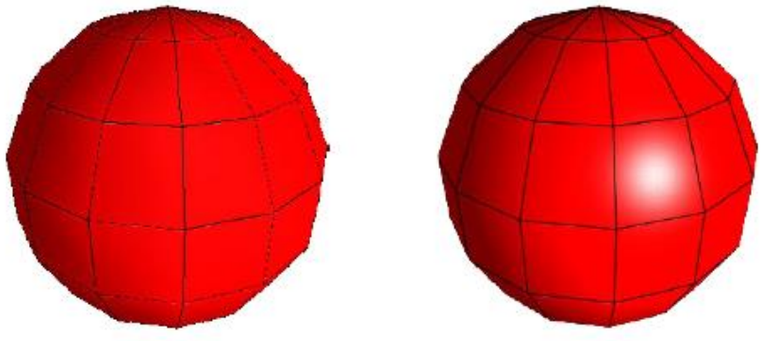
Illumination d'un objet 3D

- Lissage de Phong plus lent que celui de Gouraud (plus de calcul d'illumination) mais plus nettement plus beau :



Flat *Gouraud* *Phong*

- Permet de calculer les effets spéculaires **contenus dans une facette**, contrairement au lissage de Gouraud



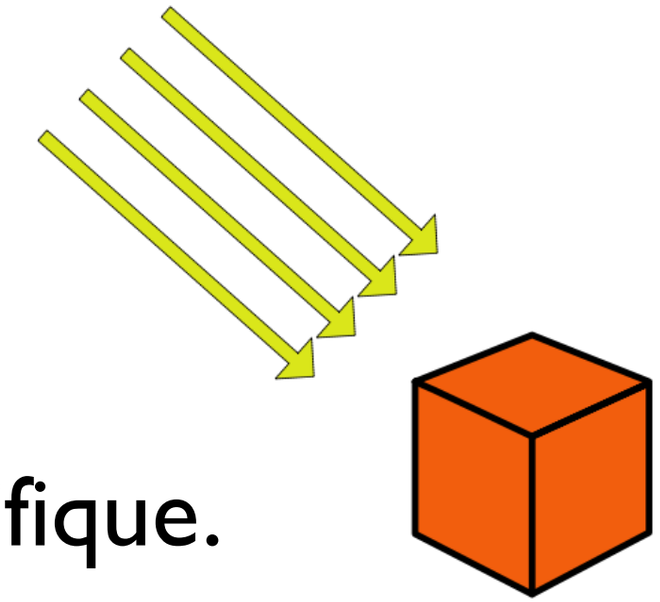
Gouraud *Phong*

Lumières

- Plusieurs types d'éclairage :
  - source directionnelle
  - source ponctuelle
  - source projecteur

Lumière directionnelle

- Définie par une direction :
  - 4 coordonnées homogènes (x, y, z, 0)



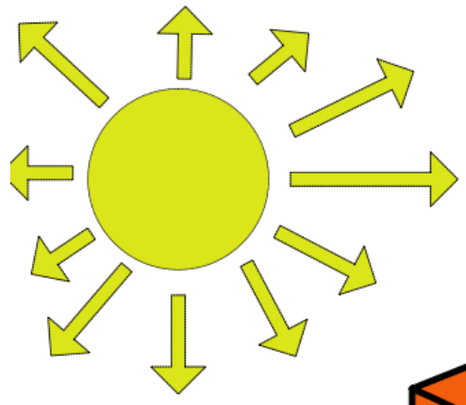
- La lumière vient d'une direction spécifique.

Coordonnées homogènes d'un vecteur

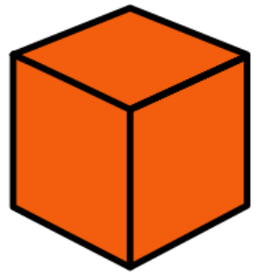
```
float position[] = {0.0, -1, 0.0, 0.0};
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Source ponctuelle

- Définie par une position :
  - 4 coordonnées homogènes (x, y, z, 1)



- La lumière vient d'un point spécifique.

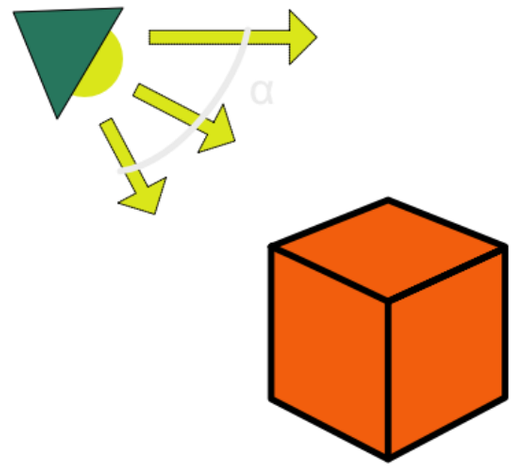


Coordonnées homogènes d'un point

```
float position[] = {0.0, -1, 0.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Lumière projecteur = « spot »

- La lumière vient d'un point spécifique,
  - intensité dépendant de la direction

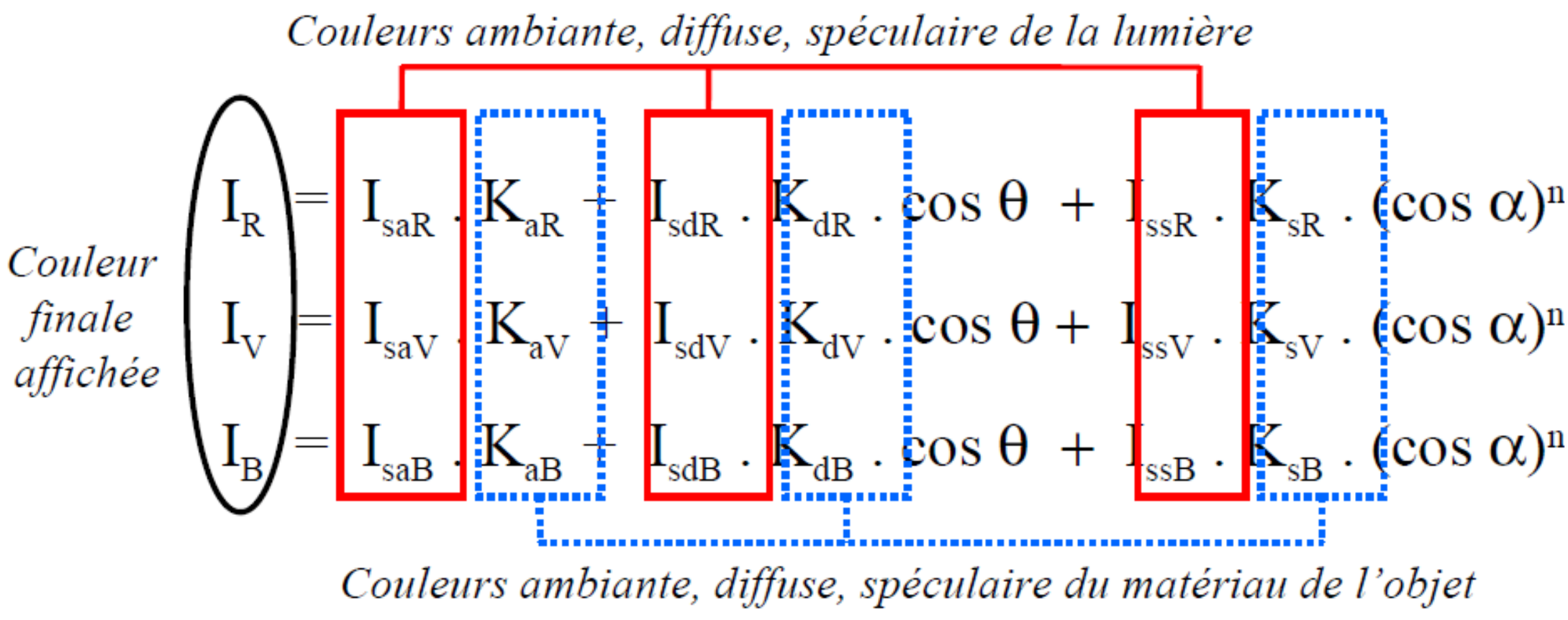


- Position : emplacement de la source
- Direction : axe central de la lumière
- Angle : largeur du rayon

```
float position[] = {0.0, 10.0, 0.0, 1.0};
float direction[] = {1.0, -1.0, 0.5};
float angle = 45.0f;
glLightfv(GL_LIGHT0, GL_POSITION, position);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, direction);
glLightf(GL_LIGHT0, GL_CUTOFF, angle);
```

Bilan

Etant donné les couleurs ambiante, diffuse, spéculaire de la lumière, les composantes du matériau d'un objet, la couleur finale sera calculée grâce à l'équation du **modèle de Phong** :

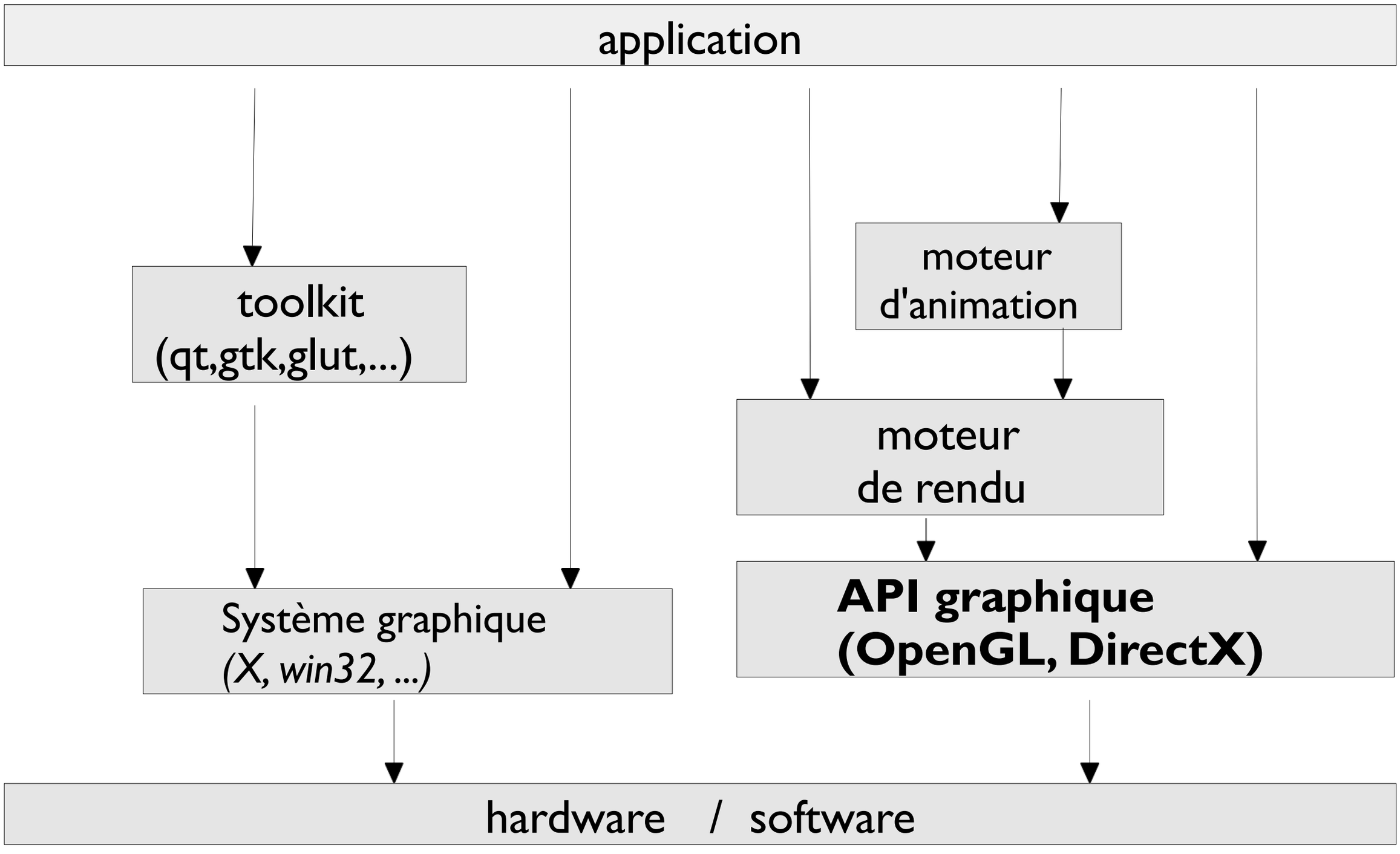


Introduction au pipeline OpenGL

App 3D OpenGL

- Manipule un ensemble de polygones, structuré par l'application
  - De la simple liste
  - Au graphe de scène complet, avec description sémantique
- Textures : images couleurs plaquées sur les polygones
- Rendu temps réel de la scène sous forme d'images couleurs affichées à l'écran
  - Boucle de rendu
- Interaction : évènements utilisateurs (e.g., clavier, souris, *touch screen*)
  - Callbacks

Architecture logicielle



Boucle de rendu

- Rendu temps-réel : en général effectué par le GPU
  - Effectue des appels à une API graphique
  - API dédiés OpenGL, DirectX, Metal, Optix, etc
- Données
  - Maillage polygonal échantillonnant la scène
  - Propriétés de surface : normal, coordonnées de textures,
  - Textures
  - Matériaux
  - Sources de lumières
  - Paramètres caméra

GPU : Données en entrée

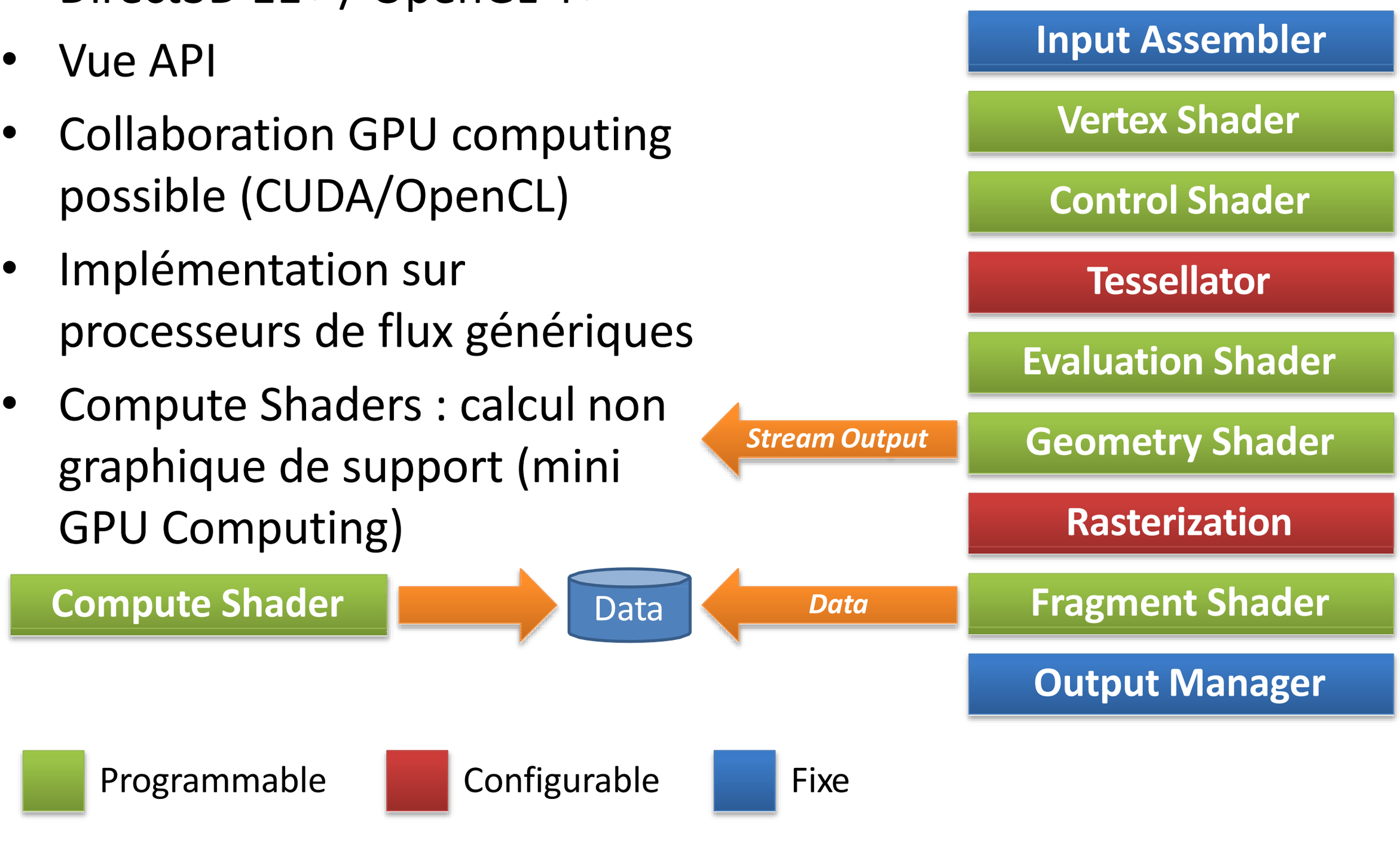
Maillage polygonal : approximation de la surface d'un objet à l'aide d'un ensemble de polygones

- **Soupe de Polygones** : suites de n-uplets de coordonnées 3D correspondants aux polygones
- **Maillages indexés** : graphe avec géométrie et topologie séparés
  - Une liste de sommets (V)
  - Une liste de relation topologique:
    - Arêtes (Edge, E)
    - Faces (F)

En pratique, {V,F} (exemple : OpenGL)

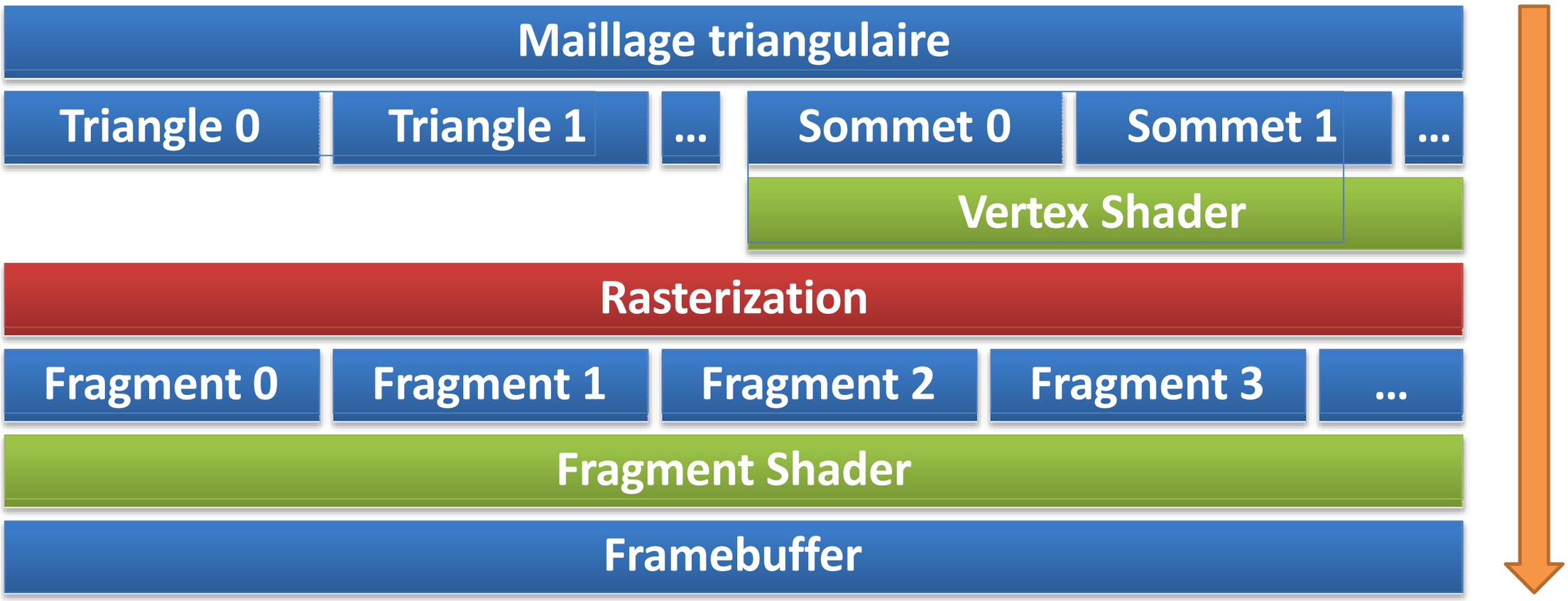
Pipeline Graphique Moderne

- Direct3D 11+ / OpenGL 4+
- Vue API
- Collaboration GPU computing possible (CUDA/OpenCL)
- Implémentation sur processeurs de flux génériques
- Compute Shaders : calcul non graphique de support (mini GPU Computing)



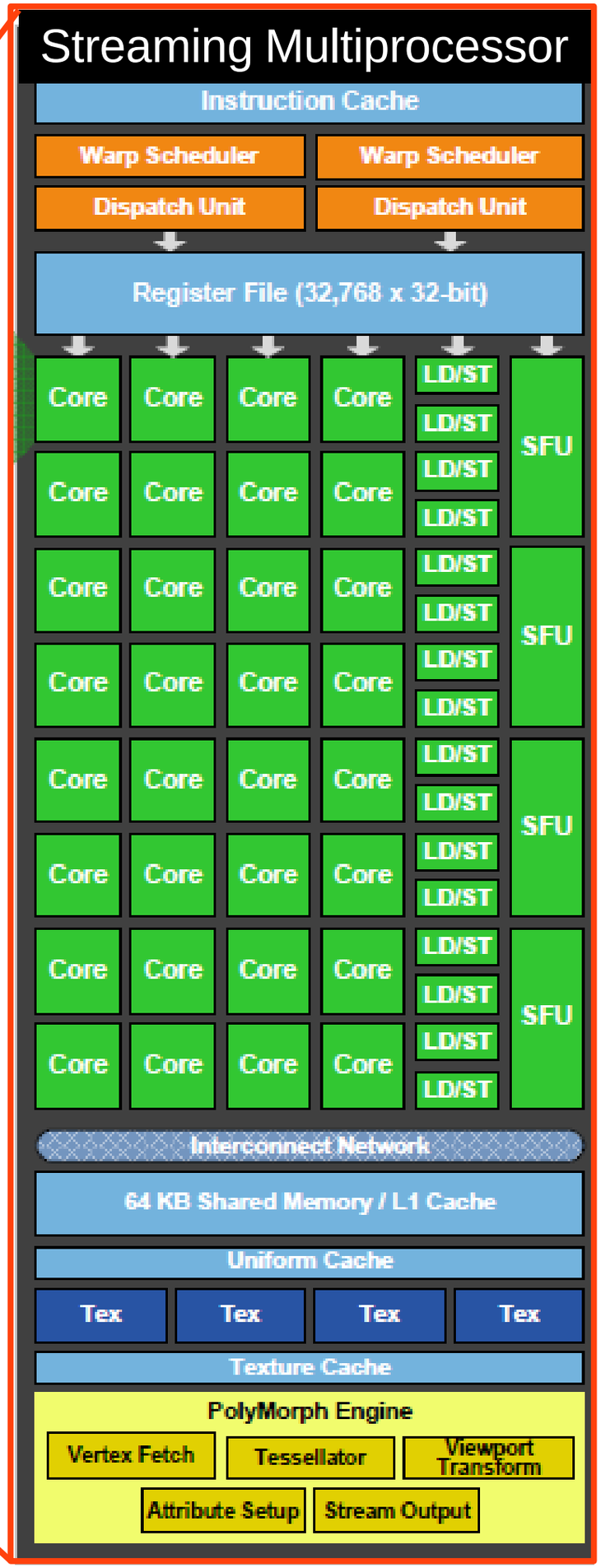
Etages majeurs

- Calcul en flux



- Intégralement parallèle
  - Chaque sommet traité indépendamment
  - Chaque fragment traité indépendamment

Architecture d'un GPU



GPU

- **GPU = Processeur Graphique**
  - permet des calculs complexes réalisés par la carte graphique
    - le CPU est libre pour réaliser d'autre tâches
- **Ce n'est pas un CPU !**
  - Hautement parallèle : jusqu'à 4000 opérations en parallèle !
    - architecture hybride (SIMD/MIMD)
  - Accès mémoire via des buffers spécialisés (*de – en - vrai*):
    - Textures (images, tableaux en lecture, 1D, 2D, 3D)
    - Vertex Buffers (tableaux de sommets, 1D)
    - Frame buffers (images en écriture, 2D)
  - Circuits spécialisés non programmable
    - rasterisation, blending, etc.

- **Accès via une API graphique**
  - **OpenGL**, DirectX, Vulkan, etc.

API OpenGL

- API grahique générique
  - Mac/PC/Linux/iOS/Android/html5/etc
- Plusieurs versions
  - OpenGL Classique (v1.2)
    - Pas de programmation GPU
    - Pipeline fixe
  - OpenGL Programmable (v2.0)
    - Programmation GPU (shaders)
    - Entrées-sorties formatées
  - **OpenGL Moderne (v3/v4)**
    - Shaders graphiques et shaders calcul,
    - Entrées sorties redéfinissables.
    - Nouveaux étages : geometry, tessellation
- Les bibliothèques GLAD et GLEW permettent de travailler avec les versions modernes d'OpenGL

Interfaçage au système d'exploitation

- Plusieurs bibliothèques existent:
  - **GLFW** (fortement recommendé)
  - FreeGlut
  - Qt
- Fournissent en général un mécanisme de **callback** pour
  - la mise à jour de l'image affichée
  - les évènements claviers
  - les évènements souris

OpenGL Moderne (v4.x)

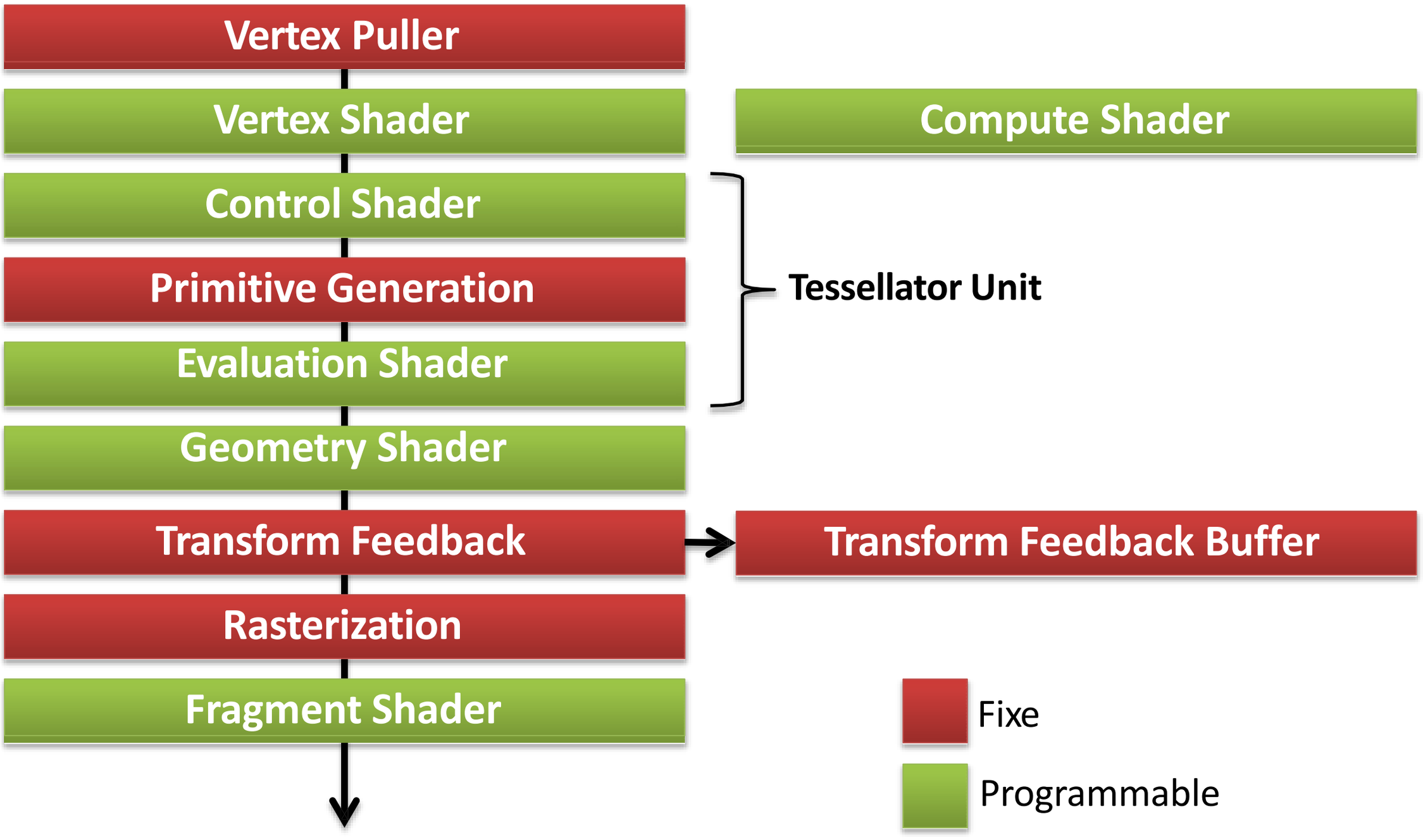
Programmes →

- Shaders
  - Vertex Shader
  - Control Shader
  - Evaluation Shader
  - Geometry Shader
  - FragmentShader
  - Compute Shader

Données →

- Buffers
  - Framebuffer
  - PixelBuffer
  - VertexBuffer
  - IndexBuffer
- Textures
  - Texture2D
  - Texture3D

Pipeline Moderne



Shaders

- **Vertex**
- **Control**
- **Evaluation**
- **Geometry**
- **Fragment**
- ***Compute (calcul non graphique)***
- ***Mesh (2018)***

<u>Pipeline Graphique = ensemble de shaders</u>

- Synchronisation I/O

Amplification geometrique

- Geometry Shader :
  - Très Flexible
  - Faible amplification
- Tessellation :
  - Peu flexible
  - Grande amplification

OpenGL

- **Bibliothèque graphique 2D/3D**

– API entre le GPU et le programme utilisateur

– Rendu d'images composées de primitives :
  - Géométriques : points, lignes, polygones ...
  - Images : bitmap, textures ....

– Bas niveau
  - **Machine à états,** contrôlé par des **commandes**
  - **Sait uniquement convertir un triangle 2D en un ensemble de pixels !**
    - – Accéléré par le matériel graphique

– Portable (Linux, Windows, MacOS, SmartPhone, WebBrowser, …)
  - langage C + interface pour tous les autres langages

(Java, Python, C#, OCaml, JavaScript, etc.)

OpenGL ?

ATTENTION aux versions !

- OpenGL 1.x, OpenGL 2.x => obsolètes !
- OpenGL 3.x avec rétro-compatibilité => à éviter !
- **OpenGL 3.x « core » → recommandé**
- OpenGL 4.x → = 3.x + nouvelles fonctionnalités
- WebGL, OpenGL-ES → proches de OpenGL 3.x « core »

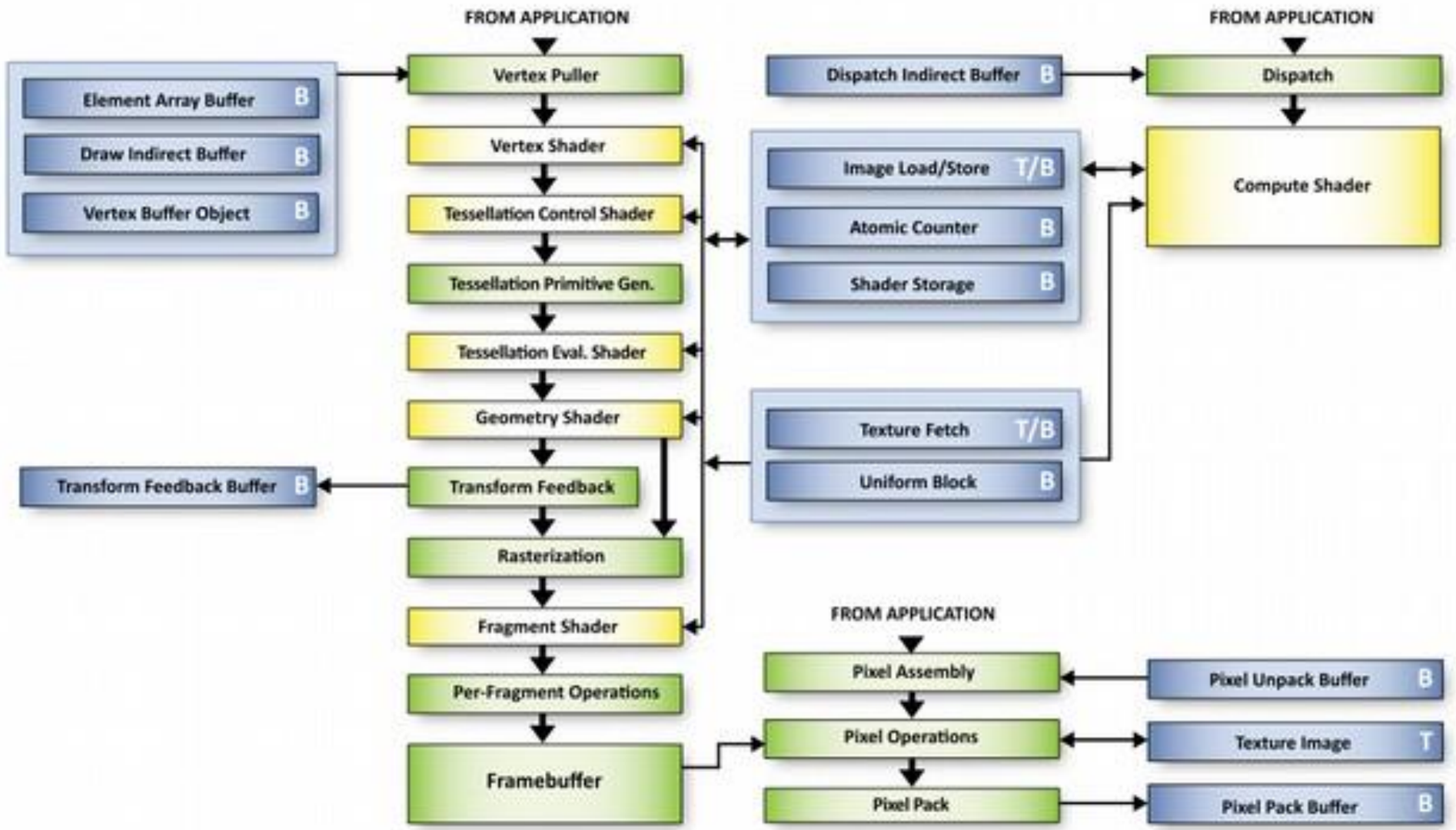
Pipeline Graphique sur GPU

OpenGL Pipeline

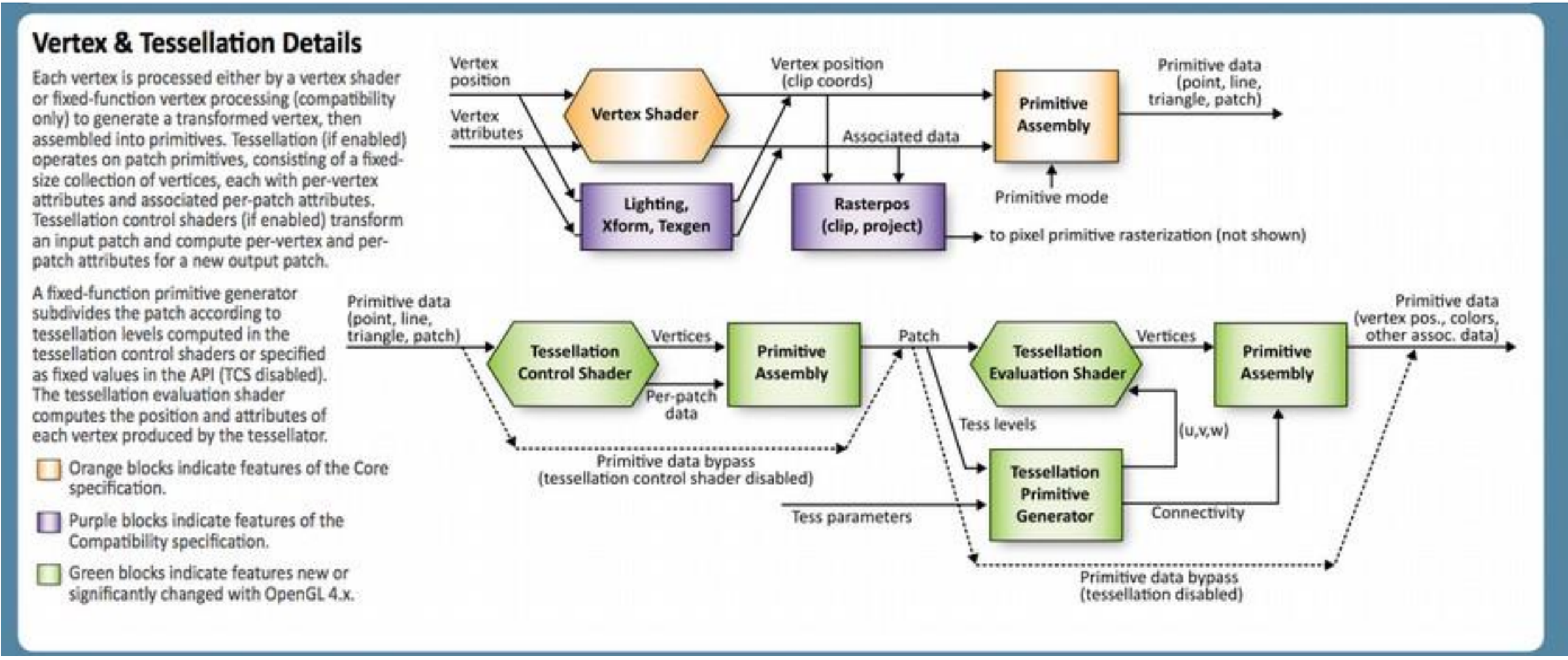
A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Calls are made to allocate a GL context which is then associated with the window, then OpenGL commands can be issued.

The heavy black arrows in this illustration show the OpenGL pipeline and indicate data flow.

- Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- Green blocks indicate fixed function stages.
- Yellow blocks indicate programmable stages.
- T Texture binding
- B Buffer binding



Pipeline Graphique sur GPU



Pipeline Graphique sur GPU

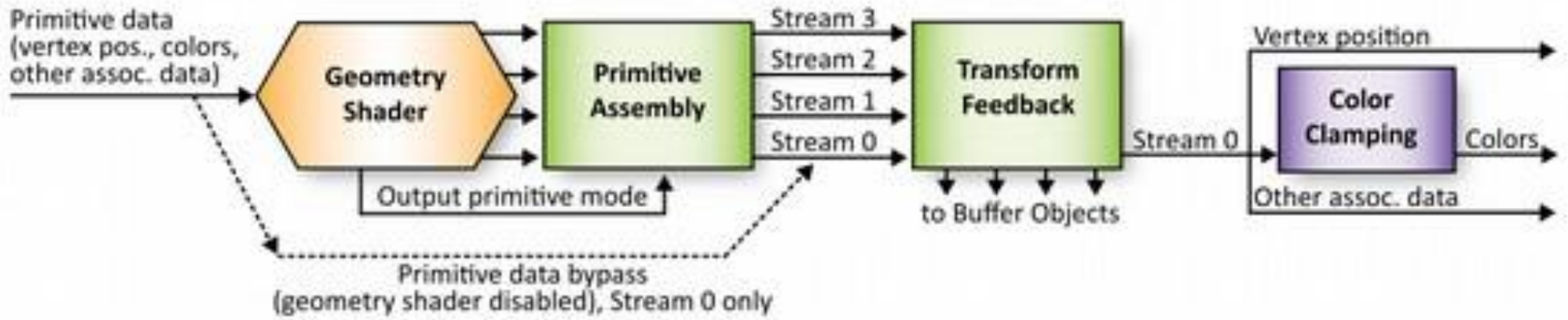
Geometry & Follow-on Details

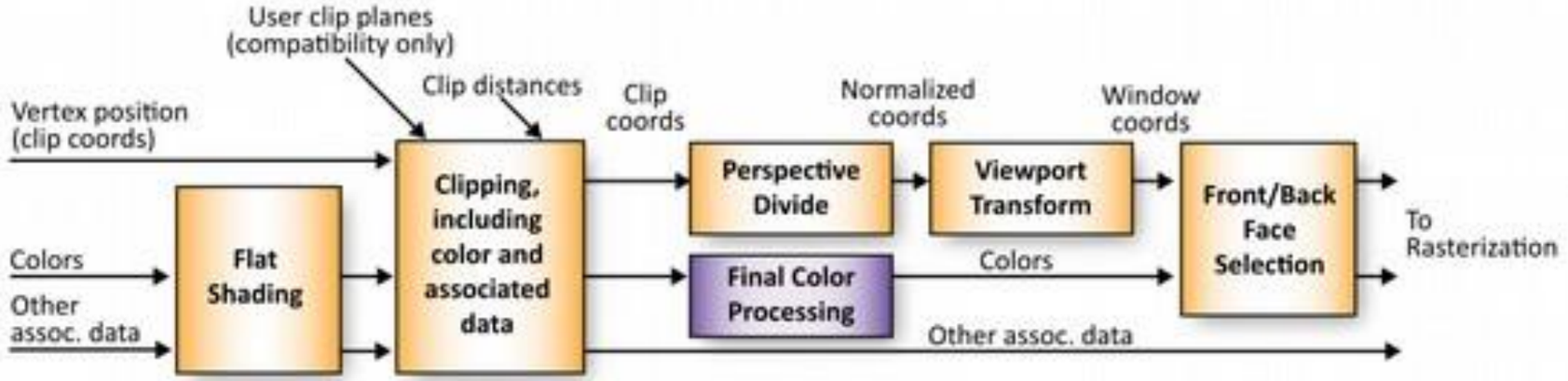
Geometry shaders (if enabled) consume individual primitives built in previous primitive assembly stages. For each input primitive, the geometry shader can output zero or more vertices, with each vertex directed at a specific vertex stream. The vertices emitted to each stream are assembled into primitives according to the geometry shader's output primitive type.

Transform feedback (if active) writes selected vertex attributes of the primitives of all vertex streams into buffer objects attached to one or more binding points.

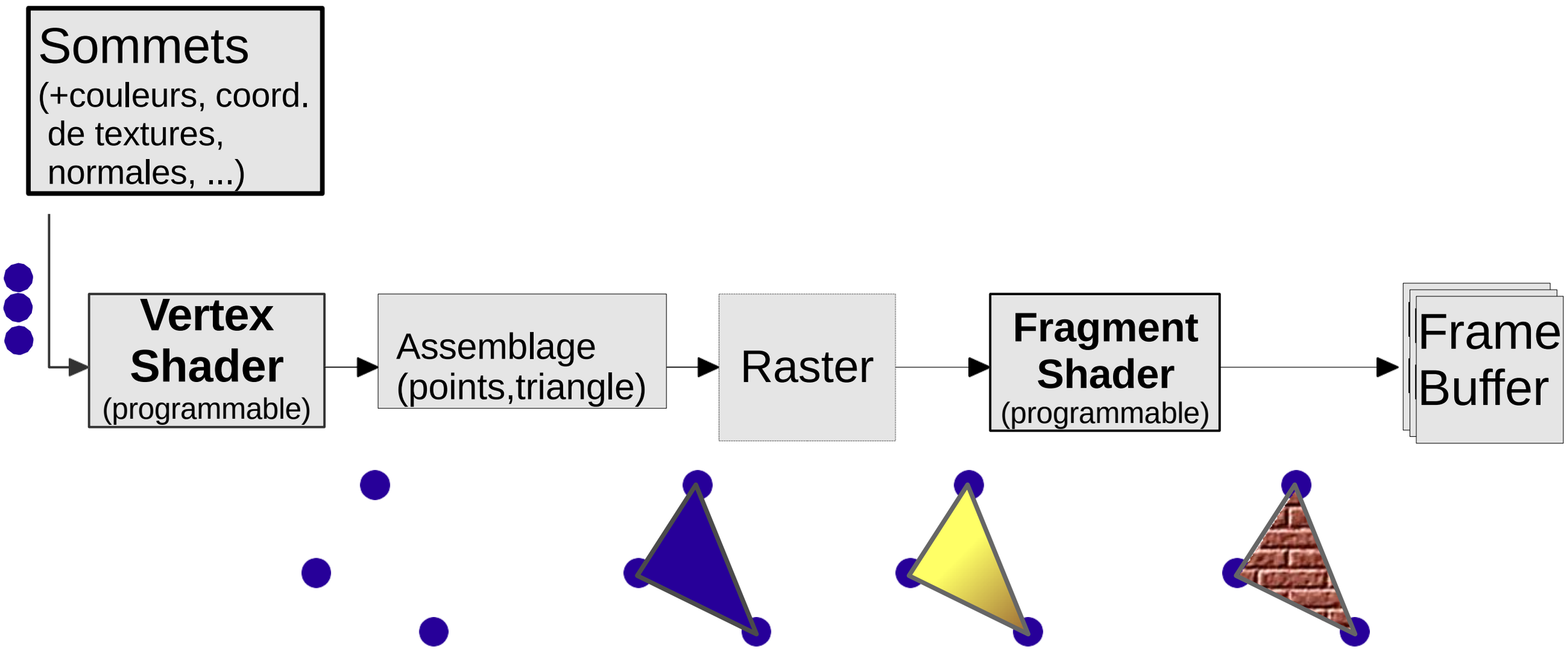
Primitives on vertex stream zero are then processed by fixed-function stages, where they are clipped and prepared for rasterization.

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.





Pipeline Graphique sur GPU



Pipeline : in/out

En entrée

- Une description numérique de la géométrie de la scène
  - = {primitives rastérisables}
  - ex. : ensemble de polygones

- Ensemble de paramètres :
  - un point de vue (caméra)
  - des attributs de matériaux associés à chaque objet
  - un ensemble de lumières
  - etc.

En sortie

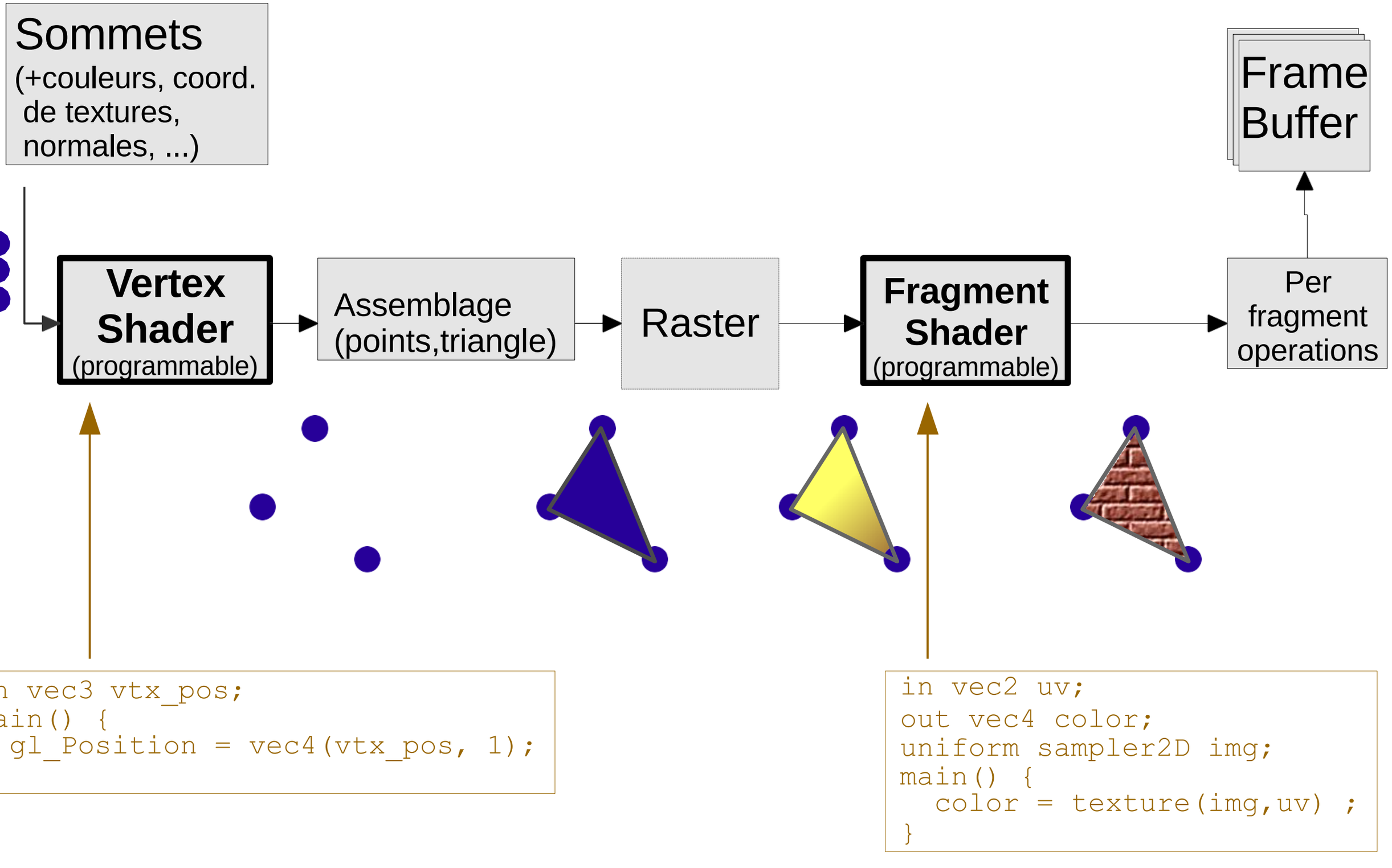
- une image = un tableau de pixels (couleurs RGB)

Algorithme de rendu par rasterisation

Pour chaque image :

- Effacer les tampons de destination (l'écran)
- Configurer la scène pour l'image courante:
  - positionner la caméra
  - positionner les sources lumineuses
  - etc.
- Pour chaque objet:
  - Charger la géométrie
  - Charger les textures
  - Configurer les shaders : matériau, transformations, etc.
  - Activer les shaders
  - **Tracer l'objet**
  - Restaurer les différents états
- Afficher l'image calculée à l'écran (double buffering)
- Calculer l'image suivante ...

Les étages programmables



Langages de programmation

Shader

– (petit) programme exécuté sur le GPU

Programmable via

– des langages de haut niveau (proche du C/C++)

- **GLSL** (*OpenGL Shading Language*)
  - compilateur intégré dans le driver OpenGL (>=2.0)
    - génération et compilation de code à la volée
  - standard ouvert

- HLSL (*Microsoft*)
  - DirectX only

OpenGL Shaders

Langage : GLSL (OpenGL Shading Language)

- Proche du C

Pas d'accès mémoire direct

- En lecture :
  - Vertex Buffer Objects – VBO (tableau de sommets avec attributs)
    - accès indirect
  - Textures (tableaux 1D, 2D, 3D, etc.)

- En écriture :
  - Frame Buffer Objects – FBO (tableaux 2D)

Compilation à la volée par le driver OpenGL

- Shader == char*
- Source code spécifié via des fonctions OpenGL

OpenGL Shaders

Deux types shaders (threads) :

- sommets et pixels

Pas de synchronisation, pas de mémoire partagée

- *« on ne veut pas que les pixels parlent entre eux... » !!*

Fonction principale

- fonction **main** (sans arguments)

```
void main(void) {
    /* … */
}
```

Paramètres constants

- variable globale avec le qualificatif « uniform »

```
uniform float intensity;
```

- valeur définie par l'hôte (`glUniform*(…)`)

OpenGL Shaders - données -

En entrée

- qualificatif « in »
  ```
  in vec3 vertex_position;
  ```
- VBO, variable spéciale, ou valeur calculée par l'étage précédent

En sortie

- qualificatif « out »
  ```
  out vec4 color;
  ```
- variable spéciale, valeur envoyée à l'étage suivant, ou FBO

GLSL : 1er exemple

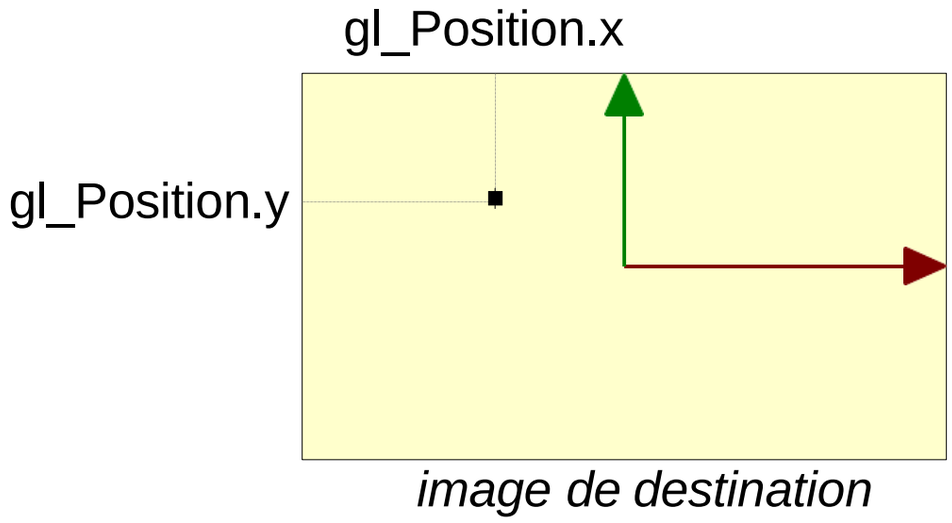
Vertex shader

```
in vec3 vtx_position ;

void main(void) {
    gl_Position.xy = vtx_position.xy;
    gl_Position.zw = vec2(0,1) ;
}
```

variable spéciale = position du sommet dans l'image 2D normalisée : [-1,1]x[-1,1]

Fragment shader

```
out vec4 color ;

void main(void) {
   color.rgb = vec3(1,0,0);
   color.a = 1 ;
}
```



GLSL : issu du C

Basé sur la syntaxe du C ANSI

fonctionalités graphiques

un peu de C++

- surcharge des fonctions
- déclaration des variables lorsqu'on en a besoin
- déclaration des *struct*
- type *bool*

qques fonctionnalités

- switch, goto, label
- union, enum, sizeof
- pointeurs, chaîne de caractères

GLSL : tableaux et structures

Tableaux

- comme en C
- Limité aux tableaux 1D

Structures

- comme en C++

Exemple

```
struct MonMateriau
{
        vec3 baseColor;
        float ambient, diffuse, specular;
};
MonMateriau mesMateriaux[12];
```

Types : vecteurs et matrices

Vecteurs:

- float, vec2, vec3, vec4
- int, ivec2, ivec3, ivec4
- bool, bvec2, bvec3, bec4

Matrices:

- mat2, mat3, mat4
  - matrice carrée de réels (en colonne d'abord)
  - utilisées pour les transformations
- mat2x2, mat2x3, mat2x4
- mat3x2, mat3x3, mat3x4
- mat4x2, mat4x3, mat4x4

GLSL : les constructeurs

Utilisés pour

- convertir un type en un autre
- initialiser les valeurs d'un type

Exemples:

```
vec3 v0 = vec3(0.1, -2.5, 3.7);
float a = float(v0);                    // a==0.1
vec4 v1 = vec4(a);                      // v1==(0.1, 0.1, 0.1, 0.1)

struct MyLight {
     vec3  position;
     float intensité;
};
MyLight light1 = MyLight(vec3(1,1,1), 0.8);
```

GLSL : manipulation des vecteurs

- **Nommage des composantes**
  - via *.xyzw* ou *.rgba* ou *.stpq*
  - ou [0], [1], [2], [3]

- **Peuvent être ré-organisées, ex:**

```
vec4 v0 = vec4(1, 2, 3, 4);
v0 = v0.zxyw + v0.wwyz;

vec3 v1 = v0.yxx;
v1.x = v0.z;
v0.wx = vec2(7,8);
```

```
// v0 = (7,5,4,7)
// v1 = (5,7,7)
// v1 = (4,7,7)
// v0 = (8,5,4,7)
```

- **Manipulation des matrices**

```
mat4 m;
m[1] = vec4(2);// la colonne   #1 = (2,2,2,2)
m[0][0] = 1;
m[2][3] = 2;
```

GLSL : les fonctions

- **Arguments : types de bases, tableaux ou structures**
- **Retourne un type de base ou void**
- ***Les récursions ne sont pas supportées***
- **les arguments peuvent être *in, out, inout***
  - Par défaut les arguments sont "in"

- **Exemple**

```
vec3 myfunc(in float a, inout vec4 v0, out float b)
{
     b = v0.y + a;   // écrit la valeur de b
     v0 /=  v0.w;    // màj de v0
     return v0*a;
}
```

GLSL : intro

- **Commentaires comme en C++**
- **Support des directives de pré-compilation**
  - #define, #ifdef, #if, #elif, #else, #endif, #pragma
- **Un shader doit avoir une fonction "main"**

```
in vec4 vert_position;
uniform mat4 MVP;
void main(void)
{
  gl_Position = MVP * vert_position;
}
```

Fonctions prédéfinies

Math

- radians(deg), degrees(rad), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x,y), atan(x_sur_y), pow(x,y), exp(x), log(x), exp2(x), log2(x), sqrt(x), invsersesqrt(x)
- abs(x), sign(x), floor(x), ceil(x), fract(x), mod(x,y), min(x,y), max(x,y), clamp(x, min, max), mix(x, y, t) = interpolation linéaire, step(t, x) = x<t ? 0 : 1
- smoothstep(t0, t1, x) = interpolation d'Hermite

Géométrie

- **length**(x), **distance**(x,y), **dot**(x,y), **cross**(x,y), **normalize**(x), reflect(I,N), refract(I,N,eta)

Relation entre vecteurs

- bvec lessThan(x,y) (composante par composante)
  lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual
- bool any(bvec x), bool all(bvec x), bvec not(bvec x)

GLSL : "type qualifiers"

const

- variable ne pouvant être modifiée
- valeur écrites lors de la déclaration

in

- déclare une variable provenant de l'étage précédent

out

- déclare une variable envoyée à l'étage suivant

uniform

- variable constante pour un groupes de primitives
- valeurs définies par le programme hôte

layout()

- contrôle sur le stockage des blocks
  `layout(row_major) uniform mat3x4 A;`

in/out/inout

- pour les fonctions

GLSL : 1er exemple

```
// vertex shader
in vec4 attrib_position;
in vec3 attrib_normal;
out vec3 normal;
uniform mat4 mvp ;                          // modelview * projection
uniform mat3 mat_normal ;                   // pour transformer les normales

void main(void) {
  gl_Position = mvp * attrib_position;
  normal = normalize(mat_normal * attrib_normal);
}

// fragment shader
in vec3 normal;
out vec3 out_color;
uniform vec3 color;
uniform vec3 light_dir;

void main(void) {
  out_color = color * max(0,dot(normalize(normal),light_dir) );
}
```