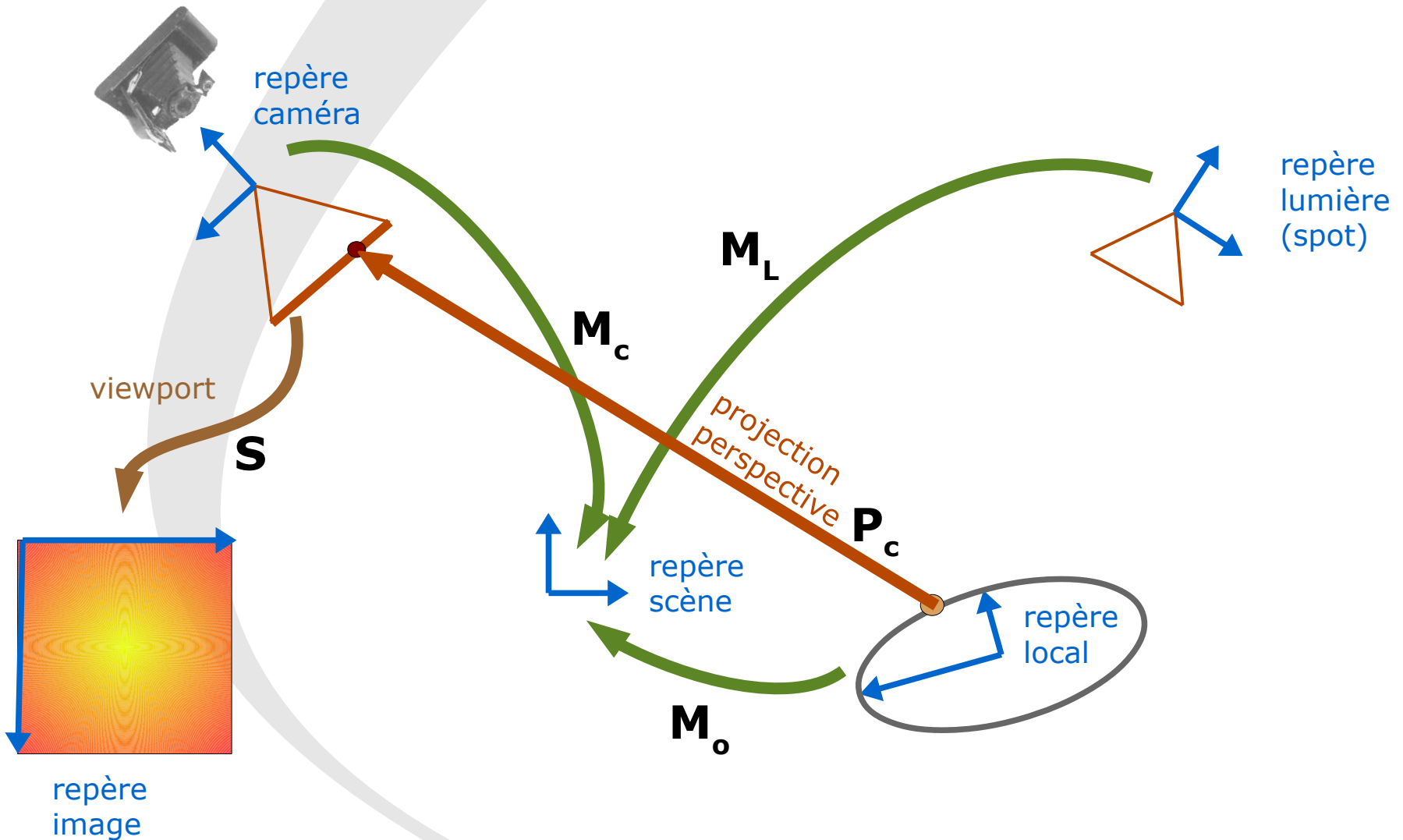


# Transformations

*[gael.guennebaud@inria.fr](mailto:gael.guennebaud@inria.fr)*

*<http://www.labri.fr/perso/pbenard/teaching/pghp>*

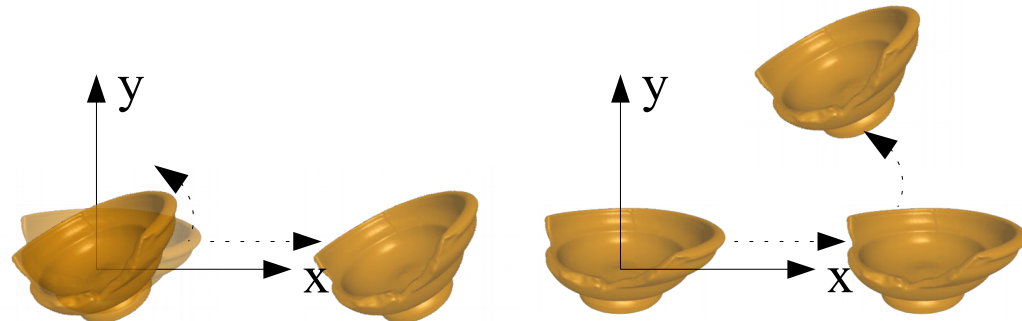
# Les transformations mises en jeu



# Rappels sur les Transformations

- Utilisation des coordonnées homogènes
  - point euclidien :
    - $v=(x,y,z,w) \sim v'=(x',y',z') = (x/w, y/w, z/w)$  si  $w \neq 0$
  - vecteur/direction :  $w = 0$  (point à l'infini !)
  - transformations affines représentées par produit matriciel
    - $v' = M * v$                        $v = M^{-1} * v'$
  - enchaînement de transformations = multiplication matricielle
    - $v' = T * R * v$
  - ATTENTION :  
non commutatif !

$$T * R * v \neq R * T * v$$



# Transformations élémentaires

- **Translation :**

- Eigen :

- $\mathbf{v}' = \mathbf{v} + \mathbf{t}$ ;
- $\mathbf{v}' = \text{Translation3f}(\mathbf{t}) * \mathbf{v}$ ;
- $\mathbf{v}' = \text{Translation3f}(t_x, t_y, t_z) * \mathbf{v}$ ;

- **Mise à l'échelle :**

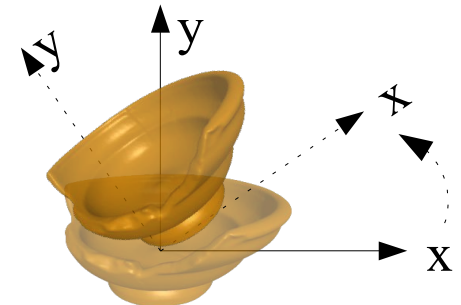
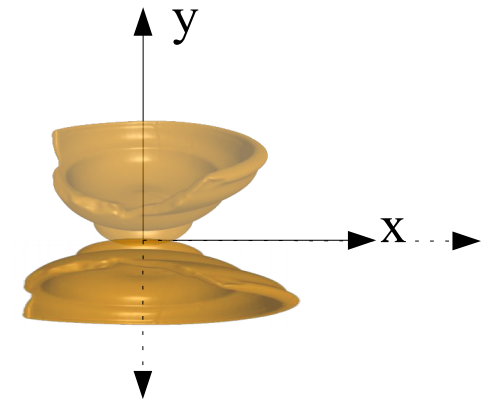
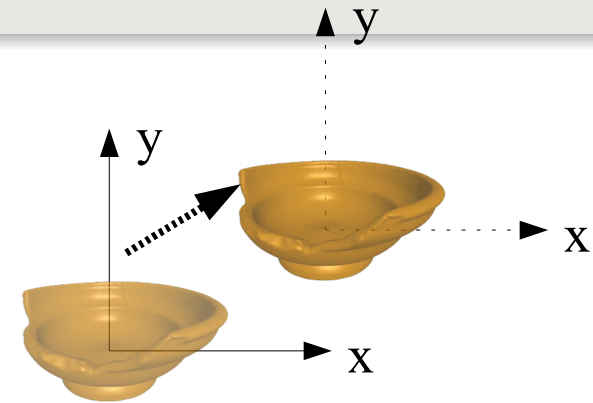
- Eigen :

- $\mathbf{v}' = s * \mathbf{v}$  ;
- $\mathbf{v}' = \text{Scaling3f}(s_x, s_y, s_z) * \mathbf{v}$
- $\mathbf{v}' = \text{Vector3f}(s_x, s_y, s_z).asDiagonal() * \mathbf{v}$

- **Rotation :**

- Eigen :

- $\mathbf{v}' = \text{AngleAxis3f}(\text{angle}, \text{axis}) * \mathbf{v}$ ;
- $\text{Matrix3f } R = \text{Matrix3f}(\text{AngleAxis3f}(\text{angle}, \text{axis}))$ ;
- $\text{Quaternion } q = \text{AngleAxis3f}(\text{angle}, \text{axis})$  ;

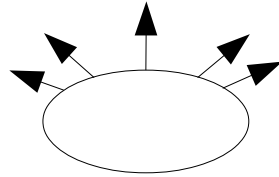
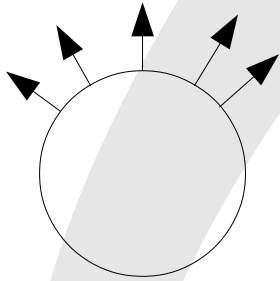


# Combinaison

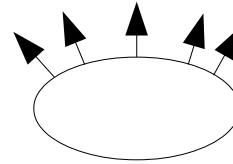
- **Combinaison with Eigen :**
  - *Affine3f M = AngleAxisf(angle1,axis1)*
    - \* *Translation3f(t1)*
    - \* *Scaling3f(s1)*
    - \* *AngleAxisf(angle2,axis2) ;*

# Transformation des normales

- **Problème d'une mise à l'échelle non uniforme M :**



$$n' = M * n \\ \Rightarrow \text{Faux !}$$



correct

- **Règle: utiliser la transposée de l'inverse:**

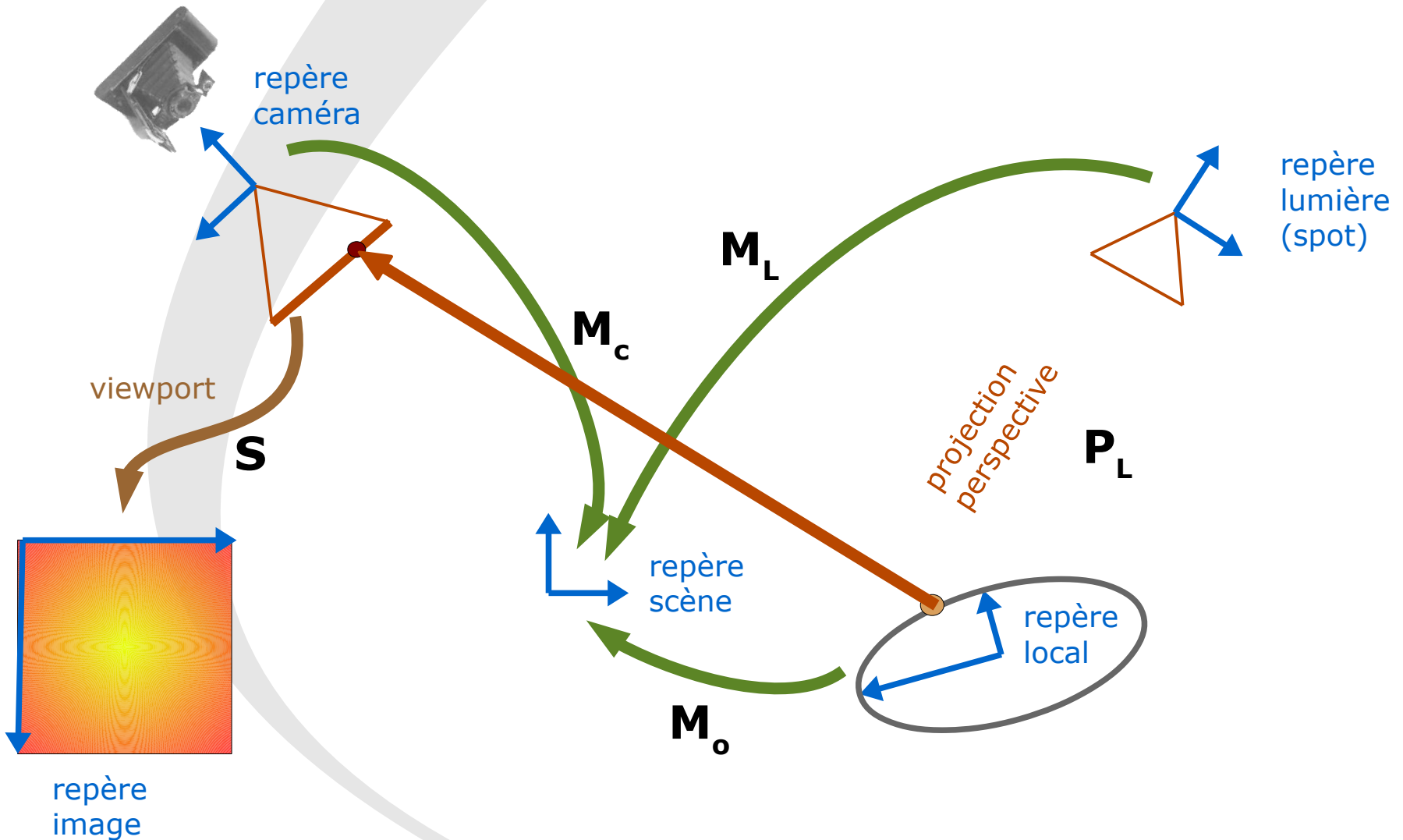
– Soit M la matrice 3x3 de transformation courante:

$$v' = M * v + T$$

– Alors :

$$n' = (M^{-1})^t * n$$

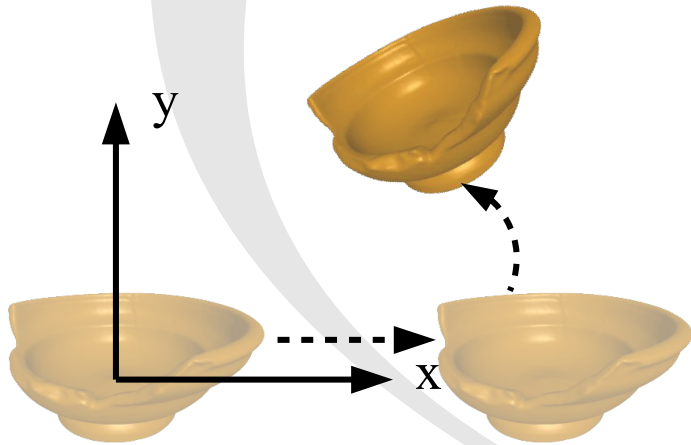
# Les transformations mises en jeu



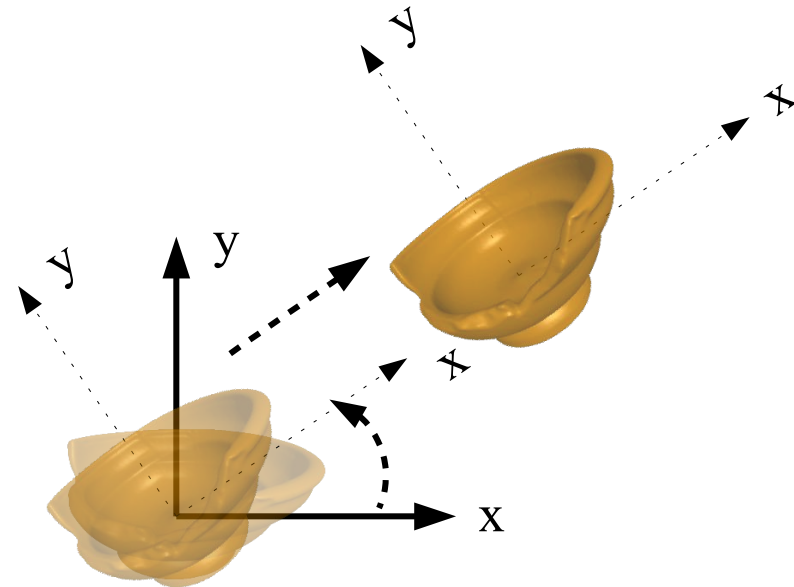
# “Penser” les transformations

- Approche “repère global”
  - le repère reste fixe
  - définir les transformations de droite à gauche

$$R * T * v$$



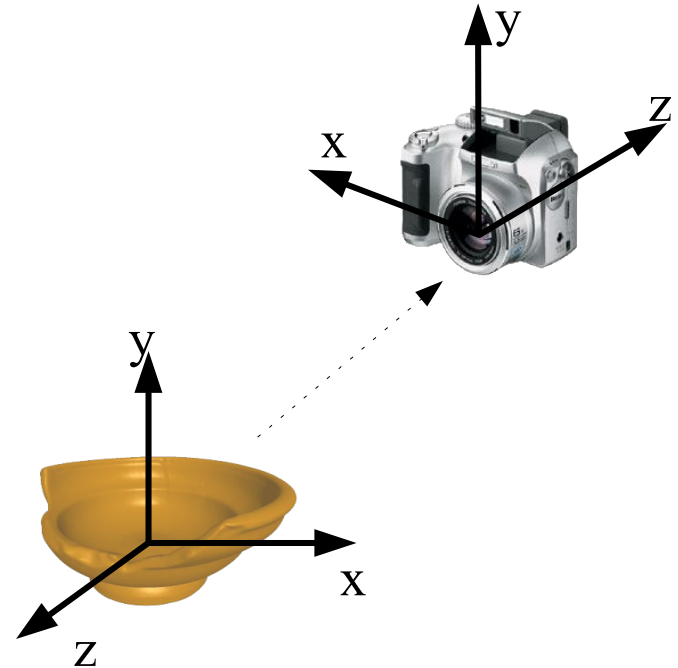
- Approche “repère local”
  - chaque transformations affecte le repère courant
  - “amener” le repère dans la position/orientation voulue
  - composer de gauche à droite





# Transformation de visualisation

- **Permet de positionner la caméra**  
transformation de visualisation  
~ inverse du repère local

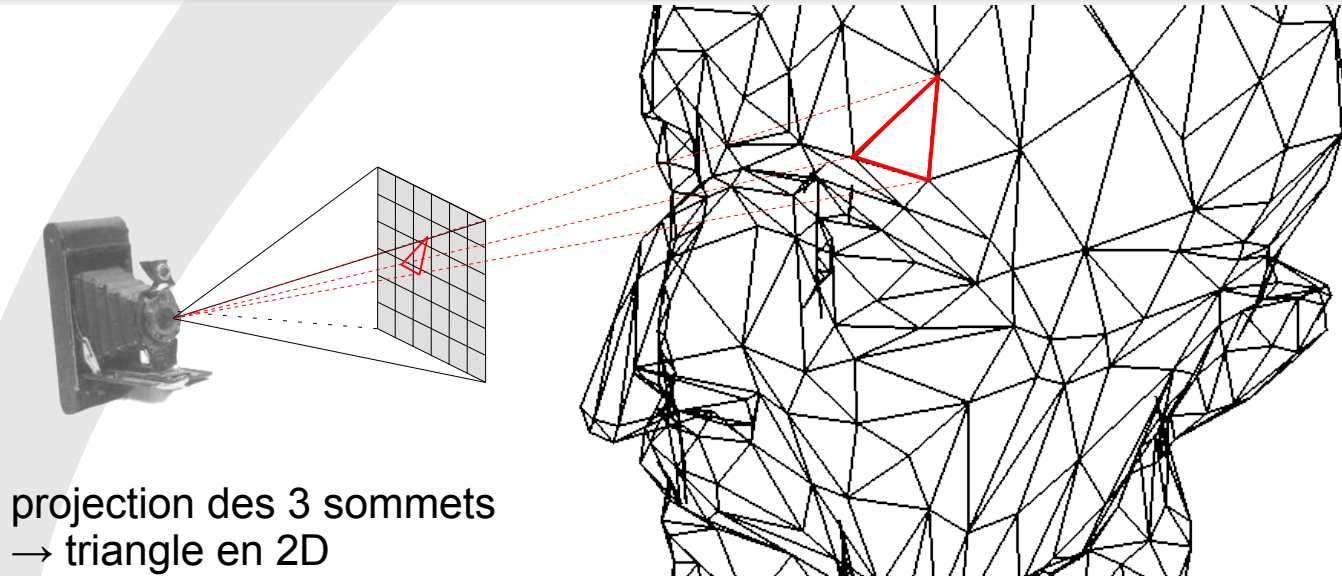


- **Exercice : positionner et orienter la caméra :**

- *Matrix4f lookAt(const Vector3f& p,  
const Vector3f& t,  
const Vector3f& u)*

*// position de la caméra  
// point de visé (target)  
// vecteur "hauteur" (up)*

# Rastérisation

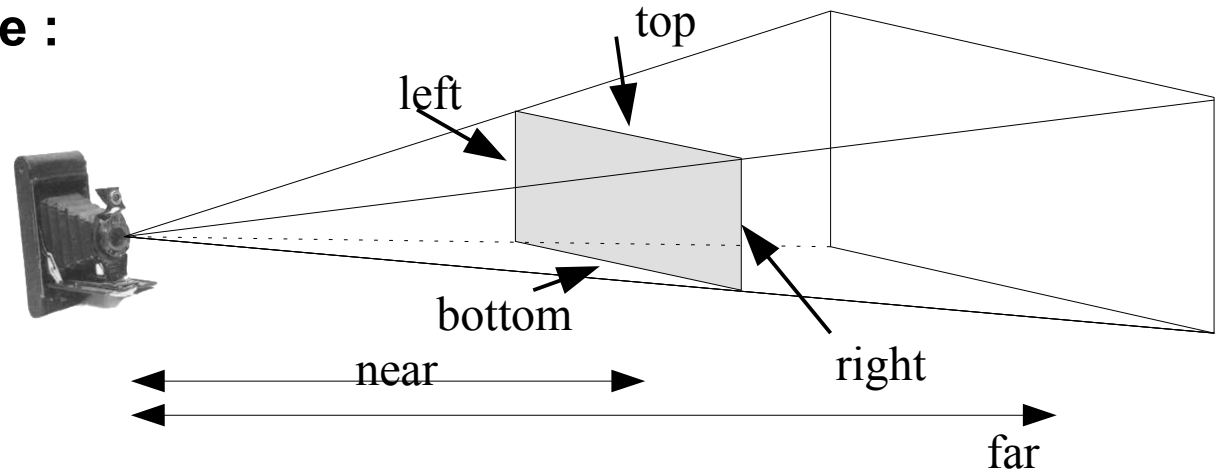


- **Rastérisation**

- Pour chaque primitive  $P_i$ , trouver les rayons intersectant  $P_i$
- Rendu en deux étapes
  - projection des primitives sur l'écran (*forward projection*)
  - discrétisation (conversion des primitives 2D en pixels)
- scène = ensemble de primitives « rastérisables »  
→ **problème : élimination des parties cachées**

# Transformation de projection

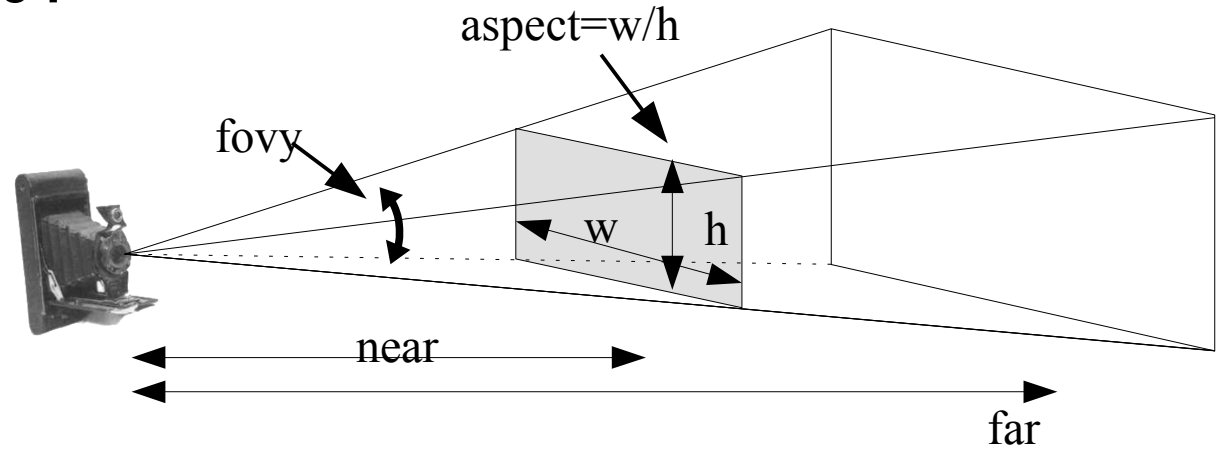
- Projection perspective :



$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

# Transformation de projection

- **Projection perspective :**

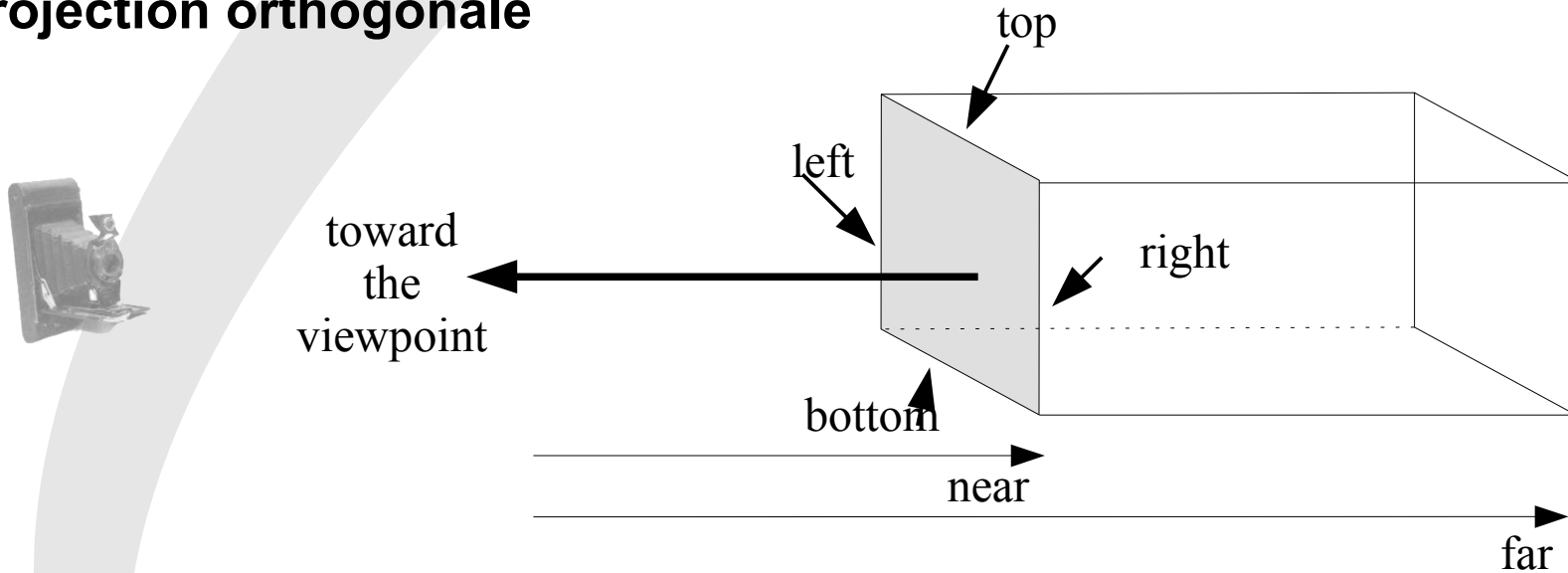


- **Exercice : écrire une fonction :**

***Matrix4f makePerspective(float fov, float aspect, float n, float f)***

# Transformation de projection

- Projection orthogonale



$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Vertex pipeline : résumé

Sommets exprimés dans le repère local de l'objet (3D)

**local** → **global** → **camera** → **projection**

Vertex  
Shader

Sommets exprimés dans le repère de la caméra normalisé (3D)  
(coordonnées homogènes)

Division  
perspective

Coordonnées écran normalisé  
(réel entre  $[-1..1]$  , 2D)

Viewport

Coordonnées écran (entier entre  $[0..w]$  , 2D)  
Viewport = zone rectangulaire de la fenêtre  
*glViewport(GLint x, GLint y, GLint w, GLint h)*

# API

	<b>GLSL</b>	<b>C++ (Eigen)</b>
<i>vecteur 2D</i>	vec2	Vector2f
<i>vecteur 3D</i>	vec3	Vector3f
<i>vecteur 4D</i>	vec4	Vector4f
<i>matrice 2x2</i>	mat2,mat2x2	Matrix2f
<i>matrice 3x3</i>	mat3,mat3x3	Matrix3f
<i>matrice 4x4</i>	mat4,mat4x4	Matrix4f
<i>matrice 3x4</i>	mat3x4	Matrix<float,3,4>
<i>accès</i>	v.x	v.x(), v[0], v(0)
	v.y	v.y(), v[1], v(1)
	v.z	v.z(), v[2], v(2)
	v.w	v.w(), v[3], v(3)
<i>(i=ligne,j=colone)</i>	M[j][i]	M(i,j)
<i>colone d'une matrice</i>	M[j]	M.col(j)
<i>ligne d'une matrice</i>		M.row(i)
<i>sous-matrice</i>		M.block(i,j,rows,cols)

# API

	<b>GLSL</b>	<b>C++ (Eigen)</b>
<i>produits</i>	$M * v$ $M1 * M2$	$M * v$ $M1 * M2$
<i>transpose</i>	<code>transpose(M)</code>	<code>M.transpose()</code>
<i>combinaison linéaire</i>	$a*v1 + b*v2$	$a*v1 + b*v2$
<i>produit scalaire</i>	<code>dot(v1,v2)</code>	<code>v1.dot(v2)</code> <code>v1.transpose()*v2</code>
<i>longueur/norme</i>	<code>length(v)</code>	<code>v.norm()</code>
<i>produit vectoriel</i>	$v3 = \text{cross}(v1,v2)$	$v3 = v1.\text{cross}(v2)$



# Eigen

## *Initialisation*

```
Matrix3 M ;  
M << 1, 2, 3,  
      4, 5, 6,  
      7, 8, 9;
```

*Translation par un vecteur v*

Translation3f(v)

*Rotation d'un angle theta autour  
d'un axe v*

AngleAxisf(theta,v)  
M = Matrix3f(AngleAxisf(theta,v))

*Mise à l'échelle*

Scaling3f(v)  
Scaling3f(s0, s1, s2)

*Transformation affine  
partie linéaire (3x3)  
translation  
matrice 4x4*

Affine3f A ;  
A.linear()  
A.translation()  
A.matrix()

*Exemple :*

```
Affine3f A ;  
A = Translation3f(p) * AngleAxisf(theta,v) * Translation3f(-p) ;  
v2 = A * v1 ;
```