

Pipeline de Rendu et Illumination

Principes d'une scène 3D - Concepts

Description

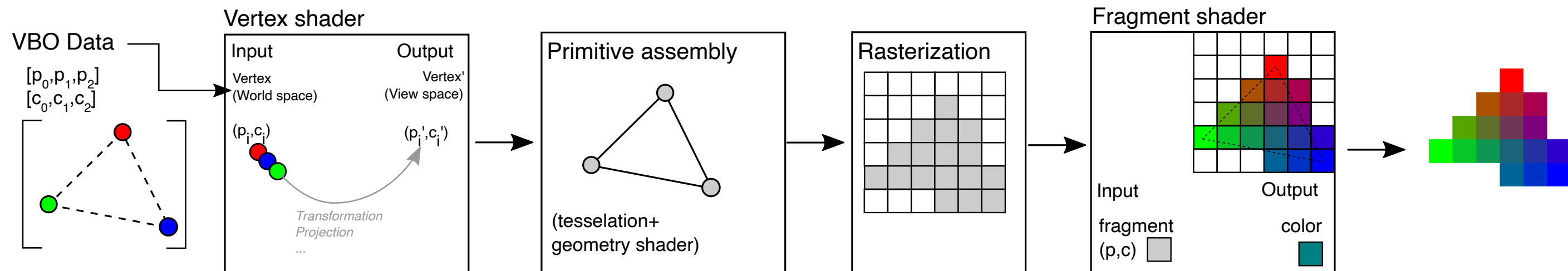
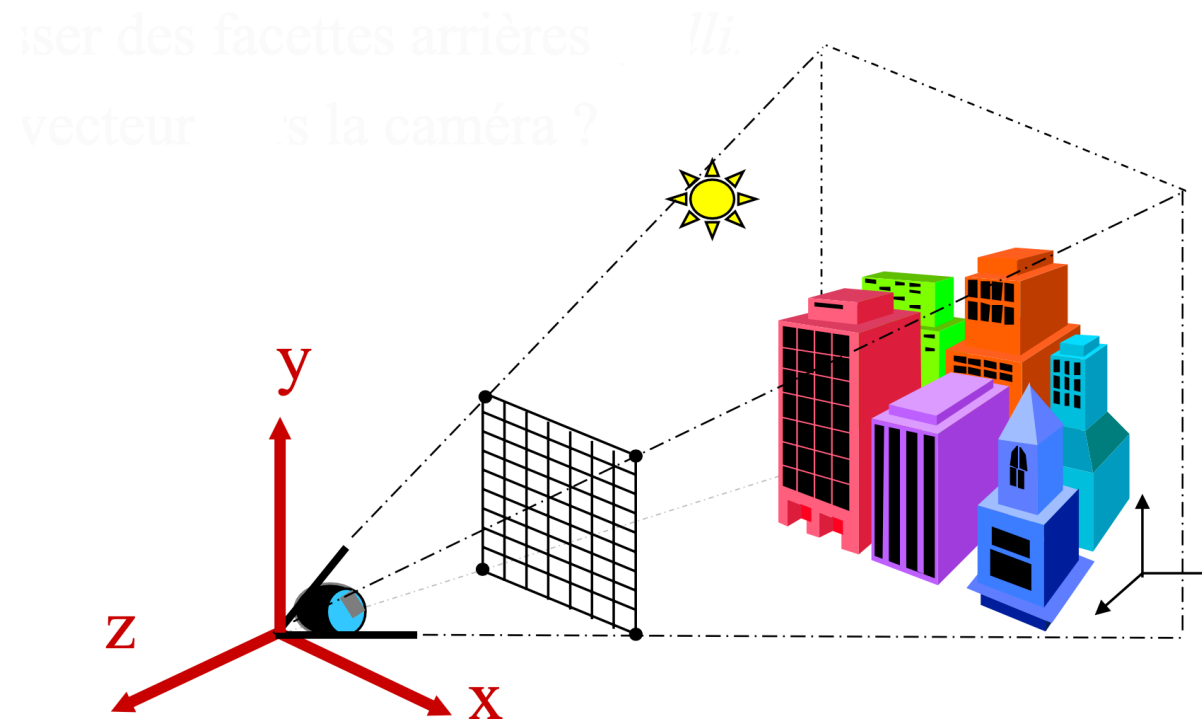
- Modèle 3D: Une surface ou un volume
- Source de lumière
- Caméra

Sortie

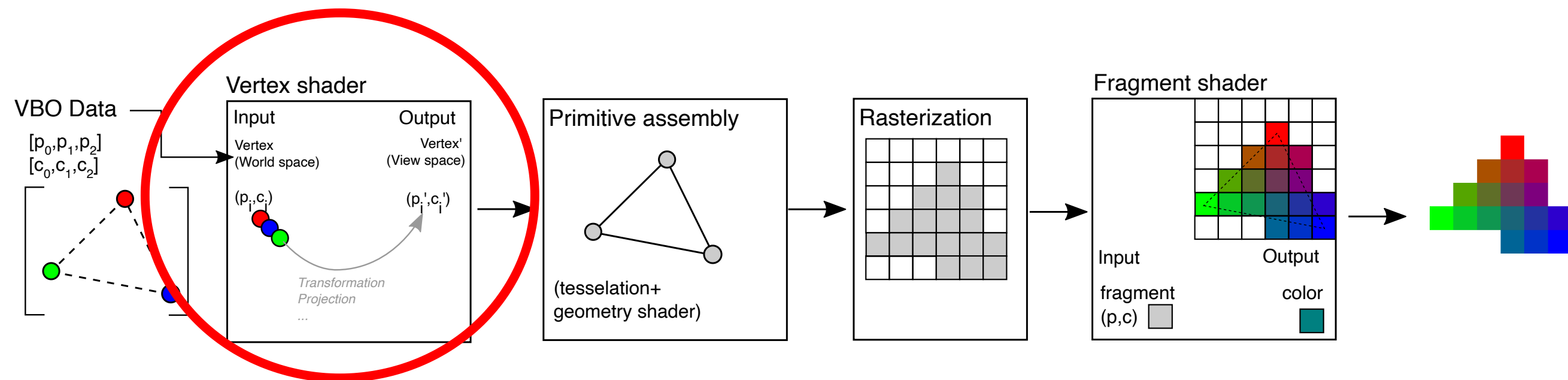
- Image vue de la caméra

OpenGL/GPU

- Pas de notion de caméra, de lumière
- Notion de sommets et d'attributs: entrée du vertex shaders
- Notion de pixels / fragments coloré: sortie du fragment shaders



Transformation des sommets Projection et perspective



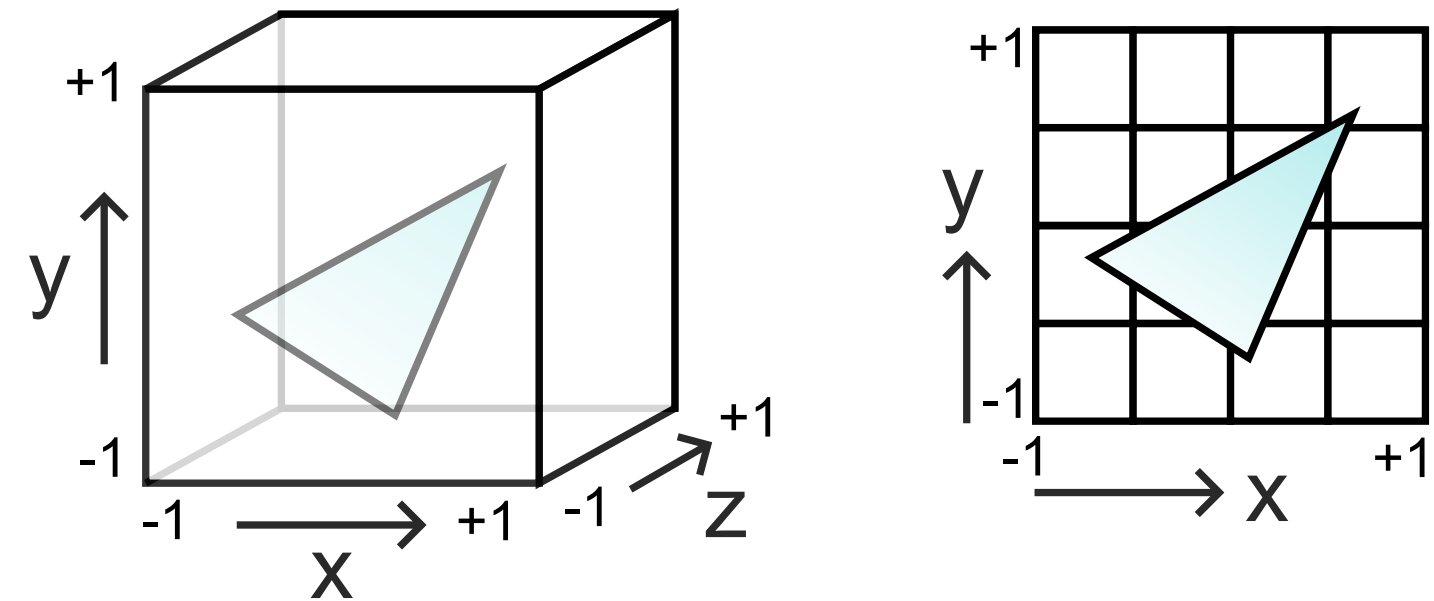
Principe de projection

OpenGL n'affiche que les triangles dans le cube $[-1, 1]^3$

Normalized Device Coordinates (NDC)

$(x, y) \rightarrow$: coordonnées de l'écran

z : profondeur



Objectif de la projection

Convertir les coordonnées globales (World space coordinates) en Normalized Device Coordinates

Décider d'un modèle de caméra

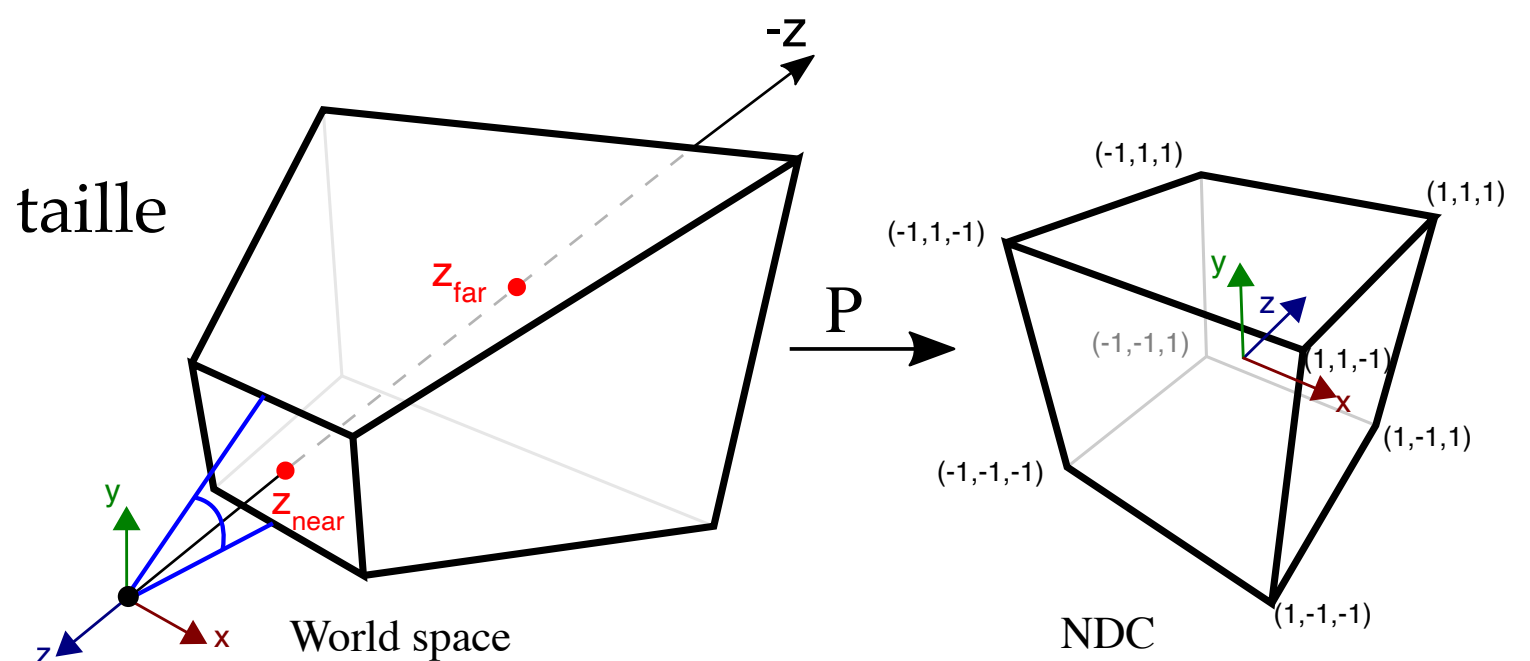
Partie visible de l'espace

Appliquer une perspective: éloignement \rightarrow réduction de taille

Perspective la plus courante:

Visibilité dans un cône rectangulaire tronqué: Frustum

Perspective en " $1/z$ "



Projection formulation

World space: (x, y, z) , NDC: $(x_{ndc}, y_{ndc}, z_{ndc})$

$$(x_{ndc}, y_{ndc}, z_{ndc}) = P(x, y, z)$$

Mapping

$$(w, h, -z_{near}) \rightarrow (1, 1, -1)$$

$$\left(w \frac{z_{far}}{z_{near}}, h \frac{z_{far}}{z_{near}}, -z_{far}\right) \rightarrow (1, 1, 1)$$

etc.

Rem: $-z$ pour garder un repère droit

Formulation

$$x_{ndc} = \frac{z_{near}}{-z} x / w$$

ou, avec un champ de vision (field of view) d'angle θ : $x_{ndc} = 1 / \tan(\theta/2) x / (-z)$

$$y_{ndc} = \frac{z_{far}}{-z} y / h$$

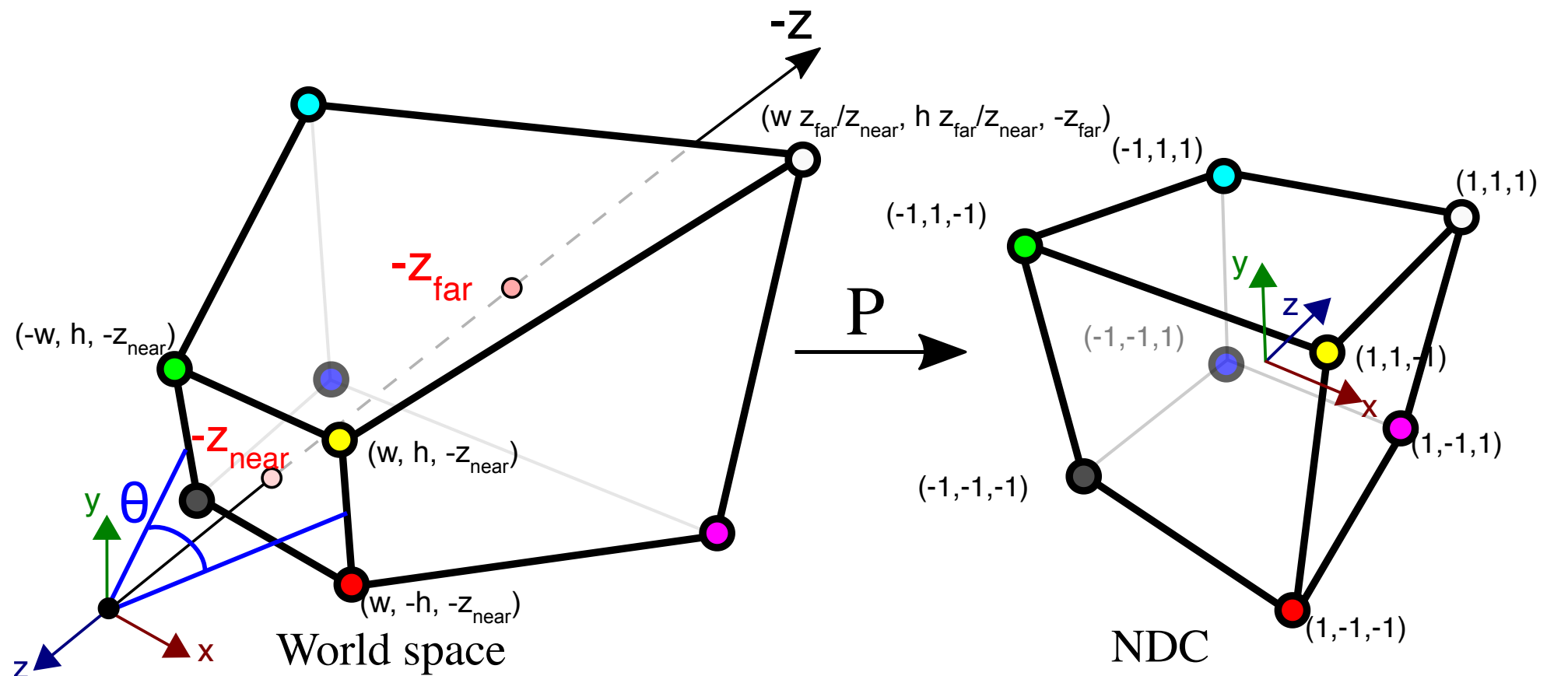
ou, avec un champ de vision (field of view) d'angle θ : $y_{ndc} = r / \tan(\theta/2) y / (-z), r = h/w$

- z_{ndc} : variation linéaire divisée par $1/z$: $z_{ndc} = (\alpha z + \beta) / (-z)$

Division par $1/z \Rightarrow$ précision plus fine proche de la caméra, grossier plus loin.

Mapping $(0, 0, -z_{far}) \rightarrow (0, 0, 1)$, et $(0, 0, -z_{near}) \rightarrow (0, 0, -1)$

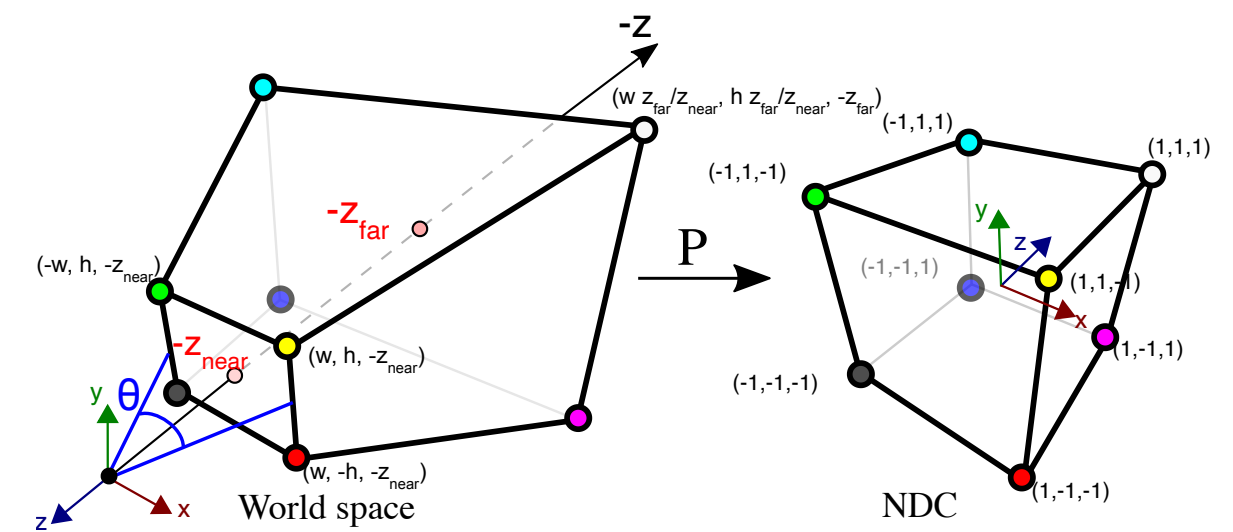
$$\Rightarrow \begin{cases} (-\alpha z_{far} + \beta) / z_{far} = 1 \\ (-\alpha z_{near} + \beta) / z_{near} = -1 \end{cases} \Rightarrow \begin{cases} \alpha = -(z_{far} + z_{near}) / (z_{far} - z_{near}) \\ \beta = -2 z_{near} z_{far} / (z_{far} - z_{near}) \end{cases} \Rightarrow z_{ndc} = \frac{1}{z} \left(\frac{z_{far} + z_{near}}{z_{far} - z_{near}} z + 2 \frac{z_{near} z_{far}}{z_{far} - z_{near}} \right)$$



Projection matrix

Formulation matricielle en coordonnées généralisées: $p_{ndc} = \text{Proj } p$

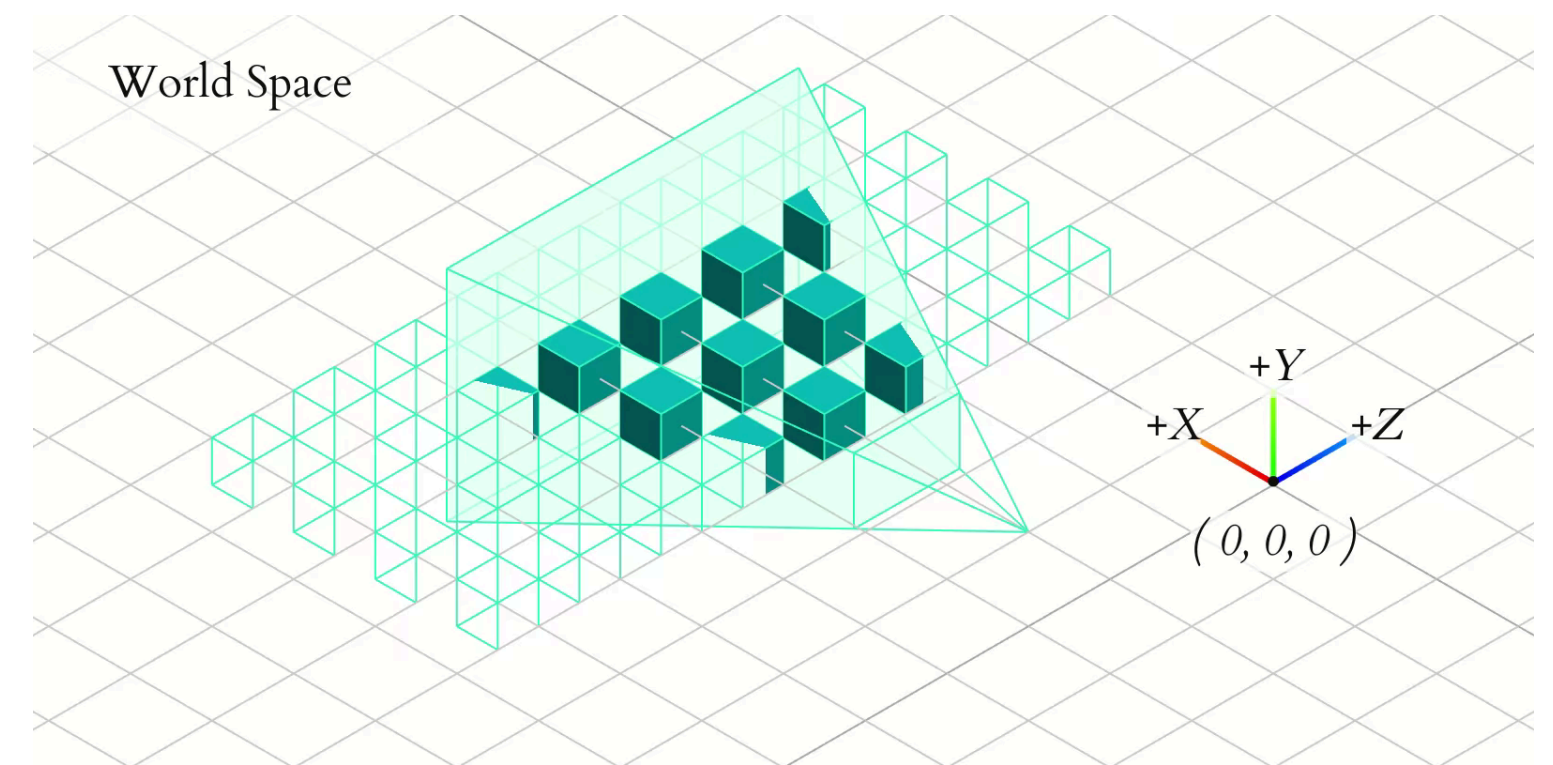
$$(x_{ndc}, y_{ndc}, z_{ndc}) = \frac{1}{-z} \left(\frac{1}{\tan(\theta/2)} x, \frac{r}{\tan(\theta/2)} y, -\frac{z_{far} + z_{near}}{z_{far} - z_{near}} z - 2 \frac{z_{near} z_{far}}{z_{far} - z_{near}} \right)$$



$$\Rightarrow p_{ndc} = \begin{pmatrix} 1/\tan(\theta/2) x \\ r/\tan(\theta/2) y \\ -\frac{z_{far} + z_{near}}{z_{far} - z_{near}} z - 2 \frac{z_{near} z_{far}}{z_{far} - z_{near}} \\ -z \end{pmatrix} = \underbrace{\begin{pmatrix} 1/\tan(\theta/2) & 0 & 0 & 0 \\ 0 & r/\tan(\theta/2) & 0 & 0 \\ 0 & 0 & -\frac{z_{far} + z_{near}}{z_{far} - z_{near}} & -2 \frac{z_{near} z_{far}}{z_{far} - z_{near}} \\ 0 & 0 & -1 & 0 \end{pmatrix}}_{\text{Proj}} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \text{Proj } p$$

Normalized Device Coordinates:

- Espace déformé dans le cube $[-1, -1]$
+ déformation perspective
- Image = Vue (x,y) du cube
- z_{ndc} = profondeur vue de la caméra dans l'espace image.



Credit: Jordan Santell

Z-Fighting

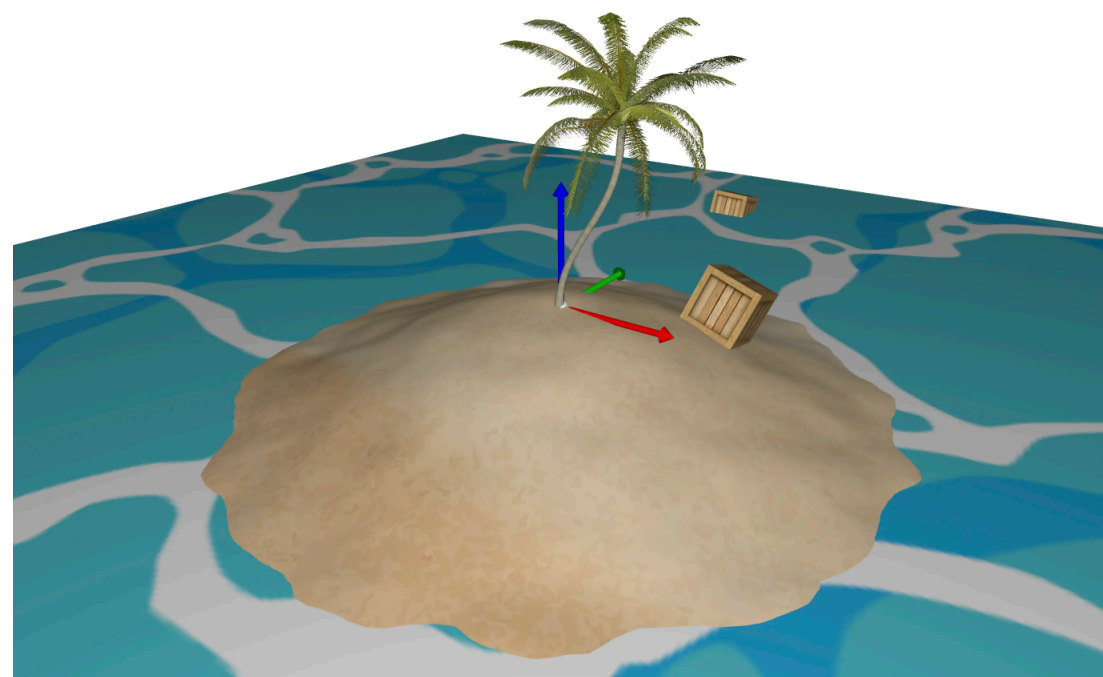
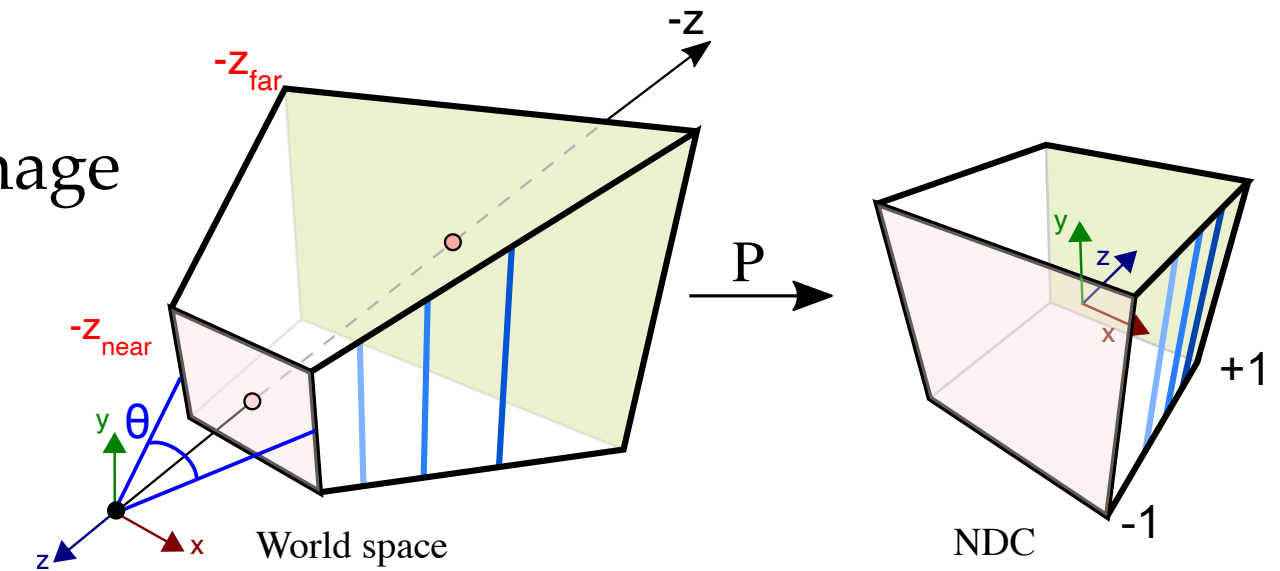
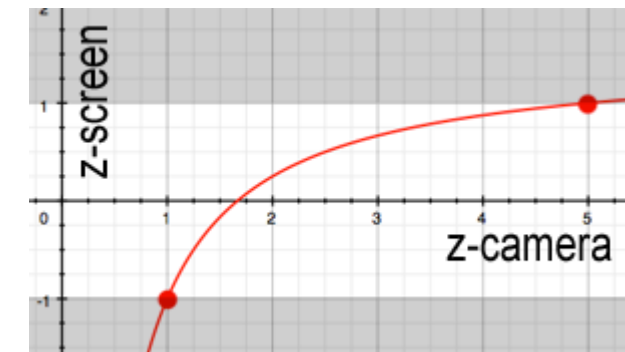
Plus de précision proche de z_{near} que de z_{far}

Si z_{near} trop proche de 0

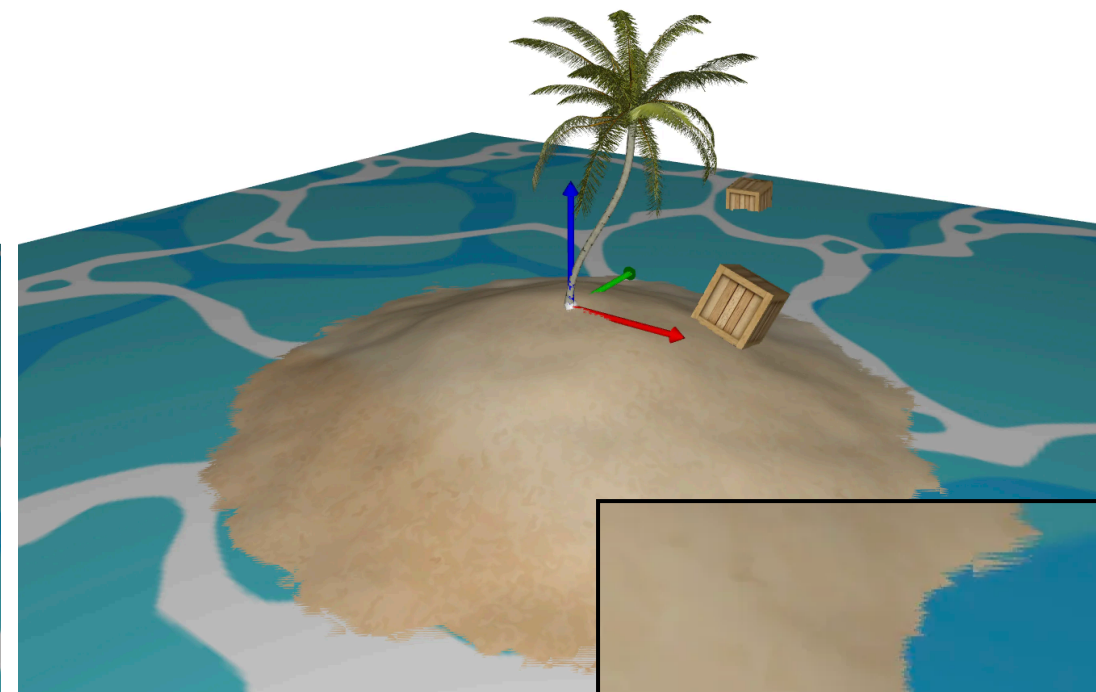
Toute la précision est concentrée sur z_{near}

Profondeur éléments éloignés = même valeur après échantillonnage

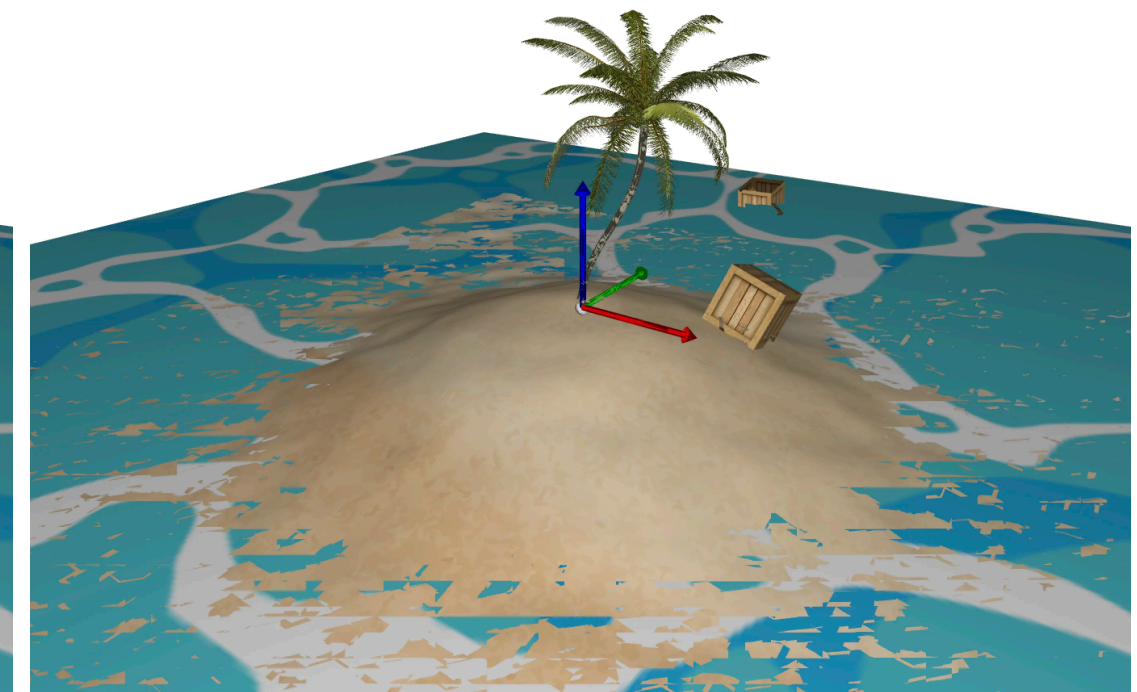
⇒ Z-fighting / Depth-fighting



$z_{near} = 10^{-2}$



$z_{near} = 10^{-4}$



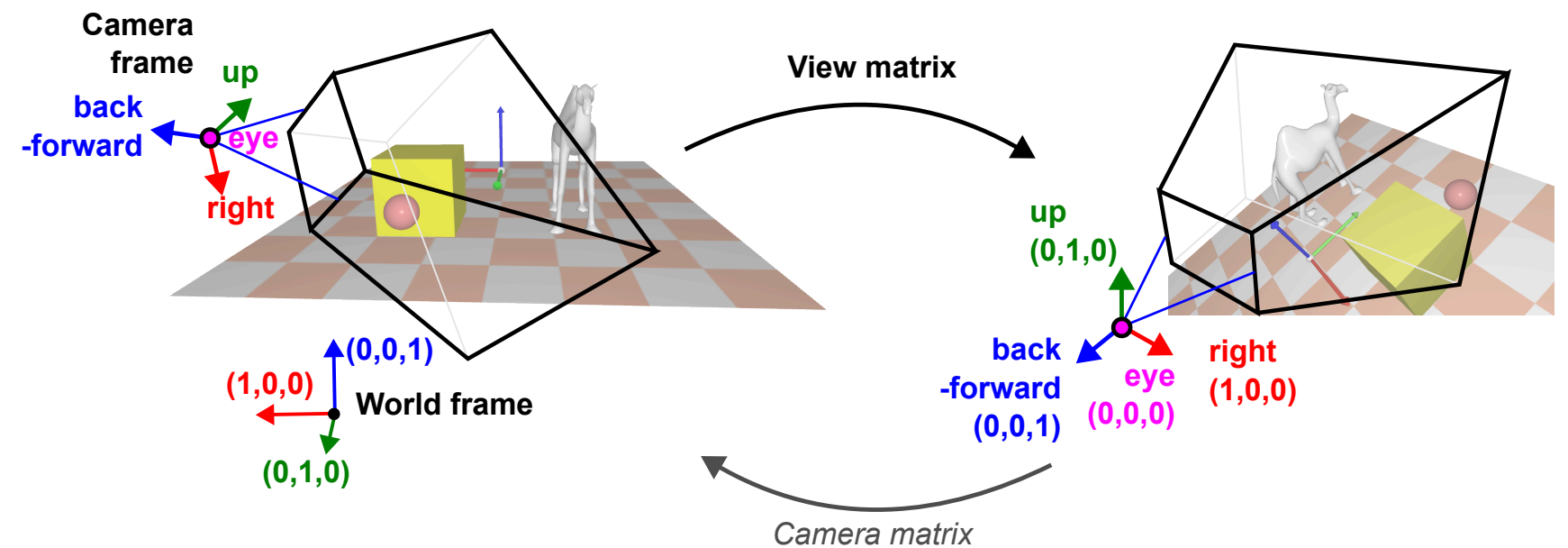
$z_{near} = 10^{-5}$

Z-Fighting Demo

View matrix - Formulation

Matrice **View**:

Transforme *World space coords.* → *View space coords.*



Coordonnées de la caméra: Cam

$$\text{Cam} \times (1, 0, 0, 0) = \textit{right}$$

$$\text{Cam} \times (0, 1, 0, 0) = \textit{up}$$

$$\text{Cam} \times (0, 0, 1, 0) = \textit{back}$$

$$\text{Cam} \times (0, 0, 0, 1) = \textit{eye}$$

View matrix: View

$$\text{View} \times \textit{right} = (1, 0, 0, 0)$$

$$\text{View} \times \textit{up} = (0, 1, 0, 0)$$

$$\text{View} \times \textit{back} = (0, 0, 1, 0)$$

$$\text{View} \times \textit{eye} = (0, 0, 0, 1)$$

$$\text{Cam} = \begin{pmatrix} | & | & | & | \\ \textit{right} & \textit{up} & \textit{back} & \textit{eye} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O & \textit{eye} \\ 0 & 1 \end{pmatrix} \quad \left| \quad \text{View} = \begin{pmatrix} (\dots & \textit{right} & \dots) & -\textit{right} \cdot \textit{eye} \\ (\dots & \textit{up} & \dots) & -\textit{up} \cdot \textit{eye} \\ (\dots & -\textit{front} & \dots) & -\textit{back} \cdot \textit{eye} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O^T & -O^T \textit{eye} \\ 0 & 1 \end{pmatrix}$$

$$\text{View} = \text{Cam}^{-1}$$

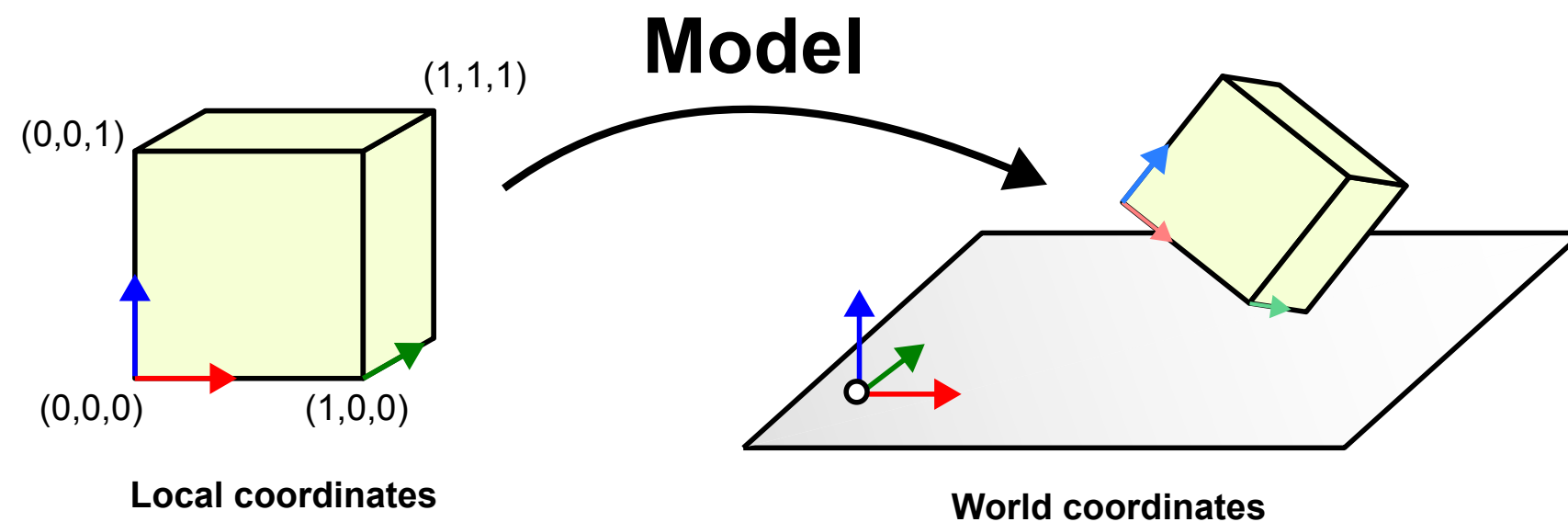
Model matrix

Positionne l'objet dans le monde

Model: coords. locales objet \rightarrow world coords.

Intêret:

- Charge une unique fois les coordonnées géométriques de l'objet dans un repère locale.
- Place / déplace en changeant uniquement la matrice Model.



$$\text{Model} = \begin{pmatrix} \text{rotation} & \text{translation} \\ 0 & 1 \end{pmatrix}$$

Synthèse matrices - Formulation

Entrée: Coordonnée $p = (x, y, z, 1)$ dans l'espace objet

- Coordonnée dans l'espace du monde $p_{ws} = \text{Model } p$

- Coordonnée dans l'espace caméra $p_{view} = \text{View } p_{ws}$

- Coordonnée dans le NDC - Normalized Device Coordinates $p_{ndc} = \text{Proj } p_{view}$

Sortie en espace image: NDC homogénéisée $p_{out} = (x_{ndc}/w_{ndc}, y_{ndc}/w_{ndc}, z_{ndc}/w_{ndc}, 1)$

Formulation globale: $p_{ndc} = \text{Proj} \times \text{View} \times \text{Model } p$

1ère étape du pipeline graphique: Vertex shader

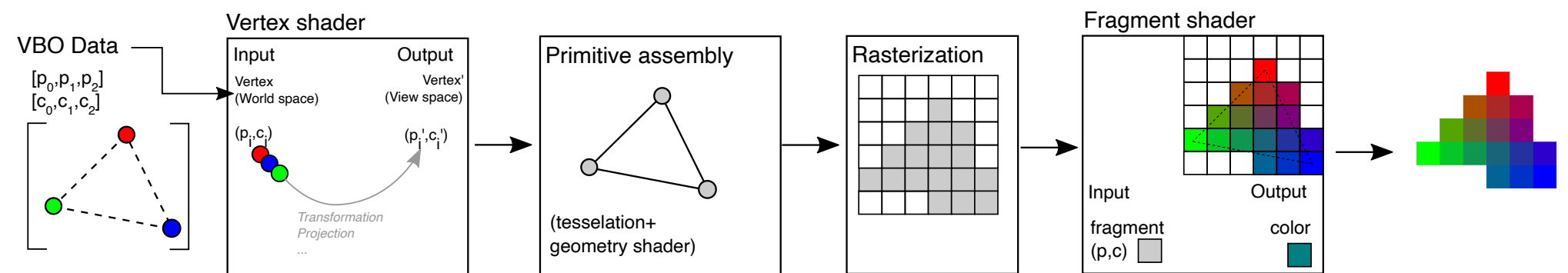
```
#version 330 core

// vertex position in local space (x,y,z)
layout (location = 0) in vec3 vertex_position;

// uniform = parameters send from C++ code
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    // gl_Position: Output position from Vertex Shader (NDC)
    gl_Position = projection * view * model * (position,1.0);

    // normalization automatically performed by the GPU
}
```



Synthèse matrices - Demo

$$p_{ndc} = \text{Proj} \times \text{View} \times \text{Model } p$$

Matrices: C++ / paramètres uniformes

Changement de coordonnées en C++:

Principalement mono-thread

- Calcul des nouvelles positions
- Copies coordonnées sur mémoire GPU

Possible, mais lent

```
std::vector<vec3> apply(mat4 Model, mat4 View, mat4 Projection,
                      std::vector<vec3> const& position_in) {
    std::vector<vec3> position_out;
    for (auto const& p : position_in) {
        vec4 p_homog{p, 1.0f};
        vec4 p_ndc_homog = Projection * View * Model * p_homog;
        vec3 p_ndc = vec3{p_homog.x, p_homog.y, p_homog.z} / p_homog.w;
        position_out.push_back(p_ndc);
    }
    return position_out;
}

void main_loop() {
    std::vector<vec3> new_position;
    // Update Model, View, Projection, ...

    // N-operations - monothread
    new_position = apply(Model, View, Projection, mesh.position);

    // N-copy to the GPU
    mesh_drawable.position.update(new_position);

    draw(mesh_drawable);
}
```

Matrices: C++ / paramètres uniformes

Changement de coordonnées via Uniforms:

Simple copie de matrices

Pas de calcul supplémentaire en C++

Calcul des nouvelles position en parallèles sur le GPU, surcout négligeable

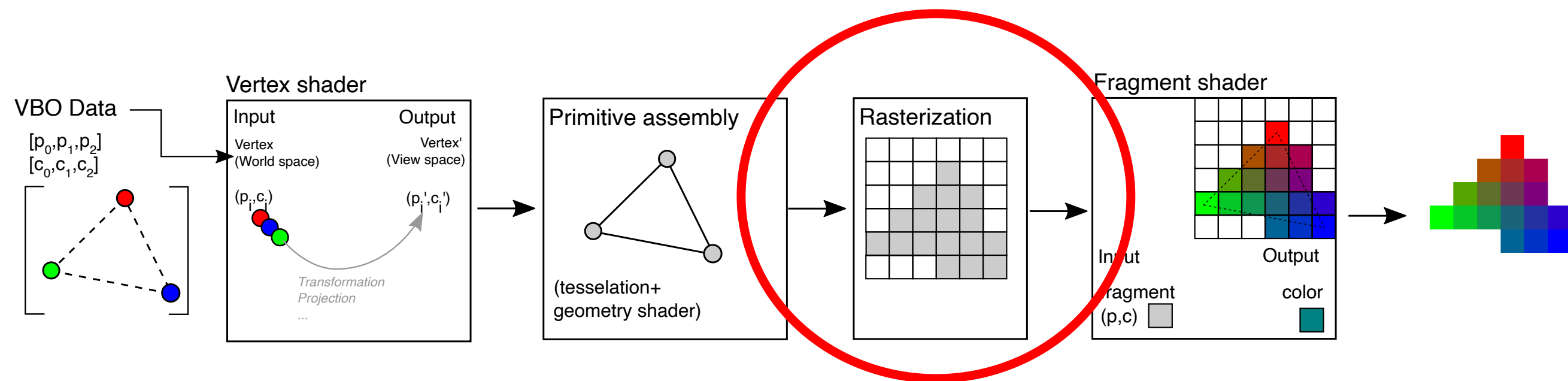
```
void main_loop() {  
    // Update Model, View, Perspective, ...  
  
    // Copy three 4x4 matrices  
    glUniform(shader, Model);  
    glUniform(shader, View);  
    glUniform(shader, Projection);  
  
    draw(mesh_drawable);  
}
```

C++

```
in vec3 vertex_position;           In parallel on all vertices  
uniform mat4 Model;  
uniform mat4 View;  
uniform mat4 Projection;  
void main() {  
    gl_Position = Projection * View * Model * (position,1.0);  
}
```

Vertex shader

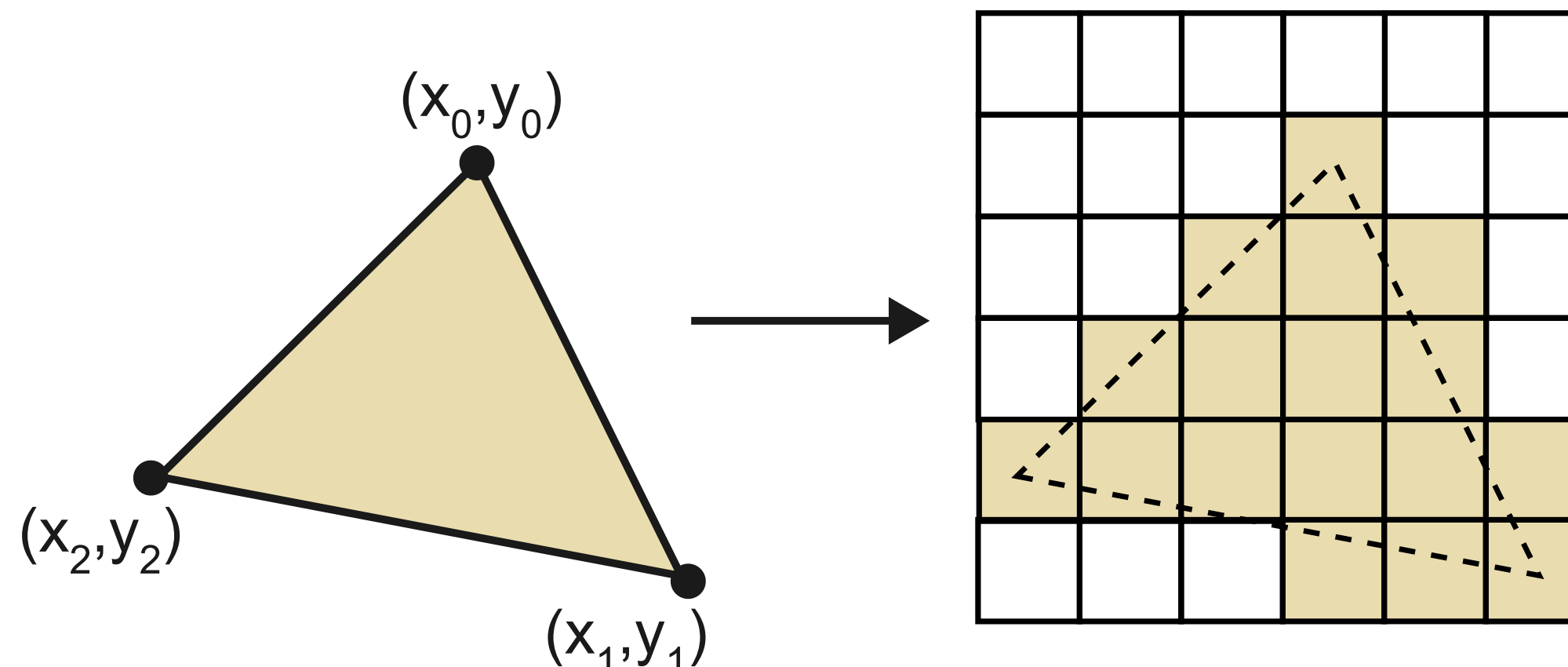
Rasterization



Rasterization

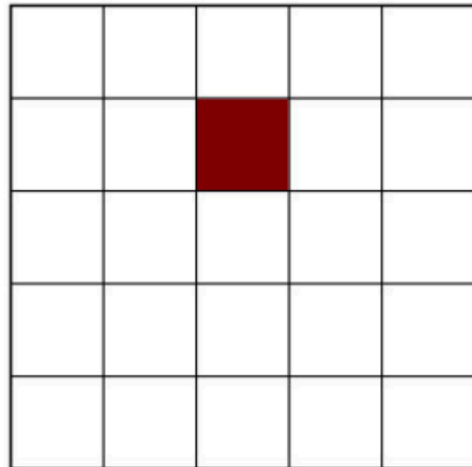
Rasterization / facetisation = conversion de données vectorielles en éléments unitaires: pixels / fragments.
Automatique en OpenGL, non programmable.

Triangle définis par 3 points 2D \Rightarrow Ensemble de coordonnées de pixels



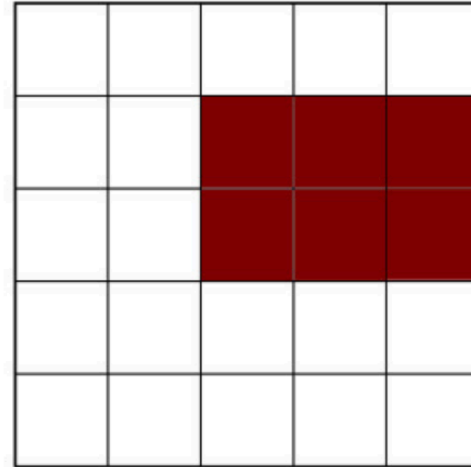
Rasterization de primitives

Point



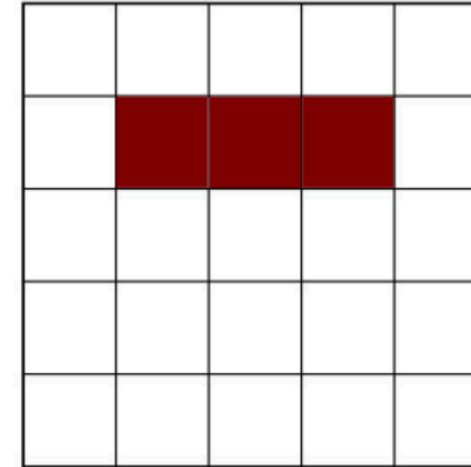
```
im(x0, y0) = c;
```

Rectangle



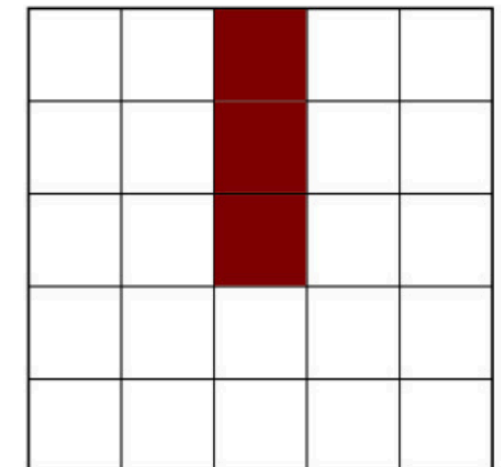
```
for(int kx=x0; kx<x1; kx++)  
  for(int ky=y0; ky<y1; ky++)  
    im(kx, ky) = c;
```

Segment horizontal



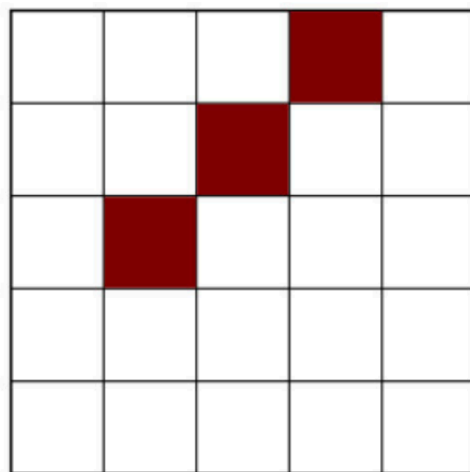
```
for(int kx=x0; kx<x1; kx++)  
  im(kx, y0) = c;
```

Segment vertical



```
for(int ky=y0; ky<y1; ky++)  
  im(x0, ky) = c;
```

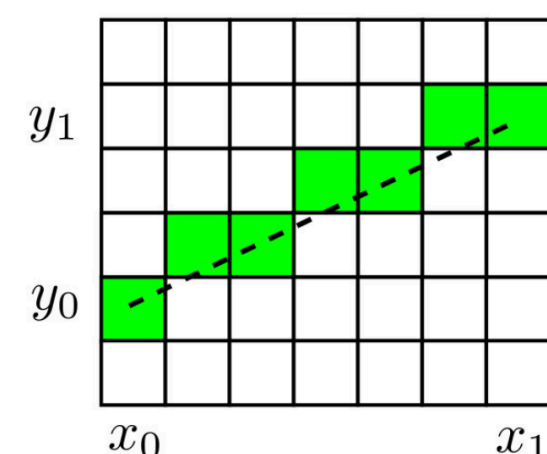
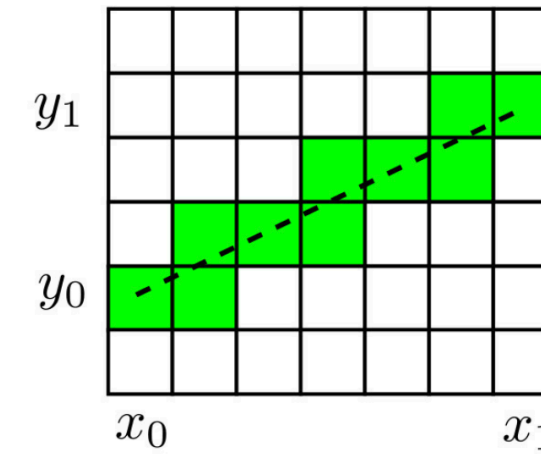
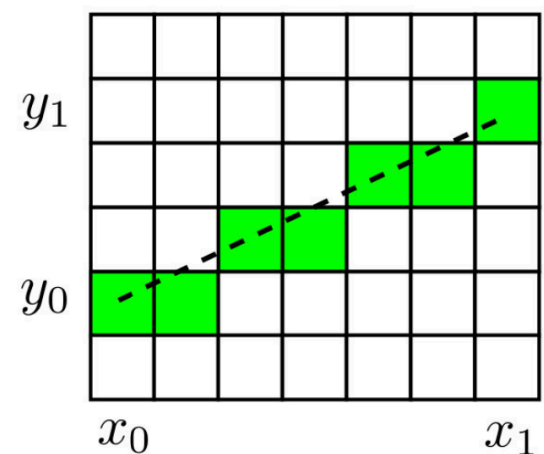
Segment diagonal



```
for(int ky=y0; ky<y1; ky++)  
  im(x0, ky) = c;
```

Segment quelconque

Plusieurs possibilités ?



Segment discret

Algorithme efficace: Algorithm de Bresenham

$$y = a(x - x_0) + y_0$$

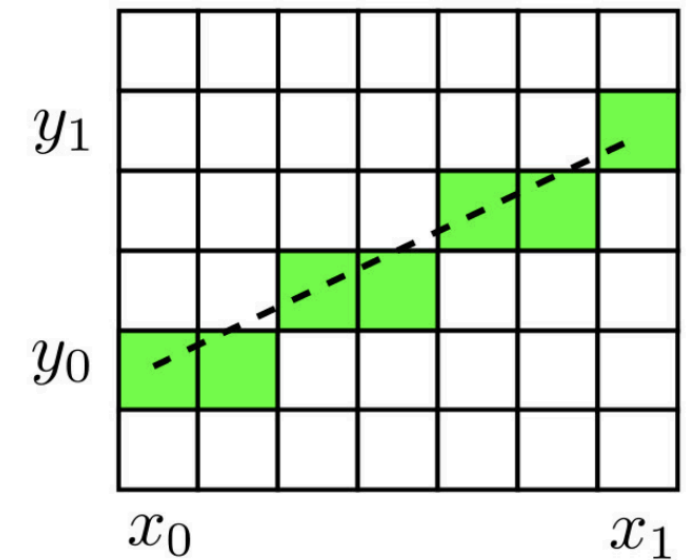
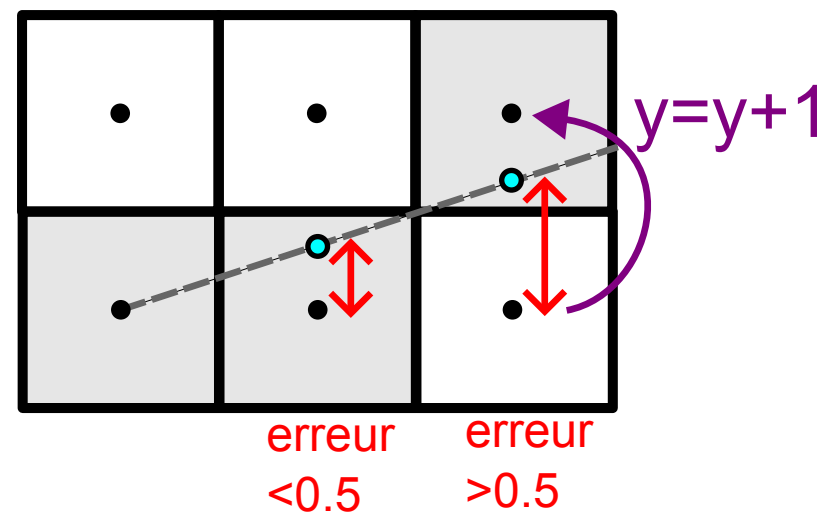
Valeur exacte (x, y) supposée au centre du pixel

Principe:

Avance suivant x de 1 pixel

Evalue si erreur en $y > 0.5$

Si oui: augment y de 1, et corrige erreur



```
int dx=x1-x0, dy=y1-y0;
float a = float(dy)/float(dx);
float erreur = 0.0f;

int y=y0;
for(int x=x0; x<=x1; x++)
{
    im(x, y) = c;
    erreur += a;
    if( erreur >= 0.5f )
    {
        y = y+1;
        erreur = erreur - 1.0f;
    }
}
```

Rem.

- Calcul entièrement sur entier possible en multipliant par $2dx$.
- Version symétrique en fonction des quadrants

Triangle: algorithme scanline

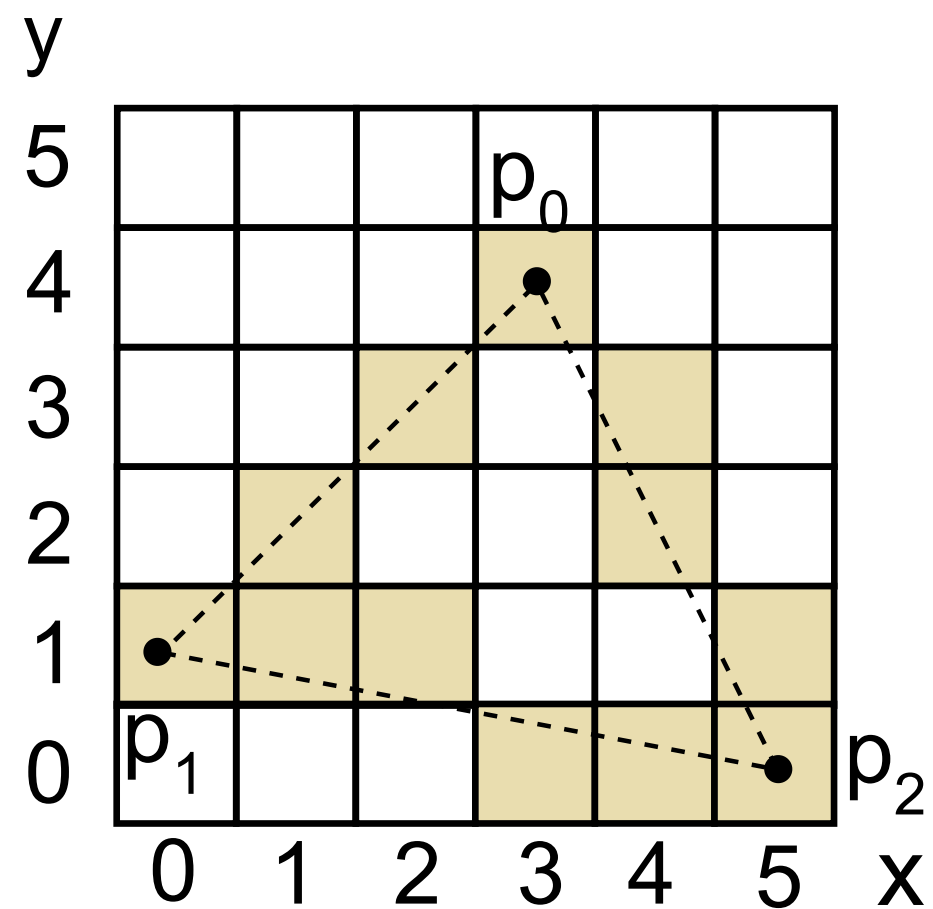
Triangle p_0, p_1, p_2

Parcours les arêtes (p_0, p_1) , (p_1, p_2) , (p_2, p_0)

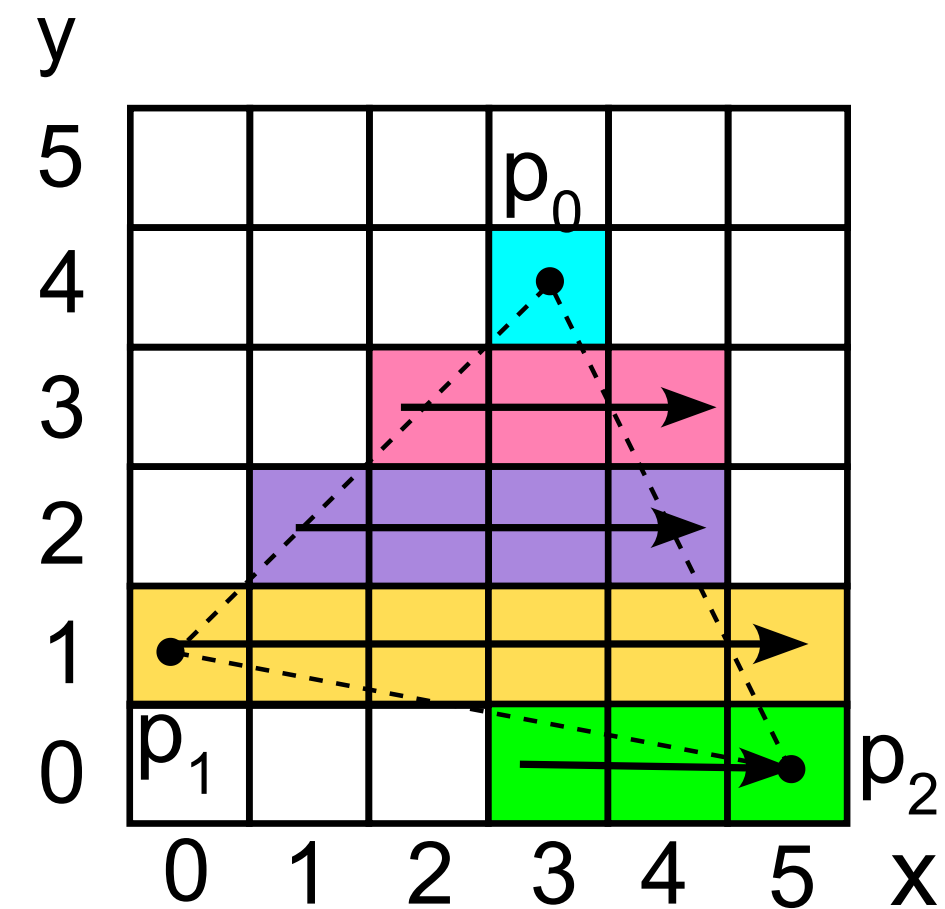
Stocke $\{y, (x_{\min}, x_{\max})\}$

Pour chaque ligne y

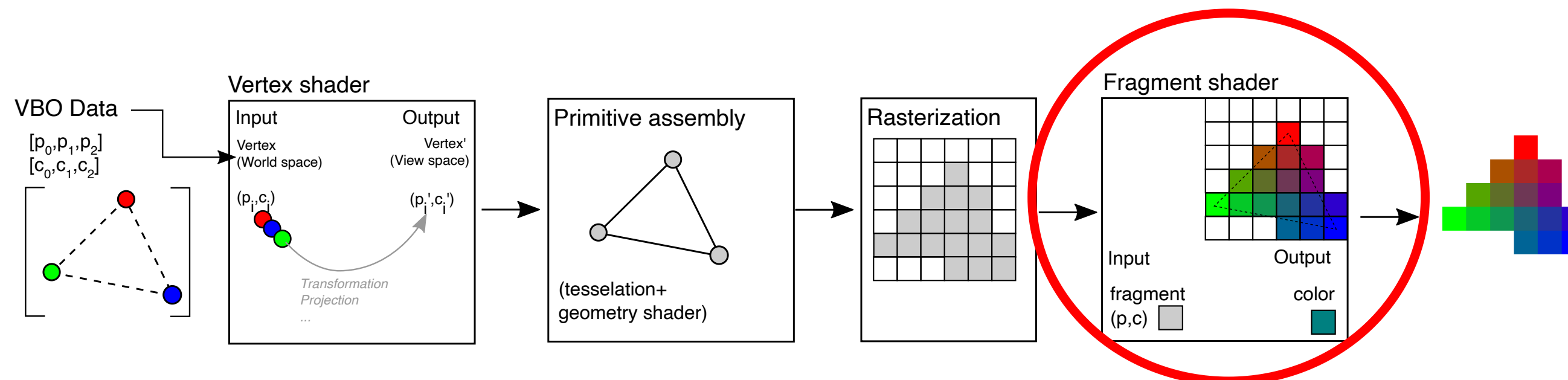
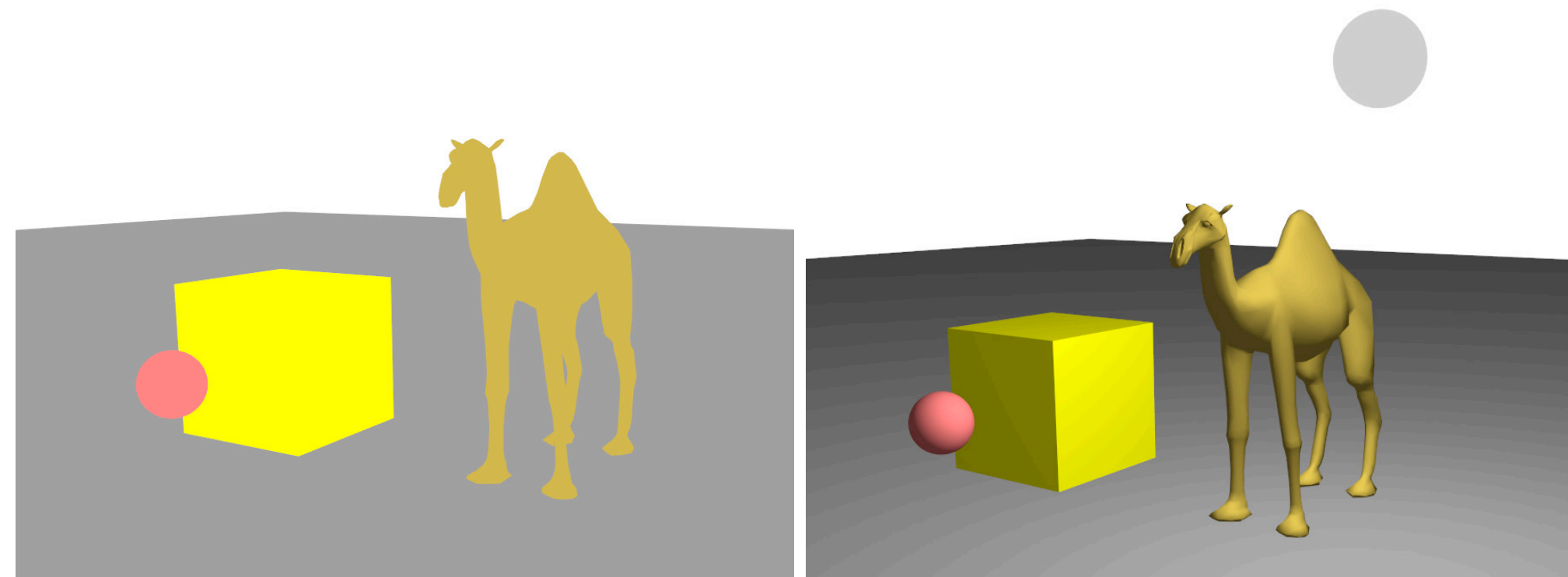
Affiche les segments horizontaux (x_{\min}, x_{\max})



y	x_{\min}	x_{\max}
0	3	5
1	0	5
2	1	4
3	2	4
4	3	3



Shading / Illumination



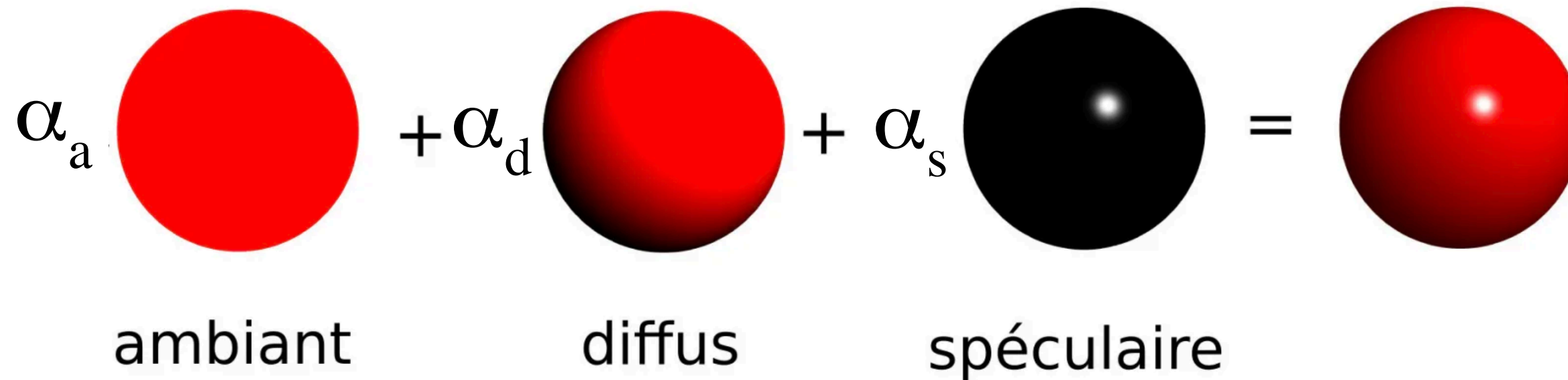
Modèle d'illumination de Gouraud

Trois composantes:

- **Ambiante/Ambiant:** Couleur générale
- **Diffuse/Diffus:** Orientation locale
- **Spéculaire/Specular:** Reflexion, brillance

Dépendance de la couleur:

- Couleur de la source lumineuse $c_l = (r_l, g_l, b_l) \in [0, 1]^3$
- Couleur de la surface $c_s = (r_s, g_s, b_s) \in [0, 1]^3$



Illumination

Couleur ambiante

$$c_a = \alpha_a c_l c_s$$

$\alpha_a \in [0, 1]$: Coefficient ambiant

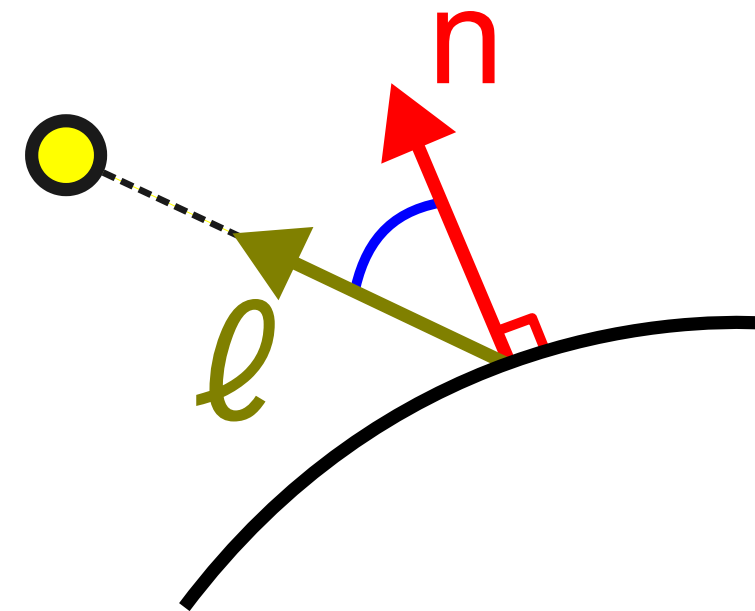
Couleur diffuse

Orientation locale de la surface / source de lumière

$$c_d = \alpha_d (n \cdot c_l)_+ c_l c_s$$

$\alpha_d \in [0, 1]$: Coefficient diffus

$(n \cdot c_l)_+ = \max(n \cdot c_l, 0) \in [0, 1]$: Composante diffuse



Illumination: Speculaire

Couleur spéculaire

Réflexion source lumineuse sur la camera

$$c_s = \alpha_s (r \cdot o)_+^\beta c_\ell$$

$\alpha_s \in [0, 1]$: Coefficient spéculaire

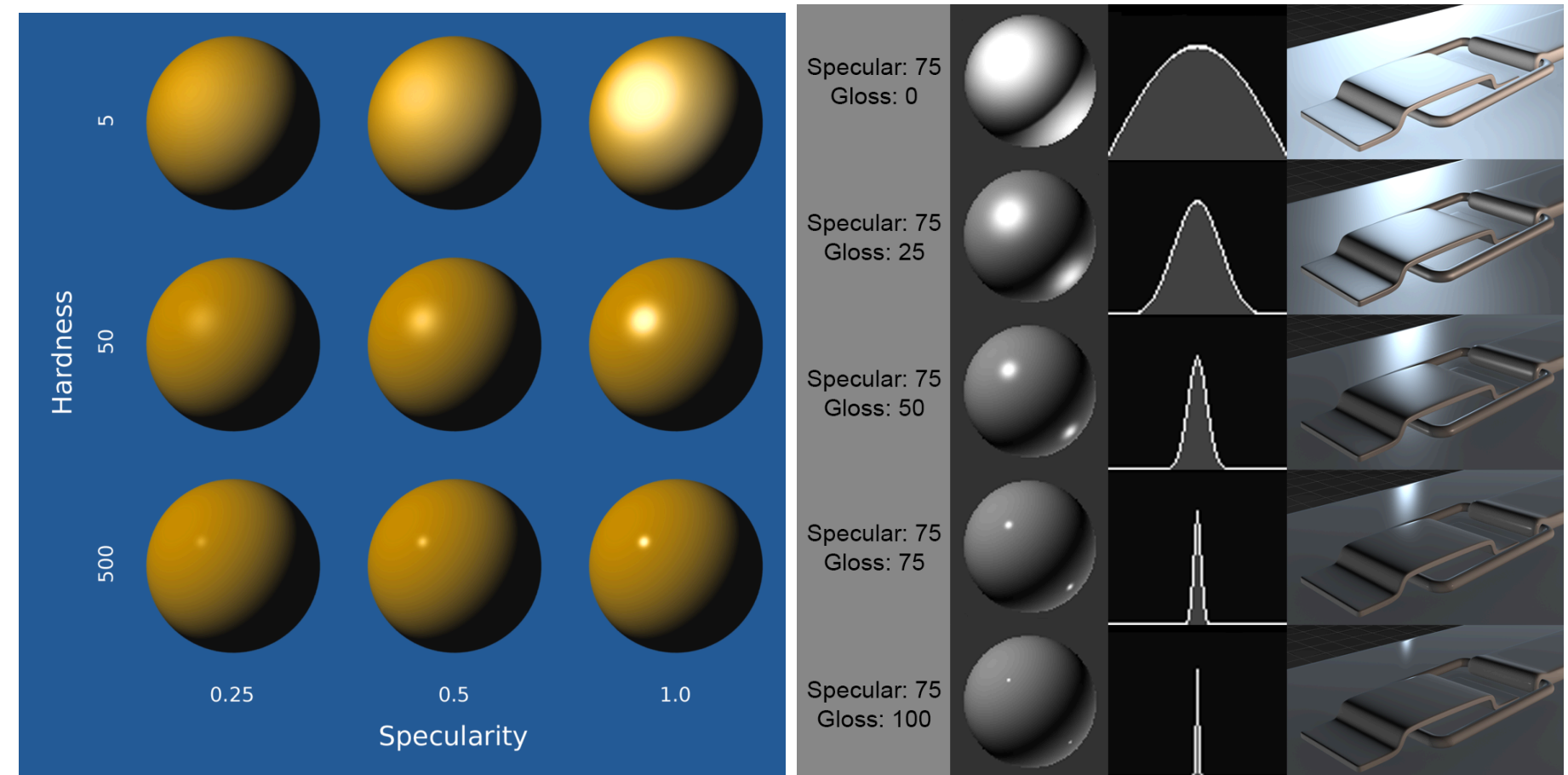
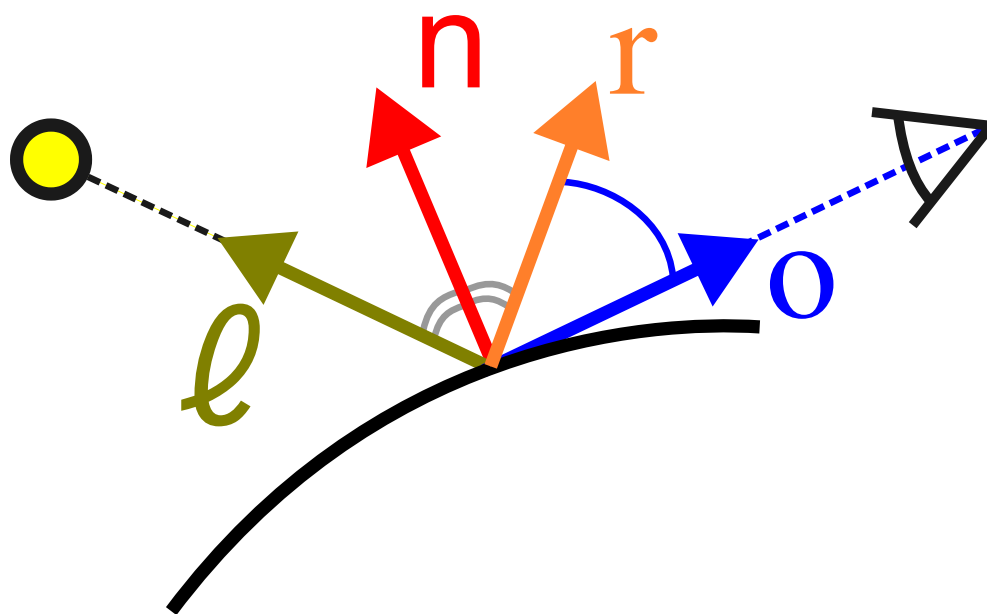
$\beta \simeq 64 - 256$: Hardness / Glossiness

r : réflexion de la source lumineuse / normale

$$r = 2(\ell \cdot n)\ell - \ell$$

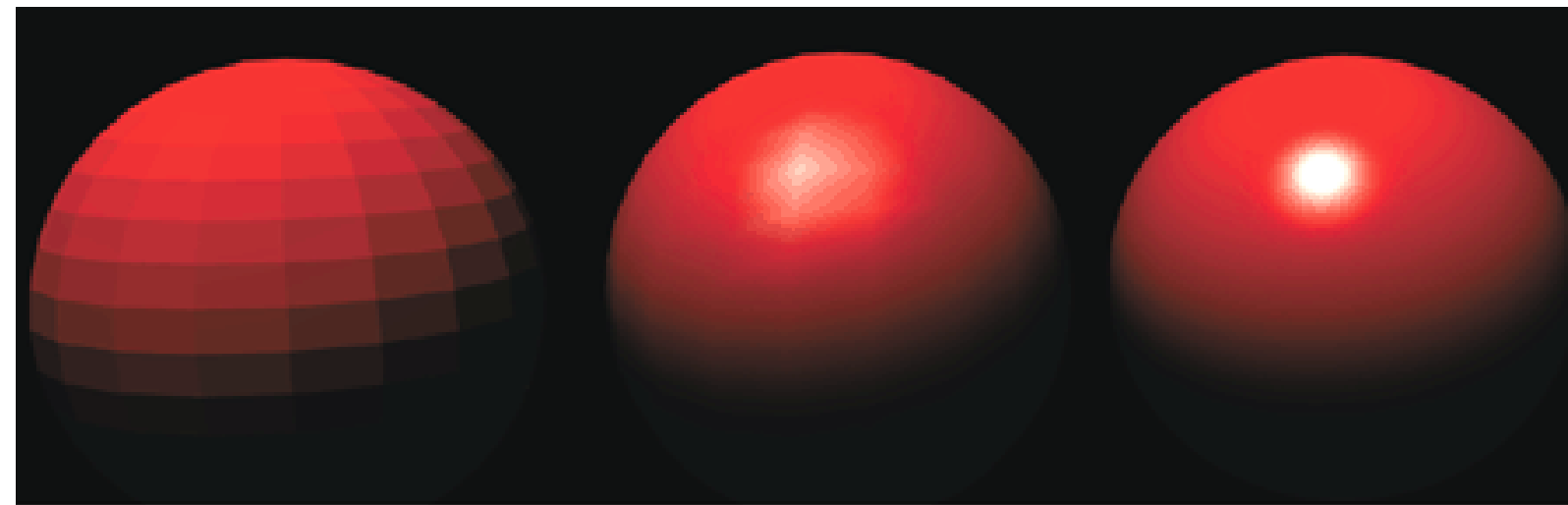
En glsl: $r = \text{reflect}(-\ell, n)$

Ne dépend que de la couleur de la source lumineuse



Interpolation de l'illumination: Flat, Gouraud, Phong

- **Flat:** Couleur uniforme par triangle
Calcul illumination dans vertex shader
- **Gouraud:** Calcul illumination par sommet, interpolation linéaire de la couleur dans le triangle.
Calcul illumination dans vertex shader
- **Phong:** Calcul illumination pour chaque fragment
Calcul illumination dans le fragment shader



Flat

Gouraud

Phong

Effets d'illumination

Sources multiples

$$C = \sum_i \alpha_a C_{l_i} C_s + \alpha_d (n \cdot C_{l_i})_+ C_{l_i} C_s + \alpha_s (r_i \cdot n)_+^\beta C_{l_i}$$

Atténuation d'intensité fonction de la distance

$$C_\ell(p) = \left(1 - \min\left(\frac{\|p - p_\ell\|}{d_{att}^0}, 1\right)\right) C_\ell^0$$

$p \in \mathbb{R}^3$: position sur la surface

$p_\ell \in \mathbb{R}^3$: centre de la lumière

d_{att}^0 : distance caractéristique d'atténuation

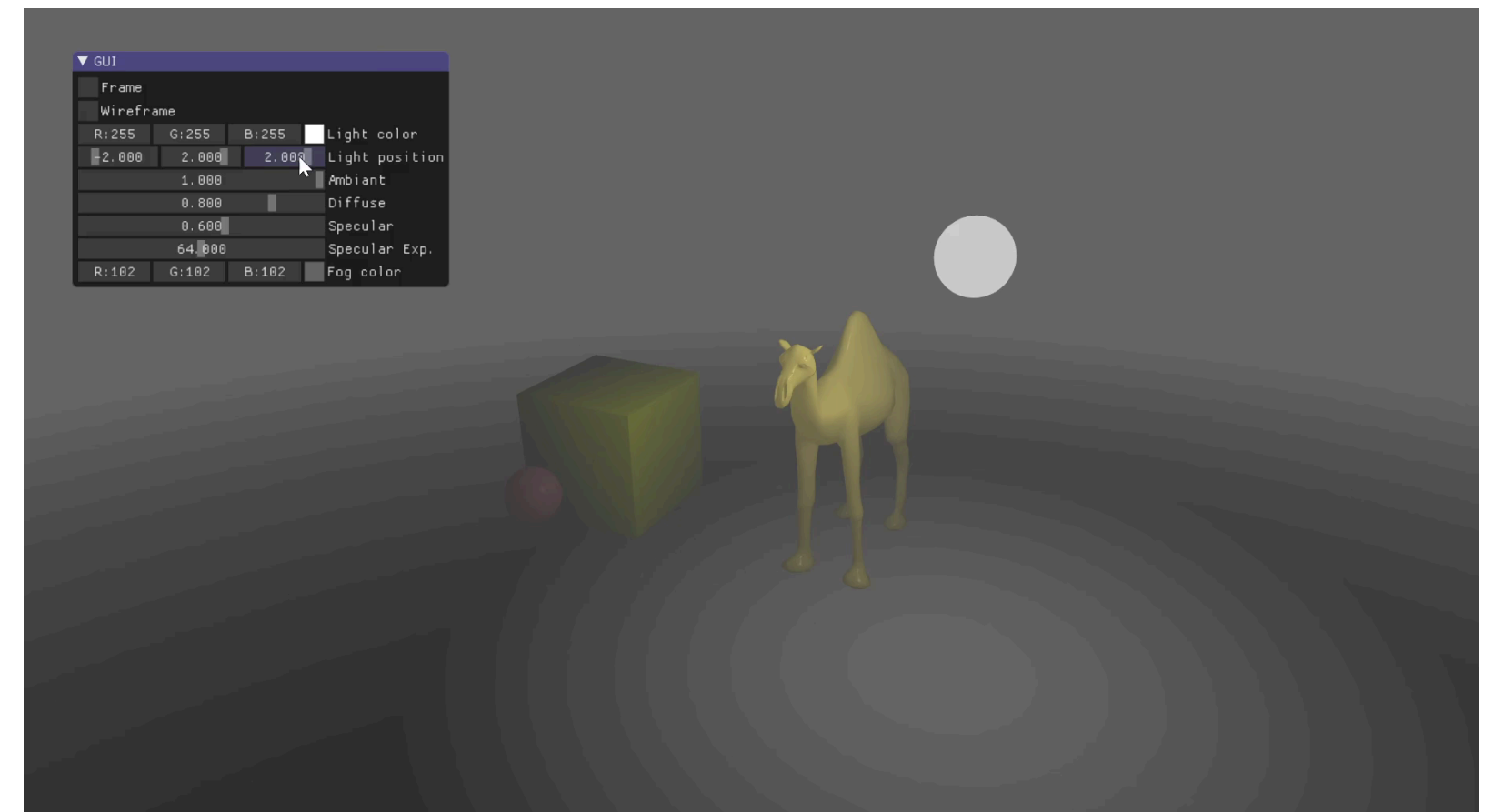
C_ℓ^0 : couleur de la lumière à la source

Effet de brume/fog

Mélange de couleur en fonction de la distance à la caméra

$$C = (1 - \gamma(p)) C_s C_\ell + \gamma(p) C_{fog}$$

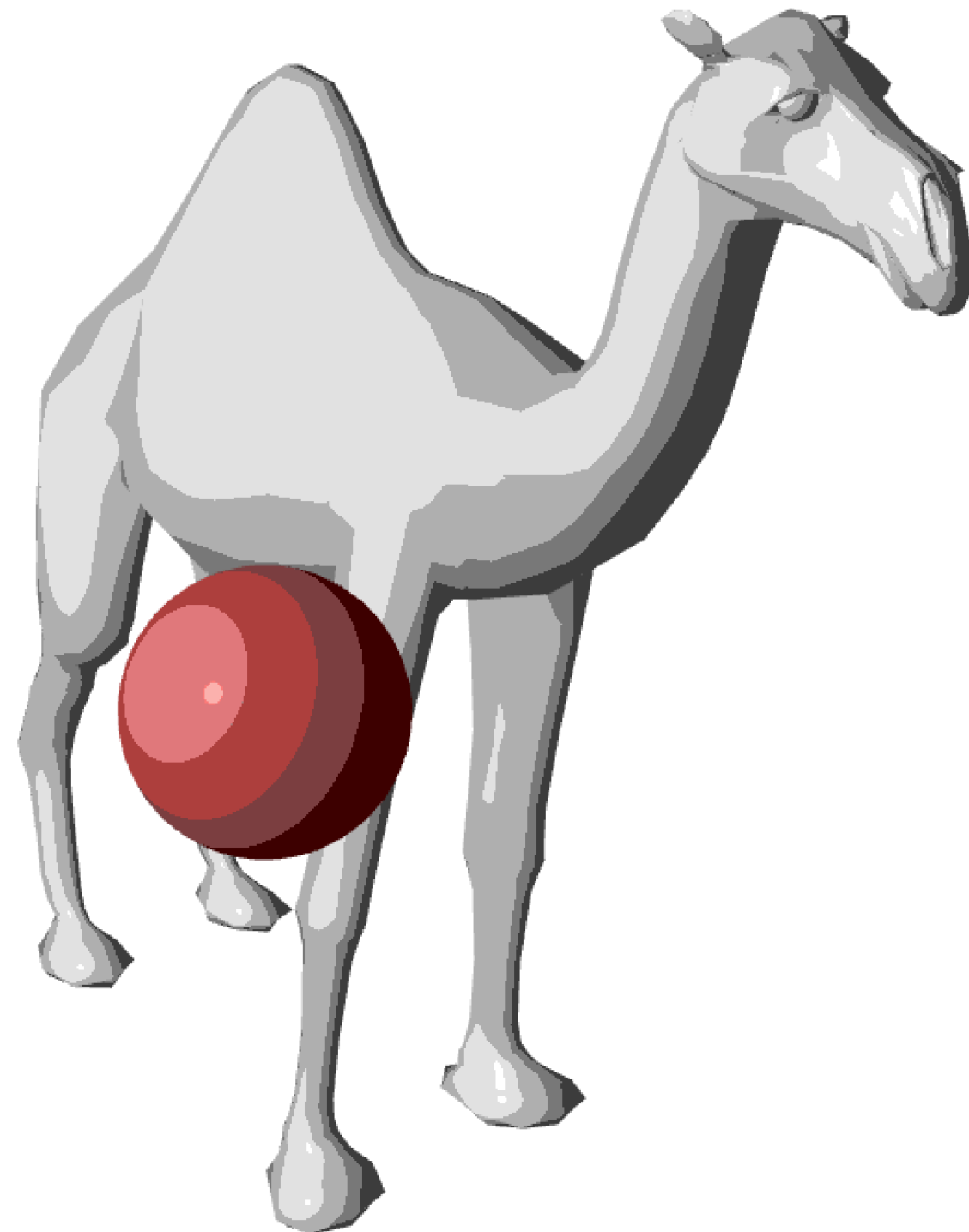
$$\gamma(p) = \min(\|p - p_{eye}\| / d_{fog}, 1)$$



Example effet: Toon Shading

Sous échantillonne les valeurs de couleurs

$$C_{toon} = \text{float}(N \text{ int}(C/N))$$



Buffer de profondeur / Depth Buffer

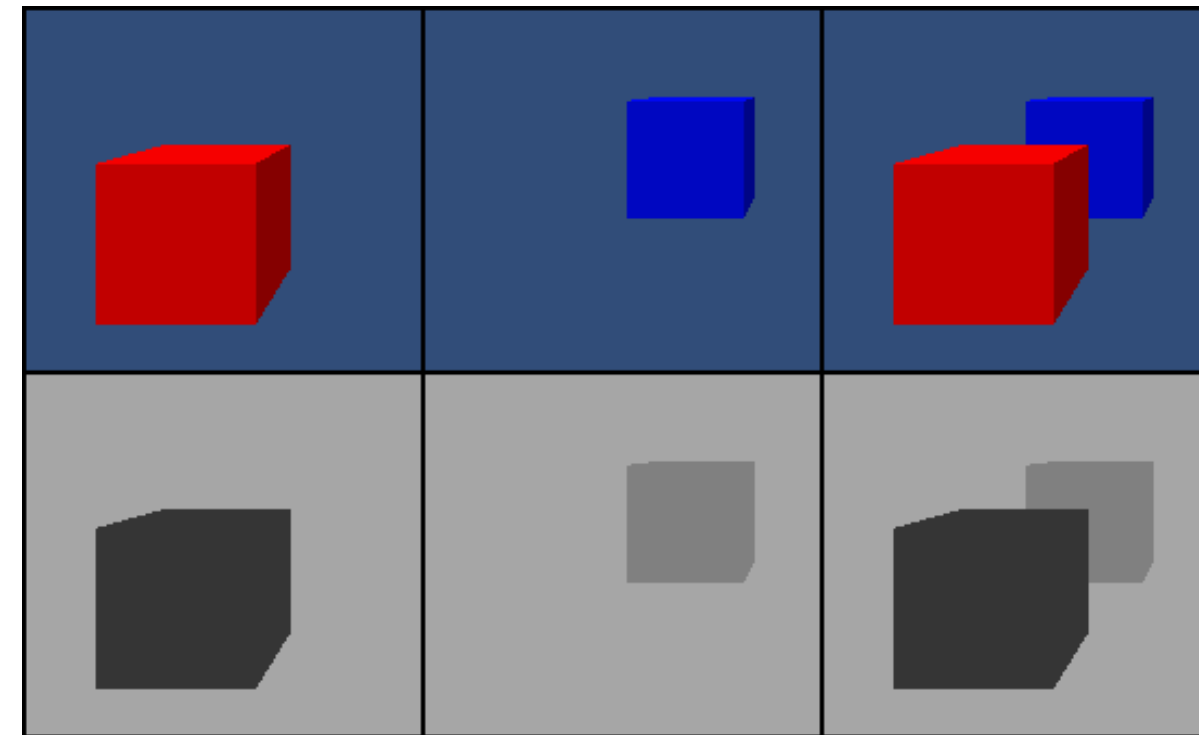


Buffer de profondeur / Depth Buffer

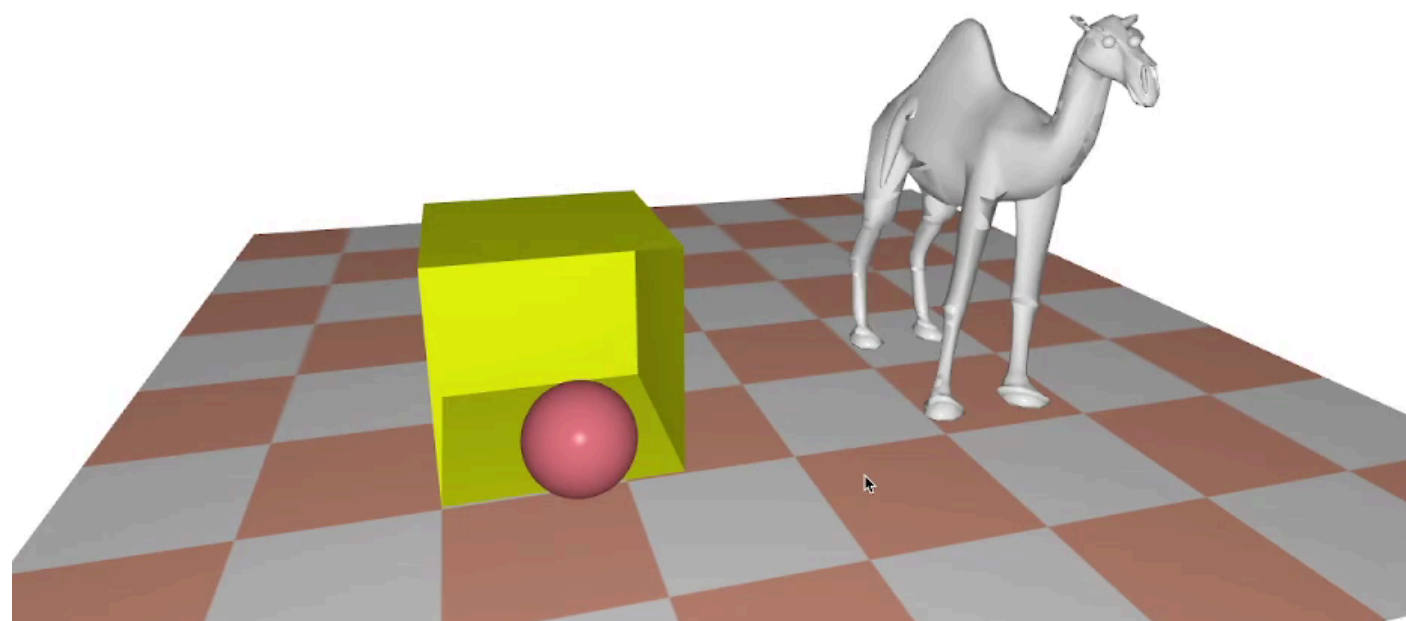
- Gestion d'un buffer/image de profondeur
 - Depth-buffer / Z-buffer
 - Stocke la valeur z_{ndc} pour chaque pixel

Algorithme

```
def Draw(position, couleur, z, image, Zbuffer):  
    if Zbuffer(position) < z:  
        image(position) = couleur  
    # else draw nothing
```

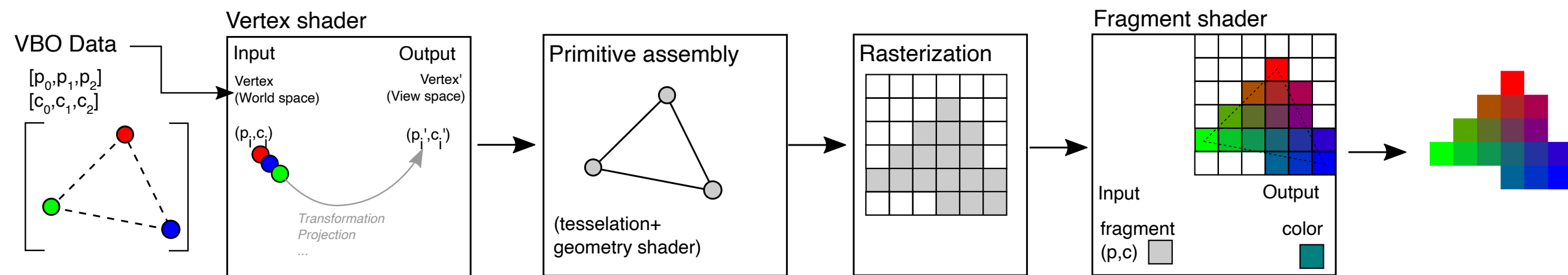


Depth buffer

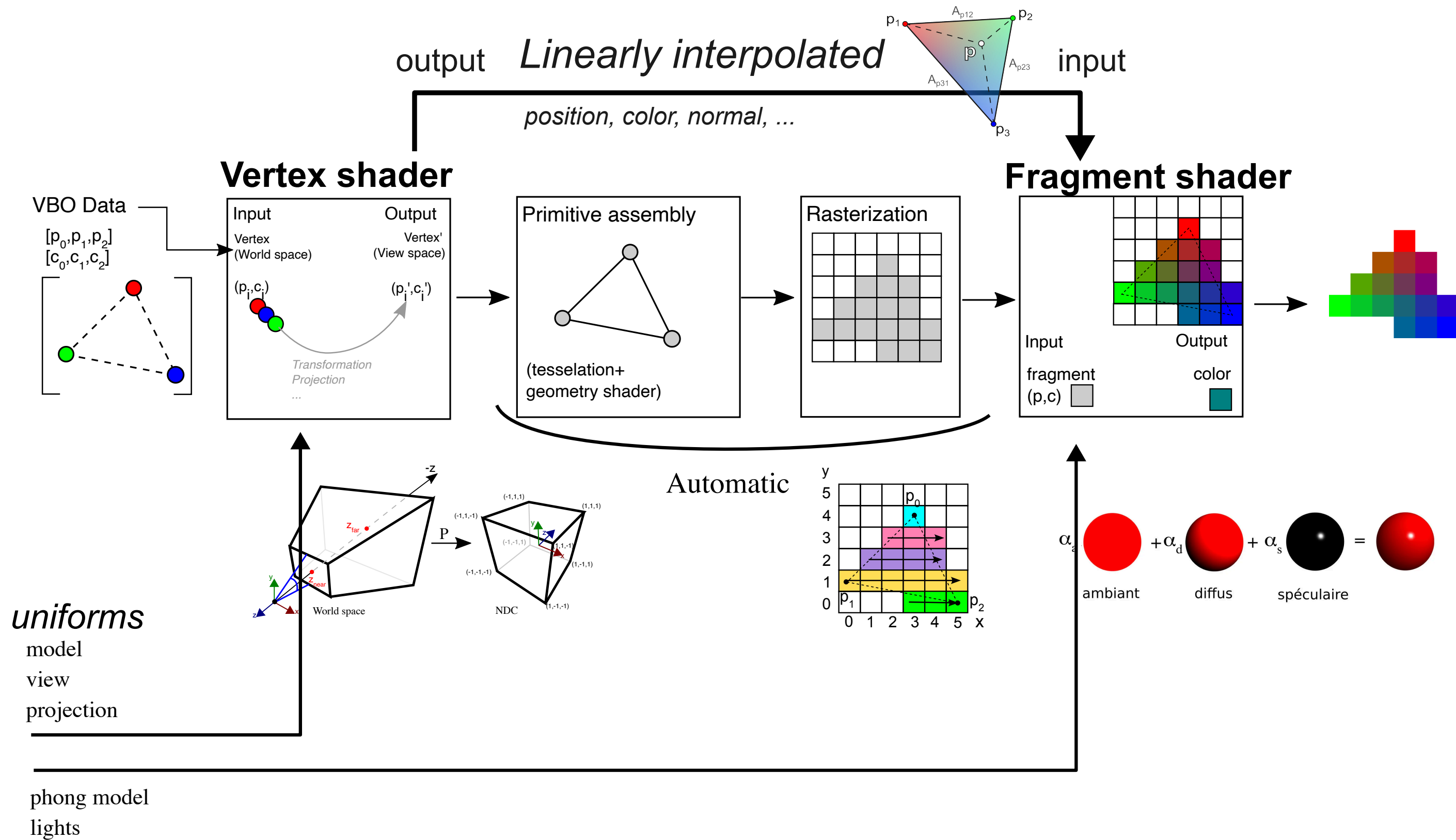


Without Depth Buffer

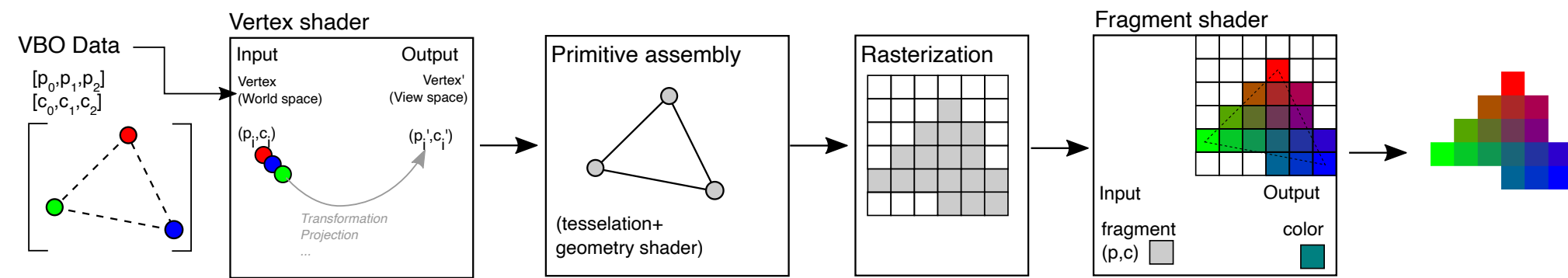
Shaders: résumé



Shaders Résumé



Shaders Résumé: Exemple de code



```

layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_normal;
layout (location = 2) in vec3 vertex_color;

out struct fragment_data {
    vec3 position;
    vec3 normal;
    vec3 color;
} fragment;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    vec4 position = model * vec4(vertex_position, 1.0);

    mat4 modelNormal = transpose(inverse(model));
    vec4 normal = modelNormal * vec4(vertex_normal, 0.0);

    fragment.position = position.xyz;
    fragment.normal = normal.xyz;
    fragment.color = vertex_color;

    gl_Position = projection * view * position;
}

```

```

// Interpolated values on the fragment
in struct fragment_data {
    vec3 position;
    vec3 normal;
    vec3 color;
} fragment;

uniform vec3 light_position;
uniform vec3 light_color;

layout(location=0) out vec4 FragColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {

    vec3 N = normalize(fragment.normal);
    vec3 L = normalize(light_position-fragment.position);

    float diffuse_magnitude = max(dot(N,L),0.0);
    // ...

    vec3 c = a_ambient * fragment.color*light_color;
    c = c + a_diffuse*diffuse_magnitude*fragment.color*light_color;
    c = c + a_specular*specular_magnitude*light_color;

    FragColor = vec4(c,1.0);
}

```