

# CPE4

## pipeline graphique et OpenGL

J.C. lehl

November 27, 2017

# Introduction

## pipeline graphique :

- ▶ *produire* une image d'une scène 3D,
- ▶ ensemble de traitements pour afficher des points, lignes, triangles,
- ▶ et *calculer* la couleur de chaque pixel...

## OpenGL :

- ▶ expose *un* pipeline graphique,
- ▶ du matériel spécialisé exécute les traitements,
- ▶ en parallèle...

# Introduction

## OpenGL :

- ▶ est une API / librairie permettant de configurer *un* pipeline graphique,
- ▶ pour dessiner des objets,
- ▶ mais aussi pour faire des calculs,
- ▶ en utilisant les processeurs parallèles d'une carte graphique.

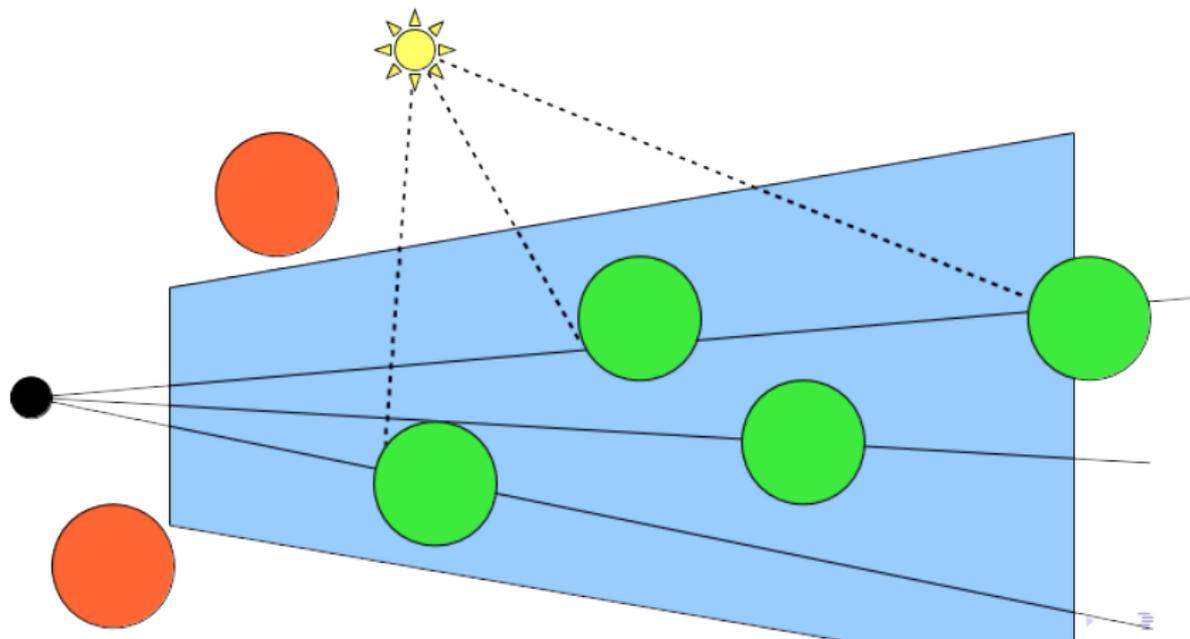
nombreuses applications : jeux, simulateurs, visualisation médicale, scientifique, réalité augmentée / virtuelle, web, apprentissage, etc...

# afficher des objets

décrire une *scène* 3D :

- ▶ chaque objet est placé et orienté dans l'espace, le "monde",
- ▶ la camera observe une région de l'espace,
- ▶ dessiner une image des objets *visibles* par la camera.

## scène 3D + camera



# afficher des objets

plusieurs problèmes :

- ▶ problème 1 : déterminer où se trouve l'objet (par rapport à la camera),
- ▶ problème 2 : déterminer l'ensemble de pixels (correspondant à la forme de l'objet),
- ▶ problème 3 : calculer la couleur de chaque pixel.

## afficher des objets

### 2 organisations :

- ▶ pour chaque objet : déterminer l'ensemble de pixels, (que se passe-t-il lorsque plusieurs objets se "dessinent" sur le même pixel ?)
- ▶ pour chaque pixel : trouver l'objet visible,

trouver l'objet visible pour chaque pixel : trouver l'objet le plus *proche* de la camera.

OpenGL représente le 1<sup>er</sup> pipeline.

# OpenGL

## c'est quoi ?

- ▶ une api 3D...
- ▶ un ensemble de fonctions permettant de paramétrer un pipeline d'affichage,
- ▶ les étapes du pipeline sont réalisées par du matériel spécialisé (carte graphique).

il vaut mieux avoir une idée des différentes étapes pour comprendre comment utiliser OpenGL.

## pipeline fragmentation / rasterization

### 2 étapes principales :

- ▶ partie 1, géométrie :  
prépare le dessin des primitives (triangles), projette les sommets dans l'image,
- ▶ partie 2, pixels :  
dessine la primitive, donne une couleur à chaque pixel occupé par la primitive dans l'image.

une carte graphique ne sait dessiner que des points, des lignes et des triangles... donc il faut trianguler la surface des objets pour les dessiner.

# triangler la surface des objets

représenter la surface des objets :

- ▶ découper la surface en triangles,
- ▶ donner les coordonnées de chaque sommet, de chaque triangle.

# triangler la surface des objets

## 1 triangle :

- ▶ 3 sommets,
- ▶ dans quel ordre ? abc, acb, ou autre chose ?
- ▶ sens trigo ou sens horaire, vu depuis l'extérieur de l'objet...

le pipeline partie 2 ne dessine que les triangles orientés correctement...

donc, il faut décrire la surface des objets, avec une orientation cohérente des sommets des triangles...

## triangler la surface des objets

un carré :

- ▶ sommets  $a = \{0, 0\}$ ,  $b = \{1, 0\}$ ,  $c = \{1, 1\}$ ,  $d = \{0, 1\}$ ,
- ▶ 2 triangles dans le sens trigo :
- ▶  $abc + acd$ , ou une autre paire ?
- ▶  $abc$  ou n'importe quelle permutation qui ne change pas l'orientation :  $abc = bca = cab$

## placer / orienter les objets

une représentation pour exprimer :

- ▶ une translation,
- ▶ une rotation,
- ▶ une "projection",
- ▶ un changement de repère.

et tout combiner / composer ensemble ?

## transformations affines et espace homogène

toutes ces transformations se représentent sous forme d'une matrice ... sauf la translation et la projection.

idée

comment représenter une translation avec une matrice ?

espace homogène et matrices  $4 \times 4$

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

## points homogènes

$$p_h = w \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

$$p_h = \begin{bmatrix} x \\ y \\ z \\ w \neq 0 \end{bmatrix}$$

on retrouve le point réel associé au point homogène en divisant par  $w$  :

$$p = p_h / w = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

## vecteurs homogènes

$$v = \begin{bmatrix} x \\ y \\ z \\ w \equiv 0 \end{bmatrix}$$

un vecteur ne subit pas de "translation".

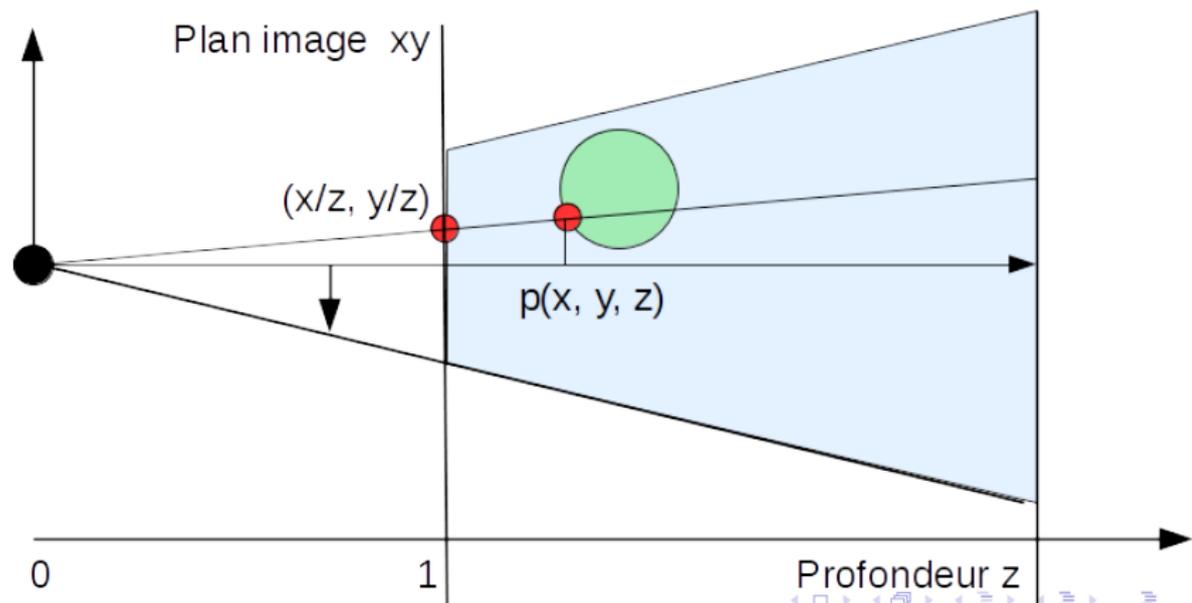
## transformation affine et projection

"projection" perspective sur le plan  $z = d$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$$

+ retrouver le point réel associé =  $\begin{bmatrix} d \frac{x}{z} \\ d \frac{y}{z} \\ d \\ 1 \end{bmatrix}$

## rappel : projection



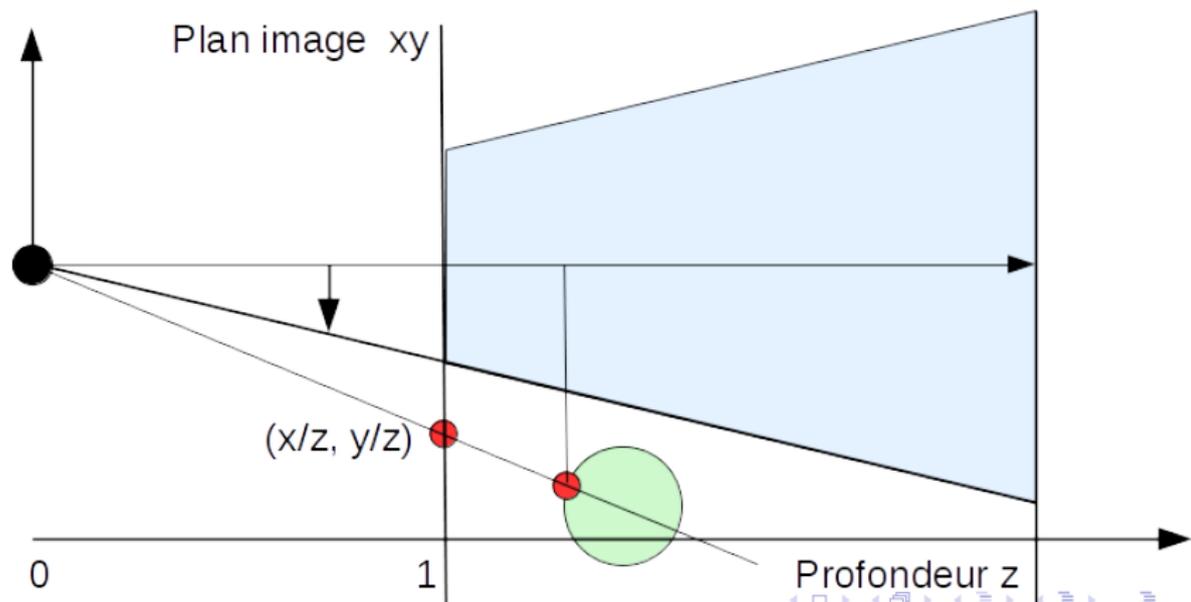
## rappel : projection

projeter  $p(x, y, z)$  :

- ▶  $(xp, yp) = (x/z, y/z, z/z) \equiv (x/z, y/z, 1)$ ,
- ▶ si le centre de projection est à l'origine du repère,
- ▶ sur quel pixel ?

par convention : le plan image est à  $z = 1$

## rappel : projection



## rappel : projection

### projection et image :

- ▶ un point se projette sur l'image si :
- ▶  $-1 < x/z < 1$ ,
- ▶  $-1 < y/z < 1$ ,
- ▶ coordonnées du pixel dans l'image *largeur* × *hauteur* pixels :
- ▶  $px = (x/z + 1) \times \text{largeur}/2$ ,
- ▶  $py = (y/z + 1) \times \text{hauteur}/2$ .

on peut aussi définir un angle d'ouverture pour *zoomer*  
sur un objet... noté *fov* (field of view)

## rappel : projection

ensemble des points *visibles / observables* :

- ▶ un point se projette sur l'image si :
- ▶  $-1 < x/z < 1$ ,
- ▶  $-1 < y/z < 1$ ,
- ▶ les points associés à un pixel se trouvent dans le volume :
- ▶  $-z < x < z$ ,
- ▶  $-z < y < z$ .

et pour les points derrière la camera ? ( $z < 0$ )

noté *frustum*.

## rappel : projection

et alors ?

- ▶ c'est exactement ce que fait la matrice homogène,
- ▶ avec tous les paramètres : fov, distance proche / loin
- ▶ ce qui la rend inversible...

## bilan

toutes les transformations se représentent sous forme de matrices.

plusieurs problèmes :

- ▶ problème 1 : déterminer où se trouve l'objet (par rapport à la camera),
- ▶ problème 2 : déterminer sur quels pixels se projettent les sommets de la surface des objets,
- ▶ problème 3 : ...

cf pipeline partie 1,

prépare le dessin des primitives (triangles), projette les sommets dans l'image

## placer / orienter les objets

les coordonnées des sommets :

- ▶ dans quel repère ?
- ▶ les objets sont créés séparément : *repère local*,
- ▶ puis placés et orientés dans le monde : *repère global / monde*,
- ▶ puis observés par la camera : *repère camera*,
- ▶ puis projetés : *repère projectif*,
- ▶ puis dessinés dans l'image : *repère image*.

un sommet à des coordonnées dans 4 ou 5 repères différents...

## composition de transformations

plusieurs repères :

objet  $\rightarrow$  scène  $\rightarrow$  observateur  $\rightarrow$  projection  $\rightarrow$  image  
 $M$   $V$   $P$   $I$

déterminer directement les coordonnées d'un point de l'objet dans le repère projectif :  $q = P(V(Mp))$

transformation globale :  $q = Tp$  avec  $T = P \cdot V \cdot M$

passer d'un repère à l'autre avec l'inverse de la transformation :

$$p = T^{-1}q$$

# transformations

## pipeline partie 1, géométrie :

- ▶ transformer les coordonnées des sommets,
- ▶ du repère local de l'objet,
- ▶ jusqu'au repère projectif,
- ▶ (la partie 2 s'occupe de la dernière étape)

## openGL et les matrices

et alors ?

- ▶ openGL doit transformer les sommets dans le repère projectif pour dessiner les triangles,
- ▶ donc il faut lui fournir la "bonne" transformation :
- ▶ en général, le passage du repère local au repère projectif,  $P \times V \times M$ ,
- ▶ et les dimensions de l'image, pour calculer la matrice  $I$ .

# vertex shader

qu'est ce que c'est ?

- ▶ une fonction exécutée pour chaque sommet, par les processeurs de la carte graphique,
- ▶ *doit* renvoyer les coordonnées dans le repère projectif,
- ▶ pour que la partie 2 du pipeline fonctionne correctement.

les shaders sont écrits en GLSL, un langage proche du C/C++.

## vertex shader

paramètres en entrée :

- ▶ uniforms : valeurs transmises par l'application,
- ▶ constantes : comme d'habitude,
- ▶ attributs de sommet : coordonnées dans le repère local.

sorties :

- ▶ `vec4 gl_Position` : coordonnées du sommet dans le repère projectif,
- ▶ `varyings` : valeurs optionnelles pour le fragment shader, cf partie 2.

## vertex shader : exemple

```
#version 330    // version de GLSL

// fonction principale du vertex shader
void main( )
{
    // declare un vecteur 4 composantes
    vec4 position= vec4(0, 0, 0, 1);

    // resultat obligatoire : coordonnees dans le repere projectif
    gl_Position= position;
}
```

## vertex shader : exemple

```
#version 330    // version de GLSL

// matrice de transformation local vers projectif
uniform mat4.mvpMatrix;
// uniform: declare une variable initialisee par l'application

const float deplace= 0.5;    // constante

// fonction principale du vertex shader
void main( )
{
    // declare un vecteur 4 composantes
    vec4 position= vec4(0, 0, 0, 1);
    // deplace le sommet
    position.x= position.x + deplace;

    // resultat obligatoire : coordonnees dans le repere projectif
    // produit matrice * vecteur, transforme le sommet
    gl_Position=.mvpMatrix * position;
}
```

## vertex shader : exemple

```
#version 330    // version de GLSL

// matrice de transformation local vers projectif
uniform mat4.mvpMatrix;
// uniform: declare une variable initialisee par l'application

// coordonnees du sommet
in vec4 position;
// in: declare une entree du shader, un attribut du sommet,
// configure par l'application

// fonction principale du vertex shader
void main( )
{
    // resultat obligatoire : coordonnees dans le repere projectif
    // produit matrice * vecteur, transforme le sommet
    gl_Position =.mvpMatrix * position;
}
```

# et alors ?

utiliser OpenGL :

- ▶ décrire la surface des objets :  
triangles + coordonnées des sommets
- ▶ ordre / orientation des triangles,
- ▶ transformation du repère local vers repère projectif,
- ▶ c'est un shader qui fait le calcul,
- ▶ mais il faut donner toutes ces informations à OpenGL.

## et alors ?

- ▶ les coordonnées des sommets doivent être stockés dans la mémoire de la carte graphique,
- ▶ + couleur, normale, etc,
- ▶ il faut stocker les *attributs* dans la mémoire de la carte graphique,
- ▶ et "expliquer" à OpenGL comment trouver les informations de chaque sommet,
- ▶ le shader doit être compilé,
- ▶ décrire l'orientation des triangles,
- ▶ l'image résultat est stockée sur la carte graphique,
- ▶ et il faut l'afficher...

et alors ?

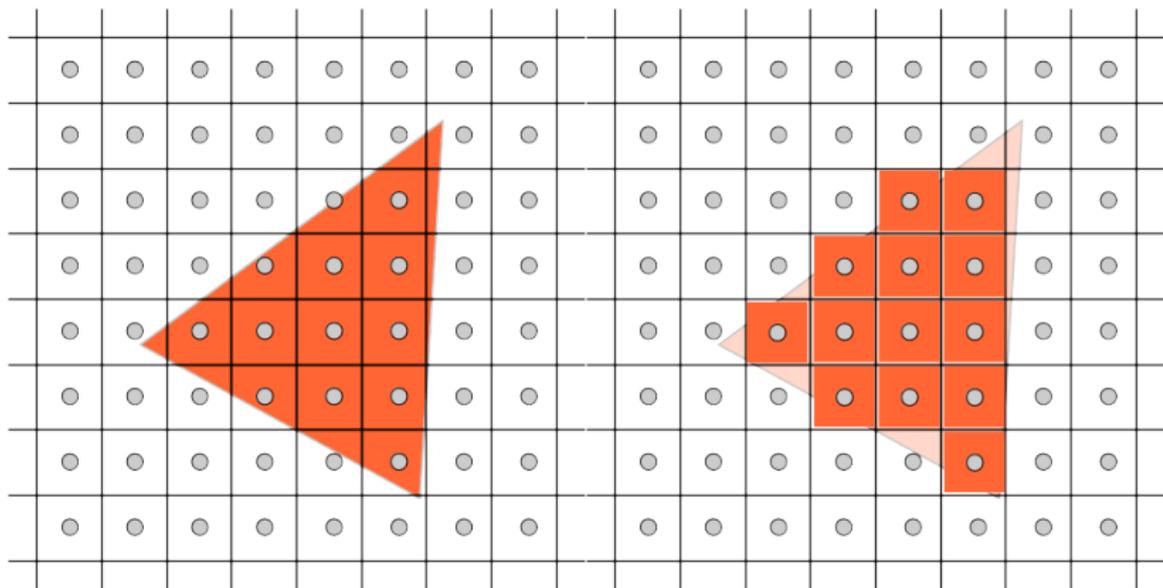
et on a toujours rien dessiné...  
cf partie 2 du pipeline

## dessiner un triangle

dessiner un triangle :

- ▶ on connaît les coordonnées des 3 sommets,  
(dans le repère projectif)
- ▶ vérifier qu'ils correspondent à des pixels de l'image,
- ▶ et trouver tous les pixels de l'image qui sont à l'intérieur du triangle.

## dessiner un triangle

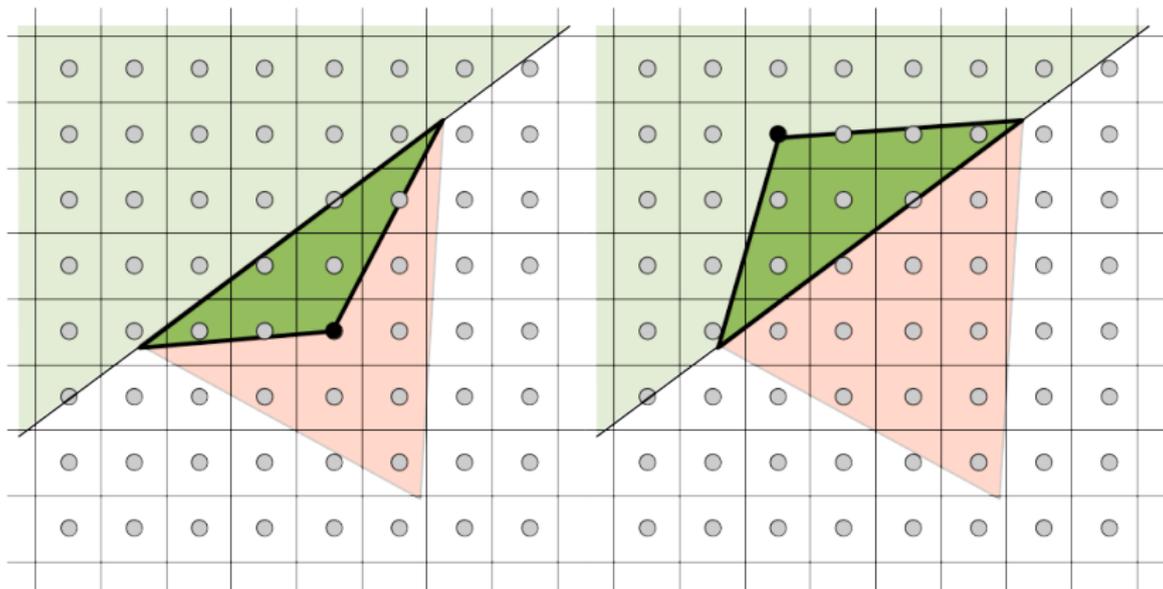


## comment ça marche ?

très simplement :

- ▶ vérifier que chaque pixel est à l'intérieur du triangle ?
- ▶ idée : si le pixel est du bon coté de chaque arête ?

comment ça marche ?



## comment ça marche ?

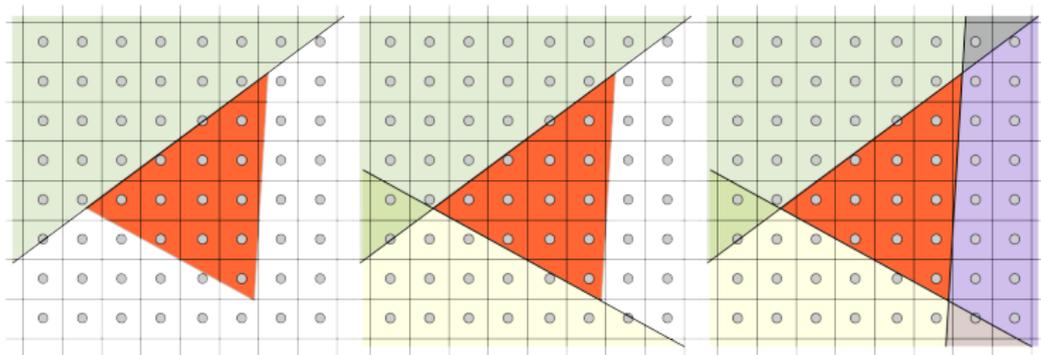
si le pixel est du bon coté ?

- ▶ un pixel et une arête forment un triangle,
- ▶ si ce triangle est bien orienté, le pixel est du bon coté...
- ▶ calculer l'aire algébrique (signée) du triangle, un coté est  $> 0$   
l'autre  $< 0$ .

si le pixel est du même coté des 3 arêtes :  
il est à l'intérieur du triangle.

les 3 aires ont le même signe que le triangle, en fonction de son orientation

## comment ça marche ?



# fragment shader

fragment shader :

- ▶ *doit* renvoyer une couleur pour le pixel,
- ▶ pour la partie du triangle qui occupe le pixel : un *fragment*

# fragment shader

paramètres en entrée :

- ▶ uniforms : valeurs transmises par l'application,
- ▶ constantes : comme d'habitude,
- ▶ varyings déclarés par le vertex shader.

sorties :

- ▶ vec4 gl\_FragColor : couleur du fragment,

## fragment shader : exemple

```
#version 330    // version de GLSL

// fonction principale du fragment shader
void main( )
{
    // resultat obligatoire : couleur du fragment
    gl_FragColor= vec4(1, 1, 0, 1);
}
```

## et avec plusieurs triangles ?

plusieurs triangles :

- ▶ peuvent se dessiner sur le même pixel...
- ▶ lequel faut-il garder ?  
(quelle couleur faut-il garder ?)

idée : l'image doit représenter ce que voit la caméra...

## plusieurs triangles ?

si les objets sont opaques :

- ▶ garder le triangle le plus proche de la camera,
- ▶ pour chaque pixel,
- ▶ ??
- ▶ celui qui a la plus petite coordonnée z dans le repère image.
- ▶ coordonnées du fragment dans le repère image ?

on ne connaît que les coordonnées des sommets dans le repère image...

# interpolation

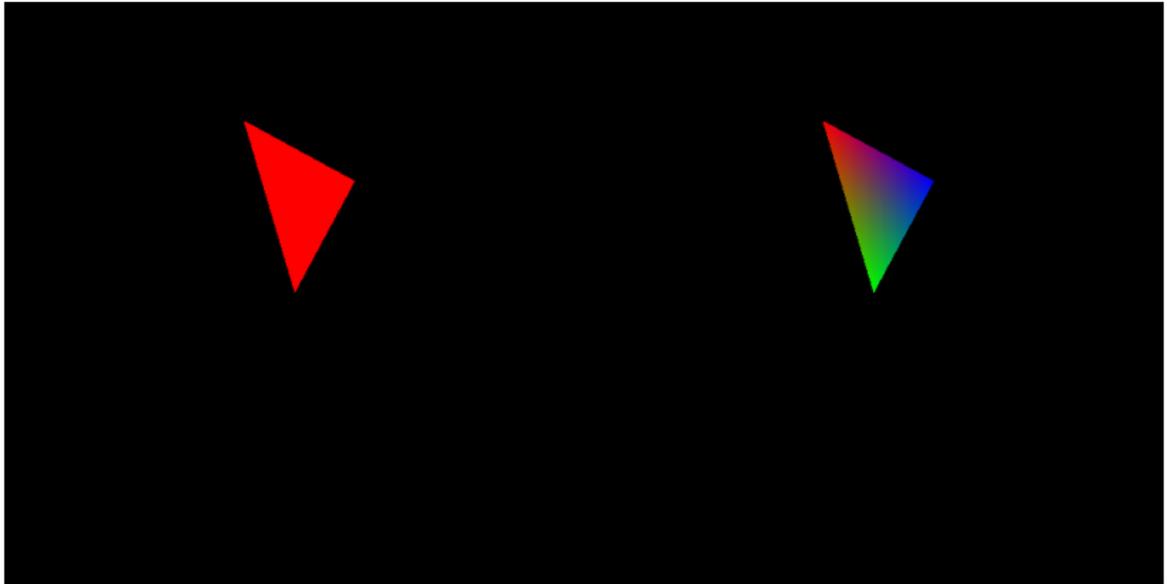
le pipeline interpole les coordonnées :

- ▶ des sommets,
- ▶ pour obtenir les coordonnées des fragments,
- ▶ on connaît donc  $x$ ,  $y$ ,  $z$  dans le repère image.

tous les attributs des sommets sont interpolés lors de la fragmentation... (position, normale, couleur, etc.)

conséquence : le repère Image est un *cube* en 3d !

## interpolation des attributs



## Ztest et Zbuffer

la profondeur du fragment :

- ▶ est conservée dans une autre "image" : le ZBuffer,
- ▶ et on peut choisir quel fragment conserver :
- ▶ le plus proche,
- ▶ le plus loin,
- ▶ le dernier dessiné.

il faut initialiser correctement la valeur par défaut du ZBuffer pour obtenir le bon résultat en fonction du Ztest.

# OpenGL et les shaders

## configuration minimale :

- ▶ le pipeline a besoin d'un vertex shader et d'un fragment shader pour fonctionner...
- ▶ chaque shader fonctionne indépendamment des autres, (en parallèle sur les processeurs de la carte graphique)
- ▶ mais un vertex shader peut transmettre des données au fragment shader qui dessine le triangle,
- ▶ paramètres *varyings* :
  - ▶ déclarés en sortie du vertex shader, `out vec4 color;`
  - ▶ déclarés en entrée du fragment shader, `in vec4 color;`
- ▶ et ils sont interpolés par le pipeline...

## varyings : exemple

```
#version 330

// vertex shader
in vec4 position;                // attribut
uniform mat4.mvpMatrix;          // uniform
out vec4 color;                  // varying / sortie

void main( )
{
    // resultat obligatoire du vertex shader
    gl_Position =.mvpMatrix * position;
    // transmet une valeur au fragment shader
    color = vec4(position.x, position.y, 0, 1);
}

// fragment shader
in vec4 color;                    // varying / entree

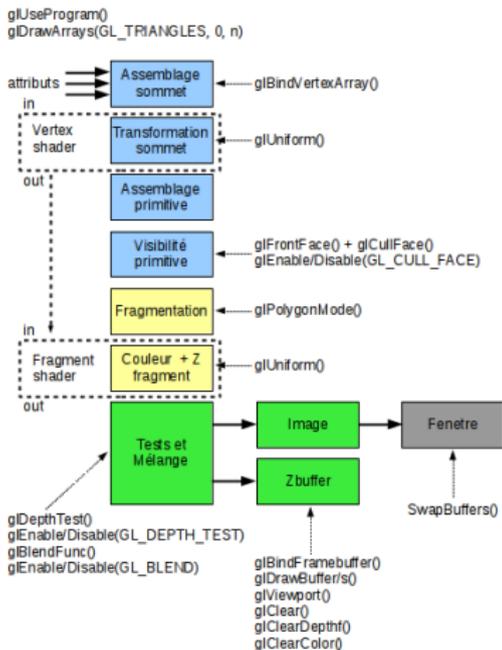
void main( )
{
    // resultat obligatoire du fragment shader
    gl_FragColor = color;
}
```

# OpenGL et les shaders

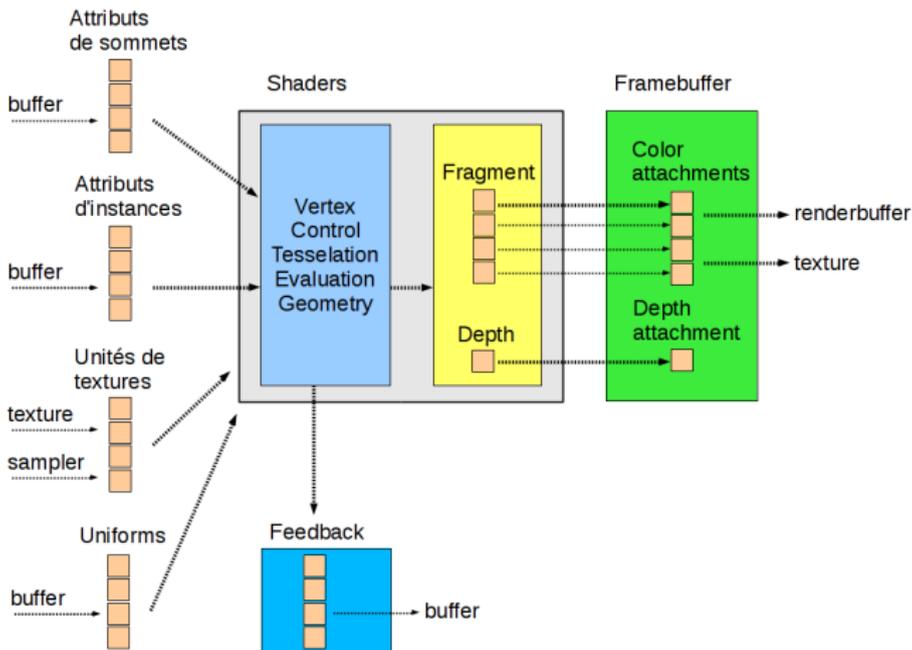
## configuration minimale :

- ▶ les uniforms sont affectés par l'application, (exemple : les matrices de transformation)
- ▶ les attributs sont stockés dans des tableaux / buffers, (uniquement accessibles aux vertex shaders)
- ▶ les varyings sont déclarés par les shaders et ne sont pas accessibles par l'application.

# pipeline simplifié



## api simplifiée



# application OpenGL

créer des objets OpenGL :

- ▶ buffers : stocker des données,
- ▶ vertex array : décrire l'organisation des attributs de sommets stockés dans des buffers,
- ▶ shader : compiler le source du vertex et du fragment shader,
- ▶ shader program : linker les 2 shaders,
- ▶ texture : stocker des images.

# application OpenGL

configurer le pipeline pour dessiner :

- ▶ le vertex array object, décrit les sommets,
- ▶ le shader program, code des shaders,
- ▶ les uniforms du shader program.

+ toutes les options de configuration...

# application OpenGL

## options de configuration :

- ▶ dimensions de l'image,
- ▶ couleur par défaut de l'image,
- ▶ Z test et Z buffer,
- ▶ profondeur par défaut du Z buffer,
- ▶ orientation des triangles,
- ▶ conserver, ou pas, les triangles à l'arrière des objets,
- ▶ remplir l'intérieur des triangles, ou ne dessiner que les arêtes, que les sommets,

+ glDraw( )

# application OpenGL

bilan :

- ▶ plutôt long pour afficher le premier triangle...
- ▶ mais faire plus n'est pas beaucoup plus compliqué...

# application OpenGL

## portabilité :

- ▶ OpenGL existe sur tous les systèmes (windows, linux, android, macos, ios, etc),
- ▶ mais ne gère pas les fenêtres, le clavier, souris, touchpad, etc.
- ▶ utiliser une librairie portable sur les mêmes systèmes : SDL2 (ou GLFW).

*remarque* : OpenGL ES 3 sur les portables / tablettes

## tp / projet

### gKit2 light :

- ▶ version très dégraissée ( $\approx 3000$  lignes) de gKit2 ( $\approx 25000$  lignes),
- ▶ version presque C, pas d'objets, pas d'héritage, pas de constructeurs, accessible pour l'option en L2,
- ▶ outils simples pour les tâches courantes :
  - ▶ fenêtre et évènements,
  - ▶ charger des images, des textures, des objets 3d,
  - ▶ compiler des shaders,
  - ▶ Point, Vector, Transform, Color pour les calculs

## tp / projet

### gKit2 light :

- ▶ mais pas mal de tutos : ( $\approx$  10000 lignes)
- ▶ et une documentation complète, source inclus,
- ▶ cf doxygen,
- ▶ compile pour l'instant :
- ▶ linux, windows, mac os,
- ▶ makefile, visual studio, code blocks, xcode,
- ▶ cf premake

# OpenGL et GLSL

référence OpenGL :

<https://www.opengl.org/sdk/docs/man/> section api

référence GLSL :

<https://www.opengl.org/sdk/docs/man/> section glsl

documentation complète OpenGL :

<https://www.opengl.org/registry/>

## SDL2 et GLFW

gKit2 / light utilisent :

<http://libsdl.org/>

mais GLFW est pas mal :

<http://www.glfw.org/>