M2-Images

openGL

J.C. Iehl

September 6, 2022

résumé des épisodes précédents

pour afficher des objets :

- il les faut les décrire...
- mais selon les traitements à faire sur les objets...
- il y a plusieurs manières...

pour afficher un objet, il faut une représentation adaptée à la méthode d'affichage...

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 / du "monde",
- dessiner une image des objets *visibles* par la 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 : donner une couleur à 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.

afficher des objets

2 cours :

- openGL et carte graphique, solution 1,
- lancer de rayons, solution 2.

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.

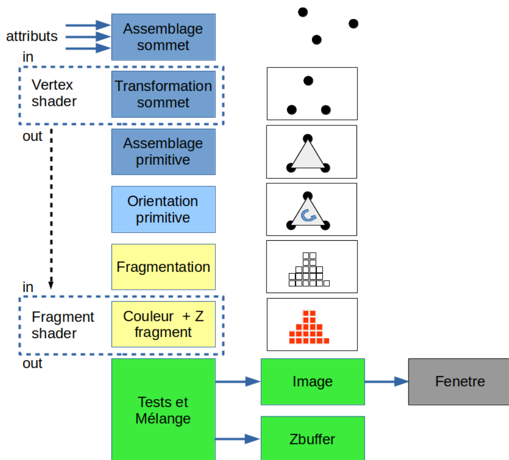
pipeline fragmentation / rasterization

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.

pipeline simplifié



pipeline simplifié

algorithme :

- pour chaque triangle :
- projetter les 3 sommets (dans l'image ?)
- si le triangle projetté est mal orienté dans l'image, stop...
- trouver tous les pixels de l'image à l'intérieur du triangle (générer les fragments du triangle)
- pour chaque fragment :
- calculer sa couleur
- si la profondeur du fragment $<$ zbuffer
  zbuffer= profondeur
  image= couleur

pipeline simplifié

fragments / pixels ?

- pixel : élément de l'image
- fragment : partie du triangle qui se projette sur un pixel

zbuffer ?

- profondeur du fragment associé au pixel...

oui, plusieurs triangles peuvent se projetter / se dessiner sur le meme pixel, il faut en choisir un...

pipeline simplifié

triangle mal orienté ?

- en 2d, on peut calculer l'aire signée d'un triangle
- si les sommets sont dans le sens trigo, aire positive
- sinon, aire négative

pipeline simplifié

et alors ?

- si les triangles projettés à l'avant d'un objet sont dans le sens trigo,
- les triangles à l'arrière de l'objet sont dans l'autre sens...
- ce test permet de ne dessiner que la moitiée des triangles de l'objet (en supposant qu'il est fermé...)

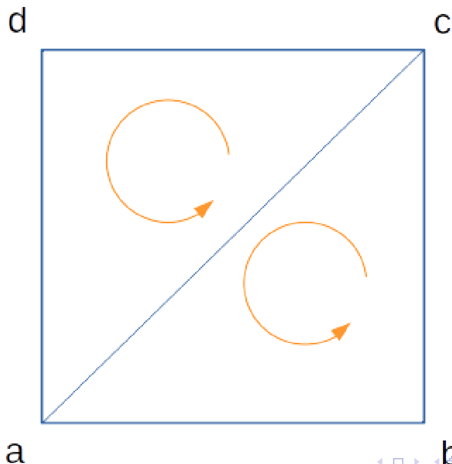
pas obligatoire de faire le test et d'orienter les triangles, mais pipeline 2 fois plus rapide...

orientation des triangles

un carré :

- sommets a= $\{0, 0\}$, b= $\{1, 0\}$, c= $\{1, 1\}$, d= $\{0, 1\}$,
- abc + cda, ou une autre paire ?
- abc ou n'importe quelle permutation qui ne change pas l'orientation : abc = bca = cab

trianguler la surface des objets



projetter les triangles

??

- les *sommets* des triangles
- comment savoir sur quels pixels dessiner les sommets d'un triangle ?
- ou sont les sommets ?
- ou est la camera ?
- ou est la projection (passage 3d vers 2d) ?
- ou sont les pixels ?

plusieurs repères...

repères et transformations standard

les coordonnées des sommets :

- dans quel repère ?
- les objets sont crées séparement : *repère local*,
- puis placés et orientés dans la scène : *repère global / monde*,
- puis observés par la camera : *repère camera*,
- puis projetés : *repère projectif*,
- puis les sommets sont projetés dans l'image : *repère image*.

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

repères et transformations standard

changements de repères :

$$\text{local} \underset{M}{\rightarrow} \text{monde} \underset{V}{\rightarrow} \text{camera} \underset{P}{\rightarrow} \text{projection} \underset{I}{\rightarrow} \text{image}$$

- M : matrice Model,
- V : matrice View,
- P : matrice Projection,
- I : matrice Viewport (v est déjà utilisé...),

les changement de repères sont représentés par des matrices 4×4...

repères et transformations standard

changements de repères :

$$\text{local} \underset{M}{\rightarrow} \text{monde} \underset{V}{\rightarrow} \text{camera} \underset{P}{\rightarrow} \text{projection} \underset{I}{\rightarrow} \text{image}$$

- sommet dans le repère local : p,
- transformé dans le repère monde : $M \times p$,
- transformé dans le repère camera : $V \times (M \times p)$
- transformé dans le repère projectif : $P \times (V \times (M \times p))$
- transformé dans le repère image : $I \times (P \times (V \times (M \times p)))$

les changement de repères sont représentés par des matrices 4×4...

repères et transformations standard

changements de repères :

local	$\leftarrow$	monde	$\leftarrow$	camera	$\leftarrow$	projection	$\leftarrow$	image
	M^{-1}		V^{-1}		P^{-1}		I^{-1}	

les changement de repères sont représentés par des matrices 4×4 et leurs inverses...

openGL et les matrices

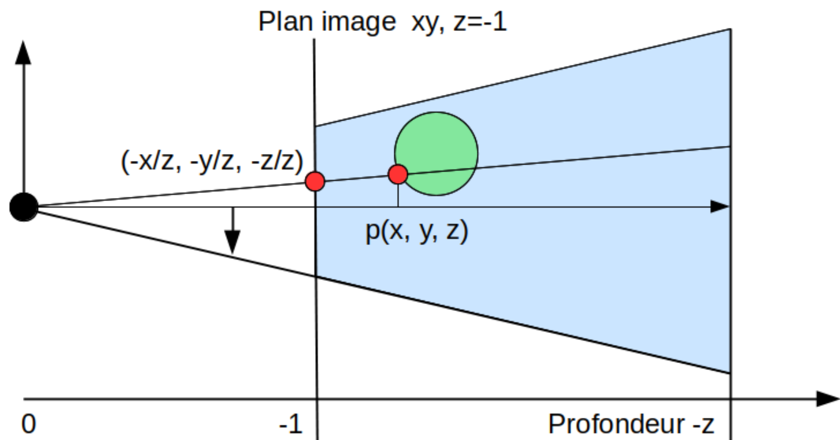
et alors ?

- 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.

gKit et les matrices :

- cf mat.h

projeter un sommet dans l'image...



projection

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

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

convention openGL : le plan image est à $z = -1$
la camera regarde -Z.

projection

projection et image :

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

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

projection

ensemble des points *visibles* / *observables* : noté *frustum*

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

et pour les points derrière la camera ?

transformation 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, s'il existe... $= \begin{bmatrix} d\frac{x}{z} \\ d\frac{y}{z} \\ d \\ 1 \end{bmatrix}$

projection

et alors ?

- c'est exactement ce que fait la matrice homogène,
- $-z$ doit "arriver" dans la composante w du point homogène résultat,
- paramètres supplémentaires : fov, distance proche / loin,
- ce qui la rend inversible...

transformation du frustum vers le cube $[-1 \,..\, 1]$

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++.
cf syntaxe GLSL

vertex shader

paramètres en entrée :

- uniforms : valeurs transmises par l'application,
- constantes : comme d'habitude,
- attributs de sommet : coordonnées dans le repère local,
- `gl_VertexID` : indice du sommet transformé.

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 un parametre 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 un parametre 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;
}
```

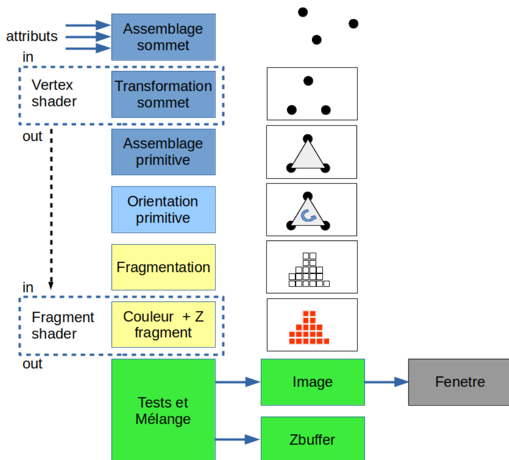
et alors ?

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 ?

et on a toujours rien dessiné...

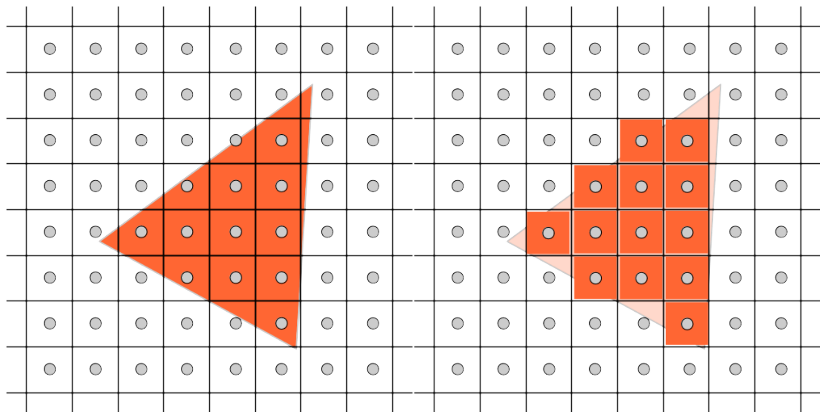


dessiner un triangle

dessiner un triangle :

- on connait les coordonnées des 3 sommets, (dans le repère projectif)
- vérifier qu'ils correspondent à des pixels de l'image (inclus dans le *frustum*, ils se projettent sur un pixel),
- et trouver tous les pixels de l'image qui sont à l'intérieur du triangle.

dessiner un triangle

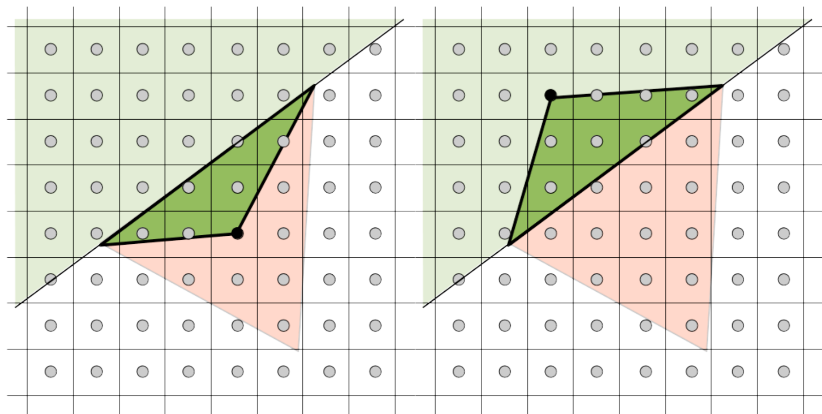


comment ca marche ?

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 ?

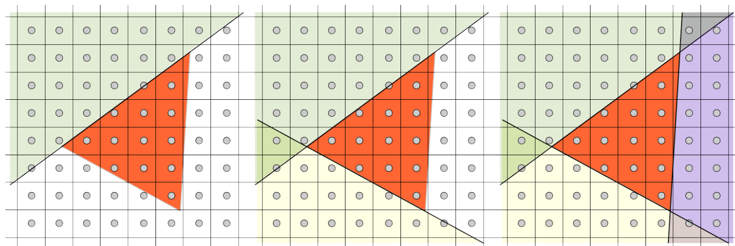
- un pixel et une arête forment un triangle,
- si ce triangle est bien orienté, le pixel est du bon coté...
- calculer l'aire algébrique (signée) du triangle, un coté est > 0 l'autre < 0.

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,
- `gl_PrimitiveID` : indice de la primitive / triangle.
- `gl_FragCoord` : coordonnées du fragment dans le repère image $p_i = (x, y, z, 1/w)$.

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 connait que les coordonnées des sommets dans le repère image...

interpolation

le pipeline interpole les coordonnées :

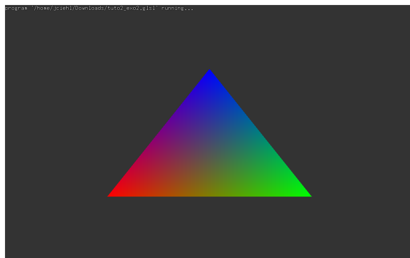
- des sommets,
- pour obtenir les coordonnées des fragments,
- on connait donc x, y, z dans le repère image.

tous les *varyings* sont interpolés lors de la fragmentation... (position, normale, couleur, etc. sorties déclarées par le vertex shader)

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

et on peut utiliser ce mécanisme pour faire autre chose...

interpolation des *varyings* définis par les vertex shaders



Ztest et Zbuffer

la profondeur du fragment :

- est conservée dans une autre "image" : le ZBuffer,
- et on peut choisir quel fragment conserver (ZTest) :
- 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

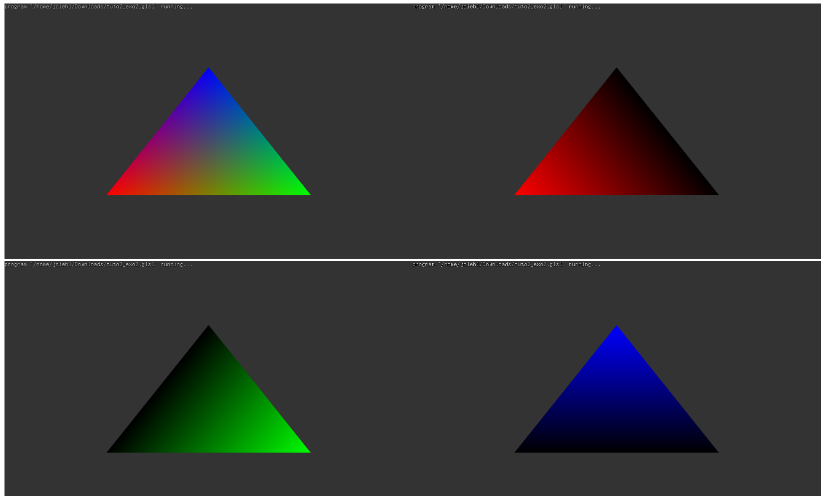
configuration minimale :

- 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

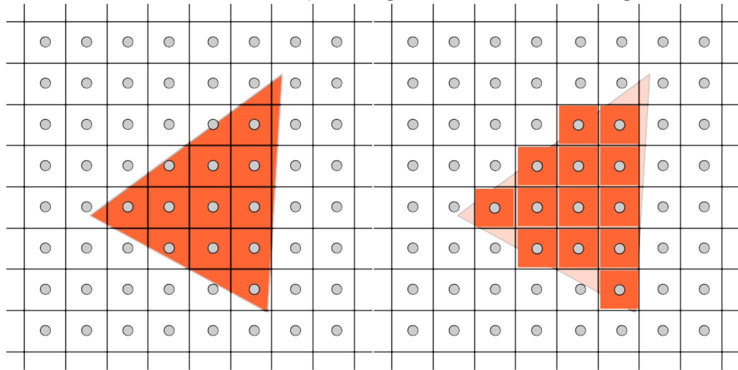
```
#version 330

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

void main( )
{
    // resulat 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;
}
```

interpolation des *varyings*

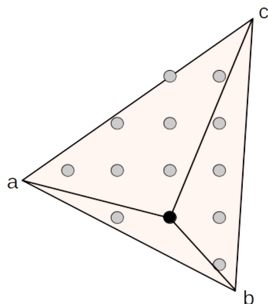


interpolation des .... *comment ça marche ?*

on sait ou se trouve chaque fragment dans le triangle :



interpolation des .... *comment ça marche ?*



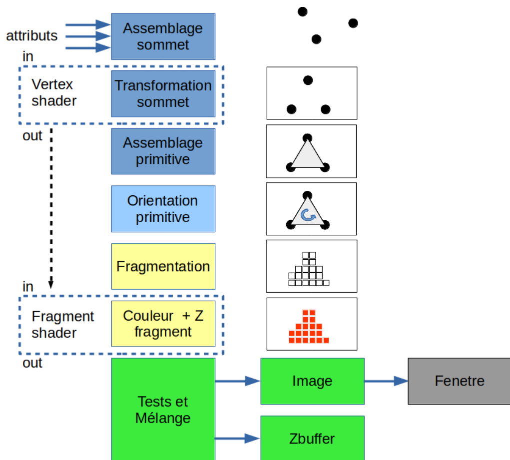
les aires des 3 petits triangles définissent les coordonnées barycentriques du centre du pixel,
le pipeline les utilise pour interpoler les sorties des 3 vertex shaders... (en plus de la profondeur / Z)

openGL et les shaders

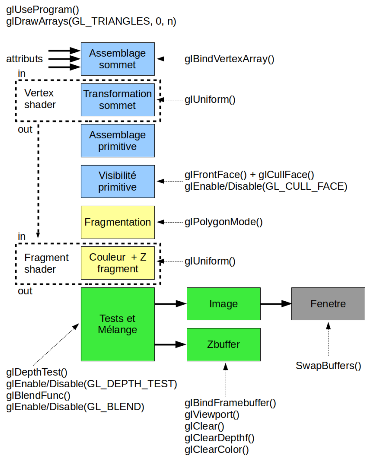
configuration minimale :

- 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



pipeline simplifié



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 / WebGL2

tp / projet

gKit2 light :

- version très dégraissée ($\approx$ 3000 lignes) de gKit2 ($\approx$ 25000 lignes),
- outils simples pour les taches courantes :
- fenêtre et evènements,
- 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 : ($\approx$ 50)
- et une documentation complète, source inclus,
- généré par doxygen,
- compile pour l'instant :
- linux, windows, mac os, (+ android, ios, webgl, avec quelques modifications)
- makefile, visual studio, code blocks, xcode,
- cf premake

git clone https://forge.univ-lyon1.fr/Alexandre.Meyer/gkit2light.git

tp / projet

écrire des shaders sans écrire d'application :

- oui c'est possible !
- cf shader_kit

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/