M2-Images

pipeline grahique et openGL

J.C. Iehl

September 14, 2016

résumé des épisodes précédents

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.

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.

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

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.

trianguler la surface des objets

représenter la surface des objets :

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

et transférer le tableau de coordonnées des sommets sur la carte graphique...

trianguler 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... (cf aire signée du triangle)

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

trianguler 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

trianguler la surface des objets

les coordonnées des sommets :

- dans quel repère ?
- cf partie 1 pipeline, projetter les sommets dans l'image...
- 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 projettés, repère projectif,
- cf partie 2 du pipeline,
- puis les sommets *visibles* sont projettés dans l'image, repère image.

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 dans un tableau dans la mémoire de la carte graphique,
- on peut aussi donner une couleur, une 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 est stockée sur la carte graphique...

et alors ?

et on a toujours rien dessiné...

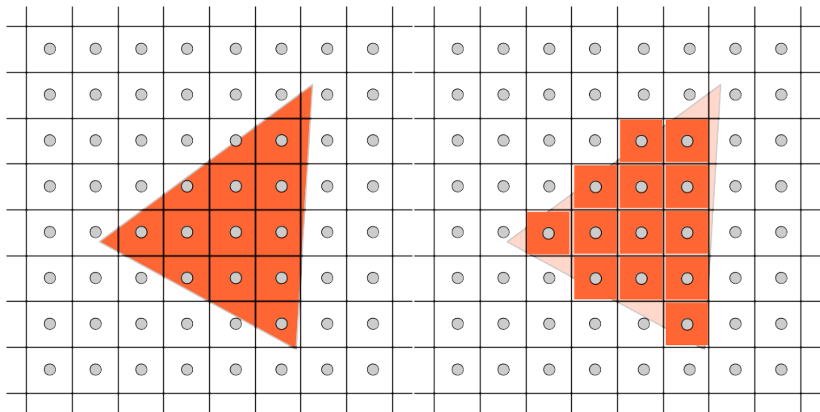
cf partie 2 du pipeline

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,
- et trouver tous les pixels de l'image qui sont à l'intérieur du triangle.

dessiner un triangle



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

fragment shader

fragment shader :

- *doit* renvoyer une couleur pour le pixel,
- pour la partie du triangle qui occuppe 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 connait que les coordonnées des sommets dans le repère image...

interpolation

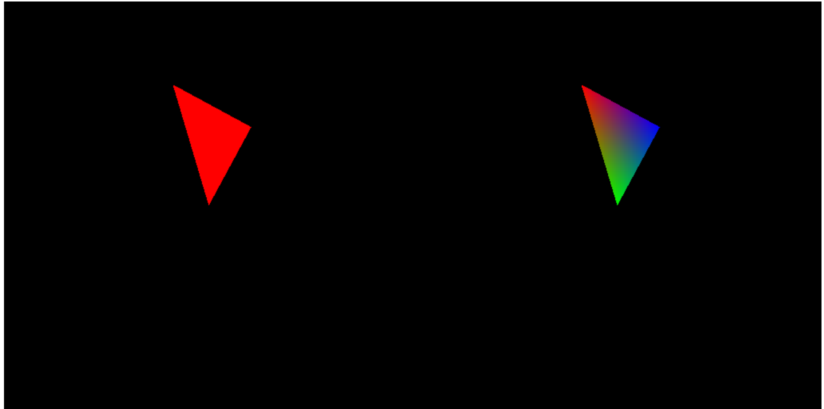
le pipeline interpole les coordonnées :

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

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

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

interplation 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

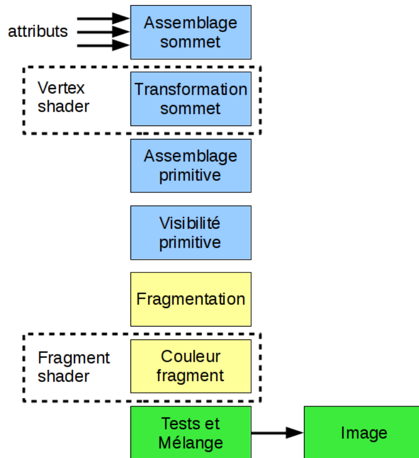
```
#version 330

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

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

openGL et les shaders

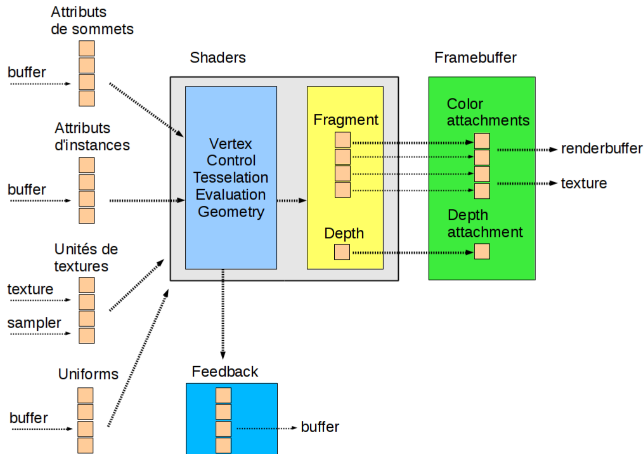
configuration minimale :

- 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( )`

openGL est une api C

convention différente :

- il n'est pas vraiment réaliste d'avoir une seule fonction draw( ) :
- qui prend des 100s de paramètres...
- donc il y a autre mécanisme :
- des paramètres implicites...

cf conception de l'api vers 1990

exemple :

version C explicite :

- tous les paramètres sont explicites,
- y compris l'objet manipulé...

```
// initialiser l'api 3d
struct Context *context= context_create();

// creer un buffer
struct Buffer *buffer= context_create_buffer(context);

// transferer des donnees dans le buffer
Point positions[]= { ... };
buffer_data(buffer, sizeof(positions), positions);
```

exemple :

version C implicite :

- l'objet manipulé est implicite,
- sélection explicite nécessaire, dans certain cas...

```
// initialiser l'api 3d
context_create();

// crer un buffer
struct Buffer *buffer= create_buffer();

// selectionner le buffer pour le manipuler
bind_buffer(buffer);

// transferer des donnees dans le buffer
Point positions[]= { ... };
buffer_data(sizeof(positions), positions);
```

exmple :

version C opaque :

- un pointeur sur une structure / objet n'est plus nécessaire,
- autant utiliser un type simple... comme int

```
// initialiser l'api 3d
context_create();

// crer un buffer
int buffer= create_buffer();

// selectionner le buffer a manipuler
bind_buffer(buffer);

// transferer des donnees dans le buffer
Point positions[]= { ... };
buffer_data(sizeof(positions), positions);
```

exemple :

pas de surchage en C :

- mais une famille de fonctions suffixees...
- glUniform1i(... , const int v);
- glUniform1f(... , const float v);
- glUniform3f(... , const float x, const float y, const float z);

exemple :

paramètres implicites :

- affecter une valeur à un paramètre uniform d'un shader :
- glUniform(...);

le shader program contenant le shader doit d'abord être sélectionné avec glUseProgram(...);

exemple :

paramètres implicites :

- dessiner dans une fenêtre $w \times h$:
- glViewport(... , w, h);

mais glViewport( ) permet aussi de calculer la transformation image (cf matrice I).

application openGL

bilan :

- plutot 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

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/

gKit2 light

base de code :

- pour les tps !
- simplifie les opérations courantes :
- compiler 2 shaders pour créer un shader program,
- surchage c++ pour affecter les paramètres uniforms d'un program,
- charger un objet 3d, format wavefront .obj,
- charger / enregistrer une image,
- charger une image et creer une texture,
- Point, Vector, Transform
- ...

gKit2 light

à lire pour la prochaine fois :

- tutos openGL (intro, interface... 5/6 premiers)
- tuto écrire une application openGL