CPE4

openGL et applications interactives

J.C. Iehl

November 27, 2017

épisode précédent

pipeline graphique openGL :

- dessiner des triangles...
- glDraw( ) est une *énorme* fonction prennant des dizaines de paramètres...

application interactive :

- afficher une image tous les 1/60s,
- animer les objets...
- *réagir* aux évènements utilisateurs : clavier, souris, etc.

aujourd'hui

résumé de l'essentiel de l'api openGL :

- comment affecter une valeur aux paramètres de glDraw() ?
- notion d'objets openGL ≡ groupes de paramètres,
- syntaxe C pur, sans surcharge,
- paramètres implicites,
- shaders : vertex et fragment.

application interactive avec SDL2 :

- initialisation, création d'une fenêtre openGL,
- évènements.

SDL2

portabilité :

- openGL existe sur beaucoup de systèmes,
- chaque système manipule différement les fenêtres, clavier, souris, etc.
- utiliser une librairie adaptée, SDL2, GLFW3

présentation SDL2...

application SDL2

application SDL2 :

- initialiser SDL2,
- créer une fenêtre pour openGL,
- initialiser openGL,
- évènements,
- afficher l'image dessinée par openGL,
- recommencer 60 fois par seconde.

initialiser SDL2

```
#include "SDL2/SDL.h"

int main( )
{
    if(SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        printf("[error] SDL_Init() failed:\n%s\n", SDL_GetError());
        return 1;        // erreur lors de l'init de sdl2
    }

    // enregistre le destructeur de sdl
    atexit(SDL_Quit);

    ...
    return 0;
}
```

créer une fenêtre pour openGL

```
...
// etape 1 : creer la fenetre
SDL_Window *window= SDL_CreateWindow("gKit",
     SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
    1024, 640, SDL_WINDOW_OPENGL);

if(window == NULL)
{
    printf("[error] SDL_CreateWindow() failed.\n");
    return 1;       // erreur lors de la creation de la fenetre
}

...
```

initialiser openGL

plusieurs versions d'openGL :

- version 3.3,
- mode debug,
- version shader / *core profile*.

plusieurs images associées à la fenêtre :

- dessiner dans une image non affichée dans la fenêtre,
- échanger l'image affichée et l'image de dessin,
- en fonction de l'écran.

pourquoi ? sinon défauts d'affichage cf *tearing*... lorsque le dessin est trop long ($> 16ms$)

initialiser openGL

```
...
// etape 2 : creer un contexte opengl core profile pour dessiner
// version 3.3
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
// version debug
SDL_GL_SetAttribute(SDL_GL_CONTEXT_FLAGS,
    SDL_GL_CONTEXT_DEBUG_FLAG);
// version shader / moderne
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
    SDL_GL_CONTEXT_PROFILE_CORE);

// dessiner dans une image differente de celle affichee
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);

SDL_GLContext context= SDL_GL_CreateContext(window);
if(context == NULL)
{
    printf("[error]␣creating␣openGL␣context.\n");
    return 1;
}

// attendre l'ecran pour echanger les images de la fenetre
SDL_GL_SetSwapInterval(1);
...
```

initialiser openGL

dernière étape :

- selon les systèmes, openGL est une librairie dynamique,
- ou pas,
- importer les (pointeurs de) fonctions openGL,
- utiliser une librairie, GLEW...

initialiser openGL

```
#include "GL/glew.h"

...
#ifndef NO_GLEW // pas necessaire sur macOS
    // initialise les extensions opengl, si necessaire
    glewExperimental= 1;
    GLenum err= glewInit();
    if(err != GLEW_OK)
    {
        printf("[error] loading extensions\n%s\n",
            glewGetErrorString(err));
        SDL_GL_DeleteContext(context);
        SDL_DestroyWindow(window);
        return 1;       // erreur lors de l'init de glew
    }

    // purge les erreurs opengl generees par glew !
    while(glGetError() != GL_NO_ERROR) {;}
#endif
...
```

application

au minimum :

- init SDL2,
- init openGL,
- tant que la fenêtre n'est pas fermée :
- // animer les objets,
- // en fonction des réactions de l'utilisateur,
- dessiner les objets,
- présenter l'image dessinée,
- recommencer 60 fois par seconde...

évènements

évènements :

- SDL2 présente l'ensemble d'évènements qui se sont produits, (dans le dernier intervalle de temps)
- traiter tous les évènements :
- au minimum : fermer la fenetre et quitter l'application.

cf SDL2 events

évènements SDL2

```
...
// etape 4 : affichage de l'application,
// tant que la fenetre n'est pas fermee.
bool done= false;
while(!done)
{
    // gestion des evenements
    SDL_Event event;
    while(SDL_PollEvent(&event))
    {
        // sortir si click sur le bouton de la fenetre
        if(event.type == SDL_QUIT)
            done= true;

        // sortir si la touche esc / echapp est enfoncee
        else if(event.type == SDL_KEYDOWN
        && event.key.keysym.sym == SDLK_ESCAPE)
            done= true;
    }

    // dessiner une image
    draw();

    // presenter le resultat
    SDL_GL_SwapWindow(window);
}
...
```

SDL2 et clavier

attention :

- les touches d'un clavier ont 2 identifiants :
- SDL_Keycode (international), dans keysym.sym
- SDL_Scancode (disposition physique) keysym.scancode

utilisez SDL_Keycode et keysym.sym

SDL2 et clavier

attention :

- appuyer sur une touche,
- et relacher la touche,
- sont 2 évènements différents...

cf event.type = SDL_KEYDOWN ou event.type = SDL_KEYUP

présenter le résultat

attention :

- si SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER , 1),
- SDL_SwapWindow() *obligatoire* !!
- sinon, rien ne s'affichera dans la fenêtre...

rappel : pipeline openGL

paramètres :

- affecter une valeur aux paramètres de glDraw() ?
- groupes de paramètres,
- configurés une fois,
- ré-utilisés pour chaque image.

en bref :
le pipeline est composé d'une partie programmable (les shaders) et d'une partie fixe (fragmentation et image résultat).

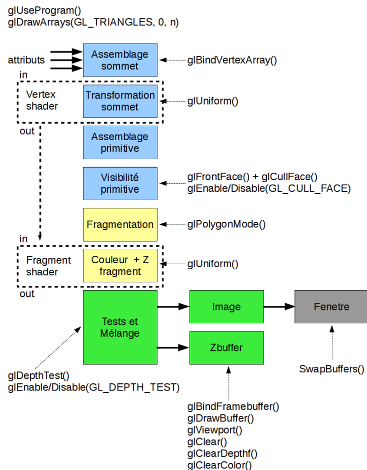
paramètres

entrées / sorties du pipeline :

- entrée : attributs des sommets,
- entrée : paramètres des shaders,
  (attributs, transformations, couleurs, etc.)
- entrée : options du pipeline fixe,
- sortie : image (crée par la fenêtre de l'application).

les shaders définissent la quasi-totalité des paramètres du pipeline...

ou est Charlie ?



GLSL

openGL Shading Language :

- proche du C/C++,
- types de base : entiers, réels,
  vecteurs 2, 3, 4 et matrices 2x2, 3x3, 4x4
- opérations de base : operations sur les vecteurs, matrices, transformations,
- tableaux (1D), et structures,
- fonctions non récursives,
- passage de paramètres par copie : mot-clés in, out, inout,

cf opengl.org / reference pages / GLSL et gKit / GLSL.

paramètres des shaders

les shaders :

- sont des fonctions "classiques",
- (mais exécutées par les processeurs graphiques, en parallèle)
- paramètres : en entrée, et en sortie,
- l'application est responsable :
- d'affecter une valeur aux entrées...
- et de fournir un stockage pour les résultats...

rappel : vertex shader

transformation d'un sommet :

- *doit* calculer les coordonnées d'un sommet dans le repère projectif homogène,
- gl_Position= ... ;

en entrée : coordonnées du sommet dans un repère arbitraire,
en entrée : (matrice de) transformation,
en sortie : pour le pipeline.

rappel : fragment shader

couleur d'un *fragment* :

- *doit* calculer une couleur,
- gl_FragColor= vec4(r, g, b, 1);

en entrée : paramètres pour le calcul de la couleur...
en sortie : pour le pipeline, image associée à la fenêtre, cf init SDL2

rappel : paramètres

coordonnées des sommets :

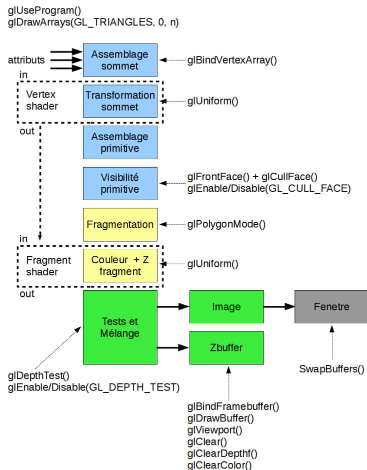
- déclarées avec le mot-clé : in,
- valeur ??
- cf vertex buffer + vertex array, glBindVertexArray()

uniforms :

- déclarés avec le mot-clé : uniform,
- valeur ??
- cf glUniform()

responsabilité de l'application.

pipeline openGL



librairie C pur

pas de surcharge :

- mais des familles de fonctions,
- suffixe dépendant du type des paramètres.

paramètres implicites :

- sélection de l'objet,
- affecter une valeur à un paramètre / propriété.

pas de pointeurs sur un objet :

- mais un identifiant opaque, de type GLuint.

openGL : compiler un vertex shader

vertex shader :

- créer un objet vertex shader,
- chaine de caractères, source du shader à compiler,
- compiler,
- vérifier les erreurs.

compiler un vertex shader

```
#include "GL/glcorearb.h"

// creer un objet openGL : vertex shader
GLuint vertex_shader= glCreateShader(GL_VERTEX_SHADER);

const char *sources[1];
sources[0]= read_source("vertex.glsl");

// fournir la chaine de caracteres
glShaderSource(vertex_shader, 1, sources, nullptr);
// compiler
glCompileShader(vertex_shader);

// verifier
GLint status;
glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
if(status == GL_TRUE)
    // pas d'erreurs de compilation
else
    // erreurs
```

openGL : compiler un fragment shader

fragment shader :

- idem vertex shader,
- mais glCreateShader(GL_FRAGMENT_SHADER);

openGL : linker un shader program

shader program :

- le pipeline utilise un vertex et un fragment shader,
- regroupés dans un shader program.

erreurs de compilation ?
cf exemples et détails : gKit program

linker un shader program

```
// creer un objet openGL : shader program
GLuint program= glCreateProgram();

// inclure les 2 shaders dans le program
glAttachShader(program, vertex_shader);
glAttachShader(program, fragment_shader);

// linker les shaders
glLinkProgram(program);

// verifier
GLint status;
glGetProgramiv(program, GL_LINK_STATUS, &status);
if(status == GL_TRUE)
    // pas d'erreurs de compilation
else
    // erreurs
```

affecter une valeur à un uniform

glUniform() :

- en fonction du type déclaré dans le shader :
- utiliser la bonne version / surcharge de glUniform()...
- mais : récupérer l'identifiant de l'uniform...
  glGetUniformLocation();
- mais : program à modifier,
  paramètre implicite de glUniform() !!
  sélectionner avec : glUseProgram();

cf détails et exemples : gKit uniforms et program

exemple : glUniform()

```
// selectionne le program a modifier
glUseProgram(program);

// shader : uniform int a;
GLint location_a= glGetUniformLocation(program, "a");
glUniform1i(location_a, 1);
// attention : le program est un parametre implicite de glUinform( ) !!

// shader : uniform float b;
GLint location_b= glGetUniformLocation(program, "b");
glUniform1f(location_b, 2.0f);

// shader : uniform vec3 c;
GLint location_c= glGetUniformLocation(program, "c");
glUniform3f(location_c, 1.0f, 2.0f, 3.0f);

// ou
float c[3]= { 1.0f, 2.0f, 3.0f };
glUniform3fv(location_c, 1, c);
```

entrées du vertex shader

le pipeline :

- est exécuté par les processeurs de la carte graphique,
- un vertex shader exécuté par sommet ,
- les processeurs doivent accéder automatiquement aux attributs / coordonnées des sommets,
- mais : mémoire graphique != mémoire de l'application,
- allouer de la mémoire graphique (buffer),
  transférer les données,
- puis :
  décrire l'organisation mémoire des attributs (vertex array),

étape 1 : buffers

allocation mémoire :

- créer un buffer,
- sélectionner le buffer en fonction de son utilisation cf GL_ARRAY_BUFFER, pour les attributs,
- transférer les données.

exemple : allouer un buffer

```
// creer un buffer
GLuint buffer;
glGenBuffers(1, &buffer);

// selectionner le buffer pour l'initialiser...
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// ... et transferer les donnees
Point positions[N]= { ... };
glBufferData(GL_ARRAY_BUFFER, N*sizeof(Point), positions,
     GL_STATIC_DRAW);
```

étape 2 : vertex array

organisation mémoire des attributs :

- attribut déclaré (et typé) par le vertex shader,
- mais : buffer, valeurs non typées,
- itérer sur les valeurs dans le buffer :
- position de la première valeur dans le buffer (offset en octets),
- distance jusqu'à la prochaine valeur dans le buffer (stride en octets).

associer un ensemble de valeurs à chaque attribut déclaré par le vertex shader.

étape 2 : vertex array

détails :

- créer un vertex array, qui conserve l'ensemble de paramètres,
- sélectionner le vertex array pour le configurer,
- identifiant de l'attribut déclaré dans le vertex shader,
- sélectionner le buffer contenant les données,
- décrire l'organisation mémoire et l'associer à l'attribut.

cf détails et exemples : gKit vertex array object

remarque : un vertex array est associé à un shader program...

exemple : vertex array

```
...
// creer un vertex array
GLuint vao;
glGenVertexArrays(1, &vao);
// selectionner le vertex array pour le configurer
glBindVertexArray(vao);

// recuperer l'identifiant de l'attribut
// vertex shader : in vec3 position;
GLint attribute= glGetAttribLocation(program, "position");
if(attribute < 0)
    // probleme, l'attribut n'existe pas...

// selectionner le vertex buffer contenant les donnees
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// configurer l'attribut
glVertexAttribPointer(attribute,
    3, GL_FLOAT,        // size et type, cf vec3 position
    GL_FALSE,           // pas de normalisation des valeurs
    sizeof(float)*3,    // ou stride 0, par defaut
    0                   // offset 0, debut du buffer
);
glEnableVertexAttribArray(attribute);
...
```

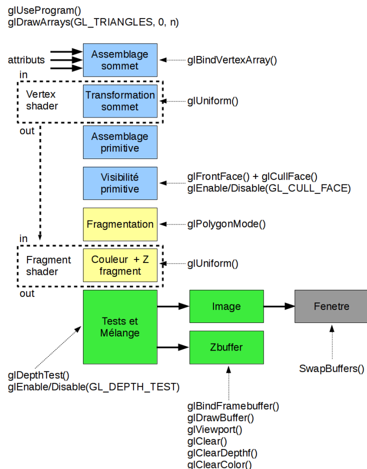
enfin glDraw() ?

bilan :

- glUseProgram();
- glBindVertexArray();
- glUniform();
- glDraw() ?

mais : manque les paramètres de la partie fixe du pipeline...
mais : décrire la géométrie des objets et utiliser la bonne version de glDraw( )...

pipeline openGL



pipeline openGL

pipeline fixe :

- l'image dans laquelle dessiner :
  cf glBindFramebuffer( ) et glDrawBuffer( ),
- les dimensions de l'image : cf glViewport( ),
- la couleur par défaut de l'image : cf glClearColor( ),
- la profondeur par défaut du zbuffer : cf glClearDepthf( ),
- le test de profondeur :
  cf glEnable/Disable(GL_DEPTH_TEST),
- l'orientation des faces avant, cf glFrontFace( ),
- l'élimination des faces arrière / mal orientées :
  cf glEnable/Disable(GL_CULL_FACE),

organisation d'une application openGL

organisation :

- init() : création / configuration des groupes de paramètres, buffers, vertex arrays, shaders,
- boucle d'affichage / gestion d'évènements : utilisation des groupes de paramètres pour dessiner,
- quit( ) : destruction des objets / groupes de paramètres.

cf application minimaliste : tuto1GL.cpp
+ utilitaires gKit : tuto1.cpp

dessiner une image

dessiner une image :

- effacer l'image couleur et le zbuffer, cf glClear()
- pour chaque objet :
- sélectionner le vertex array, cf glBindVertexArray(),
- sélectionner le shader program, cf glUseProgram(),
- affecter une valeur à tous les uniforms, cf glUniform() (cf transformation, couleur)
- glDraw()

+ Swapwindow( ) !!

utilitaires gKit

simplifier l'écriture d'une application openGL :

- charger, compiler des shaders,
  et afficher lisiblement les messsages d'erreurs,
- charger des objets 3d, format wavefront .obj,
- affecter une valeur aux uniforms (surcharge C++),
- création fenêtre, contexte openGL,
- gestion d'évènements, du temps,
- version C, ou classe de base C++ à dériver,
- vecteurs, points, couleurs, matrices classiques, + composition.