OpenGL, OpenGL/ES, Introduction

Renaud BLANCH

Université Joseph Fourier
Laboratoire d'Informatique de Grenoble

http://iihm.imag.fr/blanch/
mailto:blanch@imag.fr

Plan

0. Introduction
1. Pipeline graphique fixe
2. Pipeline graphique programmable

Plan

0. Introduction
1. Pipeline graphique fixe
2. **Pipeline graphique programmable**

2. Pipeline graphique programmable

2.0 *Program, vertex shader* et *fragment shader*
2.1 GLSL : *OpenGL Shading Language*
2.2 lien programme hôte / *program*
2.3 *Vertex shader* / *fragment shader*
2.4 Spécificités d'OpenGLIES 2.x

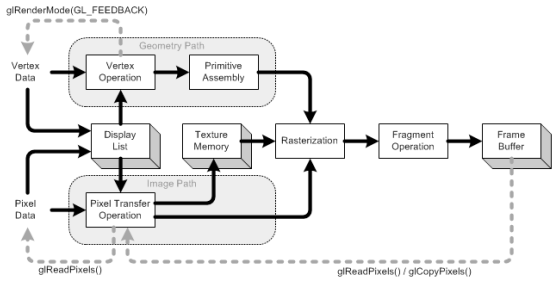
4

2. Pipeline graphique programmable

2.0 *Program, vertex shader* et *fragment shader*
2.1 GLSL : *OpenGL Shading Language*
2.2 lien programme hôte / *program*
2.3 *Vertex shader* / *fragment shader*
2.4 Spécificités d'OpenGLIES 2.x

5

2.0 *Program, shaders*



Les étapes ***Vertex Operation*** (transformation et éclairement par vertex) et ***Fragment Operation*** (couleur par fragment) deviennent **programmables**.

6

2.0 *Program, shaders*

Un *program* peut définir **deux fonctions** (`main`) qui **se substitueront** aux opérations du **pipeline fixe**.

Elles seront appelées :
• pour **chaque vertex** *(vertex shader)* ; et
• pour **chaque fragment** *(fragment shader)*
indépendamment.
Ces appels sont parallélisés par le GPU.

2. Pipeline graphique programmable

2.0 *Program, vertex shader* et *fragment shader*
2.1 GLSL : *OpenGL Shading Language*
2.2 lien programme hôte / *program*
2.3 *Vertex shader* / *fragment shader*
2.4 Spécificités d'OpenGLIES 2.x

2.1 GLSL

GLSL *(OpenGL Shading Language)*
est le **langage de programmation** des *shaders.*
Sa spécification est maintenue et publiée
par le Khronos Group.

2.1 GLSL

GLSL est un langage de programmation avec :

- une **syntaxe à la C** ;
- des **types adaptés** à l'informatique graphique ;
- des ***qualifiers*** supplémentaires ;
- des **fonctions graphiques *built-in*** ;
- du **sucre syntaxique** ;
- certains paramètres et valeur de retour **implicites** (via des variables globales) ; ...

10

2.1 GLSL

Les **types** de GLSL :

- `void`, `bool`, `int`, `float`, `struct`
- `vec2`, `vec3`, `vec4` (et [`ib`]`vec`[`234`])
- `mat2`, `mat3`, `mat4`
- `sampler1D`, `sampler2D`, `sampler3D`
- `struct`
- tableaux

11

2.1 GLSL

Les **fonctions** de GLSL :

- `radians`, `degrees`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- `pow`, `exp`, `log`, `exp2`, `log2`, `sqrt`, `inversesqrt`
- `abs`, `sign`, `floor`, `ceil`, `mod`, `min`, `max`
- `clamp`, `mix`, `step`, `smoothstep`
- `length`, `distance`, `dot`, `cross`, `normalize`
- `faceforward`, `reflect`, `refract`
- `texture[123]D`
- `dFdx`, `dFdy`, `fwidth`
- `noise[1234]`

12

2.1 GLSL

```
uniform bool lighting;

void main() {
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

   if(lighting) {
      vec4 ambient = gl_Color * gl_LightModel.ambient;
      float d = dot(normalize(gl_NormalMatrix*gl_Normal.xyz),
                    normalize(gl_LightSource[0].position.xyz));
      vec4 diffuse = gl_Color * gl_LightSource[0].diffuse * max(0., d);
      gl_FrontColor = clamp(ambient + diffuse, 0., 1.);
   } else {
      gl_FrontColor = gl_Color;
   }
}
```

13

2.1 GLSL

```
uniform bool lighting;

void main() {
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

   if(lighting) {
      vec4 ambient = gl_Color * gl_LightModel.ambient;
      float d = dot(normalize(gl_NormalMatrix*gl_Normal.xyz),
                    normalize(gl_LightSource[0].position.xyz));
      vec4 diffuse = gl_Color * gl_LightSource[0].diffuse * max(0., d);
      gl_FrontColor = clamp(ambient + diffuse, 0., 1.);
   } else {
      gl_FrontColor = gl_Color;
   }
}
```

14

2.1 GLSL

```
uniform bool lighting;

void main() {
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

   if(lighting) {
      vec4 ambient = gl_Color * gl_LightModel.ambient;
      float d = dot(normalize(gl_NormalMatrix*gl_Normal.xyz),
                    normalize(gl_LightSource[0].position.xyz));
      vec4 diffuse = gl_Color * gl_LightSource[0].diffuse * max(0., d);
      gl_FrontColor = clamp(ambient + diffuse, 0., 1.);
   } else {
      gl_FrontColor = gl_Color;
   }
}
```

15

2. Pipeline graphique programmable

2.0 *Program, vertex shader* et *fragment shader*
2.1 GLSL : *OpenGL Shading Language*
2.2 lien programme hôte / *program*
2.3 *Vertex shader* / *fragment shader*
2.4 Spécificités d'OpenGLIES 2.x

2.2 programme hôte / *program*

OpenGL fournit une **API** pour
• **compiler** ;
• faire l'**édition de lien** ;
• **charger** dans le GPU ; et
• **paramétrer**
les programmes depuis l'application hôte.

2.2 programme hôte / *program*

Un ***shader*** est une unité de compilation.

```
source = """
   void main() { gl_FrontColor = vec4(0.); }
"""

shader = glCreateShader(GL_TYPE_SHADER)
glShaderSource(shader, source)
glCompileShader(shader)
if glGetShaderiv(shader, GL_COMPILE_STATUS) != GL_TRUE:
   raise RuntimeError(glGetShaderInfoLog(shader))
```

2.2 programme hôte / *program*

Un ***program*** est un “exécutable” résultant
de l’édition de lien d’un ou de plusieurs *shaders*.

```
program = glCreateProgram()
for shader in shaders:
   glAttachShader(program, shader)

glLinkProgram(program)
if glGetProgramiv(program, GL_LINK_STATUS) != GL_TRUE:
   raise RuntimeError(glGetProgramInfoLog(program))

glUseProgram(program)
```


2.2 programme hôte / *program*

Les **paramètres** et les **résultats** des *shaders*
sont passés implicitement par des variables :

- les ***uniforms*** gardent une valeur constante par primitive (e.g. état d'OpenGL) ;
- les ***attributes*** prennent une valeur constante par vertex (e.g. couleur, normale, etc.) ;
- les ***varyings*** sont calculés par vertex puis interpolés linéairement par fragment.

2.2 programme hôte / *program*

```
vert_source = """
   uniform bool lighting;
   attribute vec3 vertex;
   varying vec3 L;

   void main() {
     gl_Position = gl_ModelViewProjectionMatrix*vec4(vertex, 1.);
     gl_FrontColor = gl_Color;
     if(lighting) L = normalize(gl_LightSource[0].position.xyz);
   }
"""

vert_shader = glCreateShader(GL_VERTEX_SHADER)
glShaderSource(vert_shader, vert_source)
glCompileShader(vert_shader)
if glGetShaderiv(vert_shader, GL_COMPILE_STATUS) != GL_TRUE:
   raise RuntimeError(glGetShaderInfoLog(vert_shader))
```

2.2 programme hôte / *program*

```
program = glCreateProgram()
glAttachShader(program, vert_shader)
glAttachShader(program, frag_shader)

attribs = ["vertex"]
locations = dict((k, v) for (v, k) in enumerate(attribs))
uniforms = ["lighting"]

for attrib in attribs:
    glBindAttribLocation(program, locations[attrib], attrib)

glLinkProgram(program)
if glGetProgramiv(program, GL_LINK_STATUS) != GL_TRUE:
    raise RuntimeError(glGetProgramInfoLog(program))

for uniform in uniforms:
    locations[uniform] = glGetUniformLocation(program, uniform)
```


2.2 programme hôte / *program*

```
glUseProgram(program)

lighting = True
glUniform1i(locations["lighting"], lighting)
glVertexAttribPointer(locations["vertex"], 3, GL_FLOAT, False,
                      record_len, vertex_offset)
```


2. Pipeline graphique programmable

2.0 *Program, vertex shader* et *fragment shader*
2.1 GLSL : *OpenGL Shading Language*
2.2 lien programme hôte / *program*
2.3 *Vertex shader / fragment shader*
2.4 Spécificités d'OpenGLIES 2.x

2.3 *Vertex shader*

Un *vertex shader* peut utiliser comme **entrées** les ***uniforms* d'état** et les ***attributs* des vertex** standard d'OpenGL :

- `gl_Vertex`,
- `gl_Normal`,
- `gl_Color`,
- `gl_MultiTexCoordi` (`i` = 0-7),
- ...

et/ou des **attributs de vertex génériques**.

2.3 *Vertex shader*

Un *vertex shader* peut donner une valeur aux *built-ins* :

- `gl_Position`
- `gl_ClipVertex`

et il peut aussi écrire dans les *varyings* :

- `gl_FrontColor`
- `gl_BackColor`
- `gl_TexCoord[i]`

2.3 *Vertex shader*

Exemple :
Animation des vertex en fonction du temps.

```
uniform float time;

void main() {
   vec4 p = gl_Vertex;
   float phase = p.x;
   phase -= time;
   phase += noise1(p.y*3.)/4.;
   p.z = sin(phase*2.*3.1415)*p.x;

   gl_Position = gl_ModelViewProjectionMatrix * p;
   gl_FrontColor = gl_Color;
}
```

2.3 *Vertex shader*

Exemple :
Reproduction d'une partie du **modèle d'éclairage**.

```
uniform bool lighting;

void main() {
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

   if(lighting) {
      vec4 ambient = gl_Color * gl_LightModel.ambient;
      float d = dot(normalize(gl_NormalMatrix*gl_Normal.xyz),
                    normalize(gl_LightSource[0].position.xyz));
      vec4 diffuse = gl_Color * gl_LightSource[0].diffuse * max(0., d);
      gl_FrontColor = clamp(ambient + diffuse, 0., 1.);
   } else {
      gl_FrontColor = gl_Color;
   }
}
```

28

2.3 *Fragment shader*

Un *fragment shader* peut utiliser comme **entrées** les ***uniforms* d'état** et les ***varyings*** standard d'OpenGL :

- `gl_Color`,
- `gl_TexCoord[i]` (`i` = 0-7),
- ...

Il peut écrire dans :
`gl_FragColor`, `gl_FragDepth`, ...

29

2.3 *Fragment shader*

Exemple :
Reproduction du ***alpha-test***.

```
const float alpha_threshold = .55;

uniform bool texturing;
uniform sampler3D texture_3d;

void main() {
   if(texturing) {
      vec4 color = texture3D(texture_3d, gl_TexCoord[0].stp);
      if(color.a <= alpha_threshold)
         discard;
   }
   gl_FragColor = gl_Color;
}
```

30

2.3 *Fragment shader*

Exemple :
Synthèse d'une **texture procédurale** : **fractale**.

```
uniform int max_i;
uniform sampler1D palette;

vec2 csquare(in vec2 c) { return vec2(c.x*c.x-c.y*c.y, 2.*c.x*c.y); }
int steps(in vec2 c) {
    vec2 z = vec2(0., 0.);
    int i;
    for(i = 0; i < max_i; i++) {
        z = csquare(z) + c;
        if(length(z) > 2.) { break; }
    }
    return i;
}
```

```
void main() {
    vec2 c = gl_TexCoord[0].st;
    int  i = steps(c);
    if(i == 0) discard;
    gl_FragColor = texture1D(palette,
                             float(max_i-i)/16.);
}
```


2.3 *Fragment shader*

Exemple :
Synthèse d'une **texture procédurale** : **cercle**.

```
void main() {
    vec2 p = gl_TexCoord[0].st;
    float v = length(dFdx(p)+dFdy(p));
    if(length(p) > 1.+v) discard;
    float alpha = clamp((1.-length(p))/v, 0., 1.);
    gl_FragColor = gl_Color;
    gl_FragColor.a *= alpha;
}
```


2. Pipeline graphique programmable

2.0 *Program, vertex shader* et *fragment shader*
2.1 GLSL : *OpenGL Shading Language*
2.2 lien programme hôte / *program*
2.3 *Vertex shader* / *fragment shader*
2.4 Spécificités d'OpenGLIES 2.x

2.4 Spécificités d'OpenGLIES 2.x

OpenGLIES 2.x est un **sous-ensemble d'OpenGL**.
Le choix de ce sous-ensemble a été guidé
par **deux critères** :
- **minimiser l'API** en supprimant tout doublon ; et
- **supprimer les fonctionnalités peu utilisées** et/ou difficiles à accélérer matériellement.

2.4 Spécificités d'OpenGLIES 2.x

En l'absence de **mode direct** (`glBegin/glEnd`),
et d'**attributs prédéfinis** (`glNormal/glColor...`)
pour les vertex,

il faut utiliser les **attributs génériques** de vertex,
et les passer dans des **tableaux**
(`glVertexAttribPointer/glDrawElements`).

On peut déléguer la gestion de ces tableaux au pilote
OpenGLIES avec les ***Vertex Buffer Objects*** *(VBO).*

2.4 Spécificités d'OpenGLIES 2.x

En l'absence des **piles de matrices** (`glMatrixMode/`
`glPush/glPop/glLoadIdentity...`),

il faut **gérer les transformations** (projection, texture ...)
dans le code hôte puis les passer au *vertex shader*
(`glUniformMatrix4fv`).

2.4 Spécificités d'OpenGLIES 2.x

En l'absence de **modèle de lumière** (`gl_LightModel, gl_LightSource, gl_NormalMatrix` ...)
il faut réimplémenter celui désiré dans le code hôte.

Il faudra en particulier prendre garde à utiliser
la bonne **transformation pour les normales**
(i.e. la transposée de l'inverse de la transformation).

2.4 Spécificités d'OpenGLIES 2.x

Manquent aussi dans OpenGLIES 2.x :

2.4 Spécificités d'OpenGLIES 2.x

Manquent aussi dans OpenGLIES 2.x :
- les **opérations *raster***
  (i.e. manipulation de pixels d'un buffer à un autre),
  il faut utiliser de la géométrie et une texture ;

2.4 Spécificités d'OpenGLIES 2.x

Manquent aussi dans OpenGLIES 2.x :
- les **opérations *raster***
  (i.e. manipulation de pixels d'un buffer à un autre),
  il faut utiliser de la géométrie et une texture ;
- les **textures 1D et 3D**
  (ne sont présentent que les textures 2D) ;

2.4 Spécificités d'OpenGLIES 2.x

Manquent aussi dans OpenGLIES 2.x :
- les **opérations *raster***
  (i.e. manipulation de pixels d'un buffer à un autre),
  il faut utiliser de la géométrie et une texture ;
- les **textures 1D et 3D**
  (ne sont présentent que les textures 2D) ;
- la possibilité de **sauver/restaurer l'état** du pipeline
  (`glPushAttrib/glPopAttrib`), si nécessaire,
  il faut garder une copie de l'état côté hôte ;

2.4 Spécificités d'OpenGLIES 2.x

Manquent aussi dans OpenGLIES 2.x :
- les **opérations *raster***
  (i.e. manipulation de pixels d'un buffer à un autre),
  il faut utiliser de la géométrie et une texture ;
- les **textures 1D et 3D**
  (ne sont présentent que les textures 2D) ;
- la possibilité de **sauver/restaurer l'état** du pipeline
  (`glPushAttrib/glPopAttrib`), si nécessaire,
  il faut garder une copie de l'état côté hôte ;
- la géométrie autre que **les points, lignes et triangles**
  (pas de `GL_QUADS`, `GL_QUAD_STRIP` ou `GL_POLYGON`).