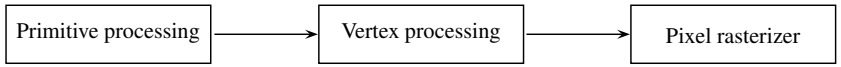
Sommaire intermédiaire

Pipeline habituel



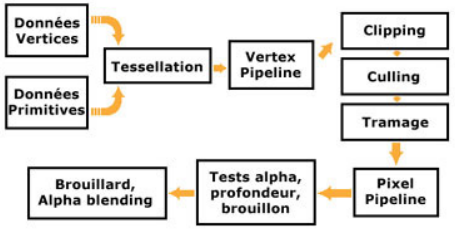
Depuis la version 1.4 l'API OpenGL permet la programmation du GPU grâce aux shaders.
Les données en entrée du pipeline graphique sont celles que transmet le programme :

- position (x, y, z, w)
- couleur (r, g, b, a)
- coordonnées de texture (u, v)

On peut rajouter les normales aux points, aux faces, ou des attributs.

Pipeline détaillé

Le pipeline graphique est en fait plus complexe, par opérations.

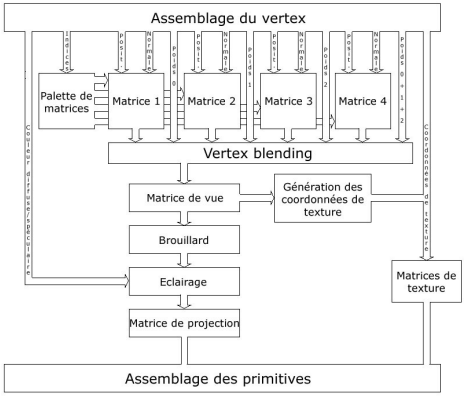


Le fragment en sortie de la projection 2D -> 3D (avant le `Pixel Pipeline`) contient pour chaque pixel :

- les coordonnées 2D entières,
- sa couleur,
- sa pseudo profondeur (profondeur relative dans le volume de vue).

Architecture d'un GPU

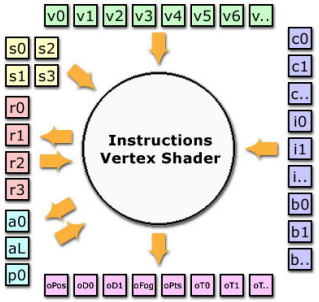
Le processeur graphique, une architecture différente...



Architecture d'un GPU

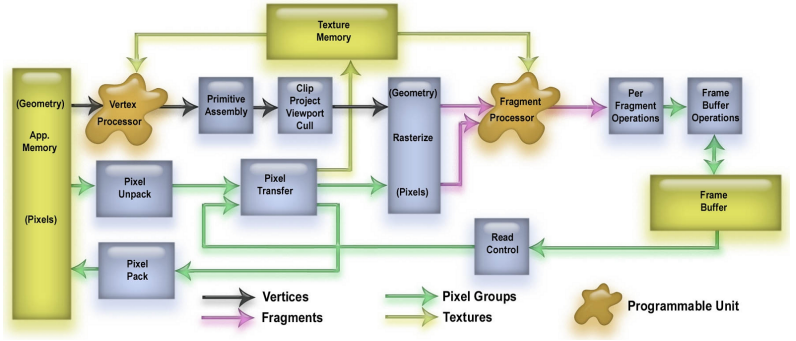
Le processeur graphique, une architecture différente...

- `V0 - V...` (15), registres d'entrée
- `R0 - R...` (12), registres de travail
- `c0 - c...` (256), constantes réelles
- `i0 - i...` (16), constantes entières
- `b0 - b...` (16), constantes booléennes
- `s0 - s3`, registres d'échantillonage, permet au vertex shader d'accéder aux textures
- a0, adresse d'un registre / aL : compteur /
- p0, branchements



OpenGL2+

Le pipeline d'OpenGL 2.0 et + et donc relativement plus compliqué car on peut communiquer entre les unités programmables.



Quels langages ?

On trouve comme d'habitude des implémentations propriétaires et des standards...

- Microsoft DirectX, se déclinant en 3 versions (de la plus ancienne à la plus récente) :
  - ASM, en assembleur comme son nom l'indique
  - HLSL, à l'aide d'un langage de haut niveau (C++ par ex.)
  - XNA (unification MsWindows/Xbox)

  Le principe est de transformer les sources du shader en assembleur binaire, par le compilateur `fxc` si le shader est précompilé, par DirectX sinon. Le driver interprète cet assembleur en byte code spécifique au hardware.

Quels langages ?

On trouve comme d'habitude des implémentations propriétaires et des standards...

- OpenGL
  - GLSL, mis en place par 3DLabs. Intégré à OpenGL2.0

  Le code GLSL du shader est fourni directement au driver de la carte graphique. Le compilateur est inclus au driver.
  - avantage : le compilateur est mis à jour avec le driver
  - inconvénient : l'obligation de développer un compilateur avec le driver

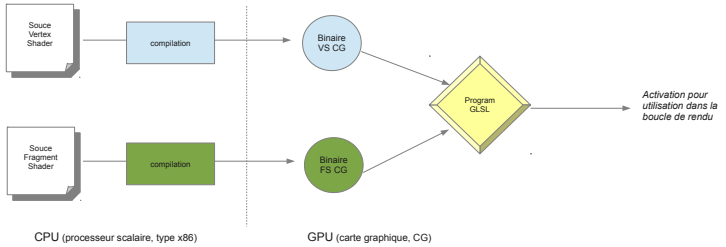
Quels langages ?

On trouve comme d'habitude des implémentations propriétaires et des standards...

- Nvidia
  - CG, cross-platform, langage spécifique
    - Transformation du langage en assembleur par le compilateur Cg. Assemblage du shader en format binaire par OpenGL ou DirectX. interprétation de cet assembleur en byte code spécifique au hardware.

Introduction

Un shader OpenGL porte sur la géométrie (« Vertex Shader » ou VS) ou sur les fragments (« Fragment Shader » ou FS) qui sont des pixels avec une pseudo-profondeur (comme dans le `Z-Buffer`). Pour les utiliser, il faut construire un « Program » qui est l'union d'un VS et d'un FS.
Chaque shader est compilé par le pilote de la carte graphique, on peut donc faire cette étape avant le début de la boucle de rendu. La création d'un « Program » suivra donc les étapes suivantes :



Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

1. charger et lire les fichier sources (attention à l'encodage, `UTF-8` ou non peut entraîner des erreurs) ;

Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

2. créer des identifiants de shader (ces variables sont de type `GLuint`) :

```
VertexShader = glCreateShader(GL_VERTEX_SHADER);
FragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

3. lier ces deux sources (chaîne de caractères) à un structure OpenGL :

```
glShaderSource(IDVertexShader, 1, &constVS, NULL);
glShaderSource(IDFragmentShader, 1, &constFS, NULL);
```

Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

4. compiler ces 2 sources :

```
glCompileShader(IDVertexShader);
glCompileShader(IDFragmentShader);
```

Note : Comme le shader est compilé par la carte graphique, on peut le modifier et relancer le programme principal. Cela recompile forcément le shader.

Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

5. créer un `ProgramObject`, qui contient les binaires du vertex et du fragment shader et qui remplacera le programme natif de la carte graphique :

```
IDProgramObject = glCreateProgram();
```

6. attacher les 2 shaders à ce programme (on peut les détacher et en attacher d'autre dynamiquement) :

```
glAttachShader(IDProgramObject, IDVertexShader);
glAttachShader(IDProgramObject, IDFragmentShader);
```

7. lier le programme au moteur graphique :

```
glLinkProgram(IDProgramObject);
```


Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

8. et pour finir, utiliser ce programme pour toutes les actions OpenGL qui suivent :

```
glUseProgram(IDProgramObject);
```

Note : `glUseProgram(0)` pour désactiver ce programme.

Le shader de base

VERTEX SHADER

```
void main(void) {
//renvoie la couleur de la face avant sur les sommets dans le pipeline
  gl_FrontColor = gl_Color;

//effectue les transfos de modélisation et de vue
  gl_Position = ftransform();
}
```

FRAGMENT SHADER

```
void main (void) {

//on prend la couleur dans le pipeline
//et on l'utilise pour l'affichage
  gl_FragColor = gl_Color;
}
```

Attention, s'applique sommet/sommet, pixel/pixel.

Qu'est-ce que l'on peut faire d'un shader ?

Mettre les pixels de la scène en niveau de gris :
FRAGMENT SHADER

```
void main (void)
{
//on récupère la couleur dans le pipeline
  vec4 vectcolor = gl_Color;
  float luminance = 0.2126*vectcolor.r + 0.7152*vectcolor.g
                    + 0.0722*vectcolor.b;

//on calcule la luminance
  vectcolor = vec4(luminance, luminance, luminance, 0.0);

//et on la renvoie pour l'affichage
  gl_FragColor = vectcolor;
}
```


types GLSL

Comme en C++, on trouve les types :

- float
- bool
- int

Plus les types vectoriels manipulés par le GPU :

- Vecteurs 2-4D
  - vec2,3,4 de réels (floats)
  - bvec2,3,4 de booléens
  - ivec2,3,4 d'entiers
- Matrices carrées 2x2, 3x3 et 4x4
  - mat2, mat3, mat4

Précautions à prendre

Attention !

```
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
vec4 c = vec4(a,b);          //donne c=vec4(1.0,2.0,3.0,4.0);
vec2 g = vec2(1.0,2.0);
float h = 3.0;               // et pas 3
vec3 j = vec3(g,h);
```

Un *tips&tricks* : le *swizzling*. On peut échanger les coordonnées dans la description, RGB vs XYZ ou indices :

```
vec3 v0 = vec3(0.0, 1.0, 2.0);
vec3 v1 = v0.zyx;               //v1 est (1.0, 2.0, 3.0)
vec3 v2.xzy = v1.zxy;           //v2 = v0 !
vec3 v3.rgb = v0.xyz;           //v3 = v0
```


Textures et structures

On trouve également les types Texture, les « Samplers » :

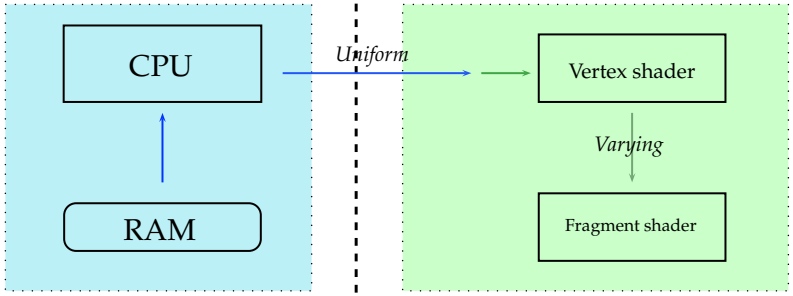
- sampler1D, texture 1D
- sampler2D, texture 2D
- sampler3D, texture 3D
- samplerCube, texture cube map
- sampler1DShadow, pour les shadow maps
- sampler2DShadow, pour les shadow maps

Et les structures comme en C :

```
struct lastructure {
  vec3 monvecteur;
  float monreel;
} mastructure;
```

Passage de valeurs

On peut aussi passer des données (`int`, `float`, `float *`, ...) entre le programme en CPU et les shaders du GPU. On peut également passer des données entre shaders (dans le sens du pipeline).



Exemple de shader

FRAGMENT SHADER

```
uniform vec3 lightDir;
varying vec3 normal;
void main()
{
  float intensity;
  vec4 color;
  intensity = dot(lightDir,normal);

  if (intensity > 0.95)
    color = vec4(1.0,0.5,0.5,1.0);
  else if (intensity > 0.5)
    color = vec4(0.6,0.3,0.3,1.0);
  else if (intensity > 0.25)
    color = vec4(0.4,0.2,0.2,1.0);
  else
    color = vec4(0.2,0.1,0.1,1.0);

  gl_FragColor = color;
}
```

VERTEX SHADER

```
varying vec3 normal;

void main()
{
    normal = gl_Normal;

    gl_Position = ftransform();
}
```

Plus dur

```
#define KERNEL_SIZE 9

float kernel[KERNEL_SIZE];
uniform sampler2D colorMap;

//taille de l'image=512, attention en dur !2000
const float step_w = 1.0 / 512.0;
const float step_h = 1.0 / 512.0;
vec2 offset[KERNEL_SIZE];

void main(void)
{
//le noyau de la convolution, i.e. le masque appliqué aux pixels pour l'instant laplacien
 kernel[0] = 0.0; kernel[1] = 1.0; kernel[2] = 0.0;
 kernel[3] = 1.0; kernel[4] = -4.0; kernel[5] = 1.0;
 kernel[6] = 0.0; kernel[7] = 1.0; kernel[8] = 0.0;

//comment sont définis les voisins du pixel courant
 offset[0] = vec2(-step_w, -step_h); offset[1] = vec2(0.0, -step_h); offset[2] = vec2(step_w, -step_h);
 offset[3] = vec2(-step_w, 0.0); offset[4] = vec2(0.0, 0.0); offset[5] = vec2(step_w, 0.0);
 offset[6] = vec2(-step_w, step_h); offset[7] = vec2(0.0, step_h); offset[8] = vec2(step_w, step_h);
```

>... suite ...>

Plus dur

```
int i = 0;
vec4 sum = vec4(0.0);

//on effectue les opérations que sur les pixels dans la moitié gauche de l'image
if(gl_TexCoord[0].s < 0.498) {
   for( i = 0; i < KERNEL_SIZE; i++ ) {
      vec4 tmp = texture2D(colorMap, gl_TexCoord[0].st + offset[i]);
      float luminosite = 0.299 * tmp.r + 0.587 * tmp.g + 0.114 * tmp.b;
        tmp.rgb = vec3(luminosite);
        sum += tmp * kernel[i];
        }
   }
   //sur la partie droite, on ne fait rien
   else if( gl_TexCoord[0].s > 0.502 ) {
      sum = texture2D(colorMap, gl_TexCoord[0].xy);
   }
   else
      sum = vec4(1.0, 0.0, 0.0, 1.0); //au milieu on met des pixels rouges

   gl_FragColor = sum;
}
```

Traitement d'une image

mots-clés : fragment shader, convolution et masque, offset de texture, uniform sampler2D.
Pour traiter une image avec un Fragment Shader, on a besoin de sommets qui passent dans un VS puis qui sont utilisés dans un FS. Le plus simple est donc d'utiliser un QUAD sur lequel on va plaquer une texture, et avec lequel on effectuera le Fragment Shader.

Le Vertex Shader contient juste la transformation des points 3D en points 2D et le passage de la texture au Fragment Shader :

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


OpenGL avancé | Comment cela fonctionne ?

Traitement d'une image

mots-clés : fragment shader, convolution et masque, offset de texture, uniform sampler2D.
Pour traiter une image avec un Fragment Shader, on a besoin de sommets qui passent dans un VS puis qui sont utilisés dans un FS. Le plus simple est donc d'utiliser un QUAD sur lequel on va plaquer une texture, et avec lequel on effectuera le Fragment Shader.

La texture est récupérée dans le Fragment Shader dans un `uniform sampler2D`. On peut alors utiliser le point (`gl_TexCoord[0].st`) de la 1ère texture ([0]) du sampler2D pour avoir sa couleur.

On peut aussi utiliser des décalages pour tenir compte des points voisins (`offset`). Cela permet de programmer des masques de traitement d'images (convolution) très facilement.

123/213 R. Raffin Prog. graphique & applis indus. v. 2019 12

Exemple de Vertex Shader

Modifier le vertex shader : comme on récupère une position dans `gl_Vertex`, on peut la faire modifier par la carte graphique. Par exemple, une translation de $(1, 1, 0)$:
VERTEX SHADER

```
void main(void) {
  //Translation
  vec4 point = gl_Vertex;
       point.x = point.x + 1.0;
       point.y = point.y + 1.0;
       gl_Position = gl_ModelViewProjectionMatrix * point;
}
```


Plus dur

Modifier le comportement du vertex shader, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type `Uniform`.
Comme elles sont stockées sur la carte graphique, on doit, depuis le prog. C++, pouvoir les modifier. Il faut connaître l'endroit où la carte les stocke (la carte renvoie une référence d'un emplacement mémoire). Par exemple, on passe une variable `mytime` incrémentée dans le prog en C++ `RenderScene` et on s'en sert pour déformer les sommets dans le vertex shader.

Plus dur

Modifier le comportement du vertex shader, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type `Uniform`.
VERTEX SHADER

```
uniform float cputime;

void main(void) {
  vec4 v = vec4(gl_Vertex);

//sin() est une fonction implémentée dans le GPU,
//cf GLSL référence

   v.z = sin(5.0*v.x + cputime)*0.5;

   gl_Position = gl_ModelViewProjectionMatrix * v;
}
```


Plus dur

Modifier le comportement du vertex shader, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type `Uniform`.
PROGRAMME C++

```
<variables en global>
GLint loc;
float mytime=0.0f;

<fonction RenderScene()>
  glRotatef(angle,1,1,0);
  glColor3f(1.0, 0.0, 0.0);

//on fait la liaison entre
//la valeur en GPU à
//l'adresse "loc" et la
//valeur dans "mytime"

  glUniform1fARB(loc, mytime);

  glutSolidTeapot(1.0);
  mytime += 0.01;
```

```
<fonction setShaders()>
 glUseProgram(IDProgramObject);

//on récupère l'adresse de
//la variable cputime du GPU,
//par son nom, dans "loc"

loc = glGetUniformLocation(
   IDProgramObject,"cputime");
```


Exemple de Varying

Varying permet de passer une valeur d'un Vertex Shader à un Fragment Shader pour ce pixel.
VERTEX

```
//les valeurs que l'on reçoit du CPU
uniform vec3 Uv3centre_deformation;
uniform vec3 Uv3vecteur_deformation;
uniform float Ufrayon_deformation;

//une valeur que l'on souhaite faire
//passer au Fragment shader
varying float VFactif;

void main(void) {
vec4 point = gl_Vertex;
VFactif = 0.0;

//on calcule la distance du centre de la déformation
//au vertex (point) courant
//distance est une fonction de GLSL
   float d = distance(vec4(Uv3centre_deformation, 1.0)
             , point);
```

```
//si la distance ci-dessus est inférieure au
//rayon d'action de la contrainte, on déforme
//de manière linéaire (max=centre, min=rayon)

if (d < Ufrayon_deformation) {
 point.xyz = point.xyz + (1.0 -
             (d / Ufrayon_deformation))
              * Uv3vecteur_deformation.xyz;
 VFactif = 1.0;
}

//et on projette ce point
gl_Position = gl_ModelViewProjectionMatrix * point;

}
```


Exemple de Varying

Varying permet de passer une valeur d'un Vertex Shader à un Fragment Shader pour ce pixel.
FRAGMENT

```
varying float VFactif;

void main (void) {

   if (VFactif == 1.0)

    //modif du pixel actif en rouge
      gl_FragColor = vec4(1.0,0.0,0.0,1.0);

   else

    //modif de tous les pixels (bleu)
      gl_FragColor = vec4(0.0,0.0,1.0,1.0);
}
```

Geometry shader

Les Geometry Shader permettent de créer à la volée des géométries, qui retourneront dans le pipeline (transformations, projections, rasterisation).

```
void main(void) {
   mat4x4 bezierBasis = mat4x4( 1, -3, 3, -1, 0, 3, -6, 3 , 0, 0, 3, -3 , 0, 0, 0, 1);
   for(int i=0; i<64; i++) {
      float t = i / (64.0-1.0);
      vec4 tvec = vec4(1, t, t*t, t*t*t);
      vec4 b = tvec*bezierBasis;
      vec4 p = gl_PositionIn[0]*b.x + gl_PositionIn[1]*b.y + gl_PositionIn[2]*b.z + gl_PositionIn[3]*b.w;

      gl_Position = p;
      EmitVertex();
      }

      EndPrimitive();
}
```

Le renvoi de géométrie se fait grâce à EmitVertex(), La primitive doit être préalablement choisie (triangle, point, ligne).

« Éditeurs » GLSL

- Shader-maker, `http://cg.in.tu-clausthal.de/teaching/shader_maker/index.shtml`
- GLSL Validate
- GLSL Parser Test
- Rendermonkey (ATI)
- ShaderDesigner (Typhoon Labs, `http://www.opengl.org/sdk/tools/ShaderDesigner/`)
- gDEBugger (Graphics Remedy)

Sommaire intermédiaire

Mode direct

... Et coûteux en temps et en débit.

```
//Dans la boucle de rendu
glBegin(GL_TRIANGLES);
   glVertex3f(0.0f, 0.0f, 0.0f);
   glVertex3f(1.0f, 1.0f, 1.0f);
   glVertex3f(2.0f, 2.0f, 2.0f);
glEnd();
```

Cela implique qu'à chaque image (à chaque fois qu'OpenGL a besoin de retracer), on re-décrit toute la géométrie de la scène et qu'on la renvoie de la RAM du CPU vers la RAM du GPU. Si l'objet n'est pas modifié, c'est beaucoup de temps et d'énergie perdus !

DisplayLists

Vous avez sans doute vus les `DisplayLists`, qui permettent la pré-compilation des objets en RAM. La carte graphique arrange les données, les prépare pour leur utilisation et repère leurs emplacements grâce à un identifiant.

```
id = glGenLists(1);

glNewList(id, GL_COMPILE);

for (int i=0; i < nbfaces; i++)
  glBegin(GL_TRIANGLES);
     glVertex3f(...);
     glVertex3f(...);
     glVertex3f(...);
  glEnd();

glEndList();
```

On effectue donc un pré-travail pour la carte graphique, il n'y a plus qu'à appeler la liste (`glCall(id)`) pour l'envoyer au traitement du GPU.

Il reste 2 inconvénients : les transmissions CPU -> GPU et l'impossibilité de modifier ces géométries sans recréer la liste.

Évidemment si l'objet est modifié, il faut détruire et refaire la DisplayList et la recompiler. Le 2ème inconvénient peut être résolu en implémentant un VertexShader qui peut modifier une géométrie dans une DisplayList.
C'est évidemment plus rapide, pas la peine de recréer la géométrie à chaque fois mais on peut peut-être faire mieux...

Mode indirect

Les géométries, en mieux : VertexArray
Depuis la version 1.2 d'OpenGL, on a la possibilité de préparer un peu mieux les données pour la carte graphique. Les VertexArray permettent de stocker la géométrie d'un objet, mais aussi sa (ses) couleur(s), ses normales, ses coordonnées de texture... dans des tableaux. Lors du rendu, on envoie ces tableaux à la carte graphique qui n'a plus qu'à lire, les données sont déjà organisées.

Vertex Array

Comment fait-on ? On crée des tableaux qui vont contenir les données de l'objet (des tableaux de `float`), on met dedans toutes les données de l'objet chargé, et on les utilise plus tard.

```
float ColorArray[nbsommets*3];
float NormalArray[nbsommets*3];
float VertexArray[nbsommets*3];
```

Dans le rendu, on active le rendu depuis des buffers, pour chaque type de données à transmettre :

```
//attention, on va se servir de sommets
glEnableClientState(GL_VERTEX_ARRAY);
//attention, on va se servir de normales
glEnableClientState(GL_NORMAL_ARRAY);
//attention, on va se servir de couleurs
glEnableClientState(GL_COLOR_ARRAY);

 //et voilà le buffer des couleurs, sur 3 float pour chaque couleur
 glColorPointer(3, GL_FLOAT, 0, ColorArray);

 //le buffer des normales, sur des floats (3 implicitement)
 glNormalPointer(GL_FLOAT, 0, NormalArray);

 //le buffer des sommets, 3 floats pour chaque coordoonées
 glVertexPointer(3, GL_FLOAT, 0, VertexArray);

 //et hop, on envoie au rendu de la CG
 glDrawArrays(GL_TRIANGLES, 0, nbfaces * 3);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```


Le Graal : les VBO

Les géométries, en encore encore mieux : `VertexBufferObject`
Depuis OpenGL1.5 on peut se servir des `VertexBufferObject`. L'intérêt de ces structures est que l'on peut stocker directement la géométrie sur la carte graphique.

Comme pour les `VertexArrays`, on utilise des tableaux pour stocker la géométrie d'un objet, on lie ensuite ces tableaux avec des zones mémoires de la carte graphique, on copie les données, et hop, c'est prêt ! (du coup on peut - peut-être - désallouer la mémoire en RAM du CPU puisque la géométrie est sur la carte graphique).

Il devient donc inutile de conserver la géométrie à la fois dans le CPU et le GPU, d'où un gain de mémoire en plus du gain de transfert.

Le mode des VBO

On peut gérer les VBO sous 3 modes :

- `GL_STREAM_DRAW` lorsque les informations sur les sommets peuvent être mises à jour entre chaque rendu = VertexArrays. On envoie les tableaux à chaque image.
- `GL_DYNAMIC_DRAW` lorsque les informations sur les sommets peuvent être mises à jour entre chaque frame. On utilise ce mode pour laisser la carte graphique gérer les emplacements mémoires des données, pour le rendu multi-passe notamment.
- `GL_STATIC_DRAW` lorsque les informations sur les sommets ne sont pas mises à jour - ce qui redonne le fonctionnement d'une DisplayList.
- Se rajoute à ces modes, des modes d'accès aux données : `READ`, `WRITE`, `COPY`, `READ_WRITE`. Ce qui permet également de copier des parties d'un objet dans un autre, et même de faire des interactions avec le CPU.

note : attention aux différents modes, cela ne fonctionne pas sur toutes les cartes graphiques.

Comment fait-on cela ?

Préparation des données :

1. on crée les tableaux de géométrie comme pour les VertexArray

   ```
   float lcolors[nbsommets*3];
   float lnormales[nbsommets*3];
   float lsommets[nbsommets*3];
   ```

2. on crée les buffers GPU qui contiendront ces données (`glBindBuffer`, `glBufferData`)

Création de VBO

```
//VBO pour les sommets
glGenBuffers((GLsizei) 1, &VBOSommets);
  glBindBuffer(GL_ARRAY_BUFFER, VBOSommets);
  glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float),
               lsommets, GL_STATIC_DRAW);

//VBO pour les couleurs
glGenBuffers((GLsizei) 1, &VBOCouleurs);
  glBindBuffer(GL_ARRAY_BUFFER, VBOCouleurs);
  glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float),
               lcolors, GL_STATIC_DRAW);

//VBO pour les normales
glGenBuffers((GLsizei) 1, &VBONormales);
  glBindBuffer(GL_ARRAY_BUFFER, VBONormales);
  glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float),
               lnormales, GL_STATIC_DRAW);
```


Rendu de VBO

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);

 glBindBuffer( GL_ARRAY_BUFFER, VBOSommets);
 glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );

 glBindBuffer( GL_ARRAY_BUFFER, VBOCouleurs);
 glColorPointer( 3, GL_FLOAT, 0, (char *) NULL );

 glBindBuffer( GL_ARRAY_BUFFER, VBONormales);
 glNormalPointer(GL_FLOAT, 0, (char *) NULL );

  glDrawArrays(GL_TRIANGLES, 0, monobjet.nbfaces * 3);

glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```


Les VBO

Et pour les images : le `FBO`
Un `Frame Buffer Object` (FBO) est l'espace de stockage qu'utilise OpenGL pour mettre, au fur et à mesure de son avancement, toutes les données issues d'algorithmes de l'espace image (lié à la résolution du rendu, en pixels). Par exemple les pixels de la scène, mais aussi le Z-buffer. Dans un fonctionnement statique, vous devez utilisez des stencils ou des masques pour interagir avec le Z-buffer, et vous ne pouvez pas modifier les pixels de l'image 2D rendue.

À partir d'OpenGL 2, on peut adresser ces espaces de stockages comme des textures (`Texture Object`) ou des buffers (`RenderBuffer Object`). Pour schématiser, les premiers sont faciles d'accès (par le programme principal ou un shader) et les seconds sont plus rapides (stockés en mémoire vive de la carte graphique, pas dans l'espace des textures).

Création d'un FBO

Pour créer un FBO, c'est comme pour les textures et les VBO, il faut générer une référence sur la carte graphique :

1. obtenir un identifiant

   `void glGenFramebuffersEXT(GLsizei quantité, GLuint* idfbo)`

Création d'un FBO

Pour créer un FBO, c'est comme pour les textures et les VBO, il faut générer une référence sur la carte graphique :

2. Pour attacher une texture à ce FBO, on écrit :

```
glFramebufferTexture2DEXT(GLenum target,
      GLenum attachmentPoint,
      GLenum textureTarget,
      GLuint textureId,
      GLint  level)
```

avec :

- target = `GL_FRAMEBUFFER_EXT`,
- attachementPoint dans `GL_COLOR_ATTACHMENT0_EXT`, ..., `GL_COLOR_ATTACHMENTn_EXT`, `GL_DEPTH_ATTACHMENT_EXT`, `GL_STENCIL_ATTACHMENT_EXT` selon le FBO à récupérer.
- textureTarget = `GL_TEXTURE_2D`,
- textureId = identifiant de votre texture
- level = ?, comme vous voulez, c'est le niveau de mipmapping de la texture finale.

Rendu d'un VBO

On active ensuite le FBO dans la boucle de rendu par :

```
void glBindFramebufferEXT(GLenum target, GLuint id)
```

avec target = `GL_FRAMEBUFFER_EXT`. Si on met target = 0, le FBO est désactivé.

Note : faire du rendu (`off-screen`) directement dans une image ou un fichier AVI -> c'est le `pbuffer`, et c'est différent.

On peut imaginer diverses utilisation des FBO, par exemple :

- effectuer un flou d'une scène selon la profondeur (comme dans les déserts par exemple) -> c'est faire 2 FBO (couleurs et profondeur) + 1 fragment shader qui parcourt la texture de profondeur et modifie la texture de couleur par un masque de cœfficients pris dans la profondeur (comme une convolution) ;
- appliquer des effets d'ombrage ou d'éclairement. On peut mettre une caméra à la place de la lumière, prendre le tampon de profondeur et l'utiliser pour faire un éclairement des sommets visibles et un ombrage des autres (cf `shadowmap`).

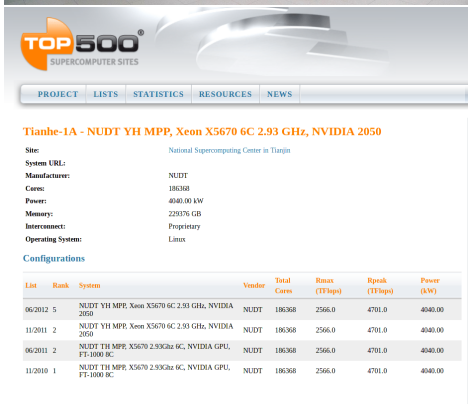
Convergence CPU/GPU

Shader model 4

- unification des unités de traitement, pas de différence vertex ou fragment
- 32 unités de shader
- Geometry Shader
- surfaces de haut-niveau (Bézier, Bspline, Nurbs, subdivision)
- rendu automatique (tesselation)

GPGPU

- programmation en calcul
- vectoriel massif sur carte graphique
- `www.gpgpu.org`



- LLNL's Gauss (Graphstream)

Programmation OpenGL

Dépendances :

- pour la gestion des fichiers images : FreeImage, bibliothèque open-source qui manipule env. 30 types de fichiers, `http://freeimage.sourceforge.net/`, paquets `libfreeimage3` et `libfreeimage-dev`,
- pour la gestion des extensions OpenGL : GLEW, bibliothèque open-source qui permet de vérifier les capacités de la carte graphique et de l'implémentation OpenGL, de manipuler les shaders et les fonctions d'OpenGL (`ARB_...`), `http://glew.sourceforge.net/`, paquets `libglew1.6` et `libglew1.6-dev`,
- pour la gestion dans l'environnement fenêtré (fenêtre, interactions, contexte OpenGL) : GLUT, `http://freeglut.sourceforge.net/`, paquets `freeglut3` et `freeglut3-dev` .