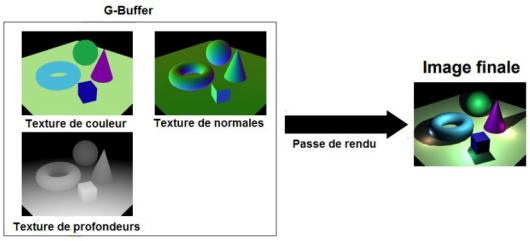
TP3 : Rendu dans une texture



Djihane BABAHENINI

djihane.babahenini@univ-biskra.dz

03/07/2019
1.0

Légende

 Entrée du glossaire

 Abréviation

 Référence Bibliographique

 Référence générale

Table des matières

Objectifs

À la fin de ce troisième TP, l'étudiant sera capable de:

- Comprendre les étapes de création d'une application OpenGL.
- Savoir créer un vertex et fragment shader.
- Construire une application de rendu d'un modèle 3D.

Pré-requis :

Pour pouvoir suivre ce TP avec succès il faut au préalable savoir:

- Des notions de base sur le langage C++.
- Des notions de base sur le concept de shaders et de GLSL.

Introduction

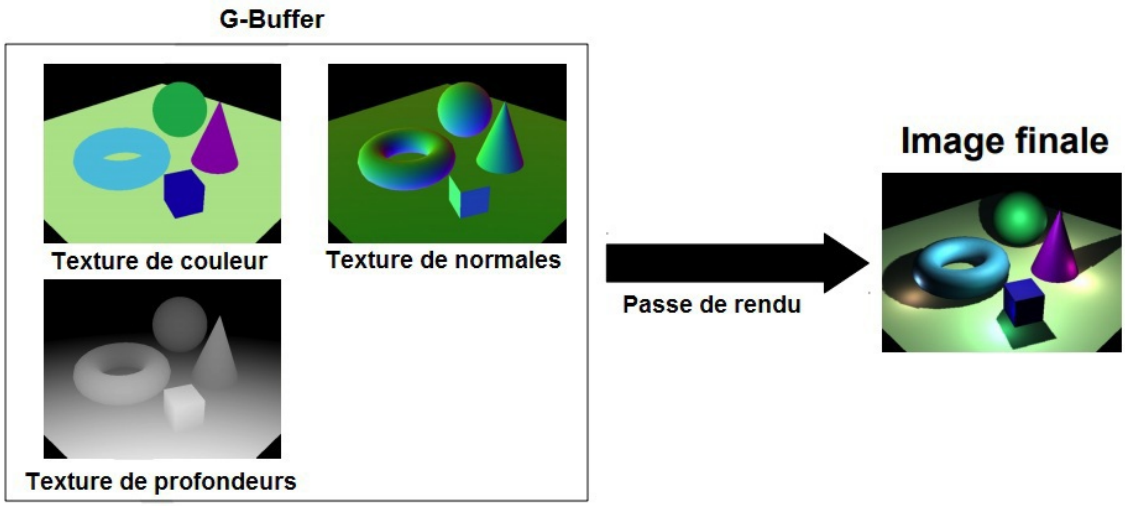
Le rendu sur une texture est une technique qui permet de calculer le rendu d'une scène 3D à partir de plusieurs textures utilisées en même temps.

Ces textures sont :

1. *Texture de positions :* contient la position 3D de chaque sommet de modèle.
2. *Texture de normales :* contient les coordonnées de normale de chaque surface de modèle.
3. *Texture de couleurs :* contient la couleur ou les cordonnées de texture de chaque sommet de modèle.
4. *Texture de profondeur :* contient la valeur z de chaque sommet de modèle.

L'ensemble de ces textures est appelé *G-Buffer*[p.23 AA] .

A partir de ces texture, on fait le rendu de la scène.



Rendu dans une texture.

Application OpenGL

1. Étapes de création des shaders

Pour pouvoir utiliser les shaders dans notre application OpenGL, il faut les charger, les compiler et de les lier au sein d'un programme OpenGL

1.1. Chargement de code source

Syntaxe

Dans cette étape, on lit le code de vertex shader et de fragment shder ligne par ligne à partir des fichiers sources :

```
1    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
2    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
3
4    // Lit le code du vertex shader à partir du fichier
5    std::string VertexShaderCode;
6    std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
7    if(VertexShaderStream.is_open())
8    {
9        std::string Line = "";
10       while(getline(VertexShaderStream, Line))
11           VertexShaderCode += "\n" + Line;
12       VertexShaderStream.close();
13   }
14
15   // Lit le code du fragment shader à partir du fichier
16   std::string FragmentShaderCode;
17   std::ifstream FragmentShaderStream(fragment_file_path, std::ios::in);
18   if(FragmentShaderStream.is_open()){
19       std::string Line = "";
```

```
        while(getline(FragmentShaderStream, Line))
            FragmentShaderCode += "\n" + Line;
        FragmentShaderStream.close();
    }
```

1.2. Compilation des shaders

Syntaxe

Les shaders de GLSL ont compilé durant l'exécution de l'application OpenGL, comme suit :

```
// Compile le vertex shader
    printf("Compiling shader : %s\n", vertex_file_path);
    char const * VertexSourcePointer = VertexShaderCode.c_str();
    glShaderSource(VertexShaderID, 1, &VertexSourcePointer , NULL);
    glCompileShader(VertexShaderID);
// Compile le fragment shader
    printf("Compiling shader : %s\n", fragment_file_path);
    char const * FragmentSourcePointer = FragmentShaderCode.c_str();
    glShaderSource(FragmentShaderID, 1, &FragmentSourcePointer , NULL);
    glCompileShader(FragmentShaderID);

```

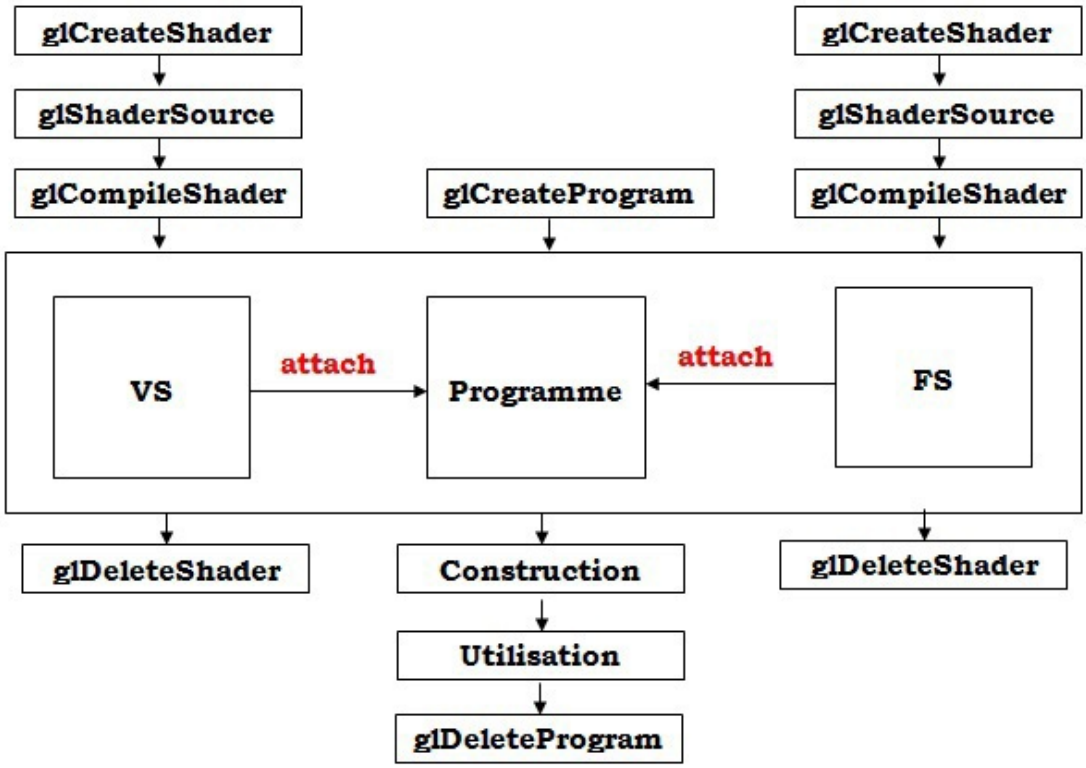
1.3. Attachement au programme

Syntaxe

Après la compilation des shaders, on doit maintenant les lier à un programme shader pour pouvoir l'utiliser:

```
// Lit le programme
    fprintf(stdout, "Linking program\n");
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);
```

Cette figure résume les étapes de création des shaders



Étapes de création de shaders.

2. Textures et FBO

2.1. Création des textures

Pour faire le rendu de la scène 3D, on doit crée les textures de positions, normales, et couleurs.

```
1 Texture* Fragmentbuffer::texturefbo(GLenum attach, GLenum typeTexture)
2 {
3   GLuint renderedTexture;
4   glGenTextures(1, &renderedTexture);
5   glBindTexture(GL_TEXTURE_2D, renderedTexture);
6
7   glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB32F, width, height, 0,GL_RGB, typeTexture, 0
  );
8   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
9   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
10  glFramebufferTexture(GL_FRAMEBUFFER, attach, renderedTexture, 0);
11  return new Texture(renderedTexture);
12 }
```

On crée un objet de la classe Fragmentbuffer, tels que windowWidth et windowHeight sont les dimensions de la fenêtre OpenGL:

```
1 Fragmentbuffer fbo(windowWidth, windowHeight);
```

Ensuite, on fait appel de la méthode texturefbo pour les trois types de textures

```
// Creation texture et depth(texture openGL et render buffer)
  Texture* diffTexture = fbo.texturefbo(GL_COLOR_ATTACHMENT0, GL_UNSIGNED_BYTE);
  Texture* normalTexture = fbo.texturefbo(GL_COLOR_ATTACHMENT1, GL_FLOAT);
  Texture* posTexture = fbo.texturefbo(GL_COLOR_ATTACHMENT2, GL_FLOAT);
```

2.2. Création des FBO

Premièrement, on va créer le tampon d'image (Framebuffer) qui contient le dessin de la scène.

```
GLuint FramebufferName = 0;
glGenFramebuffers(1, &FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

Ensuite, on associe les textures au tampon d'image :

```
Texture* diffTexture = fbo.texturefbo(GL_COLOR_ATTACHMENT0, GL_UNSIGNED_BYTE);
Texture* normalTexture = fbo.texturefbo(GL_COLOR_ATTACHMENT1, GL_FLOAT);
Texture* posTexture = fbo.texturefbo(GL_COLOR_ATTACHMENT2, GL_FLOAT);
GLenum drawBuffersFBO[3] = {GL_COLOR_ATTACHMENT0,
                GL_COLOR_ATTACHMENT1,
                GL_COLOR_ATTACHMENT2};
glDrawBuffers(drawBuffersFBO, 3);
```

Et finalement, on active le tampon d'image en ecriture :

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
```

3. Fenêtre d'affichage

Après avoir créer les shaders, les textures et le FBO (tampon d'image), on doit maintenant créer la fenêtre d'affichage.

On utilise la bibliothèque SFML pour la création de la fenêtre d'affichage :

```
  // création da fenêtre SFML
  sf::ContextSettings settings;
  settings.depthBits = 32;
  settings.stencilBits = 0;
  settings.antialiasingLevel = 4;
  settings.majorVersion = 4;
  settings.minorVersion = 0;
  sf::Window window(sf::VideoMode(windowWidth, windowHeight), "OpenGL", sf::
Style::Default, settings);
  window.setVerticalSyncEnabled(true);
```

Et dans la boucle de rendu :

```
bool running = true;
  while (running)
  {
    // gestion des évènements
    sf::Event event;
    while (window.pollEvent(event))
    {
```

```
8
9       switch(event.type)
10      {
11      case sf::Event::Closed : // Croix de fermeture
12        running=false;
13        break;
14      case sf::Event::Resized :
15        glViewport(0, 0, event.size.width, event.size.height);
16
17        break;
18      case sf::Event::KeyPressed : // Appui sur une touche
19        {
20          if(event.key.code == sf::Keyboard::Escape)
21          {
22            // Touche echap
23            running = false;
24          }
25
26        }
27        break;
28      }
29    }
```

4. Exercice

[solution n°1 p.21]

Répondez aux questions suivantes :

4.1. Exercice

Les étapes de création de shader sont :

- ☐ Édition des liens
- ☐ Chargement de code source
- ☐ Écriture de code avec GLSL
- ☐ Compilation
- ☐ Attachement au programme

4.2. Exercice

G-buffer a besoin de trois textures qui sont:

- ☐ Texture de position
- ☐ Texture procédurale
- ☐ Texture de couleur
- ☐ Texture de normale
- ☐ Image de texture

4.3. Exercice

La fonction OpenGL qui permet de créer les shaders est :

4.4. Exercice

La fonction OpenGL qui permet de compiler les shaders est :

4.5. Exercice

La fonction OpenGL qui permet de lier un vertex ou un fragment shader au programme shader est :

4.6. Exercice

L'ensemble des textures crées par OpenGL sont appelées:

4.7. Exercice

On crée la fenêtre d'affichage à l'aide de la bibliothèque :

Les shaders

Le rendu dans une texture se fait en deux étapes :

1. La génération de G-Buffer.
2. Le shading (ombrage).

Les shaders sont des fichiers texte crées par n'importe quel éditeur de texte (bloc-note, wordPad, notePad, ...)

1. Passe 01 : G-Buffer

Le G-Buffer est un ensemble d'images de même résolution que l'image finale

Il stocke pour chaque pixel toutes les informations propre au pixel et nécessaire pour le calcul d'éclairage : position, normal, texture, color, ..

Les deux shaders vertex et fragment de ce premier shader passe ont défini comme suit :

1.1. Vertex shader

Méthode

On envoie les trois textures (position, normale, couleur) crées sur CPU (application OpenGL) au GPU (exactement au vertex shader) :

```
1 layout(location = 0) in vec3 vertexPosition;
2 layout(location = 1) in vec2 vertexUV;
3 layout(location = 2) in vec3 vertexNormal_modelspace;
```

Ensuite, on communique les valeurs de position, normale, et texture entre le vertex et le fragment

shaders par interpolation en utilisant les qualifiers in et out.

On passe le MVP par le qualifier uniform :

```
1 out vec2 UV;
2 out vec3 Normal_world;
3 out vec3 Position_world;
4 uniform mat4 MVP;
```

Le code principal de vertex shader :

```
1 void main(){
2
3   gl_Position =  MVP * vec4(vertexPosition,1);
4   Position_world = vertexPosition;
5   Normal_world = vertexNormal_modelspace;
6
7   UV = vertexUV;
8 }
9
```

1.2. Fragment shader

Méthode

On récupère les données envoyées par le vertex shader par interpolation, et si l'objet est texturé, on récupère la texture à partir de CPU (diffuseTexture):

```
1 in vec2 UV;
2 in vec3 Normal_world;
3 in vec3 Position_world;
4
5 layout(location = 0) out vec3 color;
6 layout(location = 1) out vec3 normal;
7 layout(location = 2) out vec3 position;
8
9 uniform sampler2D diffuseTexture;
10
```

Et voilà le fragment shader :

```
1 void main(){
2   color = texture2D(diffuseTexture, vec2(UV.y,1.f-UV.x) ).rgb;
3   normal = Normal_world;
4   position = Position_world;
5 }
```

2. Passe 02 : shading

Dans ce passe de rendu, on fait le rendu dans une texture :

- Récupération des données à partir des textures du G-Buffer
- Déduire la position 3D à partir des coordonnées écran du pixel et du z
- Appliquer le modèle d'éclairage

2.1. Vertex shader

Méthode

On récupère les positions des sommets de l'objet à partir de CPU et on partage les coordonnées de texture UV entre le vertex et le fragment par interpolation :

```
layout(location = 0) in vec3 vertexPosition;

out vec2 UV;

void main(){
  gl_Position = vec4(vertexPosition,1);
  UV = vec2(vertexPosition.x*0.5f + 0.5f, vertexPosition.y*0.5f + 0.5f);

}

```

2.2. Fragment shader

Méthode

On récupère les positions des sommets de l'objet à partir de CPU et on partage les coordonnées de texture UV entre le vertex et le fragment par interpolation :

```
in vec2 UV;
//in vec4 Position_modelspace;
//in vec3 Normal_cameraspace;
//in vec3 EyeDirection_cameraspace;
//in vec3 LightDirection_cameraspace;

layout(location = 0) out vec3 color;

uniform sampler2D positionSampler;
uniform sampler2D normalSampler;
uniform sampler2D diffuseSampler;

uniform vec3 LightPosition_worldspace;
void main(){
  vec3 LightColor = vec3(1,1,1);
  float LightPower = 25.0f;
  vec3 MaterialDiffuseColor = texture2D( diffuseSampler, UV ).rgb;
  vec3 MaterialAmbientColor = vec3(0.5) * MaterialDiffuseColor;
  vec3 MaterialSpecularColor = vec3(0.3,0.3,0.3);

  vec3 position = texture2D( positionSampler, UV ).rgb;
  float distance = length( LightPosition_worldspace - position );
  vec3 n = normalize( texture2D( normalSampler, UV ).rgb );
  vec3 l = normalize(LightPosition_worldspace - position);

  float cosTheta = clamp( dot( n,l ), 0,1 );

```

```
28   // MaterialAmbientColor
29 //avec eclairage
30   //color = MaterialAmbientColor + MaterialDiffuseColor * LightColor * LightPower
   * cosTheta / (distance*distance);
31
32 //sans eclairage
33   color = texture2D(diffuseSampler,UV).rgb;
34 }
35
```

3. Exercice

[solution n°2 p.22]

Placer les actions ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

color = texture2D(positionSampler,UV).rgb;

layout(location = 0) out vec3 color;

out vec3 Position_world;

uniform vec3 LightPosition_worldspace;

uniform sampler2D diffuseSampler;

in vec2 UV;

gl_Position = MVP * vec4(vertexPosition,1);

uniform mat4 MVP;

layout(location = 1) in vec2 vertexUV;

out vec2 UV;

Vertex shader	Fragment shader

Communication entre l'application et les shaders

III

1. Les variables uniformes

Syntaxe

Cette variable peut être dans le vertex ou dans le fragment shader .

On a deux variables qui ont envoyé comme uniform :

- MVP : matrice de transformation.
- LightPosition_worldspace : position de la source lumineuse.

ils ont envoyé à partir de l'application Opengl comme suit :

```
Shader shader1("vertex.txt","fragment.txt");
GLuint MatrixID = glGetUniformLocation(program, "MVP");
Shader shader2("vertexquad.txt","fragmentquad.txt");
GLuint LightID = glGetUniformLocation(program, "LightPosition_worldspace");
```

Les deux variables ont déclaré dans le vertex et le fragment shader comme suit :

```
//vertex shader
uniform mat4 MVP;
//fragment shader
uniform vec3 LightPosition_worldspace;
```

Pour les texture uniformes, l'application OpenGL permet d'envoyer les textures de G-Buffer au programme shader :

```
glUniform1i(glGetUniformLocation(program, "positionSampler"), 0);
glUniform1i(glGetUniformLocation(program, "normalSampler"), 1);
glUniform1i(glGetUniformLocation(program, "diiffuseSampler"), 2);
```

Et dans les shaders, ils ont passé comme suit:

```
//vertex shader
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

```

```
6 //fragment shader
7 uniform sampler2D positionSampler;
8 uniform sampler2D normalSampler;
9 uniform sampler2D diffuseSampler;
```

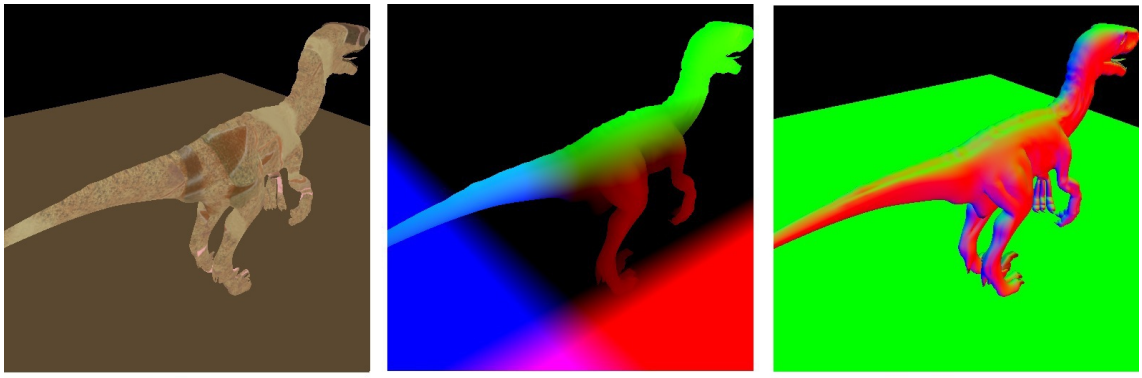
TP3

IV

1. Énoncé de TP

Ecrire un programme OpenGL permettant de :

1. Charger un modèle obj de votre choix.
2. Générer les trois textures : de position, de normale et de couleur.
3. Créer les deux programmes shaders qui affiche ces texture.



Texture de couleur | Texture de position | Texture de normale

Résultat de G-Buffer.

Conclusion

Dans ce TP, nous avons vu :

1. Le rôle de rendu dans une texture.
2. Comment faire le rendu dans une texture.
3. Le passage de données entre les deux shaders.
4. Le passage de données entre l'application OpenGL et les shaders.

Solutions des exercices

> **Solution** n°1

Exercice p. 11

Exercice

Les étapes de création de shader sont :

- ☐ Édition des liens
- ☑ Chargement de code source
- ☐ Écriture de code avec GLSL
- ☑ Compilation
- ☑ Attachement au programme

Exercice

G-buffer a besoin de trois textures qui sont:

- ☑ Texture de position
- ☐ Texture procédurale
- ☑ Texture de couleur
- ☑ Texture de normale
- ☐ Image de texture

Exercice

La fonction OpenGL qui permet de créer les shaders est :

glCreateShader

Exercice

La fonction OpenGL qui permet de compiler les shaders est :

glCompileShader

Exercice

La fonction OpenGL qui permet de lier un vertex ou un fragment shader au programme shader est :

glAttachShader

Exercice

L'ensemble des textures crées par OpenGL sont appelées:

G-Buffer

Exercice

On crée la fenêtre d'affichage à l'aide de la bibliothèque :

SFML

> **Solution** n°2

Exercice p. 16

Placer les actions ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

Vertex shader	Fragment shader
out vec2 UV;	layout(location = 0) out vec3 color;
gl_Position = MVP * vec4(vertexPosition,1);	uniform sampler2D diffuseSampler;
uniform mat4 MVP;	color = texture2D(positionSampler,UV).rgb;
out vec3 Position_world;	in vec2 UV;
layout(location = 1) in vec2 vertexUV;	uniform vec3 LightPosition_worldspace;

Abréviations

G-Buffer : Geometry Buffer