TP3 : Indexation et Shader de Phong

Introduction

L'objectif du TP est de construire un outil de visualisation de modèle 3D simple permettant l'importation et l'affichage de modèle 3D.

La base de code qui vous est fournie reprend les éléments de la dernière séance. Pour la récupérer, allez à l'adresse suivate :

`https://team.inria.fr/imagine/modelisation-synthese-dimage-ricm4-polytech-2014-2015/`.

Pour l'utiliser, créez un dossier temporaire depuis lequel vous utiliserez CMake pour générer un Makefile avant de lancez la compilation.

```
mkdir build             # Creation du dossier de build
cd build                # Acces au dossier de build
cmake ..                # Creation du Makefile
make                    # Compilation + edition de liens
./polytech_ricm4_tp3    # Execution
```

Buffers

Pour rappel, un buffer est une zone en mémoire sur la carte graphique qui contient des données utile pour l'affichage de l'image. Pour l'instant, nous avons seulement utilisé un buffer pour stocker les positions des sommets de nos objets. Nous allons maintenant voir comment créer des buffers pour gérer d'autres types de données : couleurs, normales, indices.

Buffer de couleurs

Pour créer un buffer de couleurs, procédez comme suit dans l'initialisation :

```
vec4 color[] = {  vec4(0.0, 0.0, 0.0, 1.0),
                  vec4(1.0, 0.0, 0.0, 1.0),
                  vec4(0.0, 1.0, 0.0, 1.0),
                  vec4(1.0, 1.0, 0.0, 1.0),
                  vec4(0.0, 0.0, 1.0, 1.0),
                  vec4(1.0, 0.0, 1.0, 1.0),
                  vec4(0.0, 1.0, 1.0, 1.0),
                  vec4(1.0, 1.0, 1.0, 1.0)};

GLuint colorBufferID;
glGenBuffers(1, &colorBufferID);
glBindBuffer(GL_ARRAY_BUFFER, colorBufferID);
glBufferData(GL_ARRAY_BUFFER, sizeof(color), color, GL_STATIC_DRAW);
```

Ensuite, il faut modifier le vertex shader :

```
#version 150

// Donnees d'entree
in vec3 in_position;
in vec4 in_color;

// Donnees de sortie
out vec4 vert_color;

// Parametres
uniform mat4 ModelMatrix;
```

```
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;

// Fonction appellee pour chaque sommet
void main()
{
  // Affectation de la position du sommet
  gl_Position = ProjectionMatrix *ViewMatrix * ModelMatrix * vec4(in_position, 1.0);

  // Affectation de la couleur du sommet
  vert_color = in_color;
}
```

Ainsi que le fragment shader :

```
// Version d'OpenGL
#version 150

// Donnees d'entree
in vec4 vert_color;

// Donnees de sortie
out vec4 frag_color;

// Fonction appellee pour chaque fragment
void main()
{
  // Affectation de la couleur du fragment
  frag_color = vec4(vert_color.rgb ,1.0);
}
```

Et enfin récupérer l'identifiant de la variable `in_color` dans l'initialisation du main :

```
GLuint colorPositionID = glGetAttribLocation(programID, "in_color");
```

Avant de l'utiliser dans la boucle de dessin :

```
// Activation de l'attribut colorPositionID
glEnableVertexAttribArray(colorPositionID);

// Definition de colorBufferID comme le buffer courant
glBindBuffer(GL_ARRAY_BUFFER, colorBufferID);

// On indique comment lire les donnees
glVertexAttribPointer(
  colorPositionID,
  4,
  GL_FLOAT,
  GL_FALSE,
  0,
  (void*)0
);
```

Sans oublier de désactiver l'attribut à la fin du dessin :

```
glDisableVertexAttribArray(colorPositionID);
```

Et de détruire le buffer à la fin du programme :

```
glDeleteBuffers(1, &colorBufferID);
```

Buffer d'indices

Nous allons maintenant créer un buffer d'indice. L'idée est la suivante : au lieu de transmettre un sommet à chaque fois qu'on en a besoin, on va transmettre une seul fois tout les sommets, puis y faire référence grâce à des indices.

Dans l'initialisation, créez votre buffer d'indices :

```
// Creation du tableau d'indices
unsigned int indices[] = {0,1,2 /* mettez vos indices ici */ }

// Creation du buffer d'indices
GLuint elementBufferID;
glGenBuffers(1, &elementBufferID);

// Copie de notre tableau d'indices dans le buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBufferID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Puis modifiez la fonction de dessin :

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBufferID);
glDrawElements(
                GL_TRIANGLES,
                sizeof(indices) / sizeof(unsigned int),
                GL_UNSIGNED_INT,
                (void*)0
                );
```

Et encore une fois détruisez le buffer à la fin du programme :

```
glDeleteBuffers(1, &elementBufferID);
```

Pas besoin de modifier les shaders.

Travail à faire : modifiez le programme pour utiliser un buffer d'indices pour le dessin du cube.

```
OFF

#vertex_count face_count  edge_count
3     1     0

#One line for each vertex
0.0   0.5   0.0
0.5   -0.5  0.0
-0.5  -0.5  0.0

#One line for each polygonal face
#vertex_number vertex_indices
3 0 1 2
```

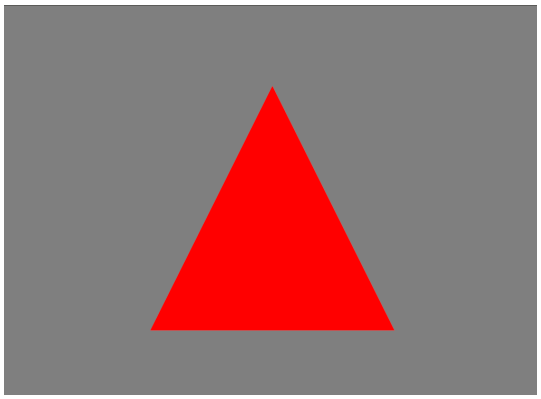


Figure 1: Exemple de fichier OFF pour un triangle

Maillage

Nous allons maintenant nous intéresser à un mode de représentation des objets en 3D : les maillages. Un maillage est une surface discrétisée sous la forme de polygones. Nous nous en servons en informatique graphique pour représenter les objets.

Un maillage est la donnée de :

- un ensemble de points, ils représentent les sommets des polygones formant notre maillage
- un ensemble d'indices, ils représentent l'appariement des sommets pour former les polygones

Pour stocker un maillage dans un fichier, il existe différents formats. Le plus simple d'entre eux est le format OFF. Pour plus de détail, voir figure 1.

Les fichiers "Mesh.h" et "Mesh.cpp" implémentent une classe permettant de représenter un maillage : cette classe contient des attributs pour les tableaux de positions, de normales, de couleurs, et d'indices. Par ailleurs, cette classe propose de charger automatiquement ces informations à partir d'un fichier OFF.

Pour l'utiliser, faites comme ceci :

```
Mesh m("../models/armadillo.off");    // chargement du maillage
m.vertices.data();                    // acces au tableau de positions
m.normals.data();                     // acces au tableau de normales
m.faces.data();                       // acces au tableau d'indices
```

Travail à faire : chargez un maillage du répertoire "models/" et affichez le dans la fenêtre.

Rendu lumineux

Maintenant que nous savons passer tout type de buffers à nos shaders, nous allons utiliser un buffer particulier pour le calcul de l'illumination.

Récupération des normales

Travail à faire : modifiez votre vertex shader pour qu'il réceptionne les normales, et affecte une couleur à chaque vertex avec la formule suivante :

```
vert_color = vec4((in_normal + vec3(1,1,1))*0.5, 1.0);
```

Modèle de Phong

Un modèle d'illumination vise à calculer une couleur à partir d'un ensemble de paramètres d'entrée :

- un ensemble de paramètres relatifs à la surface de l'objet :
  - une position dans l'espace
  - une normale à la surface en ce point
  - une couleur de la surface en ce point
- un ensemble de paramètres relatifs à la surface de l'objet :
  - une position dans l'espace ou une direction d'illumination
  - une couleur

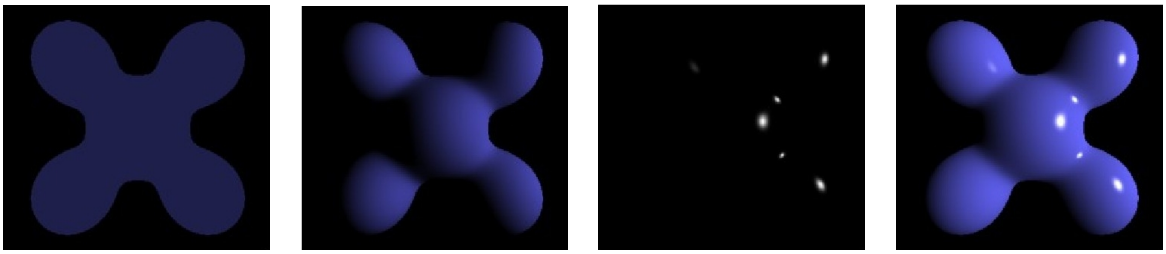
La modèle le plus simple et le plus répandu est le modèle de Phong. Il s'exprime comme la somme de diférentes composantes, comme nous pouvons le voir sur la figure 2.

Plus précisément, on note :

$$L_f = \rho_a L_a + \rho_d L_d max(\mathbf{n}.\mathbf{l}, 0) + \rho_s L_s max(\mathbf{r}.\mathbf{e}, 0)^s$$

où :

- ρ_a, ρ_d, ρ_s, sont les coefficients associés à chaque composante
- L_a, L_d, L_s, sont les couleurs de chaque composante
- L_f est la couleur résultante
- n est la normale à la surface

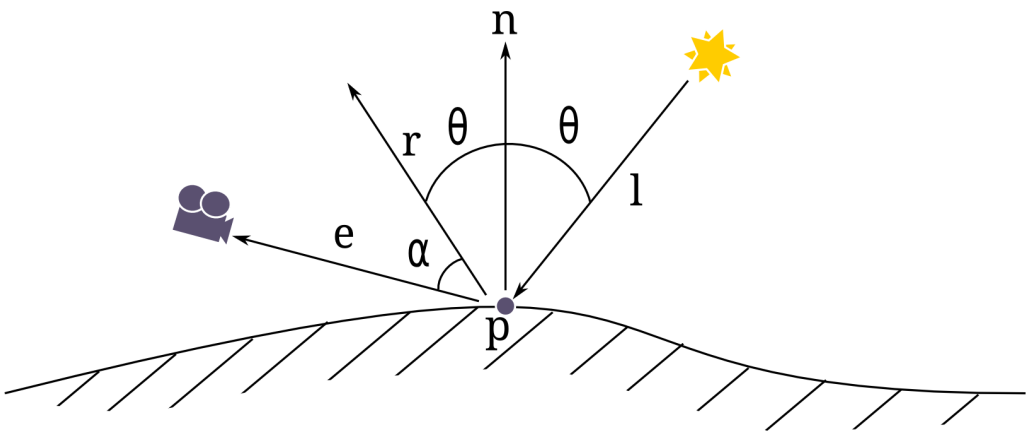


Ambient Diffus Spéculaire Total

Figure 2: Les différentes composantes du modèle d'illumination de Phong

- l est a direction de la lumière
- r est la réflexion de l par rapport à n
- e est la direction de vision
- s est un nombre (souvent puissance de deux) pondérant l'étalement de la tâche spéculaire, autrement dit plus ce nombre est élevé et plus la surface paraitra lisse

La figure suivante montre la sigification des paramètres ci-dessus :



Gouraud Shading

Travail à faire : implémentez le modèle d'illumination de Phong dans le vertex shader. Pour ce faire, vous calculerez toutes les composantes nécessaire au calcul dans le vertex shader, puis vous écrirez le résultat du calcul dans la variable `vert_color`.

Phong Shading

Travail à faire : implémentez le modèle d'illumination de Phong dans le fragment shader shader. Pour ce faire, vous calculerez toutes les composantes nécessaire au calcul dans le vertex shader, puis vous les transmettrez au fragment shader, depuis lequel vous ferez le calcul.

Quel différence constatez vous ? Comment interprêtez vous celle-ci ?