Examen de synthèse d'images
Documents non autorisés, Durée : 2h

Conseil : prenez le temps de lire correctement le sujet. Il n'est pas très long et vous avez le temps. N'écrivez pas de romans. Des réponses très courtes sont suffisantes pour répondre à l'ensemble des questions. Bon courage !

1 Questions de cours - 5 pts

1. Citer les 3 matrices principales utilisées pour projeter un point du repère du monde vers l'écran. Pour chacune d'entre elles, donner l'espace de départ et d'arrivé.
2. A quoi sert le buffer de profondeur (ou *z-buffer*) dans le pipeline graphique ?
3. Ou va se projeter le point $(0, 1, 0)$ sur l'écran si on ne lui applique aucune transformation ?
4. Rappeler (avec un algorithme simplifié) comment fonctionne le shadow-mapping.
5. Dans un vertex shader, on applique la transformation suivante : vec3 P = (modelviewMatrix*vec4(position,1)).xyz
   (a) dans quel espace se trouve le point P ?
   (b) que représente le scalaire float d = length(P) ? (rappel : length calcule la norme d'un vecteur)
6. Qu'est-ce qu'une interpolation bilinéaire et pourquoi est-elle utilisée par OpenGL ?

2 Maillage et texture - 6 pts

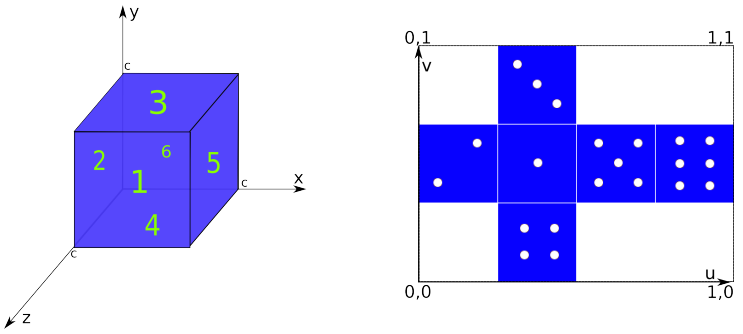


FIGURE 1 – A gauche : le cube de coté c que l'on veut modéliser. A droite, la texture que l'on souhaite plaquer. Les numéros correspondent : le chiffre 1 dans la texture doit être plaqué sur la face 1 (la face avant) du cube, la partie contenant le chiffre 2 doit être plaquée sur la face 2 (à gauche) du cube et ainsi de suite.

On souhaite faire apparaître un dé (de taille c) dans une image de synthèse. Pour cela, on le modélise avec un cube texturé (Fig. 1). La texture doit être plaquée sur les 6 faces du cube, de sorte à ce que les numéros du dé apparaissent correctement. Nous modélisons le dé avec un tableau de coordonnées, en mode indexé (avec des tableaux d'attributs plus un tableau séparé contenant des indices reliant les sommets pour former les triangles).

1. Ecrivez le tableau contenant les coordonnées de positions des sommets, en allant à la ligne après chaque face. Le tableau doit contenir 6 faces * 4 sommets par face * 3 coordonnées par sommets = 72 valeurs. Notez que comme les sommets partagent tous 3 faces, chacun d'entre eux sera répété 3 fois dans le tableau, de sorte à ce que les faces soient indépendantes les unes des autres. Vous respecterez l'ordre donné pour les faces (1 : face avant, 2 : face gauche, 3 : face dessus, 4 : face dessous, 5 : face droite, 6 : face arrière). Enumérez les sommets dans le sens inverse des aiguilles d'une montre pour chaque face. La première ligne du tableau vous est donnée :

   ```
   0,0,c, c,0,c, c,c,c, 0,c,c // 1 : face avant
   ```

2. De la même manière, écrivez le tableau contenant les normales en chacun de ces points. Ce tableau contient autant d'éléments que dans le tableau précédent. Les normales devraient toutes être égale sur les sommets d'une même face.
3. Ecrivez maintenant le tableau contenant les coordonnées de texture de ces points. Pour plus de clarté, écrivez les nombres sous forme de fraction (la texture du dé est divisée en quarts sur l'abscisse et en tiers sur l'ordonnée).

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec2 coord;

uniform mat4 mdvMat;      // modelview matrix
uniform mat4 projMat;     // projection matrix

void main() {
  gl_Position = projMat*mdvMat*vec4(position,1.0);
}
```

Vertex shader

```
uniform sampler2D texDe; // la texture du dé

out vec4 outBuffer; // framebuffer (affichage)

void main() {
  outBuffer = vec4(1,0,0,1);
}
```

Fragment shader

FIGURE 2 – Cube texturé

4. Ecrivez maintenant le tableau contenant les indices des sommets qui forment les triangles à afficher. Ce tableau contient 6 faces * 2 triangles par face * 3 indices = 36 valeurs (respectez toujours l'ordre des faces et revenez à la ligne à chacune d'entre elles).

On dispose des shaders de la figure 2 pour afficher le cube.

1. Avec quelle couleur le cube est-il affiché par défaut ?
2. Combien de fois le vertex shader sera-t-il exécuté ?
3. Ecrivez ce qu'il manque dans les vertex et fragment shaders pour appliquer la texture sur le cube.
4. Est-ce que l'on pourrait obtenir le même résultat avec un tableau d'attributs contenant 8 sommets (au lieu de 24 actuellement) ? Pourquoi ?

3 Shading - 4 pts

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;

uniform mat4 mdvMat;      // modelview matrix
uniform mat4 projMat;     // projection matrix
uniform mat4 normalMat;   // normal matrix

out vec3 normalV;

void main() {
  normalV     = normalMat*normal;
  gl_Position = projMat*mdvMat*vec4(position,1.0);
}
```

Vertex shader

```
in  vec3 normalV;
out vec4 outBuffer;

void main() {
  vec4 ambientColor = vec4(0.2,0.2,0.2,1);
  vec4 diffuseColor = vec4(0,0,1,1);
  vec4 specularColor = vec4(1,1,0,1);
  float r = 10;

  vec3 l = normalize(vec3(-1,1,0));
  vec3 n = normalize(normalV);
  vec3 v = normalize(vec3(0,0,-1));

  float d = max(dot(n,l),0);
  float s = pow(max(dot(reflect(l,n),v),0),r);

  outBuffer = ambientColor + d*diffuseColor + s*specularColor;
}
```

Fragment shader

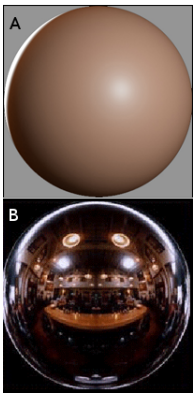
FIGURE 3 – Phong shading

On affiche une sphère avec les shaders de la figure 3 qui appliquent un éclairage de Phong. Les vecteurs l, n et v du fragment shader représentent respectivement la direction de la lumière, la normale et le vecteur vue utilisés pour obtenir le rendu final (tous ces vecteurs sont définis dans le repère de la caméra).

1. Pourquoi le vecteur $0, 0, -1$ est-il utilisé comme vecteur vue ?
2. Pourquoi la normale est-elle (re)normalisée dans le fragment shader (variable "n") ?
3. Que représentent les variables "d" et "s" dans le fragment shader ?
4. Ou se trouve la lumière par rapport à la caméra (variable "l") ?
5. Représentez la sphère sur une figure en faisant (approximativement) apparaître :
   (a) le point où le reflet est maximum
   (b) la couleur en ce point
   (c) une ligne représentant la limite de la zone éclairée par la lumière directionnelle
   (d) la couleur hors de cette zone

4 Lit-Sphere - 3 pts

On souhaite pouvoir obtenir des rendus variés sur des objets quelconques facilement et rapidement. Pour cela, on utilise les textures et shaders ci-dessous. Le vertex shader est le même que dans l'exercice précédent. Les questions portent sur le fragment shader (la normale est positionnée dans le repère de la caméra). Les textures A et B sont utilisées comme entrée du shader (sphereTex) :



2 textures

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;

uniform mat4 mdvMat;      // modelview matrix
uniform mat4 projMat;     // projection matrix
uniform mat4 normalMat;   // normal matrix

out vec3 normalV;

void main() {
  normalV     = normalMat*normal;
  gl_Position = projMat*mdvMat*vec4(position,1.0);
}
```

Vertex shader

```
in  vec3 normalV;
out vec4 outBuffer;

uniform sampler2D sphereTex;

void main() {
  vec4 c = vec4(1,0.8,0.8,1);

  vec3 n = normalize(normalV);
  vec2 c = n.xy*0.5+0.5;
  vec4 t = texture(sphereTex,c);

  outBuffer = c * t;
}
```

Fragment shader

FIGURE 4 – Un rendu mystère

1. Dans quel intervalle se trouvent les 2 premières coordonnées de la normale n.xy ?
2. Comment est orientée la surface de l'objet par rapport à la caméra si la normale est égale à (0,0,1) ? à (1,0,0) ? à (0,1,0) ?
3. Quelle seront les coordonnées de texture obtenues dans ces 3 cas ?
4. Quel sera le type de matériau obtenu si on utilise la texture A en entrée (dans sphereTex) ? la texture B ?
5. Que fait ce shader et en quoi est-il utile ? Quelle est sa principale limitation ?

5 Post-process - 3 pts

On souhaite appliquer un effet sur une image de 512x512 pixels. Pour cela, on affiche un rectangle de la taille du viewport, et on plaque l'image sur celui-ci. Dans le code ci-dessous, la variable "texcoord" $\in [0,1]$ permet d'accéder à l'image que l'on veut modifier (texcoord=vec2(0,0) dans le coin bas-gauche, et texcoord=vec2(1,1) dans le coin haut-droit). L'utilisation de texture(img,texcoord) afficherait alors simplement l'image dans le viewport.

```
in vec2 texcoord;

uniform sampler2D img;

void main() {
  vec2 ps = 1.0/vec2(textureSize(img,0)); // size of a pixel

  vec4 gx = texture(img,texcoord+vec2(ps.x,0))-texture(img,texcoord-vec2(ps.x,0));
  vec4 gy = texture(img,texcoord+vec2(0,ps.y))-texture(img,texcoord-vec2(0,ps.y));

  float m = length(gx.xyz)+length(gy.xyz);

  outBuffer = texture(img,texcoord)-vec4(vec3(m),0);
}
```

FIGURE 5 – Fragment shader

1. Combien de fois le fragment shader est-il exécuté ?
2. Que représentent les variables gx et gy ?
3. Que contient le scalaire m ?
4. Quel est l'effet produit ?