Tuto 4

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: Tuto 4

Imprimé par: Theo bonnet
Date: vendredi 28 février 2020, 14:54

Table des matières

Principe de l'éclairage

La lumière est nécessaire pour voir : ce sont les rayons lumineux, issus d'une source lumineuse qui interagissent avec les objets avant d'être reçus par nos capteurs (notre rétine ou un capteur ccd). Pour que le monde virtuel que nous sommes en train de créer soit le plus réaliste possible, il faut donc simuler les phénomènes qui interviennent dans l'éclairage.
Jusqu'à mantenant si notre scène virtuelle se passait d'éclairage c'est parce que nos objets émettaient leur propre lumière, c'est à dire leur teinte. Tous les points de la face rouge du cube avaient toujours exactement la même teinte à l'écran, indépendamment de l'inclinaison de la face ou de l'éloignement entre la face et la caméra.
Avec une source lumineuse externe, il va falloir tenir compte :

- de la position de la source lumineuse par rapport à l'objet éclairé, en particulier de l'angle avec lequel les rayons lumineux arrivent sur la surface ; mais aussi de l'éloignement de cette source car il y a naturellement une perte d'énergie avec la distance ;
- de la nature de la source lumineuse car la lumière peut être colorée et plus ou moins puissante. Elle peut être pontuelle ou étendue ;
- de la nature de l'interaction de la lumière et de l'objet : quelle quantité de lumière reçue est absorbée et quelle quantité est réfléchie ? dans quelle(s) direction(s) ? Un objet mat et un objet brillant ne diffusent pas de la même manière. Un objet rouge absorbe la composante cyan lorsqu'il est éclairé par une lumière blanche. Donc si la lumière est bleue le même objet apparaitra noir...
- certaines parties de l'objet ne reçoivent aucune lumière directe car les rayons lumineux issus directement de la source sont stoppés mais sont quand même éclairés par des rayons lumineux réfléchis par d'autres parties de la scène..

Les informaticiens ont proposé des modèles informatiques de plus en plus perfectionnés pour tenir compte de toutes ces interactions ; le plus simple, qui date de 1973 s'appelle **modèle de Phong**. C'est un modèle d'éclairage *local* car il ne prend en compte que la position de la source lumineuse et de l'observateur et ne tient pas compte d'un éventuel éclairage indirect.

Modèle de Phong

Le modèle de Phong additionne le résultat de 3 composantes :

1. une composante **ambiante** qui simule un éclairage indirect diffus qui teinte de la même façon tous les éléments de la scène indépendamment de leur position ou de lur inclinaison ;
2. une composante **diffuse** qui tient compte de l'orientation de la face vis à vis des rayons lumineux : plus les rayons lumineux font un angle proche de 90° avec la face plus la face reçoit de lumière ; en revanche avec une lumière rasante (angle proche de 0° ou 180°) l'éclairage est faible. Si l'angle est négatif, l'éclairage est nul car la lumière arrive pardessous. La composante diffuse est émise (diffusée) dans toutes les diretcions en quantité égale.
3. une composante **spéculaire** qui produit un reflet spéculaire "brillant". Cette composante dépend à la fois de l'angle des rayons lumineux incidents et de la position de l'observateur. C'est quand l'observateur se trouve dans une direction bien précide qu'il va recevoir le maximum de lumière réfléchie par la surface.

Composante ambiante

Il faut que la source lumineuse ait une composante ambiante, qui est un vecteur $L_a = (R, G, B)$ peu saturé. On prend souvent un gris foncé. Ce type de lumière va permettre d'éviter des zones trop sombre dans la scène car tous les points vont recevoir cet éclairage.
La composante ambiante de la lumière émise E_a par la surface est le produit (terme à terme) de la componente ambiante de la lumière par la composante ambiante de la couleur de l'objet C_a qui est souvent la même que sa couleur diffuse.

$$E_a = L_a \times C_a$$

Composante diffuse

Si L_d est la composante diffuse de la source lumineuse et C_d est la composante diffuse de la lumière de l'objet, alors la composante diffuse de la lumière émise E_d est

$$E_d = L_d \times C_d \times \cos(\alpha)$$

où α est l'angle entre le rayon lumineux incident et un vecteur N perpendiculaire à la surface de l'objet appelé *vecteur normal*.

Composante spéculaire

Si L_s est la composante diffuse de la source lumineuse et C_s est la composante diffuse de la lumière de l'objet, alors la composante diffuse de la lumière émise E_s est

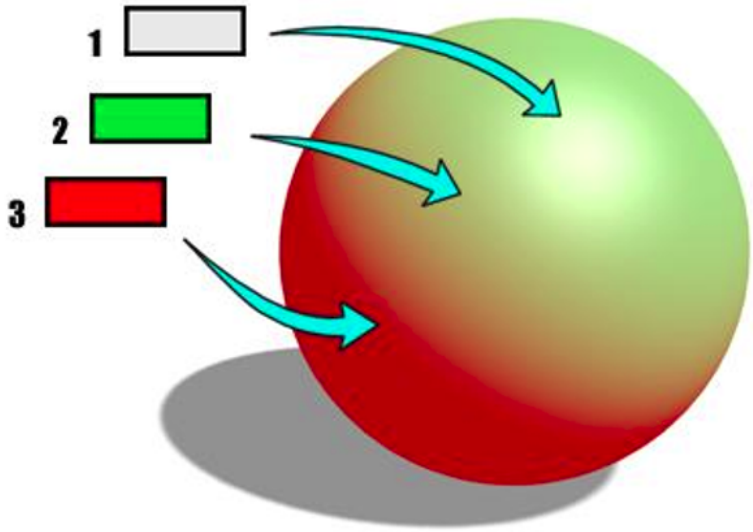
$$E_s = L_s \times C_s \times \cos^{n_s}(\beta)$$

où β est l'angle entre le rayon lumineux réfléchi spéculairement et le *vecteur de visée* qui pointe dans la direction de l'observateur et n_s est un coefficient de réfléxion spéclaire.

Au total

Selon le modèle de Phong la couleur finale de l'objet est donc

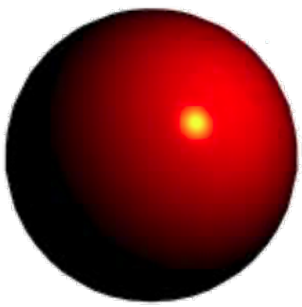
$$Col = E_a + E_d + E_s$$

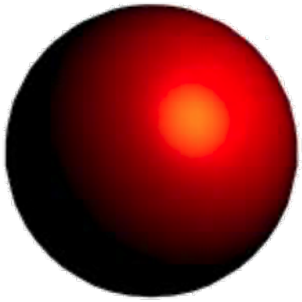


1. Couleur spéculaire
2. Couleur diffuse
3. Couleur ambiante

Remarques

- En général une source lumineuse a la composante diffuse et spéculaire idéntique (souvent blanc d'ailleurs) et une composante ambiante de type gris ;
- Pour un objet la composante diffuse et la composante spéculaire sont souvent les mêmes et sont souvent donnée la couleur lue dans une texture ;
- Pour un objet la composante spéculaire est souvent du blanc (même pour un objet qui n'est pas blanc) : c'est ce qui va produire le reflet blanc à sa surface d'autant plus clair que son coefficient spéculaire est élevé. Mais dans certains cas la composante spéculaire peut aussi être donnée par une texture, soit pour représenter le fait que le coefficient de réflexion spéculaire varie selon les zones de l'objet, soit pour simuler la réflexion de l'environnement autour de l'objet (et on utilise dans ce cas une texture cubique).





Implémentation dans un fragment shader

Pour avoir un meilleur résultat le calcul de l'éclairage est implémenté au niveau des fragments et non des vertex. Les calculs se font donc dans le fragment shader dans les coordonnées du repère de la scène. Pour cela le vertex shader doit fournir pour chaque fragment :

- la position du fragment (dans le repère de la scène) `outPos`
- le vecteur normal à la face à laquelle le fragment appartient toujours dans le repère de la scène) `outNormal`

Voici le code du vertex shader :

```
attribute highp vec3 rv_Position;
attribute highp vec2 rv_TexCoord;
attribute highp vec3 rv_Normal;

uniform   highp mat4 u_ModelMatrix;
uniform   highp mat4 u_ViewMatrix;
uniform   highp mat4 u_ProjectionMatrix;
uniform   highp float u_opacity;
uniform   highp vec4 u_color;

varying   highp vec4 outColor;
varying   highp vec2 outTexCoord;
varying   highp vec3 outPos;
varying   highp vec3 outNormal;

void main(void)
{
    outPos = vec3(u_ModelMatrix * vec4(rv_Position, 1.0));
    outNormal = vec3(u_ModelMatrix * vec4(rv_Normal, 0.0));
    outColor = vec4(u_color.rgb, u_opacity);
    outTexCoord = rv_TexCoord;

    gl_Position = u_ProjectionMatrix * u_ViewMatrix * vec4(outPos, 1.0);
}
```

Remarques :

- Au lieu de passer au vertex shader en une seule matrice le produit des trois matrices ModelViewProjection, comme on l'a fait jusqu'à maintenant, on a besoin des trois matrices séparément dans le sahder : en particulieron a besoin de la matrice Model qui transforme les coordonnées de l'objet depuis les repère local à l'objet au repère de la scène. Car on doit passer au fragment shader les coordonnées du sommet et son vecteur normal *dans ce repère*. Ensuite on multiplie par les matrices vue puis projection pour avoir `gl_Position` comme avant ;
- Quand on compléte les `vec3` en `vec4` pour les utiliser dans la multiplication matricielle par une matrice 4x4 on ne travaille pas de la même façon avec `rv_Position` qui est un point et `rv_Normal` qui est un vecteur. Voir la leçon sur les coordonnées homogènes pour en comprendre la raison.

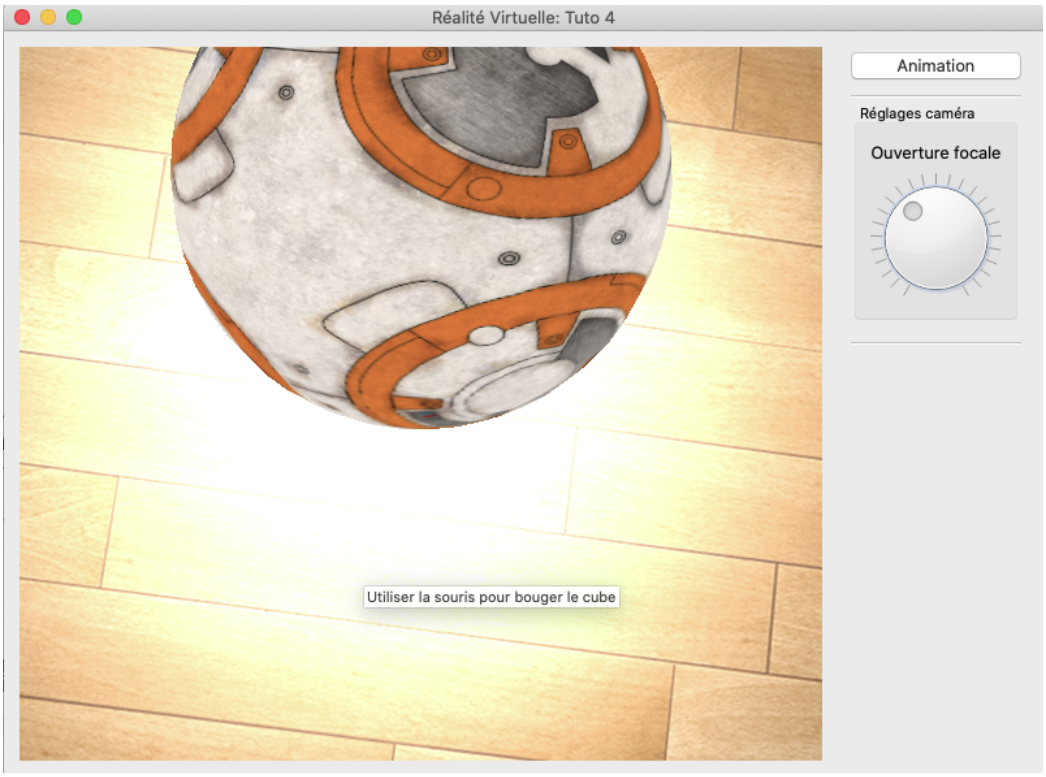
Dans le fragment shader, il y aura donc beaucoup de nouvelles variables uniformes, correspondant aux caractéristiques de la source lumineuse ansi que la position de l'observateur.

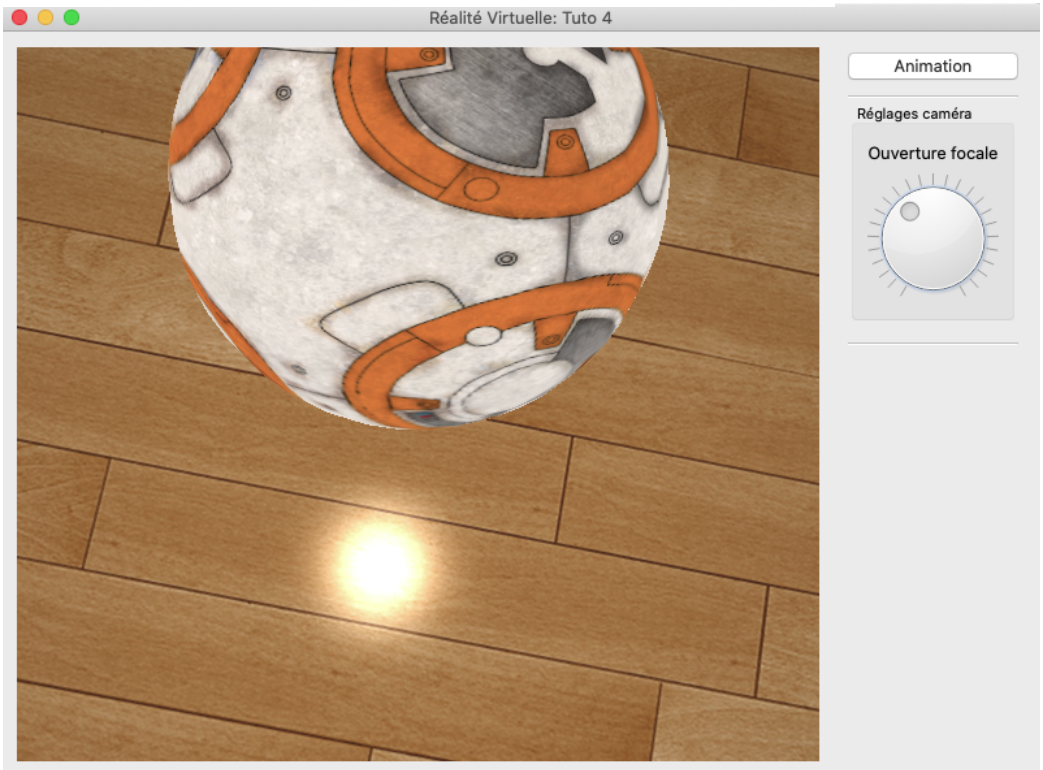
```
varying highp vec2 outTexCoord;
varying   highp vec3 outPos;
varying   highp vec3 outNormal;

uniform sampler2D texture0;

uniform vec4 light_ambient_color;
uniform vec4 light_diffuse_color;
uniform vec4 light_specular_color;
uniform float light_specular_strength;
uniform vec3 light_position;
uniform vec3 eye_position;

void main(void)
{
    //Lecture de la texture
    vec4 color = texture2D(texture0, outTexCoord);

    //calcul de la composante ambiante
    vec4 ambient = color * light_ambient_color;

    //calcul de la composante diffuse
    vec3 norm = normalize(outNormal);
    vec3 lightdir = normalize(light_position - outPos);
    float coeff_diffusion = max(dot(lightdir, norm), 0.0);
    vec4 diffuse = (coeff_diffusion * light_diffuse_color) * color;

    //calcul de la composante spéculaire
    vec3 viewdir = normalize(outPos - eye_position);
    vec3 reflectdir = reflect(lightdir, norm);
    float coeff_specular = pow(max(dot(viewdir, reflectdir), 0.0), light_specular_strength);
    vec4 specular = (coeff_specular * light_specular_color) * vec4(1.0, 1.0, 1.0, 1.0);

    //couleur finale
    gl_FragColor = ambient + diffuse + specular ;
    gl_FragColor.a = outColor.a;
}
```

Remarques :

- la fonction `dot` calcule le produit scalaire entre deux vecteurs. Comme les vecteurs sont unitaires, cela donne le cosinus de l'angle entre les vecteurs ;
- la `max` sert à donner un éclairage nul dès que l'angle dépasse 90° (cosinus négatif) ;
- la fonction `reflect` calcule le vecteur obtenu par réflection par rapport à la droite donnée par son second argument : ici il sert à calculer le vecteur de réflexion spéculaire.

Voici quelques images (sur notre sol lambrissé) de halo donné par la réflexion spéculaire avec des coefficients spéculaires différents.




Réalité Virtuelle: Tuto 4
Animation
Réglages caméra
Ouverture focale

Normales

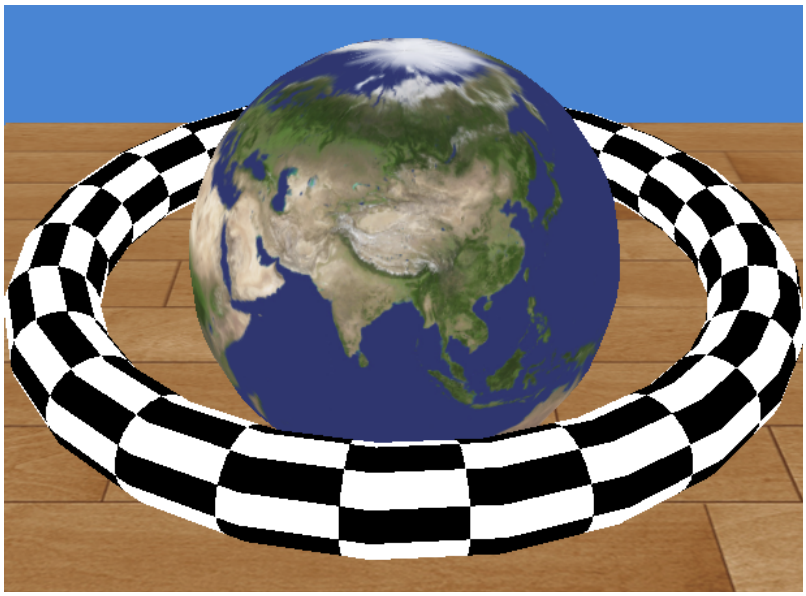
Pour que les algorithmes d'éclairage puissent fonctionner (et quel que soit l'algorithme choisi) il faut obligatoirement que le modèle 3D intègre des informations sur les normales. C'est à dire que dans le vertex buffer, chaque sommet doit porter (en plus de ses propres coordonnées dans le repère local) un vecteur qui pointe dans une direction perpendiculaire à la face à laquelle appartient ce sommet dans un sens sortant. On fait en sorte que ce vecteur soit normalisé (c'est à dire de norme 1) même si concrètement les algorithmes vont toujours passer par une phase préalable de normalisation avant de l'utiliser dans les calcules d'ombrage.

Eclairer le plan

Mise en place

Le point de départ du tuto4 est un projet qui contient les objets suivants

- plan texturé sans ombre
- sphère texturée avec une seule texture
- tore avec texture en damier
- camera simple



Vecteurs normaux

Dans `RVBody.h` on change le `struct RVVertex` pour lui ajouter un nouvel attribut appelé `normal` de type `QVector3D` . On modifie le constructeur de `RVVertex` pour que ce dernier attribut ait une valeur par défaut nulle.

```
    struct RVVertex {
    QVector3D position;         //! position du sommet
    QVector2D texCoord;         //! coordonnées texture
    QVector3D normal;           //! vecteur normal

    RVVertex(QVector3D pos = QVector3D(),
             QVector2D tex = QVector2D(),
             QVector3D normalVector = QVector3D())
    {
        position = pos;
        texCoord = tex;
        normal = normalVector;
    }
};
```

Dans la méthode `RVBody::initializeVAO()` il faut faire le lien entre ce troisième attribut du vertex et une nouvelle variable du *vertex shader* qui s'appelle `rv_Normal`. Ce nouvel attribut se compose de 3 *float* et commence (dans le VBO) après `sizeOf(RVVertex.position)+sizeOf(RVVertex.texCoord)` octets. Donc :

```
m_program.setAttributeBuffer("rv_Normal", GL_FLOAT, sizeof(RVVertex::position)+sizeof(RVVertex::texCoord),
                             3, sizeof(RVVertex));
m_program.enableAttributeArray("rv_Normal");
```

On modifie le vertex buffer de `RVPlane` dans `RVPlane::initializeBuffer` pour que chacun des 4 vertex ait en plus des coordonnées et de la coorodnnée texture, un vecteur normal qui pointe vers le haut.

```
QVector3D up(0, 1, 0);

RVVertex vertexData[] = {
    RVVertex(A, SW, up),
    RVVertex(B, SE, up),
    RVVertex(C, NE, up),
    RVVertex(D, NW, up)
};
```

programmes de shader

Ajouter aux ressources, dans "shaders" deux nouveau programmes de shader `VS_lit_texture.vsh` et `FS_lit_texture.fsh` avec le code indiqué plus haut dans le cours.
Dans le constructeur de `RVBody` utiliser par défaut ces deux fichiers texture.

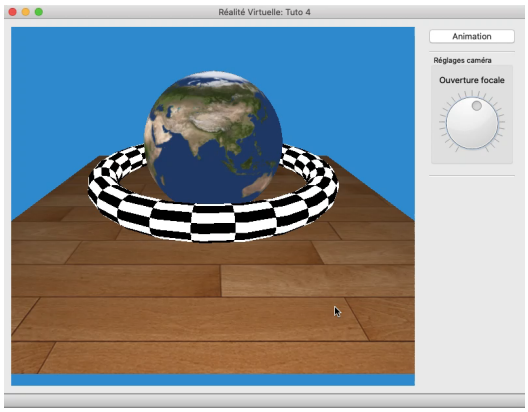
Utilisation des nouveaux shaders dans le plan

Pour pouvoir utiliser, dans le `draw()` de `RVPlane` les nouveaux shaders il y a des variables uniformes à modifier et des nouvelles variables uniformes à définir.

- A modifier : au lieu de passer une seule matrice au Vertex Shader on en passe trois :
  - `u_ModelMatrix` qui est la matrice `model` du plan ;
  - `u_ViewMatrix` qui est la matrice `view` de la camera ;
  - `u_ProjectionMatrix` qui est la matrice projection de caméra.
- A rajouter : les caractéristiques de la source lumineuse au Fragment Shader :
  - `light_ambient_color` qui est la couleur (un `QColor`) de la lumière ambiante. Un gris moyen fait l'affaire.
  - `light_diffuse_color` qui est la composante diffuse de la lumière: blanc ;
  - `light_specular_color`qui est la composante spéculaire de la lumière : blanc aussi ;
  - `light_specular_strength` qui est le coefficient de réflexion spéculaire (par exemple 20) ;
  - `light_position` qui est un `QVector3D` qui donne la position de la source lumineuse ponctuelle dans le repère de la scène ;
  - `eye_position` qui est la position de l'observateur dans le repère de la vue (donc la propriété position de la caméra).

```
m_program.setUniformValue("u_ModelMatrix", this->modelMatrix());
m_program.setUniformValue("u_ViewMatrix", m_camera->viewMatrix());
m_program.setUniformValue("u_ProjectionMatrix", m_camera->projectionMatrix());
m_program.setUniformValue("u_opacity", m_opacity);
m_program.setUniformValue("u_color", m_globalColor);
m_program.setUniformValue("texture0", 0);

m_program.setUniformValue("light_ambient_color", QColor(100, 100, 100));
m_program.setUniformValue("light_diffuse_color", QColor(255, 255, 255));
m_program.setUniformValue("light_specular_color", QColor(255, 255, 255));
m_program.setUniformValue("light_specular_strength", 20.0f);
m_program.setUniformValue("light_position", QVector3D(10, 0, 10));
m_program.setUniformValue("eye_position", m_camera->position());class="hljs-string">"eye_position", QVector3D());
```

Pour pouvoir tester efficacement le programme, il faut modifier le code de `RVWidget::mouseMoveEvent` pour que ce soit le plan qui soit manipulable à la souris et non pas la sphère.



Éclairer la sphère et le tore

Ajouter des normales aux surfaces

La sphère et le tore sont obtenus à partir de la classe `RVSurface` qui construit le maillage de la surface à partir des 3 *fonctions coordonnées* $x(s,t), y(s,t), z(s,t)$ où les *paramètres* $(s,t) \in I \times J$.
Pour puvoir éclairer ces surfaces, il faut que dans le vertex buffer il y ait un vecteur perpendicualaire unitaire sortant pour chaque valeur de s et de t.
Or les deux vecteurs

$$\nabla_s = \begin{pmatrix} \frac{\partial x}{\partial s} \\ \frac{\partial y}{\partial s} \\ \frac{\partial z}{\partial s} \end{pmatrix}$$

et

$$\nabla_t = \begin{pmatrix} \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial t} \end{pmatrix}$$

sont deux vecteurs tangents à la surface (ils définissent le plan tangent à la surface en $(x(s,t), y(s,t), z(s,t))$).
Donc le vecteur obtenu par le *produit vectoriel* des deux vecteurs tangents est perpendiculaire au plan tangent donc va donner la normale à la surface que nous voulons.

$$N = \nabla_s \times \nabla_t$$

Concretement nous allons approcher les dérivées partielles par des différences finies (en prenant des petits incréments `ds` et `dt`).

```
    QVector3D RVSurface::gradS(double s, double t)
{
    return pos(s+ds,t)-pos(s,t);
}

QVector3D RVSurface::gradT(double s, double t)
{
    return pos(s,t+dt)-pos(s,t);
}
```

Ce qui permet d'obtenir la normale par

```
QVector3D RVSurface::normal(double s, double t)
{
    QVector3D n = -QVector3D::crossProduct(gradS(s,t), gradT(s,t));
    n.normalize();
    return n;
}
```

Et on utilse donc la fonction `normal()` dans `RVSurface::initializeBuffer()` pour renseigner le champs `normal` de `RVVertex` :

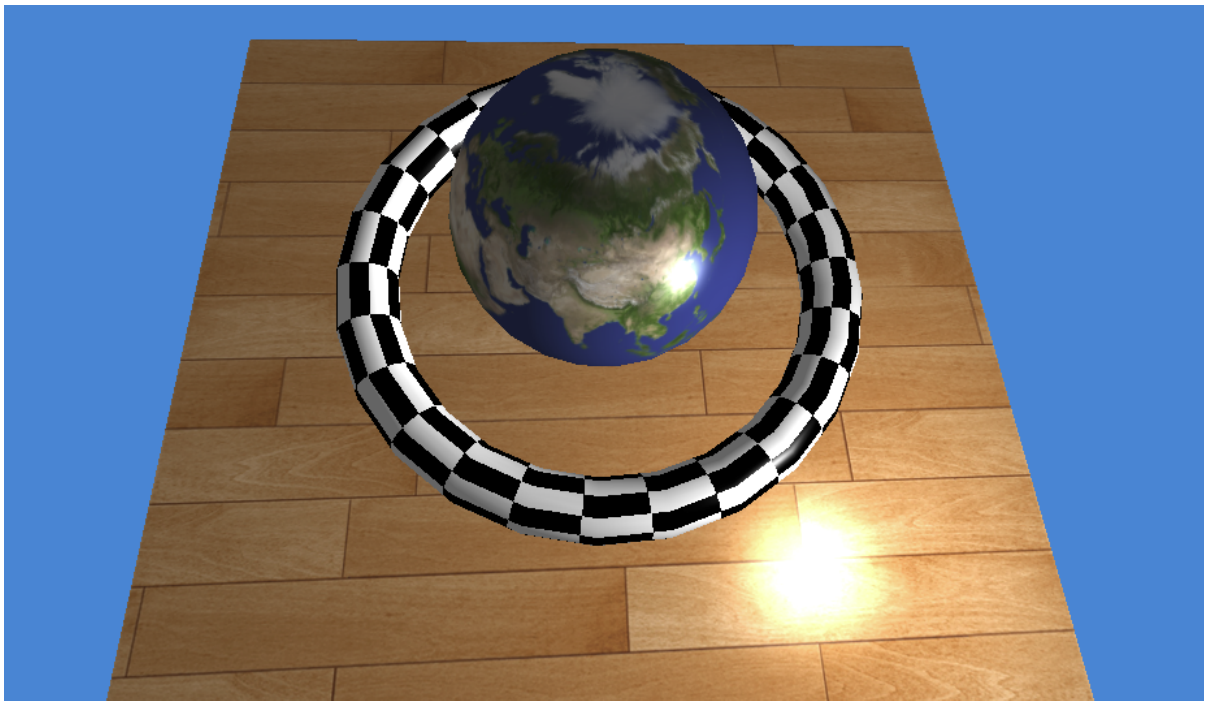
```
    vertexData[cptPoint].normal = normal(s,t);
```

Finalement il n'y a rien d'autre à faire pour avoir l'éclairage sur la sphère texturée (en utilisant les nouveaux shaders).

Éclairer le tore

En revanche si on veut éclairer le tore à damier il faut créer un nouveau fragment shader `FS_lit_damier.fsh` qui mélange le code qui donne la couleur à damier qui va être la couleur de base (diffuse et ambiante de l'objet) avec le code qui gère l'éclairage :

```
varying highp vec4 outColor;
varying highp vec2 outTexCoord;
varying   highp vec3 outPos;
varying   highp vec3 outNormal;

uniform sampler2D texture0;

uniform vec4 light_ambient_color;
uniform vec4 light_diffuse_color;
uniform vec4 light_specular_color;
uniform float light_specular_strength;
uniform vec3 light_position;
uniform vec3 eye_position;

void main(void)
{
    vec4 col1 = outColor;
    vec4 col2 = vec4(0.0, 0.0, 0.0, 1.0);

    float res = mod(floor(26.0*outTexCoord.s)+floor(10.0*outTexCoord.t), 2.0);
    vec4 color = (col1*res + col2*(1.0 - res));

    //calcul de la composante ambiante
    vec4 ambient = color * light_ambient_color;

    //calcul de la composante diffuse
    vec3 norm = normalize(outNormal);
    vec3 lightdir = normalize(light_position - outPos);
    float coeff_diffusion = max(dot(lightdir, norm), 0.0);
    vec4 diffuse = (coeff_diffusion * light_diffuse_color) * color;

    //calcul de la composante spéculaire
    vec3 viewdir = normalize(outPos - eye_position);
    vec3 reflectdir = reflect(lightdir, norm);
    float coeff_specular = pow(max(dot(viewdir, reflectdir), 0.0), light_specular_strength);
    vec4 specular = (coeff_specular * light_specular_color) * vec4(1.0, 1.0, 1.0, 1.0);

    //couleur finale
    gl_FragColor = ambient + diffuse + specular ;
    gl_FragColor.a = outColor.a;

}
```



Rendu Offscreen

Principe

Par défaut le moteur OpenGL fait le rendu dans un *frame buffer* qui est attaché à l'écran. Un frame buffer de OpenGL est composé de plusieurs buffers :

- le *color buffer* qui contient les couleurs des fragments
- le *depth buffer* dans lequel sont stockées les profondeurs des fragments visibles ; c'est grace au depth buffer que OpenGL met en place l'algorithme du *Z-buffer* pour l'élimination des parties cachées ;
- le *stencil buffer* peut être utilisée pour limiter le rendu à certaines parties de l'écran...

On peut choisir de demander à OpenGL de ne pas faire le rendu dans le framebuffer principal mais dans un framebuffer secondaire (en mémoire). De cette façon on peut récupérer sous forme de texture (par exemple) le résultat du rendu du color buffer.
On appelle cette technique le *rendu offscreen* car l'image produite n'est pas affichée à l'écran.
Cela peut être utile pour plusieurs raisons :

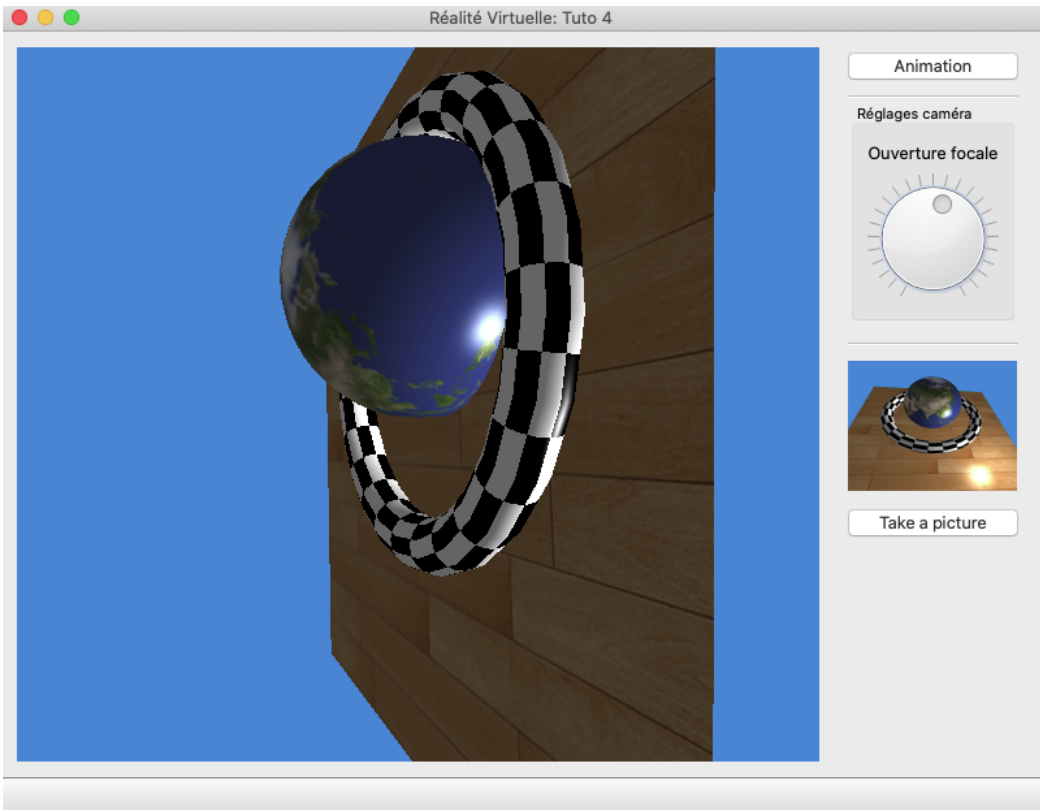
- Récupérer le rendu dans une image à sauvegarder dans un fichier ;
- Mettre en place des effets de réflexion ;
- Implémenter le picking, c'est à dire la possibilité de choisir, en cliquant dessus, l'un des objets de la scène ;
- Ajouter au rendu OpenGL des effets de postprocessing (flou, extraction de contours, etc...) calculés par le GPU.

Frame Buffer Object

Le frame buffer associé à l'écran est créé automatiquement par le contexte de rendu de OpenGL. Si l'on veut un second frame buffer pour faire du rendu off screen, on doit explicitemt créer un *Frame Buffer Object* (abrégé en FBO). Cette création est fastidieuse et délicate en général mais Qt offre une classe `QOpenGLFrameBufferObject` qui simplifie cette étape de création.
En revanche, une fois que le FBO est créé, le rendu "en texture" est très simple : il suffit de *lier* le FBO au contexte OpenGL (avec la commande `bind`) avant de lancer la commande de rendu usuelle; automatiquement c'est le FBO qui sera utilisé par OpenGL.

Concrètement

On veut dans le tuto4 ajouter un bouton `takeImage` qui utilise un FBO pour récupérer dans un QImage le résultat du rendu et l'afficher en petit dans l'IHM



- On ajoute à `RVWidget` une variable `m_fbo` de type `QOpenGLFrameBufferObject *`.
- On initilaise le FBO dans `RVWidget::initializeGL()` avec

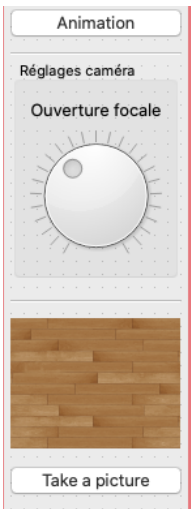
```
m_fbo = new QOpenGLFramebufferObject(this->width(), this->height());
m_fbo->setAttachment(QOpenGLFramebufferObject::CombinedDepthStencil);
```

- On ajoute à `RVWidget` une méthode `QImage RVWidget::getImage()` qui tuilise le FBO pour faire le rendu, puis récupère à partir du FBO l'image du color buffer et la renvoie après l'avoir convertie au bon format.

```
QImage RVWidget::getImage()
{
    makeCurrent();
    m_fbo->bind();
    paintGL();
    QImage img(m_fbo->toImage(true));
    m_fbo->release();

    return QImage(img.constBits(), img.width(), img.height(), QImage::Format_ARGB32);
}
```

Modification de l'IHM

On ajoute à `mainwindow.ui`



- Un QLabel appelé `image` avec une taille minimale 130x100 et une politique *preferred* . Dans sa propriété `pixmap` on met une image enressource (le logo ou autre). Il faut cocher la case `scaledContents`
- Un QButton "Take a Picture"
  dont l'action `clicked` est asscoiée à un *slot* `takePicture` de `MainWindow`
- Dans le slot `MainWindow::takePicture` on appelle la méthode `getPicture()` de `RVWidget` et on utilise l'image obtenue pour la mettre dans le label `image`.

```
void MainWindow::takePicture()
{
    QImage img(ui->widgetRV->getImage());
    ui->image->setPixmap(QPixmap::fromImage(img));
}
```