

# Tuto 3

Site: [lms.univ-cotedazur.fr](https://lms.univ-cotedazur.fr)  
Cours: Realite virtuelle - EIMAD919  
Livre: Tuto 3

Imprimé par: Theo bonnet  
Date: vendredi 28 février 2020, 14:53

## Table des matières

### **1. Principe des textures**

### **2. Texture sur le plan**

- 2.1. Modification du VBO
- 2.2. Modification du Vertex Shader
- 2.3. Modification du VAO
- 2.4. Chargement de la texture
- 2.5. Modification du Fragment Shader
- 2.6. Affichage
- 2.7. Test
- 2.8. Ajout du dégradé

### **3. Texture dans RVBody**

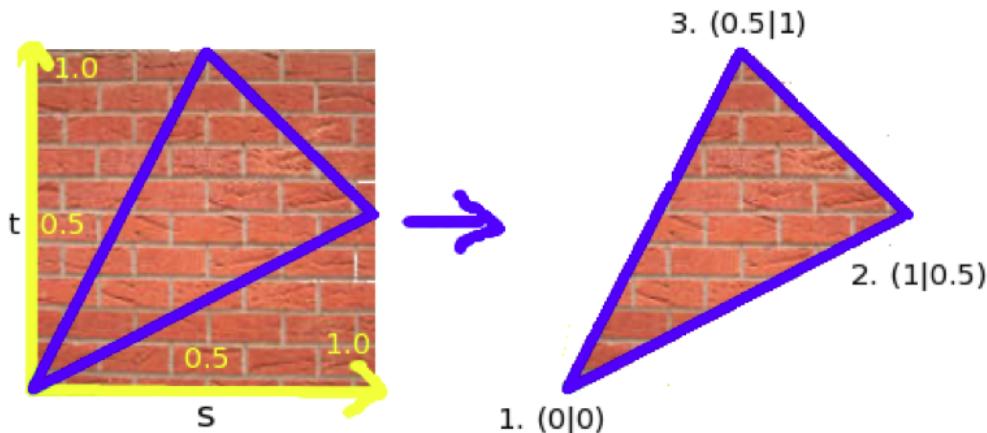
- 3.1. Mélange de deux textures
- 3.2. Texture procédurale

## 1. Principe des textures

En synthèse d'images les textures sont utilisées pour ajouter du réalisme aux objets rendus. Souvent c'est une image (bitmap) qui est utilisé pour donner la couleur diffuse de l'objet. Mais une texture peut aussi être procédurale (c'est à dire que générée par un algorithme, comme du bruit) et être utilisée dans des contextes plus généraux : pour définir un reflet spéculaire, pour simuler un éclairage sphérique, pour être utilisé comme source de donnée dans les programmes de shader. Les textures OpenGL ont un format spécifique, indépendant du périphérique et de la plateforme. Il faut donc en général convertir l'image lue à partir d'un fichier en une texture OpenGL. Heureusement, dans Qt, il y a la classe `QOpenGLTexture` qui offre tout un tas de méthodes utiles.

### Coordonnées texture

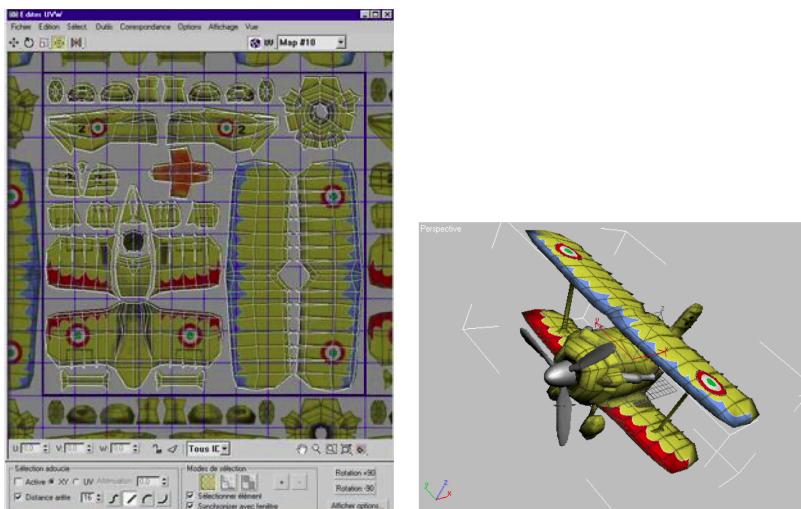
Pour pouvoir appliquer la texture sur les faces du maillage, il faut que chaque sommet possède des *coordonnées textures*, c'est à dire un couple de réels (souvent entre 0 et 1) qui est utilisé par l'échantillonneur de texture pour trouver la couleur du pixel correspondant.



Souvent, un seul fichier texture (dont la taille est souvent une puissance de 2) contient toutes les images utilisées par les différentes parties du modèle,



Le travail qui consiste à appliquer une texture sur un modèle 3D s'appelle le texture mapping et peut être plutôt complexe.

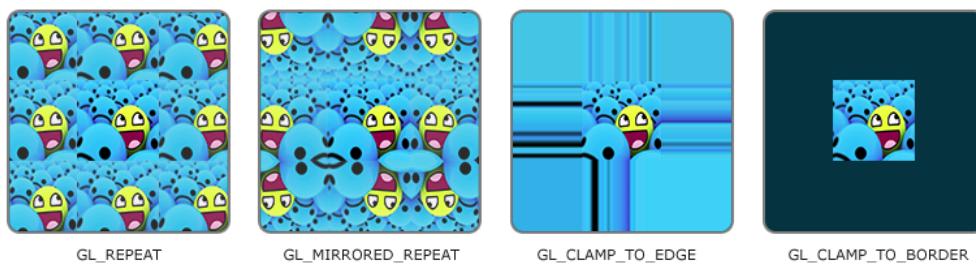


## Texture wrapping

Que se passe-t-il lorsqu'une coordonnée texture **sort** de l'intervalle  $[0, 1]$  ?

OpenGL offre différentes façons de répondre à cette question (qui s'appelle *wrapping mode*) avec la variable d'environnement `GL_TEXTURE_WRAP_S` ou `GL_TEXTURE_WRAP_T` selon que cela concerne la première ou la seconde coordonnée texture :

- `GL_REPEAT` : c'est le comportement par défaut. La texture est répétée à l'identique : par exemple l'intervalle  $[1, 2]$  est identique à  $[0, 1]$ . En quelque sorte on raisonne *modulo 1*;
- `GL_MIRRORED_REPEAT` : comme précédemment sauf qu'on applique une réflexion spéculaire à chaque répétition
- `GL_CLAMP_TO_EDGE` : toute variable qui dépasse 1 prend valeur 1 et toute variable inférieure à 0 vaut 0. Autrement dit on reste sur le bord de la texture.
- `GL_CLAMP_TO_BORDER` : pour les valeurs en dehors de l'intervalle on affecte une couleur fixe (par exemple noir).



## Types de texture

En général on utilise des textures bidimensionnelles que l'on plaque sur les triangles du maillage, mais OpenGL offre aussi la possibilité d'utiliser

- des textures cubiques (cubemap) pour définir des skybox ou pour les réflexions spéculaires (chaque fois que l'on doit simuler

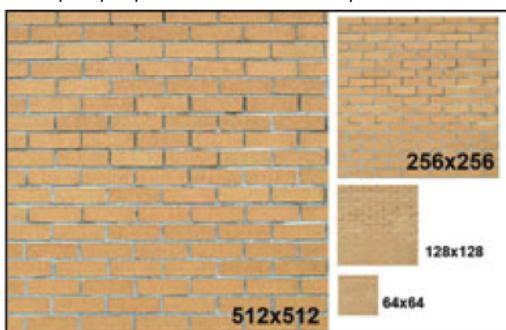


l'environnement qui englobe la scène)

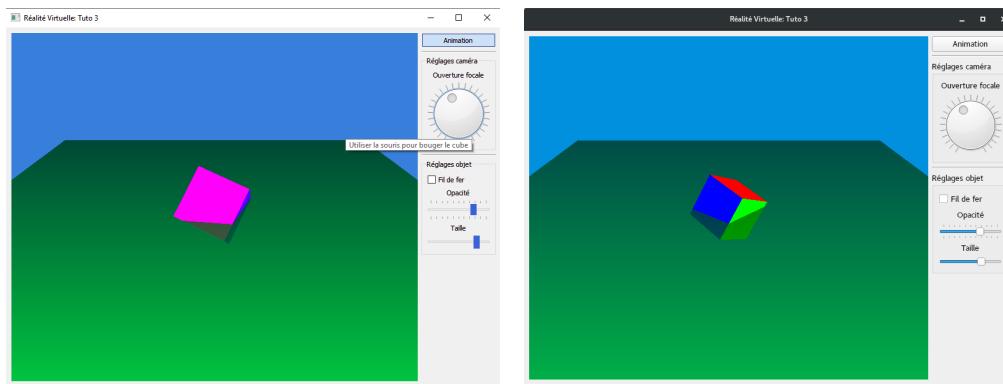
- des textures tridimensionnelles, souvent de type procédural, pour simuler l'aspect à l'intérieur des objets



- des mipmaps qui sont des textures à plusieurs résolutions différentes selon la proximité de l'observateur à l'objet texturé.



## 2. Texture sur le plan



Ouvrir le projet `RVTuto3` de `RVTuto3_start`. Il est essentiellement basé sur le TP2 et contient les classes suivantes :

- classe abstraite `RVBody`;
- classe `RVCamera`;
- classe `RVPlane` et `RVCube` qui héritent de `RVBody`;
- classe `RVWidget` qui affiche le cube et le plan, qui gère la souris et le clavier ainsi que les widgets de l'IHM.

Du point de vue des shaders, il y a :

- `VS_simpler.vsh` qui est le vertex shader par défaut de `RVBody` et qui utilise comme unique attribut les coordonnées des sommets du VBO. Utilise comme variable uniforme (en plus de la matrice) une couleur uniforme et une opacité. C'est ce VS qu'utilise `RVPlan`.
- `FS_simple.fsh` qui est le fragment shader par défaut qui ne fait rien d'autre que prendre la couleur en entrée et l'écrire dans `g1_FragColor`. C'est le FS utilisé par `RVCube`.
- `VS_simple.vsh` est le vertex shader utilisé par le cube coloré `RVCube` qui utilise comme attributs la position du sommet et sa couleur.
- `FS_plan.fsh` qui utilise la coordonnée z (la profondeur) du fragment pour assombrir sa couleur. Utilisé, comme son nom l'indique, par `RVPlane`.

## 2.1. Modification du VBO

Dans un premier temps, nous allons ajouter une texture sur le plan. Pour cela nous allons modifier la méthode `initializeBuffer()` de `RVPlane` pour que chacun des 4 sommets du vertex buffer objet ait, en plus des coordonnées dans l'espace, des coordonnées textures. Pour cela nous allons définir, à l'intérieur de la classe `RVPlane`, un `struct RVVertex` pour associer le `QVector3D` des coordonnées et un `QVector2D` pour les textures :

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

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

Grâce à ce nouveau type, il est facile de remplir le VBO dans `initializeBuffer()`, avec 4 `RVVertex` :

```
QVector3D A(-m_length/2, 0, m_width/2);
QVector3D B(m_length/2, 0, m_width/2);
QVector3D C(m_length/2, 0, -m_width/2);
QVector3D D(-m_length/2, 0, -m_width/2);

QVector2D SW(0, 0);
QVector2D SE(1, 0);
QVector2D NE(1, 1);
QVector2D NW(0, 1);

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

Remarquez que maintenant `vertexData` est un tableau de `RVVertex`. Le reste de la méthode ne va pas changer.

## 2.2. Modification du Vertex Shader

On crée une copie de `VS_simpler.vsh` que l'on renomme `VS_simple.texture.vsh` dans lequel :

- on ajoute une nouvel attribut `rv_TexCoord` de type `vec2`;
- on ajoute une nouvelle variable de type `varying` appelée `outTexCoord`, toujours de type `vec2`;
- dans le `main` du vertex shader on copie l'attribut dans la variable varying pour que ces données soient bien associées au vertex modifié et passé ensuite au fragment shader.

Ce qui donne pour `VS_simple.texture.vsh`

```
attribute highp vec3 rv_Position;
attribute highp vec2 rv_TexCoord;
uniform    highp mat4 u_ModelViewProjectionMatrix;
uniform    highp float u_opacity;
uniform    highp vec4 u_color;
varying    highp vec4 outColor;
varying    highp vec2 outTexCoord;

void main(void)
{
    gl_Position = u_ModelViewProjectionMatrix * vec4(rv_Position,1);
    outColor = vec4(u_color.rgb, u_opacity);
    outTexCoord = rv_TexCoord;
}
```

Après avoir modifié le fichier dans la zone `Other files`, il faut l'ajouter en tant que ressources dans `RVResources.qrc` avec un clic-droit sur shaders puis Ajouter des fichiers existants...

## 2.3. Modification du VAO

Comme nous avons des nouvelles données dans le VBO **et** un nouvel attribut dans le vertex shader, il faut donc modifier la méthode `initializeVAO()` de `RVPlane` pour faire le lien du premier avec le deuxième.

Contrairement à ce qui a été fait dans la classe `RVCube` (la aussi on avait mis dans le vertex buffer les positions et les couleurs) maintenant dans le vertex buffer les informations sur la position et les informations sur les coordonnées textures sont *entrelacées* (alors que dans le cube, on avait les positions des 24 sommets puis les couleurs des 24 sommets).

On va donc utiliser le 5ème argument de la méthode `setAttributeBuffer` (qui est 0 par défaut) qui indique le *stride* (en français le pas, l'enjambée) c'est à dire le nombre d'octets qui séparent les informations sur le premier vertex, de celles sur le second ; dans notre cas le stride sera `sizeof(RVVertex)` (5 floats donc 20 octets).

```
void RVPlane::initializeVAO()
{
    //Initialisation du VAO
    m_vao.bind();
    m_vbo.bind();
    m_ibos.bind();

    //Définition de l'attribut position
    m_program.setAttributeBuffer("rv_Position", GL_FLOAT, 0, 3, sizeof(RVVertex));
    m_program.enableVertexAttribArray("rv_Position");

    //Définition de l'attribut de coordonnée texture
    m_program.setAttributeBuffer("rv_TexCoord", GL_FLOAT, sizeof(RVVertex)::position, 2, sizeof(RVVertex));
    m_program.enableVertexAttribArray("rv_TexCoord");

    //Libération
    m_vao.release();
    m_program.release();
}
```

### Explication en détail

```
m_program.setAttributeBuffer("rv_Position", GL_FLOAT, 0, 3, sizeof(RVVertex));
```

signifie que l'attribut `rv_Position` du vertex shader :

- est composé de floats
- commence au début du VBO (position 0)
- est composé de 3 valeurs (donc 3 floats)
- que pour passer au vertex suivant il faut se déplacer d'autant d'octets qu'est la taille d'un `RVVertex` (20 octets)

```
m_program.setAttributeBuffer("rv_TexCoord", GL_FLOAT, sizeof(RVVertex)::position, 2, sizeof(RVVertex));
```

signifie que l'attribut `rv_TexCoord` du vertex shader :

- est composé de floats
- ne commence pas au début du VBO mais après un certain nombre d'octets égal à la taille d'un `QVector3D` (12 octets) puisque `RVVertex::position` qui représente le champs `position` du `struct` est bien un `QVector3D`;
- est composé de 2 valeurs (donc 2 floats)
- que pour passer au vertex suivant il faut se déplacer d'autant d'octets qu'est la taille d'un `RVVertex` (20 octets)

## 2.4. Chargement de la texture

On ajoute à la classe `RVPlane` une variable membre `m_texture` qui est un pointeur sur un `QOpenGLTexture` et une méthode `void setTexture(QString textureFilename)` qui permet de décider quelle texture on veut coller sur le plan.

Pour le code de cette méthode, on construit une instance de `QImage` à partir du nom de fichier (qui sera souvent une image contenue en ressources) puis une instance de `QOpenGLTexture` à partir de cette `QImage` :

```
void RVPlane::setTexture(QString textureFilename)
{
    if (m_texture)
        delete m_texture;
    m_texture = new QOpenGLTexture(QImage(textureFilename));
}
```

## 2.5. Modification du Fragment Shader

On copie `FS_simple.fsh` en `FS_simple_texture.fsh` et on le rajoute en tant que ressource de type shaders au projet.  
Ce qui change :

- on a une deuxième variable `varying` qui vient du vertex shader qui s'appelle `outTexCoord`
- on aura besoin d'une variable uniforme qui représente la texture qu'il faut échantillonner pour récupérer la couleur du fragment : le type de cette variable est `sampler2D` et on va l'appeler `texture0` (car, comme on va le voir on peut superposer plusieurs textures)
- dans le code du fragment shader, cette fois la couleur finale du fragment `gl_fragColor` est obtenue grâce à la fonction GLSL qui s'appelle `texture2D` et qui prend comme premier argument la texture à échantillonner et comme second argument les coordonnées texture. Comme résultat on obtient un `vec4` avec les 4 composantes de la couleur lue dans la texture.

```
varying highp vec4 outColor;
varying highp vec2 outTexCoord;
uniform sampler2D texture0;

void main(void)
{
    gl_FragColor = texture2D(texture0, outTexCoord.ts);
}
```

Remarque : dans cette nouvelle version du fragment shader (toujours une seule ligne de code!) on n'utilise pas du tout la variable `outColor`. On aurait pu éviter de la passer depuis le vertex shader vers le fragment shader. C'est vrai, mais on va voir dans la suite que l'on pourra *mélanger* la couleur issue de la texture avec la couleur globale venant du vertex shader...

## 2.6. Affichage

### Constructeur

Il faut définir les bons shaders à utiliser.

```
RVPlane::RVPlane(float lenght, float width)
    :RVBody(), m_texture()
{
    m_VSFileName =(":/shaders/VS_simple_texture.vsh";
    m_FSSFileName =(":/shaders/FS_simple_texture.fsh";
    m_length = lenght;
    m_width = width;
}
```

### Méthode draw()

On teste s'il y a bien un fichier texture qui a été défini puis on active dans OpenGL différents réglages liés aux textures :

- Le fait qu'on utilise des textures 2D;
- Le fait qu'on utilise une seule texture, la numéro 0;
- On définit les filtres à utiliser dans l'échantillonage en cas de grossissement ou rapetissement;
- On lie notre texture `m_texture` au contexte de rendu, et donc à la texture 0.

```
if (m_texture) {
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE0);
    m_texture->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
    m_texture->setMagnificationFilter(QOpenGLTexture::Linear);
    //Liaison de la texture
    m_texture->bind();
}
```

Ensuite il faut associer la variable uniforme `"texture0"` du fragment shader à la texture numéro 0 que l'on a activée (toujours avant les commandes de rendu).

```
m_program.setUniformValue("texture0", 0);
```

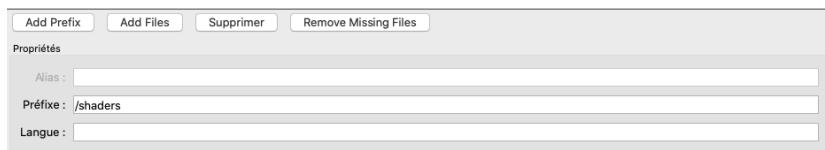
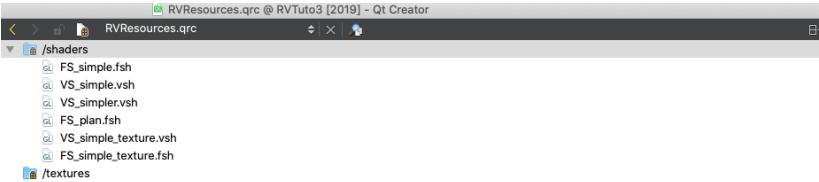
Et après les commandes de rendu on désactive tout ce qu'on a activé.

```
if (m_texture) {
    m_texture->release();
    glDisable(GL_TEXTURE0);
    glDisable(GL_TEXTURE_2D);
}
```

## 2.7. Test

### Nouvelle ressource

On ajoute la texture aux ressources : avec un clic-droit sur `RVResources.qrc` puis `Open in Editor` on a accès à l'éditeur de ressources.



Ce qui permet de :

- créer un nouveau préfixe `Add Prefix` appelé `textures`
- ajouter le fichier `wood.png` dans `textures`.

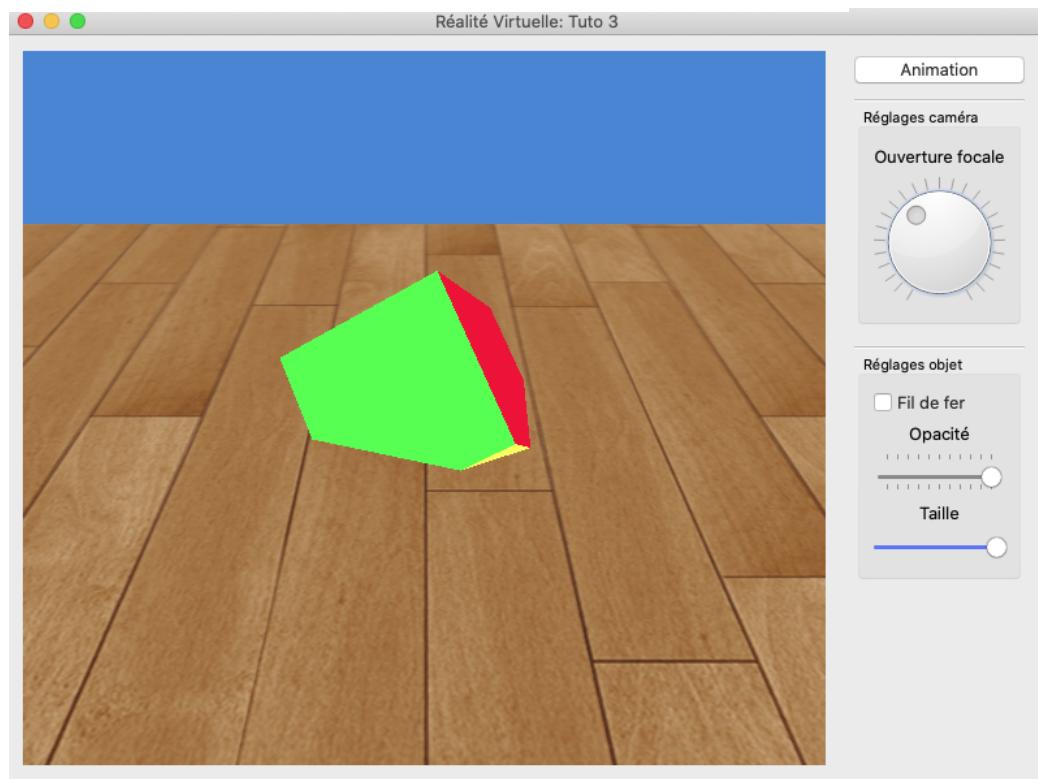
### Modification dans RVWidget

Dans la méthode `initializeGL` de `RVWidget` il n'y a pratiquement rien à changer sauf définir quelle est la texture à utiliser pour `m_plane`.

Même si `m_plane` est une instance de `RVPlane`, la variable a été définie comme pointeur sur `RVBody`. Pour pouvoir accéder à sa méthode `setTexture`, il faut faire un *transtypage dynamique descendant* (downcast) :

```
dynamic_cast<RVPlane*>(m_plane)->setTexture(":/textures/wood.png");
```

Et la compilation donne :



## 2.8. Ajout du dégradé

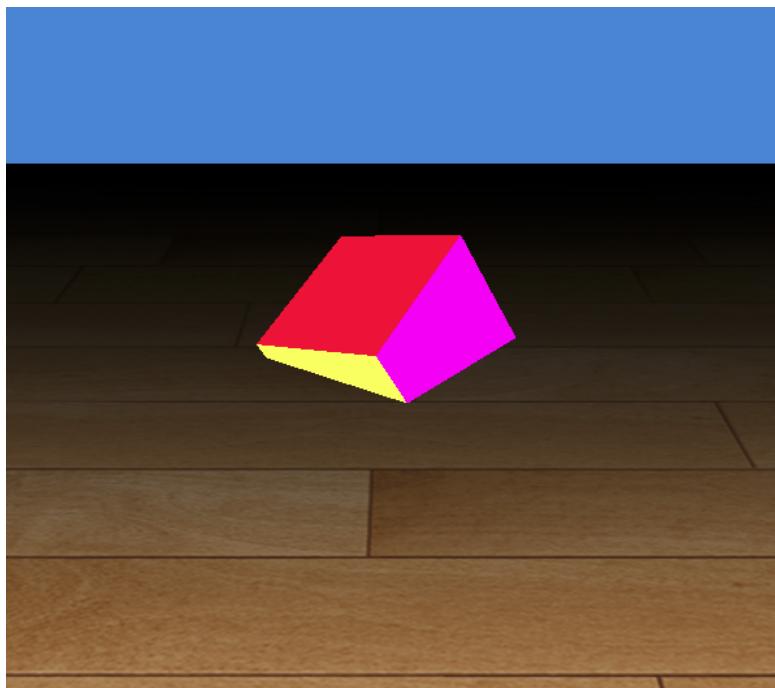
### Ajout du dégradé

On peut bien sûr ajouter au fragment shader le code qui permet d'assombrir les fragments les plus lointains de l'observateur comme on l'avait fait dans le TP2. Pour cela on va dupliquer le shader en un nouveau `FS_plan_texture`. Et dans ce fichier, pour avoir une transition plus douce, nous allons utiliser la fonction GLSL qui s'appelle `smoothstep(min, max, t)` qui renvoie 0 si  $x < min$ , 1 si  $x > max$  et varie de façon lisse (comme un polynôme de degré 3 de 0 à 1 pour  $x \in [min, max]$ ).

```
void main(void)
{
    float t = 1.0 - gl_FragCoord.z;
    t = smoothstep(0.2, 0.8, t);
    gl_FragColor = t*texture2D(texture0, outTexCoord);
    gl_FragColor.a = outColor.a;
}
```

La valeur  $t$  issue de `smoothstep` multiplie la couleur issue de la texture. Remarquez que nous ne voulons pas que la transparence  $\alpha$  varie avec la profondeur !

Le résultat est :



### 3. Texture dans RVBody

Toutes les modifications faites à `RVPlane`, on va les faire passer à `RVBody` pour que toutes ses classes filles puissent utiliser, si elles le veulent, des textures.

- on déplace le `#include <QOpenGLTexture>` dans `rvbody.h`
- on déplace la définition de la structure `RVVertex` à l'intérieur de la définition de la classe `RVBody`
- on déplace la variable membre `m_texture` dans `RVBody`, et on rajoute son initialisation dans le constructeur
- on déplace la méthode `setTexture` dans `RVBody`
- dans le constructeur de `RVBody` on utilise maintenant comme shader par défaut `VS_simple_texture.vsh` et `FS_simple_texture.fsh` qui utilisent les textures
- dans le destructeur de `RVBody` on libère la mémoire allouée à la texture

```
if (m_texture)
    delete m_texture;
```

- dans la méthode `RVBody::initializeVAO()` on reprend le code de `RVPlane`; on enlève donc la méthode `RVPlane::initializeVAO()`.
- dans le constructeur de `RVPlane` on redéfinit le programme du fragment shader avec `FS_plan_texture` (pour le VS on laisse celui par défaut)
- dans `RVWidget`, on n'a plus besoin de transtypage descendant dans `initializeGL()` lorsqu'on appelle `setTexture` sur `m_plane`.
- dans le constructeur de `RVCube` il faut spécifier les programmes de shader simples (sans utilisation de texture) à la place des shaders de texture (qui sont maintenant les shaders par défaut).

Rien ne devrait être changé dans le rendu, mais maintenant grâce à la version 2 de `RVBody`, ce sera très facile d'utiliser des objets 3D avec `texture`.

La raison pour laquelle le cube coloré est affiché comme avant (bien qu'il hérite de `RVBody` qui maintenant intègre les textures) c'est que la classe `RVCube` :

1. intègre les bons shaders
- redéfinit sa propre méthode `initializeVAO()` qui n'utilise pas la structure `RVVertex`
  - bien sur, définit son propre `initializeBuffer()` et `draw()`, ce qui est obligatoire car ce sont des méthodes virtuelles pures.

En revanche `RVPlane` n'a pas besoin de redéfinir `initializeVAO()` car la version par défaut lui convient.

### 3.1. Mélange de deux textures

#### Un globe terrestre

Pour tester le fonctionnement de la v2 de `RVBody`, mais aussi pour ajouter de nouveaux objets à notre scène, on va utiliser `RVSurface` et `RVSphere` qui ont (peut être) été vus dans la session 2. Vous pouvez ajouter à votre projet les 4 fichiers correspondant à ces deux classes (il s'agit de la version 2 de `RVSurface` qui intègre les coordonnées textures et `m_texture` pour être compatible avec la version 2 de `RVBody`).

On ajoute donc à `RVWidget` :

- l'inclusion de `rvsphere.h` dans `rvwidget.h`
- une variable membre `RVBody *m_world;` toujours dans `rvwidget.h`
- on ajoute aux ressources la texture "`earth_daymap.jpg`"
- dans la méthode `initializeGL()` on applique le code typique pour l'initialisation d'un objet 3D
  - on initialise `m_world` comme instance de `RVSphere`
  - on lui affecte la caméra
  - on lui donne une position
  - on lui passe le nom de la texture à utiliser
  - on appelle sa méthode `initialize()` pour terminer l'initialisation
- dans la méthode `paintGL()` on appelle aussi le `draw()` du nouvel objet
- dans la méthode `update()` on appelle aussi sa méthode `rotate` ainsi que dans `mouseMoveEvent()`

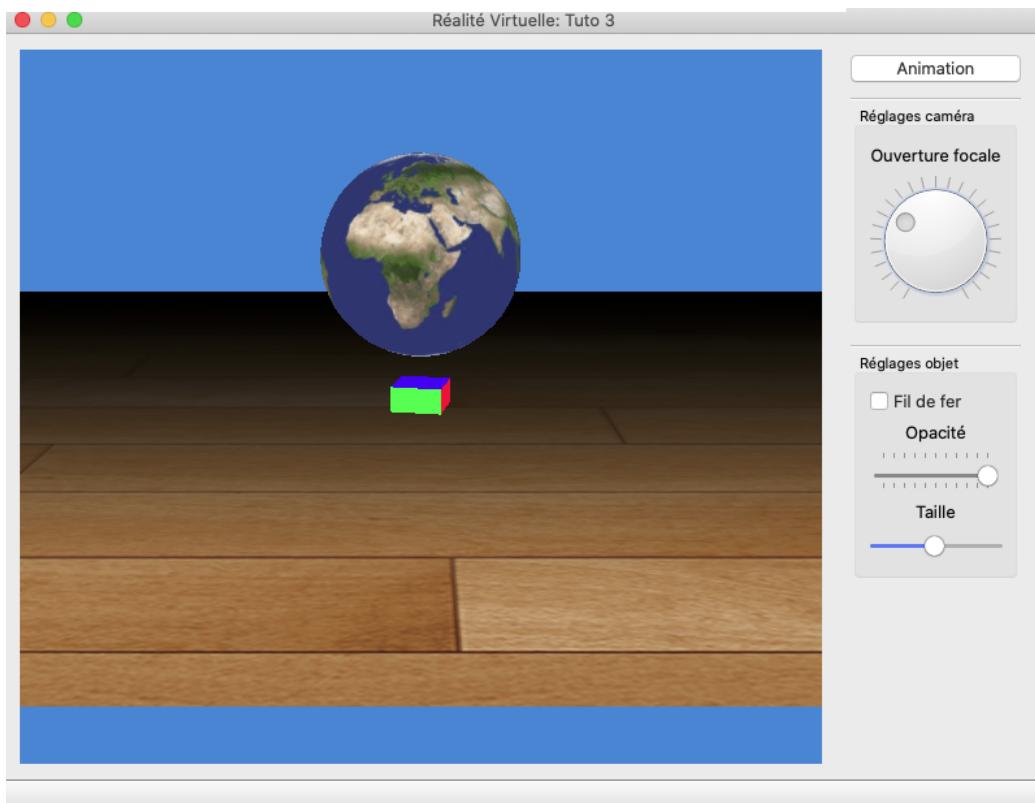


On voit que la texture est à l'envers : cela vient du fait que OpenGL et QImage ne gèrent pas les bitmaps de la même façon. La solution consiste à modifier la méthode `setTexture` de `RVBody` pour qu'elle ait la possibilité d'appliquer un renversement spéculaire au bitmap lu dans le fichier (car `QImage` contient une méthode `mirrored` qui fait ça).

```
// Dans rvbody.h
void setTexture(QString textureFilename, bool mirrored = true);

//Dans rvbody.cpp
void RVBody::setTexture(QString textureFilename, bool mirrored)
{
    if (m_texture)
        delete m_texture;
    m_texture = new QOpenGLTexture(QImage(textureFilename).mirrored(mirrored));
}
```

Donc par défaut `setTexture` appliquera une réflexion sur les textures en argument, sauf si on ajoute un deuxième argument `false`.



## Une deuxième texture

Souvent on peut avoir besoin de deux textures sur le même objet pour obtenir des résultats intéressants comme des transparences, des réflexions spéculaires, des effets d'éclairage.

Sur notre terre nous allons ajouter une texture qui donne l'aspect de la planète la nuit avec les éclairages des villes ; la texture `2k_earth_nightmap` vient du même site que la texture diurne : <https://www.solarsystemscope.com/textures/>. Contient aussi les textures du soleil et des planètes du système solaire (et quelques lunes).

Comment gère-t-on deux textures dans OpenGL ? D'abord il faut ajouter cette nouvelle texture aux ressources du projet (dans /textures). Puis il faut modifier la classe `RVSphere` pour qu'elle utilise une deuxième texture. Enfin il faut modifier le `fragment shader` pour qu'il utilise deux textures au lieu d'une pour donner la couleur aux fragments.

### Modification de RVSphere

- Ajouter une variable membre `m_texture2` de type `QOpenGLTexture *`;
- Ajouter une méthode publique `setTexture2` identique à `RVBody::setTexture` sauf qu'elle initialise la nouvelle variable membre
- On ne peut plus utiliser pour `RVSphere` la méthode `draw()` de sa classe parent `RVSurface` donc on est obligés de surcharger cette méthode (avec `override`) et de recopier (pour l'instant) le code de `RVsurface::draw()` dans `RVsphere::draw()`.
- Dans `draw()`:
  - il faut activer les deux textures en associant l'identifiant `GL_TEXTURE0` à la première et `GL_TEXTURE1` à la seconde :

```

if (m_texture) {
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE0);
    glActiveTexture(GL_TEXTURE0);
    m_texture->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
    m_texture->setMagnificationFilter(QOpenGLTexture::Linear);
    m_texture->bind(GL_TEXTURE0);
}

if (m_texture2) {
    glEnable(GL_TEXTURE1);
    glActiveTexture(GL_TEXTURE1);
    m_texture2->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
    m_texture2->setMagnificationFilter(QOpenGLTexture::Linear);
    m_texture2->bind(GL_TEXTURE1);
}

```

- On doit passer au *fragment shader* l'identifiant de la texture 1 comme nouvelle variable uniforme

```
m_program.setUniformValue("texture1", 1);
```

- Enfin, après le rendu, on remet tout en place (sinon on cours le risque que tous les objets suivants aient deux textures (par exemple le plan - essayez)) :

```

if (m_texture2) {
    m_texture2->release();
    glDisable(GL_TEXTURE1);
}

if (m_texture) {
    m_texture->release();
    glActiveTexture(GL_TEXTURE0);
    glDisable(GL_TEXTURE0);
    glDisable(GL_TEXTURE_2D);
}

```

- dans `RVWidget::initializeGL()` il faut spécifier le nom de la deuxième texture que va utiliser `m_world` avec la méthode `setTexture2(...)`. Remarquez que pour cela on est obligés d'utiliser la fonction `dynamic_cast`.

## Modification du shader

Avec un clic droit sur `FS_simple_texture.fsh` (qui, je le rappelle, est le fragment shader par défaut de toute classe qui hérite de `RVBody`) on peut choisir `Duplicate File...` pour créer un nouveau FS spécifique à la terre qui s'appellera `FS_earth_texture.fsh`. Dans le constructeur de `Rvsphere`, il faut modifier la variable `m_FSFfileName` pour qu'elle prenne en compte la nouvelle version du shader.

Quelles modifications ?

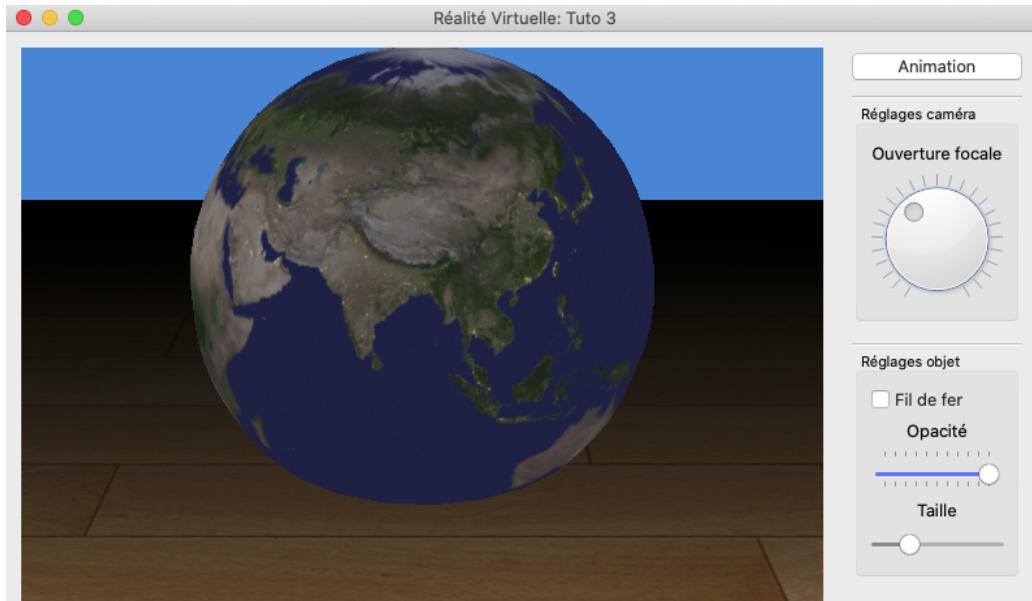
- on ajoute la déclaration d'une nouvelle variable uniforme `texture1`;
- dans le main, on stocke dans deux variables locales de type `vec4`, la couleur issues de la lecture de chaque texture (avec la même coordonnée texture `outTexCoord`);
- on utilise la fonction `mix` de GLSL pour mélanger ces deux couleurs en fonction du troisième paramètre (que l'on peut mettre à 0.5 pour l'instant) et ainsi définir la couleur finale du fragment `gl_FragColor`

```

varying highp vec4 outColor;
varying highp vec2 outTexCoord;
uniform sampler2D texture0;
uniform sampler2D texture1;

void main(void)
{
    float c = 0.5;
    vec4 col0 = texture2D(texture0, outTexCoord);
    vec4 col1 = texture2D(texture1, outTexCoord);
    gl_FragColor = mix(col0, col1, c) ;
}

```



On pourrait ajouter un slider dans l'IHM pour modifier la valeur du paramètre de mélange `c` mais on va plutôt opter par une variation en fonction de temps.

Donc on ajoute à la classe `RVSphere` une variable uniforme `m_time` de type `float` initialisée à 0 et incrémentée au début de la méthode `draw` de 0.05.

Cette variable est passée au shader en l'associant à une nouvelle variable uniforme appelée `u_daytime`.

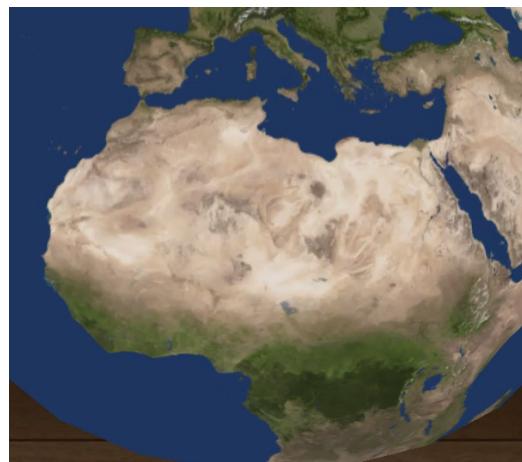
Dans le fragment shader, on applique à cette nouvelle variable uniforme une bonne fonction périodique pour obtenir régulièrement une valeur entre 0 et 1 qui sera utilisée pour définir la variable de mélange `c`.

```

varying highp vec4 outColor;
varying highp vec2 outTexCoord;
uniform sampler2D texture0;
uniform sampler2D texture1;
uniform float u_daytime;

void main(void)
{
    float c = pow(sin(u_daytime/10.0), 2.0);
    vec4 col0 = texture2D(texture0, outTexCoord);
    vec4 col1 = texture2D(texture1, outTexCoord);
    gl_FragColor = mix(col0, col1, c) ;
}

```



Il y a bien une alternance des deux textures de façon lisse mais le résultat n'est pas plus réaliste puisque cela correspond à une nuit qui tombe au même moment sur tous les points de la planète.

Pour avoir une frontière dégradée entre le jour et la nuit qui tourne autour de la terre, il faut utiliser la première coordonnée texture du fragment `s` qui correspond à la *longitude* en même temps que la fonction GLSL appelée `smoothstep`. Par exemple (sans animation) si le shader est :

```
void main(void)
{
    float x = outTexCoord.s;
    float c = smoothstep(0.4, 0.5, x);
    vec4 col0 = texture2D(texture0, outTexCoord);
    vec4 col1 = texture2D(texture1, outTexCoord);
    gl_FragColor = mix(col0, col1, c) ;
```



On a le lever de soleil sur Nice...  
rotation avec le temps vous trouverez ma solution dans le corrigé du tuto.

Pour que la frontière soit en



### 3.2. Texture procédurale

Pour montrer comment on peut définir une *texture procédurale*, c'est à dire une texture produite par un algorithme et non par un bitmap, on va utiliser comme support un *tore*, c'est à dire un pneu. La texture procédurale va simplement produire un damier bicolore noir/couleur globale de l'objet.

L'algorithme qui produit un damier  $n \times m$  est très simple : si  $i$  est l'indice de ligne (entre 0 et  $n-1$ ) et  $j$  est l'indice de colonne (entre 0 et  $m-1$ ) alors la **parité** de

$$i + j$$

dit si on est dans une case d'une couleur ou de l'autre.

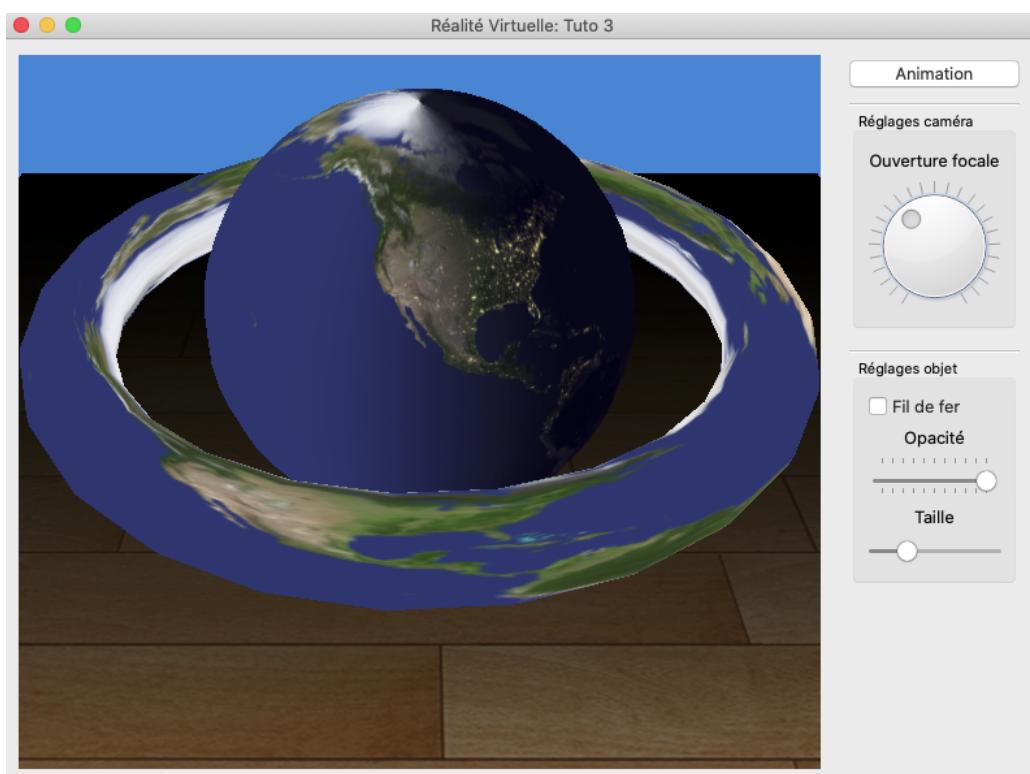
#### Mise en place

- ajouter les fichiers `rvtorus.cpp` et `rvtorus.h` au projet ;
- ajouter l'include de "rvtorus.h" dans `rvwidget.h` ;
- ajouter à `RVWidget` une variable membre `m_torus` de type pointeur sur `RVBody` ;
- initialiser `m_torus` dans `RVWidget::initializeGL()` comme d'habitude (sans définir de texture) ;
- appeler la méthode `draw()` de `m_torus` dans `RVWidget::paintGL()`.

Le résultat ci-dessous (où j'ai volontairement augmenté le grand rayon du tore pour qu'il englobe la planète) est surprenant car sans lui avoir défini aucune texture, le tore apparaît texturé avec l'image de la terre !

Cela vient du fait que le fragment shader utilisé par défaut par `RVtorus` est défini dans `RVBody` comme étant

`FS_simple_texture.fsh` qui dit que la couleur du fragment doit être lue dans `texture0`. Cette variable uniforme ne correspond à rien dans le cas du tore, mais comme on avait précédemment utilisé la texture de la terre pour la sphère, ces données sont restées en mémoire (de la carte graphique) et donc par chance elles apparaissent (notez que je ne suis pas sûr que ce comportement apparaisse sur toutes les architectures, mais en tout cas c'est le cas sous MacOS).



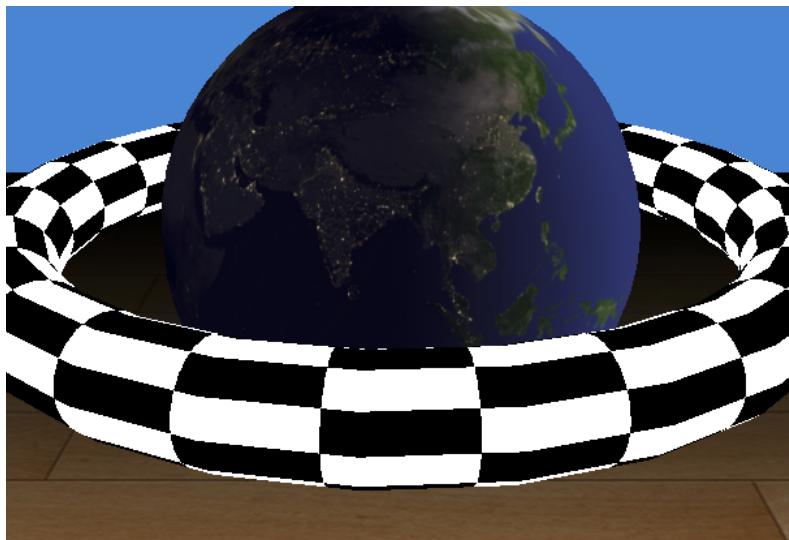
#### Ajout de la texture procédurale

- on crée une copie de `FS_simple_texture.fsh` en `FS_texture_damier.fsh` dans l'éditeur de ressources avec `Duplicate...` ;
- on change le nom du fragment shader à utiliser dans `RVWidget::initializeGL()` avec l'appel de `setFS()` (avant l'appel de `initialize()` évidemment) ;
- on modifie le code du shader pour avoir  $26 \times 10$  carreaux :

```
varying highp vec4 outColor;
varying highp vec2 outTexCoord;
uniform sampler2D texture0;

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);
    gl_FragColor = (col1*res + col2*(1.0 - res));
}
```



Améliorations possibles :

- changer la couleur globale du tore avec `setGlobalColor`
- ajouter des variables uniformes au shader pour choisir dynamiquement
  - la seconde couleur
  - le nombre de cases en s et en t
- mais pour cela il faut surcharger la méthode `draw()` de `RVSurface`.
- faire en sorte que la composante alpha de `gl_FragColor` soit lue à partir de celle de `outColor` pour que l'on puisse changer l'opacité du tore avec le slider comme pour les autres objets.