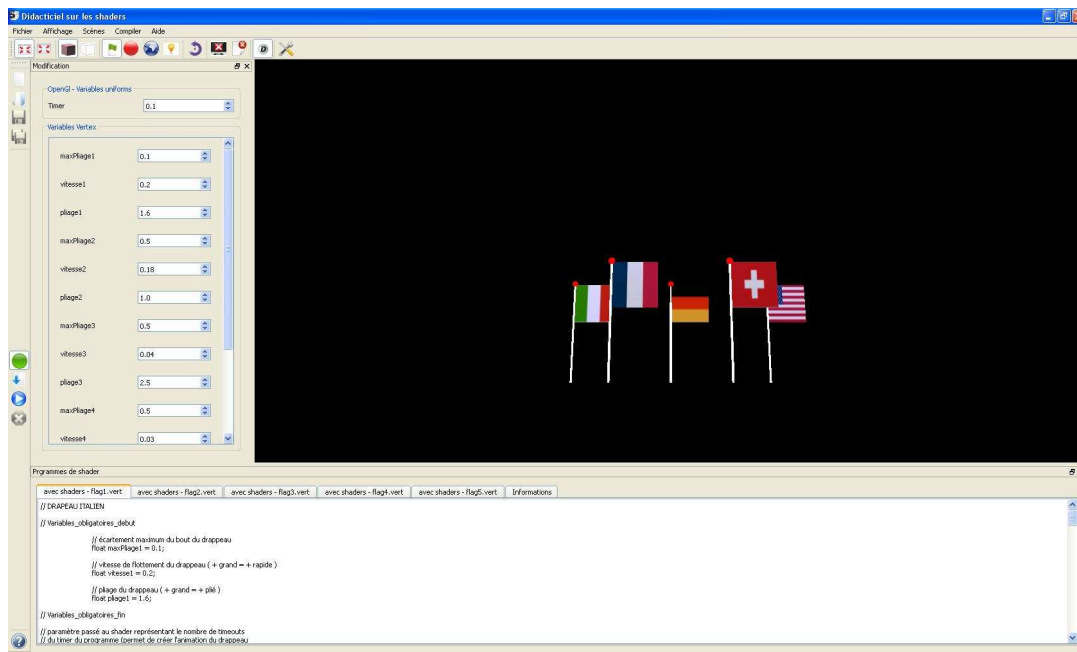


# Didacticiel pour les shaders en OpenGL

## Rapport



**Développeur :** Christophe Magri  
**Responsable :** Pierre Chatelain  
**Expert :** Matthieu Saner  
**N° de projet :** 11INF-TB203  
**Date de Remise :** 08.08.2011

## Tables des matières

1	Introduction.....	4
1.1	Résumé.....	4
1.2	Objectifs :.....	6
1.3	Technique :.....	6
2	Cahiers des charges.....	6
	Ce travail fait suite au projet 11INF-TP203. ....	6
3	Analyse .....	7
3.1	Liste des effets possibles .....	7
3.2	QGLWidget .....	9
3.2.1	Fonctions .....	9
3.3	Les Shaders .....	11
3.3.1	Les types de shaders.....	11
3.3.2	Les langages de shader .....	12
3.3.3	Le langage GLSL .....	12
3.3.4	Les qualifieurs.....	12
3.4	QGLShader.....	22
3.4.1	Fonctions .....	22
3.5	QGLShaderProgram.....	22
3.5.1	Fonctions .....	23
4	Développement.....	24
4.1	Fichier sht .....	24
4.2	Interface .....	25
4.3	Enregistrement.....	27
4.4	La structure.....	28
4.5	Effet de lumière.....	29
4.6	Effet sans shaders.....	29
4.7	Type d’affichage .....	31
4.8	Facettes .....	32
4.9	Scène lumière .....	33
4.10	Récupération de la position de la lumière .....	33
4.11	Le didacticiel.....	35
4.11.1	Parser le fichier.....	35
5	Problèmes rencontrés .....	36

5.1	Shader pour les ombres. ....	36
5.2	Scène « Drapeaux » avec shader.....	36
5.3	Récupération de la position de la lumière .....	37
5.4	Polymorphisme .....	37
5.5	Scène « Earth ».....	37
6	Améliorations possibles .....	38
6.1	Ajout d'autres scènes .....	38
6.2	Ajouter d'autres types de variables gérées par le didacticiel .....	38
6.3	La scène « Drapeaux ».....	38
7	Conclusion .....	39
8	Références.....	39
9	Liste des figures .....	40
10	Annexes .....	40

# 1 Introduction

## 1.1 Résumé

Ce projet a pour but de créer un didacticiel permettant de comprendre le fonctionnement, grâce à quelques exemples de façon didactique, des shaders en OpenGL. Le logiciel est composé des 4 scènes suivantes :

1. Drapeaux : Scène composée de cinq drapeaux associés à un programme de vertex de shader, permettant la simulation du flottement d'un drapeau.
2. Sphère : Scène composée d'une sphère rouge associée à un programme de vertex de shader, permettant la morphose (morphing) de celle-ci.
3. Earth : Scène composée de la Terre associée à un programme de vertex de shader et un programme de fragment de shader, permettant la simulation jour/nuit en fonction de la position de la lumière.
4. Lumière : Scène composée d'un cube, d'un cône et d'une sphère associés à trois programmes de shader différents :
  - Un programme de shader permettant d'avoir un éclairage avec trois lumières.
  - Un programme de shader permettant d'avoir un éclairage avec de la lumière diffuse.
  - Un programme de shader permettant d'avoir un éclairage avec de la lumière spéculaire.

Chaque programme est composé d'un vertex et d'un fragment de shader.

Le logiciel sera principalement utilisé dans le cours d'infographie basé sur OpenGL. Bien que les shaders ne soient pas étudiés durant ce cours (dû aux manques de périodes), il sera parfois nécessaire de les utiliser durant le projet de printemps.

Le didacticiel a été réalisé selon les trois parties suivantes :

1. Projet d'automne : Etude des shaders et réalisation d'exemples.
2. Projet de printemps : Debug et amélioration des scènes précédentes.
3. Projet de Bachelor : Restructuration complète du logiciel précédent. Le tout a été refait à l'exception de quelques classes. Amélioration des scènes et programmes de shader réalisés précédemment. Rendre le programme didactique, ajout de la scène lumière associée à trois programmes de shader différents.

Le logiciel explique les shaders de façon didactique et une scène contenant différents programmes de shaders a été ajoutée. Le programme contient, selon la scène, un ou plusieurs programme/s de vertex de shader ou un programme de vertex de shader ainsi qu'un programme de fragment de shader. Les deux types de shader ont donc bien été utilisés.

*The goal of this project was to develop a tutorial which would help understanding, with examples, how shaders with OpenGL work. The software contains the four following scenes :*

- 1. Drapeaux : This scene contains five flags which are associated to one vertex shader program which simulates the floating flag effect.*
- 2. Sphère : This scene contains a red sphere which is associated to one vertex shader program which simulates the morphing effect.*
- 3. Earth : This scene contains the earth which is associated to one vertex shader program and one fragment shader program which simulate the day/night effect according to the light position.*
- 4. Lumière : This scene contains a cube, a cone , a sphere which are associated to three different shader program :*
  - The first program gives a three lights lighting.*
  - The second program gives a diffuse lighting.*
  - The third program gives a specular lighting.*

Each program consists of a vertex and a fragment shader.

*The software will be mainly used in the course of computer graphics which is based on OpenGL. Although shaders are not studied in this course due to time's lack, it will sometimes be necessary to use them during the spring project. This tutorial has been realized according to the next three steps:*

- 1. Fall project : Study and realization of shader examples.*
- 2. Spring project : Debug and improvement of previous scenes.*
- 3. Bachelor project : Complete restructuring of the previous software. Everything has been rebuilt, except a few classes. Improvement of previous scenes and shader programs. Render the program didactic and add the scene "Lumière" associated to three different shader program.*

*The software explains the shaders with a didactic way and a scene containing different shader programs has been added. The program contains, according to the scene, one or many shader's vertex programs, or a shader's vertex program and a shader's fragment program. Both types of shader have therefore been used.*

## 1.2 Objectifs :

Mon objectif était de rendre le logiciel existant didactique, d'améliorer les scènes et programmes de shader existant et d'en ajouter d'autres.

La méthode que j'ai suivie est la suivante :

- 1) Restructuration complète du programme. La structure et toutes les classes ont été refaites, exceptées les classes concernant les fichiers.
- 2) Amélioration des scènes précédentes et des programmes de shader.
- 3) Ajout d'un widget permettant de modifier les shaders de façon didactique.
- 4) Ajout d'une scène «Lumière » comprenant trois programmes de shaders différents.

## 1.3 Technique :

Le didacticiel utilise OpenGL et est développé avec le framework Qt. Les programmes de shader ont été développés en GLSL (OpenGL Shading Language).

# 2 Cahiers des charges

### Situation initiale :

Ce travail fait suite au projet 11INF-TP203.

### Buts du projet :

- Il s'agit maintenant de réaliser un didacticiel qui permettra à un utilisateur de comprendre ce qu'est un programme de shaders et comment le réaliser.

### Démarche proposée :

- La démarche proposée est la suivante :
  - Analyser la façon la plus conviviale pour créer un didacticiel sur la base du programme existant (en particulier, quelles sont les boîtes de dialogue qui pourraient le mieux aider à comprendre le fonctionnement des shaders déjà réalisés).
  - Bien montrer en particulier l'existence et la différence entre un « vertex shader » et un « fragment shader » (on laissera de côté les « geometry shaders »).
  - Ajouter des shaders simples mais spectaculaires, toujours avec un dialogue convivial pour pouvoir agir au mieux sur leurs effets.
  - Ajouter un programme de shader pour les ombres (toujours avec boîtes de dialogue conviviales). Le projet de référence pour les ombres se trouve :  
P:\Formation\010-Bachelor\040-TravailBachelor\Archives\2010\INF10\INF-TB208-ShadowsVolumes

### Contraintes :

On utilisera Qt pour le développement et OpenGL pour le graphisme 3D.

Les directives de travail sont détaillées dans le document annexe.

### 3 Analyse

#### 3.1 Liste des effets possibles

Durant mon projet de semestre d'automne, il m'a été demandé d'étudier les différents effets possibles grâce aux shaders et d'en faire une liste :

Nom	Description	Source
<b>Occlusion ambiante</b>	Améliorer le réalisme d'un rendu 3D.	RenderMonkey
<b>Filtre anistrophe (Anisotropic filtering)</b>	Lisse les textures éloignées. Donc évite les effets escalier sur les textures éloignées.	RenderMonkey
<b>Rebonds (Bounce on Deforming Surface)</b>	Permet la simulation de rebonds sur une surface élastique.	RenderMonkey
<b>Complexité de profondeur (Depth complexity)</b>		RenderMonkey
<b>Profondeur de champ (Depth of field)</b>	Transition entre image flouée et image nette	RenderMonkey
<b>Eclairage Disco (Disco lighting)</b>	Simule une boule de lumière se trouvant dans les discothèques.	RenderMonkey
<b>Terre (Earth with clouds)</b>	Simulation de la terre avec ajout de nuages. Transition entre le jour et la nuit (image de la terre la nuit -> image de la terre le jour).	RenderMonkey
<b>Erosion</b>	Ajout d'érosion sur un objet 3D.	RenderMonkey
<b>Tissu (Fabric)</b>	Permet de mettre du tissu sur un objet 3D.	RenderMonkey
<b>Feu (fire)</b>	Simulation de feu et d'explosion.	RenderMonkey
<b>Fractale</b>	Dessin de fractales.	RenderMonkey
<b>Texture fourrure (Textured Fur effect)</b>	Ajouter une texture en fourrure sur un objet 3D.	RenderMonkey
<b>Verre (Complex Glass)</b>	Dessiner un objet en verre.	RenderMonkey
<b>Scintillement (glitter)</b>	Appliquer une texture scintillante à un objet 3D.	RenderMonkey
<b>HDR High Dynamic range</b>	C'est un filtre qui a pour but de tordre le pixel, de lui plaquer un effet calculé, comme poussiéreux, délavé ou pastel.	RenderMonkey
<b>Illumination (Specular Bump)</b>	Applique de la lumière spéculaire sur un objet.	RenderMonkey
<b>Illumination avancé (PerPixel effect)</b>	Illumine un objet en travaillant avec les pixels.	RenderMonkey
<b>Morphing</b>	Consiste à fabriquer une animation qui transforme de la façon la plus naturelle et la plus fluide possible un dessin initial vers un dessin final.	RenderMonkey

<b>Flou cinétique</b> <b>(Animate Motion Blur)</b>	Flou visible dans une animation, dû au mouvement rapide de l'objet affiché.	RenderMonkey
<b>Normal Map Filter</b>	Technique utilisée pour feindre l'éclairage de relief.	RenderMonkey
<b>Bump mapping</b>	Placage de relief. Sert à donner du relief aux objets.	
<b>NPR</b> <b>(Non-Photorealistic rendering)</b>	Sert à modifier les textures. Offre une grande variété de styles différents.	RenderMonkey
<b>Ooze Effect</b>	Permet de soumettre un objet à une mutation.	RenderMonkey
<b>Order Independent transparency</b>	Permet d'effectuer un rendu correct des objets semi-transparents se chevauchant, sans avoir à les trier dans l'ordre.	RenderMonkey
<b>système de particules</b> <b>(Particle system)</b>	Permet de simuler de nombreux phénomènes naturels tels que feu, explosion, fumée, eau, nuage, poussière, neige, feux d'artifices, et animés à l'aide de propriétés telles que la gravité, du vent, l'inertie, etc...	RenderMonkey
<b>Plastic</b>	Permet de changer la texture d'un objet en plastique.	RenderMonkey
<b>Reflections Refractions</b>	Permet d'ajouter des effets de réflexions et de réfractions.	RenderMonkey
<b>Moteur de rendu</b> <b>(RenderMan – based)</b>	Spécification de communication entre un modèleur 3D et un moteur de 3D, ce qui permet d'utiliser librement des outils compatibles, très nombreux et surtout libres. RenderMan est un moteur de rendu créé par Pixar Animations Studios.	RenderMonkey
<b>Render to texture</b>	Permet de rendre une scène en texture puis d'appliquer cette texture sur une surface.	RenderMonkey
<b>Screen Space</b>	Occlusion ambiante en temps réel.	RenderMonkey
<b>Show effect</b>		RenderMonkey
<b>Speaker</b>	Simule le comportement d'une membrane d'haut-parleur.	RenderMonkey
<b>Pierre</b> <b>(Stones)</b>	Permet d'appliquer à un objet une texture en pierre.	RenderMonkey
<b>Bois</b> <b>(Wood)</b>	Permet d'appliquer à un objet une texture en bois.	RenderMonkey

RenderMonkey est un programme de AMD permettant de visualiser certains effets de shader et de les modifier. Il est possible de le télécharger à l'adresse suivante :

<http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx#download>



## 3.2 QGLWidget

Le framework Qt permet l'utilisation de la librairie OpenGL. La classe QGLWidget est une widget permettant l'affichage de scènes 3D en OpenGL. Elle fournit des fonctions pour l'affichage de scènes 3D en OpenGL intégrées dans une application utilisant le framework Qt.

### 3.2.1 Fonctions

QGLWidget fournit trois fonctions virtuelles pratiques pouvant être implémentées dans la sous-classe afin d'afficher des scènes 3D en OpenGL. Ces fonctions sont les suivantes :

#### 3.2.1.1 paintGL()

Cette fonction affiche la scène OpenGL. Elle est appelée chaque fois que la scène doit être redessinée. C'est donc dans cette fonction que les objets 3D seront dessinés.

Pour que cette fonction soit appelée en boucle, il est nécessaire d'ajouter un timer afin de rafraîchir la scène :

```
widgetopengl.cpp

23  // timer de rafraichissement de la scène
24  refreshTimer= new QTimer(this);
25  connect(refreshTimer,SIGNAL(timeout()),this, SLOT(refresh()));
26  refreshTimer->setInterval(25);
```

Ne pas oublier non plus de créer un SLOT pour appeler la méthode updateGL() :

```
widgetopengl.cpp

636  void WidgetOpenGL::refreshDisplay()
637  {
638      // SLOT de rafraichissement de la scène
        ....

651      if(selectedScene == -1)
652          refreshTimer->stop();
653
654      // Rafraichissement
655      updateGL();
656  }
```

### 3.2.1.2 `resizeGL(int width, int height)`.

Cette fonction est utilisée pour tout changement de taille du widget. C'est dans cette méthode que la perspective de scène OpenGL est créée et que les paramètres du viewport sont réglés.

widgetopengl.cpp

```
433 void WidgetOpenGL::resizeGL(int _width, int _height)
434 {
435     // Appelée lors de chaque changement de taille du widget;
436     glViewport(0,0, _width, _height); // Affichage sur tout l'écran.
437     ....
438     gluPerspective(45, (double)_width /_height, 0.1, 100.0);
439     ....
440 }
```

### 3.2.1.3 `initializeGL()`

La méthode `initializeGL()` permet d'initialiser et d'activer certaines fonctionnalités d'OpenGL, telles que :

- l'éclairage,
- les matériaux,
- les textures,
- les callistes,
- etc.

widgetopengl.cpp

```
366 void WidgetOpenGL::initializeGL()
367 {
368     ....
369     //Initialisation éclairage
370     glEnable(GL_LIGHTING);
371     glEnable(GL_LIGHT0);
372     ....
373     //Initialisation des matériaux
374     GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f};
375     GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8f, 1.0f};
376     GLfloat specularLight[] = { 1.0f, 1.0f, 1.0f, 1.0f};
377     ....
378     glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
379     glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
380     glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
381     ....
382     for(int i = 0 ; i < listScene->length() ; i ++ )
383     {
384         listScene->at(i)->initCallList();
385         listScene->at(i)->loadTexture(this);
386         listScene->at(i)->generateCallList();
387     }
388 }
```

Elle n'est appelée qu'une seule fois, avant la première fois que `resizeGL()` ou `paintGL` soit appelés, puis à chaque fois que le widget est assigné à un nouveau `QGLContext`, ce qui n'arrive que très rarement.

### 3.3 Les Shaders

Les shaders sont des petits programmes utilisés en 3D pour paramétrer une partie du processus de rendu réalisé par le processeur de la carte graphique (GPU). Ils permettent un traitement vectorisé soulageant ainsi l'unité centrale (CPU) afin d'obtenir un rendu plus rapide. Les shaders permettent, entre autre, de décrire :

- l'absorption de la lumière,
- la diffusion de la lumière,
- la texture à utiliser,
- les réflexions,
- les réfractions,
- l'ombrage,
- le déplacement de primitives et
- des effets post-traitements.

Ces effets sont créés au niveau des sommets (vertex) des objets 3D et/ou au niveau des pixels de l'écran.

Afin de pouvoir utiliser des shaders, l'ordinateur doit disposer d'une carte graphique permettant leur exécution. Il est nécessaire de disposer d'une carte permettant l'utilisation d'OpenGL 2.0 pour pouvoir supporter les shaders version 1.0.

#### 3.3.1 Les types de shaders

Il existe trois types de shaders, listés ci-dessous :

##### 3.3.1.1 Les vertex shaders

Les vertex shaders, aussi appelés shaders de sommet, permettent d'agir sur chaque sommet d'un objet 3D. Cela permet par exemple de changer la position, la couleur, voire même la texture de cet objet. Ils peuvent être associés à un fragment shader.

##### 3.3.1.2 Les fragments shaders

Les fragments shaders ou pixels shaders permettent d'agir sur chaque pixel de l'écran. Leur nombre dépend donc de la résolution d'affichage de l'écran. Ils permettent par exemple de changer la couleur de ces pixels. Ils peuvent être associés à un vertex shader.

##### 3.3.1.3 Les geometry shaders

Les geometry shaders permettent d'ajouter ou de supprimer des sommets à un objet 3D.

Ce type de shader n'est à l'heure actuelle malheureusement pas supporté par Qt et n'a donc pas pu être testé.

### 3.3.2 Les langages de shader

Les quatre principaux langages de shader en temps réel sont listés ci-dessous.

#### 3.3.2.1 GLSL

Le langage de shader GLSL (OpenGL Shading Language) s'utilise avec OpenGL et tout comme lui, il est multiplateforme. C'est le seul supporté par le framework Qt. C'est donc le langage que j'ai utilisé.

#### 3.3.2.2 GLSL/ES

Le langage de shader GLSL/ES (OpenGL Shading Language for Embedded System) s'utilise avec OpenGL ES (OpenGL for Embedded System). Il est basé sur GLSL et adapté aux plateformes mobiles ou embarquées telles que les téléphones mobiles, les assistants personnels (PDA), les consoles de jeux vidéo portables et encore les lecteurs multimédia de poche ou de salon.

#### 3.3.2.3 HLSL

Le langage de shader HLSL (DirectX High-Level Shader Language) de Microsoft s'utilise avec Direct3D et ne fonctionne malheureusement que sous Windows et Xbox. C'est probablement le langage le plus populaire. Ce fût le premier à fournir une interface de programmation haut niveau pour les shaders avec une syntaxe proche du C.

#### 3.3.2.4 Cg

Le langage Cg (programming language) de NVIDIA est très proche du langage HLSL car ils ont été développés en parallèle, il possède également une syntaxe proche du C. Rarement employé (sauf sur Playstation 3 de Sony), il n'est pourtant pas inintéressant, pouvant utiliser indifféremment OpenGL ou Direct3D.

Il favorise également son utilisation, étant fourni avec un large choix d'outils libres permettant à un débutant de pouvoir créer ses propres shaders. La fonctionnalité la plus intéressante que Cg propose, réside dans l'utilisation de connecteurs de types de données spéciales, pour lier les différentes étapes du processus.

### 3.3.3 Le langage GLSL

OpenGL shading language étant le langage utilisé pour la réalisation de mon projet, je l'ai analysé en détail.

Tout comme les deux langages expliqués ci-dessus, le langage GLSL ressemble beaucoup au C. Simple à comprendre, il permet notamment d'effectuer des boucles et des branchements conditionnels.

#### 3.3.4 Les qualifieurs

Il existe deux types de communication entre programmes. Le premier est la communication entre le programme en OpenGL et le programme de shader, le second est la communication entre programmes de shader.

En GLSL, ces types sont représentés par les qualifieurs suivants :

- attribute,
- uniform,
- varying,
- const,
- centroid varying,
- invariant.

Voir l'explication des trois principaux qualifieurs ci-dessous. Les exemples ci-après sont simplement exposés à titre indicatif et ne sont pas forcément utilisés dans le code.

#### 3.3.4.1 attribute

Le qualifieur attribute est une variable globale dont la valeur peut changer d'un sommet à l'autre (vertex, normal, couleur, etc...). Les valeurs sont passées depuis l'application OpenGL. Il est uniquement utilisable dans les vertex et geometry shaders et est en lecture seule.

Exemples de déclaration dans le programme de shader :

```
attribute vec4 position;  
attribute vec3 normal;  
attribute vec2 texCoord;
```

Exemple de passage de paramètre depuis le programme OpenGL :

```
prog->setAttributeValue("position", 1.0f,1.0f,1.0f,1.0f);  
prog->setAttributeValue("normal", 1.0f,0.0f,0.0f);  
prog->setAttributeValue("texCoord ", 1.0f,-1.0f);
```

#### 3.3.4.2 uniform

Ce qualifieur est utilisé pour déclarer une variable globale dont la valeur est semblable dans l'ensemble de la primitive en cours d'exécution. Elle permet de passer une valeur à un programme de shader (vertex et fragment) depuis le programme en OpenGL. Cette variable est en lecture seule.

Toutes les variables uniform sont en lecture seule et sont initialisées à l'extérieur, soit au moment de liaison ou par l'intermédiaire de l'API.

Exemples de déclaration dans le programme de shader :

```
uniform vec3 lightPosition ;  
// valeur assignée au moment de liaison  
uniform vec3 color = vec3(0.7,0.7,0.2) ;
```

Exemple de passage de paramètre depuis le programme OpenGL :

```
prog->setUniformValue("lightPosition", 1.0f,1.0f,1.0f) ;
```

### 3.3.4.3 **varying**

Ce qualifieur est utilisé pour passer une valeur entre un programme de vertex de shader, un programme de fragment de shader et un programme de geometry de shader. La déclaration d'un qualifieur varying doit être faite dans tous les programmes de shader communiquant ensemble.

Exemple de déclaration :

```
varying vec3 normal ;
```

### 3.3.4.4 **Les opérateurs**

En OpenGL Shading Language, les opérateurs sont semblables à ceux utilisés en C. Ils sont listés et décrits ci-dessous :

Priorité	Type	Opérateur	Associativity
1 (Haut)	groupement entre parenthèses	()	
2	unaire	defined	Droite à Gauche
		+ - ~ !	
3	multiplicatif	* / %	Gauche à Droite
4	additionnelle	+ -	Gauche à Droite
5	décalage bit à bit	<< >>	Gauche à Droite
6	relationnel	< > <= >=	Gauche à Droite
7	égalitaire	== !=	Gauche à Droite
8	ET bit à bit	&	Gauche à Droite
9	OU Exclusif bit à bit	^	Gauche à Droite
10	OU bit à bit		Gauche à Droite
11	ET logique	&&	Gauche à Droite
12 (Bas)	OU logique		Gauche à Droite

### 3.3.4.5 **Les variables**

Le langage GLSL contient de nombreuses variables intégrées faisant référence à des états OpenGL. Les variables commencent toutes par le préfixe gl\_.

#### 3.3.4.5.1 **Les variables intégrées**

**Vertex Shader et Geometric Shader :**

Les variables intégrées du vertex de shader et du geometric shader sont identiques :

- gl\_position
- gl\_PointSize
- gl\_ClipVertex
- gl\_FrontColor - (écriture)
- gl\_BackColor - (écriture)
- gl\_FrontSecondaryColor - (écriture)
- gl\_BackSecondaryColor - (écriture)
- gl\_TexCoord[n] - (écriture)
- gl\_FogFragCoord - (écriture)

### Fragment Shader :

Les variables intégrées du fragment de shader sont :

- `gl_FragColor` - (écriture)
- `gl_FragData` - (écriture)
- `gl_FragDepth` - (écriture)
- `gl_FragCoord` - (lecture seule)
- `gl_FrontFacing` - (lecture seule)
- `gl_FrontColor` - (lecture seule)
- `gl_BackColor` - (lecture seule)
- `gl_FrontSecondaryColor` - (lecture seule)
- `gl_BackSecondaryColor` - (lecture seule)
- `gl_TexCoord[n]` - (lecture seule)
- `gl_FogFragCoord` - (lecture seule)

#### 3.3.4.5.2 Les attributs intégrés

Les attributs intégrés par vertex (sommet) correspondent aux fonctions du programme OpenGL que l'on peut spécifier par sommet :

Attributs GLSL	Fonctions OpenGL	Définition
<code>gl_Color</code>	<code>glColor*()</code>	Définit la couleur des sommets
<code>gl_FogCoord</code>	<code>glFogCoord*()</code>	Définit les coordonnées du brouillard
<code>gl_MultiTexCoord[n]</code>	<code>glMultiTexCoord*()</code>	Définit les coordonnées de texture pour une unité de texture particulière
<code>gl_MultiTexCoord[n]</code>	<code>glTexCoord*()</code>	Définit les coordonnées de texture pour l'unité de texture courante
<code>gl_Normal</code>	<code>glNormal*()</code>	Définit la normale du sommet
<code>gl_Vertex</code>	<code>glVertex*()</code>	Définit les coordonnées du sommet

En plus des attributs expliqués ci-dessus, il est possible de spécifier ces propres attributs (scalaire, vecteur ou matrice) par sommet (masse, vitesse, etc.).

Grâce à la fonction `glGet*()` et au token `GL_MAX_VERTEX_ATTRIBS`, le nombre maximal d'attributs peut être spécifié.

Le passage d'attributs depuis le programme en OpenGL, se fait à l'aide des fonctions suivantes :

- `glGetAttribLocation()`
- `glBindAttribLocation()`
- `glVertexAttrib*()`.

### 3.3.4.5.3 Les uniformes intégrées

Les variables uniformes intégrées correspondent à des variables d'état OpenGL modifiées peu fréquemment (exemple : éclairage). Elles sont nombreuses et ne seront donc pas toutes détaillées ci-dessous. Il est possible de se référer à l'annexe GLSL\_Full.1.20.8.pdf, à partir de la page 50.

Uniform	Correspondance
gl_ModelViewMatrix	Matrice de modélisation visualisation ModelView
gl_ModelViewProjectionMatrix	Matrice de ModelViewProjection
gl_NormalMatrix	Matrice Normale (passage de la normale dans le repère caméra)
gl_ProjectionMatrix	Matrice de projection
gl_TextureMatrix[n]	Matrice de texture (max = gl_MaxTextureCoords)

En plus des variables uniformes détaillées ci-dessus, il est possible de spécifier des variables uniformes grâce aux fonctions :

- glGetUniformLocation()
- glUniform\*().

### 3.3.4.5.4 Les varying intégrées

#### Vertex Shader et Geometric Shader :

Les variables varying intégrées du vertex de shader et du geometric shader sont identiques :

- gl\_FrontColor - (*écriture*)
- gl\_BackColor - (*écriture*)
- gl\_FrontSecondaryColor - (*écriture*)
- gl\_BackSecondaryColor - (*écriture*)
- gl\_TexCoord[gl\_MaxTextureCoords] - (*écriture*)
- gl\_FogFragCoord - (*écriture*)



## Fragment Shader :

Les variables varying intégrées du fragment de shader sont :

- `gl_FrontColor` - (lecture seule)
- `gl_BackColor` - (lecture seule)
- `gl_FrontSecondaryColor` - (lecture seule)
- `gl_BackSecondaryColor` - (lecture seule)
- `gl_TexCoord[gl_MaxTextureCoords]` - (lecture seule)
- `gl_FogFragCoord` - (lecture seule)

Exemples :

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[];
varying float gl_FogFragCoord;
```

### 3.3.4.6 Les fonctions

OpenGL Shading Language dispose de tellement de méthodes qu'il prendrait beaucoup de temps de toutes les expliquer en détails. Elles sont détaillées à partir de la page 13 du fichier GLSL\_Full.1.20.8.pdf annexé.

L'identifiant `genType` représente les types génériques, soit les float, double, int, etc...

#### 3.3.4.6.1 Fonctions trigonométriques

L'angle est toujours spécifié en radians.

<code>genType radians(genType degrees)</code>	Converti les degrés en radians
<code>genType degrees(genType radians)</code>	Converti les radians en degrés
<code>genType sin(genType angle)</code>	Fonctions sinus trigonométrique standard
<code>genType cos(genType angle)</code>	Fonctions cosinus trigonométrique standard
<code>genType tan(genType angle)</code>	Fonctions tangente trigonométrique standard
<code>genType asin(genType x)</code>	Arcsin : Retourne l'angle dont le sinus est x.
<code>genType acos(genType x)</code>	Arccos : Retourne l'angle dont la tangente est x.
<code>genType atan(genType y, genType x)</code>	Arctan : Retourne l'angle dont la tangente est y/x.

### 3.3.4.6.2 Fonctions exponentielles

genType <b>pow</b> (genType x, genType y)	Retourne $x^y$
genType <b>exp</b> (genType x)	Retourne $e^x$
genType <b>log</b> (genType x)	Retourne y tel que $x = e^y$
genType <b>exp2</b> (genType x)	Retourne $2^x$
genType <b>log2</b> (genType x)	Retourne y tel que $x = 2^y$
genType <b>sqrt</b> (genType x)	Retourne $\sqrt{x}$
genType <b>inversesqrt</b> (genType x)	Retourne $1/\sqrt{x}$

### 3.3.4.6.3 Fonctions communes

genType <b>abs</b> (genType x)	Retourne x si $x \geq 0$ , sinon retourne -x
genType <b>sign</b> (genType x)	Retourne 1.0 si $x > 0$ , 0.0 si $x=0$ , sinon -1.0 si $x < 0$
genType <b>floor</b> (genType x)	Retourne l'entier $\leq x$ le plus proche de x
genType <b>ceil</b> (genType x)	Retourne l'entier $\geq x$ le plus proche de x
genType <b>fract</b> (genType x)	Retourne $x - \text{floor}(x)$
genType <b>mod</b> (genType x, genType y)	Retourne $x - y * \text{floor}(x/y)$
genType <b>min</b> (genType x, genType y)	Retourne le plus petit entre x et y
genType <b>max</b> (genType x, genType y)	Retourne le plus grand entre x et y

### 3.3.4.6.4 Fonctions géométriques

float <b>length</b> (genType x)	Retourne la longueur d'un vecteur
float <b>distance</b> (genType p0, genType p1)	Retourne la distance entre les points P0 et P1.
float <b>dot</b> (genType x, genType y)	Retourne le produit scalaire de x et y
vec3 <b>cross</b> (vec3 x, vec3 y)	Retourne le produit vectoriel de x et y
genType <b>normalize</b> (genType x)	Retourne un vecteur unitaire

### 3.3.4.6.5 Fonctions matricielles

mat <b>matrixCompMult</b> (mat x, mat y)	Multiplie chaque composante de la matrice m1 avec chaque composante de la matrice m2  mat=m1[0][0]*m2[0][0] mat=m1[0][1]*m2[0][1] mat=m1[1][0]*m2[1][0] ...
mat2 <b>transpose</b> (mat2 m) mat2x3 <b>transpose</b> (mat3x2 m) ...	Transpose une matrice (inverse ces lignes et ces colonnes)

### 3.3.4.6.6 Fonctions de texture

<code>vec4 texture2D (sampler2D sampler, vec2 coord)</code>	Retourne la couleur du pixel de coordonnée coord de la texture 2D sampler.
---	--

### 3.3.4.7 Les types

Le langage OpenGL Shading Language supporte tous les types suivants :

<b>void</b>	Fonction ne retournant aucune valeur
<b>bool</b>	Booléen – Type conditionnel (true ou false)
<b>int</b>	Entier
<b>float</b>	Flottant
<b>vec2</b>	Vecteur de 2 composantes – flottant
<b>vec3</b>	Vecteur de 3 composantes – flottant
<b>vec4</b>	Vecteur de 4 composantes – flottant
<b>bvec2</b>	Vecteur de 2 composantes – booléen
<b>bvec3</b>	Vecteur de 3 composantes – booléen
<b>bvec4</b>	Vecteur de 4 composantes - booléen
<b>ivec2</b>	Vecteur de 2 composantes – entier
<b>ivec3</b>	Vecteur de 3 composantes – entier
<b>ivec4</b>	Vecteur de 4 composantes – entier
<b>mat2</b>	Matrice de 2*2 composantes – flottant
<b>mat3</b>	Matrice de 3*3 composantes – flottant
<b>mat4</b>	Matrice de 4*4 composantes – flottant
<b>mat2x2</b>	Matrice à 2 colonnes et à 2 lignes - flottant (Equivalent à mat2)
<b>mat2x3</b>	Matrice à 2 colonnes et à 3 lignes – flottant
<b>mat2x4</b>	Matrice à 2 colonnes et à 4 lignes – flottant
<b>mat3x2</b>	Matrice à 3 colonnes et à 2 lignes - flottant
<b>mat3x3</b>	Matrice à 3 colonnes et à 3 lignes - flottant (Equivalent à mat3)
<b>mat3x4</b>	Matrice à 3 colonnes et à 4 lignes – flottant
<b>mat4x2</b>	Matrice à 4 colonnes et à 2 lignes - flottant
<b>mat4x3</b>	Matrice à 4 colonnes et à 3 lignes - flottant
<b>mat4x4</b>	Matrice à 4 colonnes et à 4 lignes - flottant (Equivalent à mat4)
<b>sampler1D</b>	Texture 1D
<b>sampler2D</b>	Texture 2D
<b>sampler3D</b>	Texture 3D
<b>samplerCube</b>	Texture de cube
<b>sampler1DShadow</b>	Texture 1D de profondeur avec comparaison
<b>sampler2DShadow</b>	Texture 2D de profondeur avec comparaison

Ces types sont détaillés à partir de la page 13 du fichier GLSL\_Full.1.20.8.pdf en annexe.

Les vecteurs `vec1`, `vec2` et `vec3` ainsi que les matrices `mat1`, `mat2` et `mat3` sont en fait des ensembles de variables fonctionnant de la manière suivante :

```
vec3 monVecteur;  
monVecteur.x = 3.0 ;  
monVecteur.y = 2.0 ;  
monVecteur.z = 1.0 ;  
// Les vecteurs peuvent également fonctionner comme des tableaux  
monVecteur[0] = 3.0 ;  
monVecteur[1] = 2.0 ;  
monVecteur[2] = 1.0 ;  
// Les matrices comme des tableaux à deux dimensions  
mat3 maMatrice;  
maMatrice [0][0] = 2.0 ;  
maMatrice [2][1] = 1.0 ;
```

### 3.3.4.8 Le casting

En OpenGL Shading Language le casting se fait de la manière suivante :

```
int maValeur = int(maVariableFloat);
```

Le cast d'une variable float en int et vice versa se fait automatiquement, ce qui n'est pas le cas lorsqu'on fait la même chose entre un `vec3` et un `vec2`.

```
vec3 monVecteur1 = ... ;  
vec2 monVecteur2 = vec2(monVecteur1);
```

### 3.3.4.9 Variables complexes

OpenGL Shading Language reprend le principe du casting pour initialiser des variables complexes. Les deux exemples plus bas, illustrent bien cette situation (initialisation d'un vecteur et initialisation d'une matrice):

### 3.3.4.10 La surcharge des opérateurs

Pour les types de données complexes (matrice et vecteur), OpenGL Shading Language utilise la surcharge d'opérateurs. De cette manière, il est très simple de, par exemple, multiplier un vecteur avec une matrice.

```
vec4 monVecteur1 = ...;  
mat3 maMatrice = ... ;  
vec4 monVecteur2 = maMatrice * monVecteur1;
```

Les opérations ne sont pas toutes possibles. Comme pour les mathématiques, la matrice et les vecteurs doivent avoir le même nombre de composante. On ne peut par exemple pas multiplier une matrice 4x4 par un vecteur de 3 composantes.

### 3.3.4.11 Initialisation d'un vecteur

L'initialisation d'une matrice s'effectue comme suit :

```
// Chargement avec toutes les composantes égales à 2.0 :[2.0,2.0,2.0]
vec3 monVecteur1 = vec3(2.0);
// Chargement des composantes égales à [1.0,2.0,3.0]
vec3 monVecteur2 = vec3(1.0,2.0,3.0);
// Chargement avec des vecteurs plus petits
vec2 monVecteur3 = vec2(1.0,2.0);
vec3 monVecteur4 = vec3(monVecteur3,3.0);
// monVecteur4 égale à [1.0,2.0,3.0]
```

### 3.3.4.12 Initialisation d'une matrice

L'initialisation d'un vecteur se fait comme suit :

```
vec4 monVecteur1,monVecteur2, monVecteur3;
monVecteur1 = vec4(1.0, 0.0, 0.0, 0.0);
monVecteur2 = vec4(0.0, 1.0, 0.0, 0.0);
monVecteur3 = vec4(0.0, 0.0, 1.0, 0.0);
// Chargement de la matrice unitaire
mat3 maMatrice1 = mat3(monVecteur1, monVecteur2, monVecteur3);
// La première ligne de maMatrice est égale a monVecteur1
// La deuxième ligne de maMatrice est égale a monVecteur2
// La troisième ligne de maMatrice est égale a monVecteur3
mat3 maMatrice2 = mat3(1.0);
// Charge la matrice unitaire. Mets des 1.0 sur la diagonale maMatrice2
```

### 3.3.4.13 Opération entre vecteur/scalaire et matrice/scalaire :

Ces opérations se font comme suit :

```
// Multiplication de chaque composante par 3
vec3 monVecteur1 = ...;
vec3 monVecteur2 = monVecteur1 * 3.0;

mat3 maMatrice1 = ... ;
mat3 maMatrice2 = maMatrice1 * 3.0;

// Addition de chaque composante de 3
vec3 monVecteur1 = ...;
vec3 monVecteur2 = monVecteur1 + 3.0;

mat3 maMatrice1 = ... ;
mat3 maMatrice2 = maMatrice1 + 3.0;
```

A noter, que les multiplications de composante par composante s'effectuent grâce à la méthode `matrixCompMult()`.

### 3.3.4.14 Opération de multiplication entre matrice/vecteur et matrice/matrice:

Ces opérations s'effectuent comme suit :

```
mat3 maMatrice1 = ... ;  
mat3 maMatrice2 = ... ;  
vec3 monVecteur1 = ... ;  
/*      Multiplication matricielle (le vecteur multiplie chaque  
ligne de la matrice) */  
vec3 monVecteur2 = monVecteur1 * maMatrice1 ;  
/*      Multiplication matricielle (Les colonnes de la matrice 2  
multiplient chaque ligne de la matrice 1) */  
mat3 maMatrice3 = maMatrice1 * maMatrice2;
```

## 3.4 QGLShader

La classe QGLShader permet uniquement aux shaders du programme OpenGL d'être compilés. Elle supporte les shaders écrits dans le langage OpenGL Shading Language (GLSL) et dans le langage OpenGL/ES Shading Language (GLSL/ES).

Elle permet de retourner le détail de la compilation des vertex et fragments de shader.

### 3.4.1 Fonctions

Les fonctions de la classe QGLShader étudiées et utilisées dans le projet sont les suivantes :

#### 3.4.1.1 bool compileSourceCode (const QString & source)

La fonction compileSourceCode() est surchargée, elle existe avec d'autre paramètre d'entrée tels que char et QByteArray. Elle permet de définir et de compiler la source du shader.

C'est une fonction de type booléen et retourne donc vrai lorsque le shader a été compilé avec succès et faux lorsque ce n'est pas le cas.

#### 3.4.1.2 bool isCompiled () const

Cette fonction est de type booléen. Elle retourne vrai lorsque le shader a été compilé avec succès et faux lorsque ce n'est pas le cas.

#### 3.4.1.3 QString log () const

La fonction log() est de type QString. Elle retourne les messages d'erreur et les messages d'avertissement de la dernière compilation.

## 3.5 QGLShaderProgram

La classe QGLShaderProgram permet au programme de shader OpenGL d'être linké et utilisé.

Cette classe supporte les shaders écrits dans le langage OpenGL Shading Language (GLSL) et dans le langage OpenGL/ES Shading Language (GLSL/ES).

Elle permet de retourner le détail de la compilation et du linkage des vertex et fragments de shader.

On utilisera donc la classe `QGLShader` plutôt pour compiler les programmes et la classe `QGLShaderProgram` pour les linker et les utiliser. Cependant, la classe `QGLShaderProgram` permet également de compiler les programmes. Par conséquent, la classe `QGLShader` n'aurait pas vraiment besoin d'être utilisée. Mais les exemples qt sont codés avec les deux classes et c'est pour cette raison que ces deux classes ont été utilisées.

### 3.5.1 Fonctions

Les fonctions de la classe `QGLShaderProgram` étudiées et utilisées dans le projet sont les suivantes :

#### 3.5.1.1 `bool addShaderFromSourceCode (QGLShader::ShaderType type, const QString &source)`

Cette fonction est une fonction surchargée. Elle compile la source du shader spécifié et l'ajoute à l'instance de `QGLShaderProgram`.

C'est une fonction booléenne qui retourne vrai lorsque la compilation est effectuée avec succès et faux lorsque ce n'est pas le cas. Les messages d'erreur et d'avertissement peuvent être récupérés via la fonction `log()` de cette même classe.

Elle est plutôt utilisée pour ajouter la source que l'on va linker et utiliser, que pour la compilation. C'est pour cette raison que cette fonction compile le programme ajouté, pour ne pas utiliser un code avec erreur.

#### 3.5.1.2 `bool link()`

Cette fonction permet de linker tous les shaders ajoutés à l'instance de `QGLShaderProgram` grâce à la fonction `addShaderFromSourceCode()`.

C'est une fonction booléenne qui retourne vrai lorsque le linkage est effectué avec succès et faux lorsque ce n'est pas le cas. Dans le cas d'une erreur, les message d'erreur et d'avertissement peuvent être récupérés via la fonction `log()` de cette même classe.

Si le programme de shader a déjà été linké, le rappel de cette fonction forcera le programme à être relinké.

#### 3.5.1.3 `bool bind`

Cette fonction permet d'activer les shaders ajoutés à l'instance de `QGLShaderProgram`. Elle est de type booléen et retourne vrai lorsque le programme a été correctement linké ou faux si ce n'est pas le cas.

#### 3.5.1.4 `void release`

Cette fonction de type void permet de désactiver les shaders ajoutés à l'instance de `QGLShaderProgram`.

### 3.5.1.5 void removeAllShaders()

La fonction removeAllShaders() permet de supprimer tous les shaders ajoutés précédemment à l'instance de QGLShaderProgram.

## 4 Développement

Ce chapitre a pour but de parler du programme qui a été développé durant et avant le travail de Bachelor.

### 4.1 Fichier sht

Les fichiers sht sont de simples fichiers texte contenant une liste des fichiers de shader (.vert et/ou .frag) associés à la scène sélectionnée. Chaque scène comprend donc un ou plusieurs fichiers sht permettant d'avoir des effets différents sur la même scène.

L'utilisateur peut également créer son propre fichier sht. Cette action va créer les

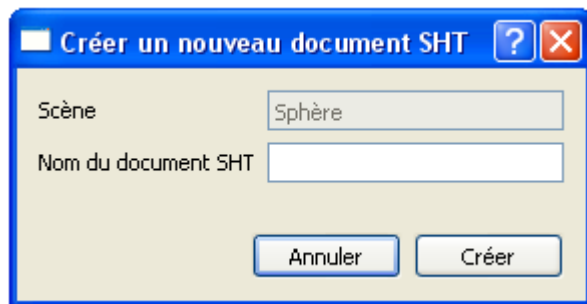


Figure 1 - Nouveau document SHT

fichiers de shaders vierges dont la scène courante a besoin. Ils seront lister dans le nouveau fichier de shaders que l'utilisateur vient de créer. Le programme va ensuite ouvrir le document sht et par conséquent ouvrir les fichiers de shader vierges. L'utilisateur pourra désormais écrire les shaders et appliquer les effets tels qu'il le désire.



## 4.2 Interface

L'interface du didacticiel sur les Shaders contient les 7 zones suivantes :

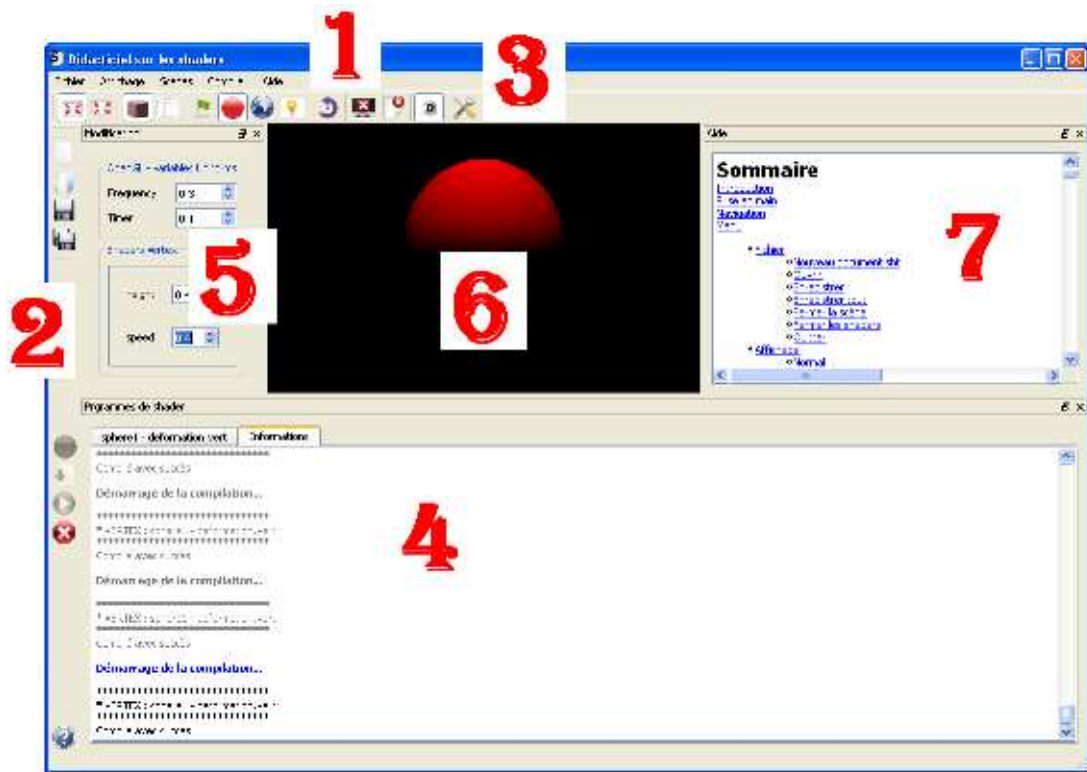


Figure 2 - Interface du didacticiel

Description des zones de la fenêtre principale du didacticiel sur les shader :

### 1) Barre de menu

La barre de menu contient tous les menus et toutes les actions du didacticiel sur les shaders. Ces menus et actions sont détaillés en annexe dans le manuel d'utilisation.

### 2) Barre d'outils principale

La barre d'outils principale du didacticiel sur les shaders contient les actions relatives au shader. Elle permet à l'utilisateur d'accéder aux actions plus rapidement. Ces actions sont également détaillées en annexe dans le manuel d'utilisation. L'utilisateur a la possibilité de l'afficher ou de l'enlever grâce au menu adéquat, de la positionner sur les quatre coins de la fenêtre ou encore en dehors de celle-ci.

### 3) Barre d'outils secondaire

La barre d'outils secondaire du didacticiel sur les shaders contient les actions relatives au rendu graphique. Tout comme la barre d'outils principale, elle permet à l'utilisateur d'accéder aux actions plus rapidement. Ces actions sont également détaillées en annexe dans le manuel d'utilisation. L'utilisateur a la possibilité de l'afficher ou de l'enlever grâce au menu adéquat, de la positionner sur les quatre coins de la fenêtre ou encore en dehors de celle-ci.

#### 4) Programmes de shader

La zone programmes du didacticiel contient tous les programmes de shaders associés à la scène en cours d'utilisation ainsi qu'une zone d'information sur la compilation et l'exécution des programmes de shaders. Ces programmes et ces informations sont affichés dans des QTextEdit. Les QTextEdit se trouvent dans un TabWidget se trouvant dans un QDockWidget fixe. L'utilisateur peut à tout moment afficher ou cacher cette fenêtre grâce au bouton et au menu adéquat. Il peut également la positionner sur les quatre côtés de la fenêtre du programme ou même la détacher de la fenêtre principale. Pour utiliser et modifier des variables grâce au didacticiel, elles doivent être initialisées et assignées entre les commentaires suivants :

```
// Variables_obligatoires_debut  
...  
// Variables_obligatoires_fin
```

Exemple :

```
// Variables_obligatoires_debut  
const float height = 2.7;  
const float speed = 2.2;  
// Variables_obligatoires_fin
```

Pour être prise en compte, une variable doit donc contenir le type, le nom de la variable et sa valeur. Pour modifier le nombre de chiffre après la virgule, il faut assigner dans le programme de shader une valeur avec le nombre de chiffre après la virgule désiré.

#### 5) Didacticiel

Le widget didacticiel de la fenêtre principale contient les outils nécessaires à la modification des variables des programmes de shader de façon didactique. Elle gère les types float et int qui doivent être initialisés et assignés de la façon décrite ci-dessus. Ce QDockWidget peut devenir externe à la fenêtre principale en cliquant sur le bouton adéquat ou en le tirant hors de celle-ci.

#### 6) Rendu 3D

La widget centrale de la fenêtre principale contient le rendu 3D du didacticiel sur les shaders. C'est ici que les scènes sont affichées.

#### 7) Aide

L'aide du didacticiel sur les shaders est un simple QTextBrowser placé dans un QDockWidget. Ce QDockWidget peut devenir externe à la fenêtre principale en cliquant sur le bouton adéquat ou en le tirant hors de celle-ci.

### 4.3 Enregistrement

Lorsqu'un ou plusieurs programme(s) de shader est/sont modifié(s), le programme donne la possibilité à l'utilisateur d'enregistrer les modifications apportées. En effet, si l'utilisateur change de fichier sht (donc de programmes de shader), ferme simplement les programmes de shaders, ferme la scène ou quitte le programme, un message d'avertissement lui demandera s'il veut enregistrer les modifications apportées précédemment.

Si au moins un programme de shader a été modifié, l'utilisateur peut soit sauvegarder le fichier en question, soit sauvegarder tous les fichiers via la barre de menu ou la barre d'outils.

## 4.4 La structure

La structure complète a été refaite et presque toutes les classes ont été redéveloppées de A à Z. Seules les classes concernant les fichiers (File, FileShader et FileSHT) sont restées intactes. La dernière version était vraiment mal conçue, beaucoup trop de code avait été développé dans la classe MainWindow et le programme ne contenait pas de polymorphisme. C'est pour ces raisons, et même si cela n'était pas demandé, que j'ai malgré tout refait la structure complète du programme. Elle n'est actuellement pas parfaite mais beaucoup mieux que la version précédente.

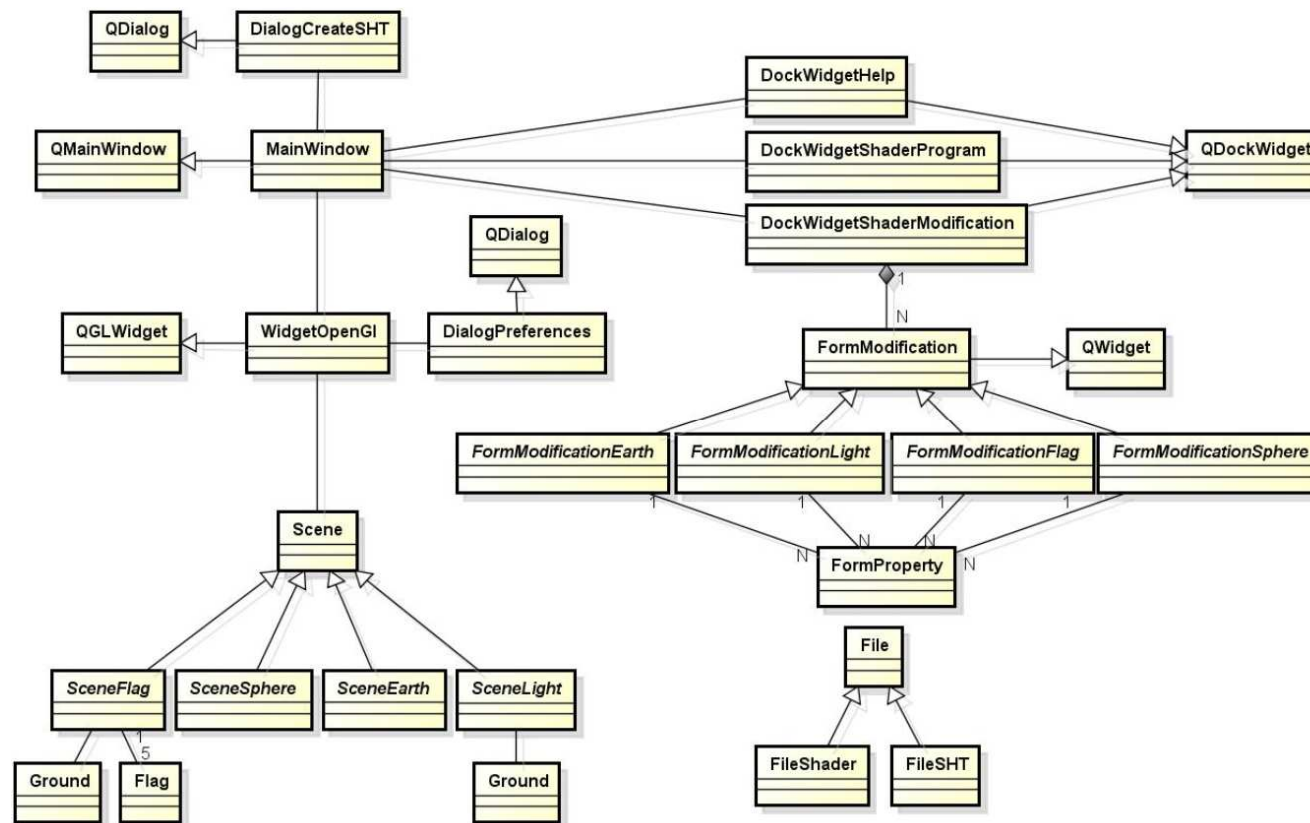


Figure 3 - Architecture

## 4.5 Effet de lumière

Dans un premier temps, le programme disposait déjà d'une source de lumière, mais celle-ci n'avait aucun effet sur les objets qui l'entourait. Lors du projet de printemps, il m'a été demandé d'y remédier en ajoutant des effets de lumières.

J'ai donc ajouté à chacune des scènes des matériaux permettant d'avoir des effets de lumière. Il fut assez compliqué dans la scène « Drapeaux » d'avoir de beaux effets de lumières, car les vertex changent continuellement de position lorsque celle-ci est en exécution.

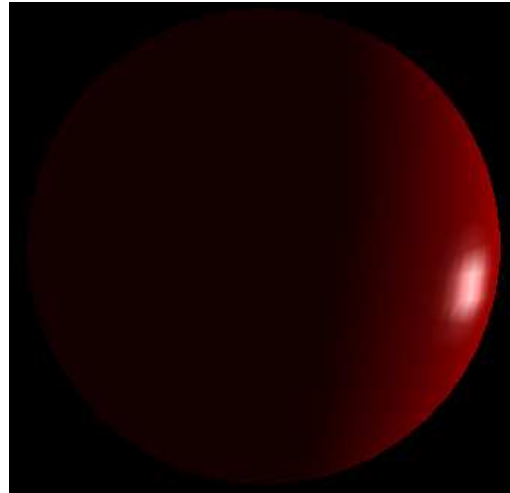


Figure 4 - Lumière appliquée à une sphère

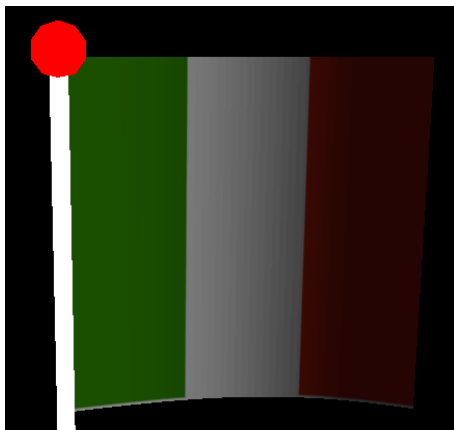


Figure 5 - Lumière appliquée à un drapeau exécuté sans shader

Ces effets de lumière furent plus simples à réaliser pour les scènes « Sphère » et « Earth » car ceux-ci ne comportent pas d'effet sans shader. En effet, ils ont été réalisés avec un simple `gluSphere` qui gère automatiquement le calcul des normales et par conséquent également les effets de lumière

Les shaders de la scène « Drapeaux » de la version du projet de printemps n'avait aucun effet de lumière lors de l'exécution du programme. Par contre, elle contenait des effets de lumière lors de l'exécution de l'effet sans shader. Pour supprimer cette différence et que l'effet soit pratiquement le même avec et sans shader, j'ai ajouté pour chaque shader la fonction `computeNormal` permettant de calculer les normales et de les appliquer.

## 4.6 Effet sans shaders

Afin de pouvoir comparer les effets de shader et les effets sans shader, j'ai dû, lors du travail de printemps, ajouter la possibilité d'effectuer un effet sur une scène sans exécuter de shader.

Pour ce qui concerne les scènes « Sphère » et « Earth », elles sont toutes deux construites à partir de la fonction `gluSphere` qui ne permet pas la modification de la forme d'une sphère créée à partir de cette fonction. Je n'ai donc pas pu ajouter la possibilité d'effectuer d'effet sans shader

La scène « Drapeaux » est donc l'unique scène à pouvoir être exécutée sans shader, car elle est dessinée à l'aide des primitives d'OpenGL. Il était donc possible de modifier les vertex sans l'exécution des shaders. Le seul souci était de calculer les normales de chaque vertex avant de les dessiner. Pour être plus clair, lors du dessin du drapeau, à chaque itération, seulement deux points sont dessinés. Le problème est que pour calculer une normal, il me fallait trois points. J'ai donc dû calculer les normales avant de dessiner chaque drapeau.

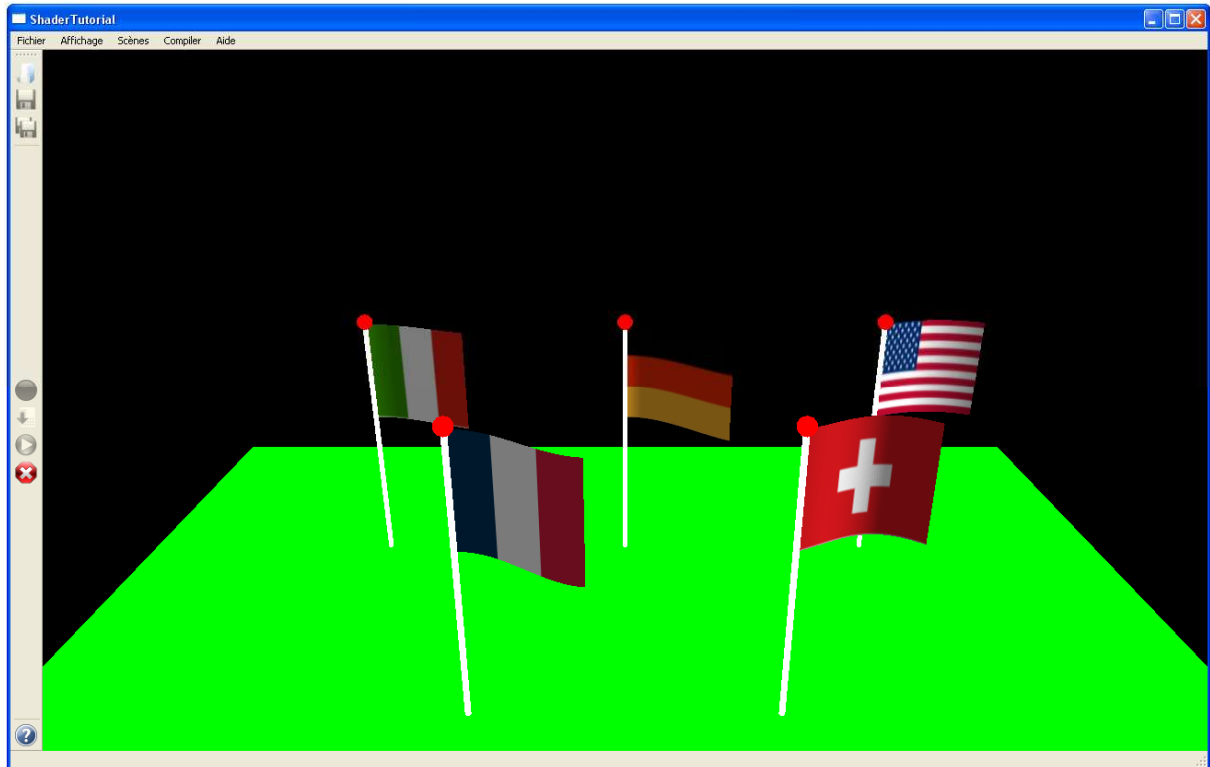


Figure 6 - Effet sans shaders



## 4.7 Type d'affichage

L'utilisateur du programme a la possibilité d'afficher les objets composant la scène visionnée en solide ou en fils de fer. Le but est de mieux comprendre comment les objets sont dessinés, de mieux comprendre les effets de lumière et de mieux comprendre les effets de shader.

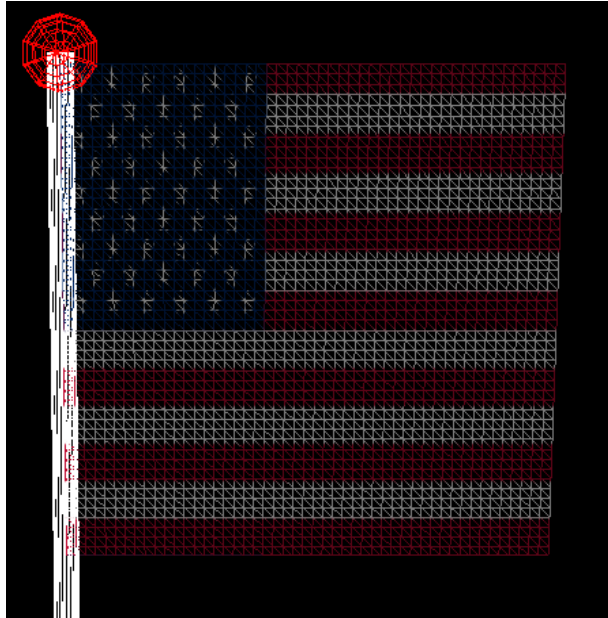


Figure 7 - Fils de fer - Drapeau

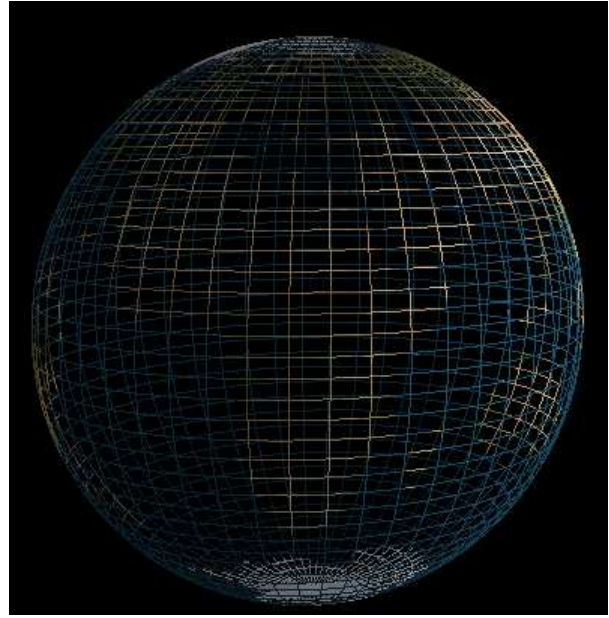


Figure 8 - Fils de fer - Earth

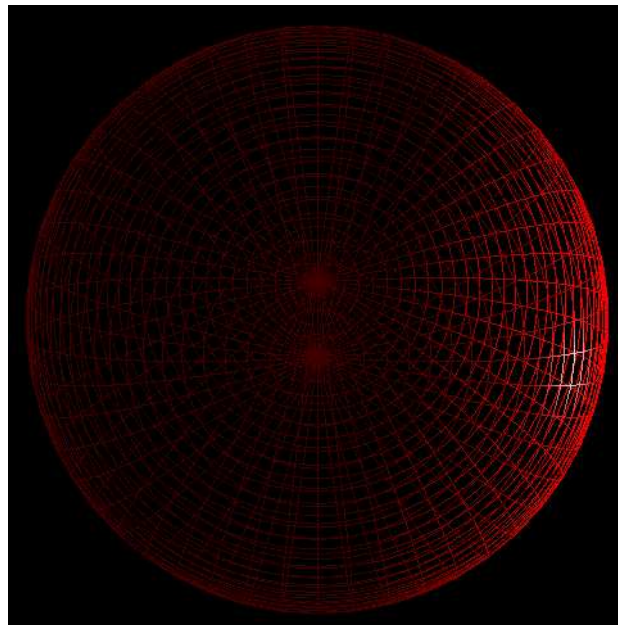


Figure 9 - Fils de fer - Sphère

## 4.8 Facettes

Il est possible pour les objets principaux de chaque scène de définir le nombre de facettes qui les composent afin de mieux comprendre les effets de shader et de comparer la vitesse d'exécution du programme en comparant le même objet avec un nombre élevé de facettes et un nombre infime de facettes. Cela m'a permis de constater qu'un nombre très élevé de facettes ralenti considérablement l'exécution de l'application.

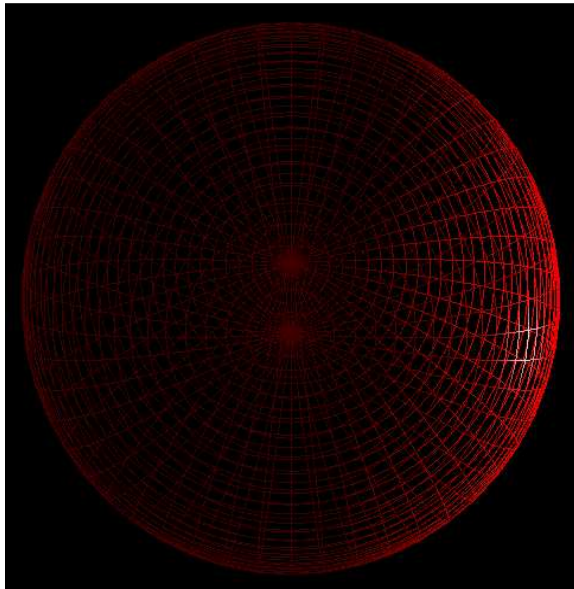


Figure 10 – 50 facettes

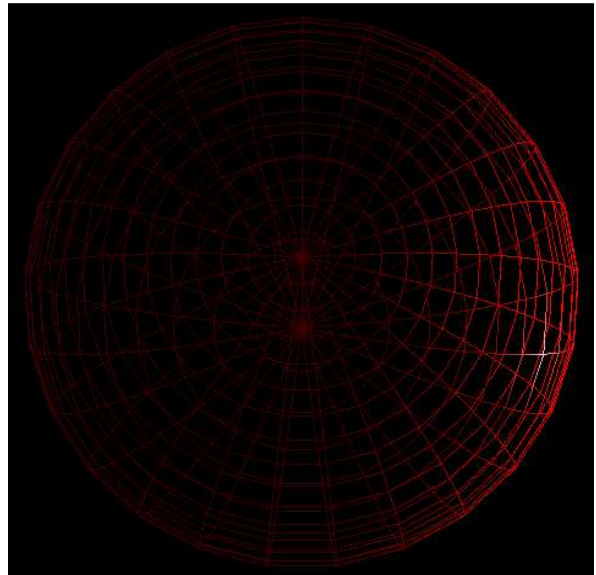


Figure 11 – 25 facettes

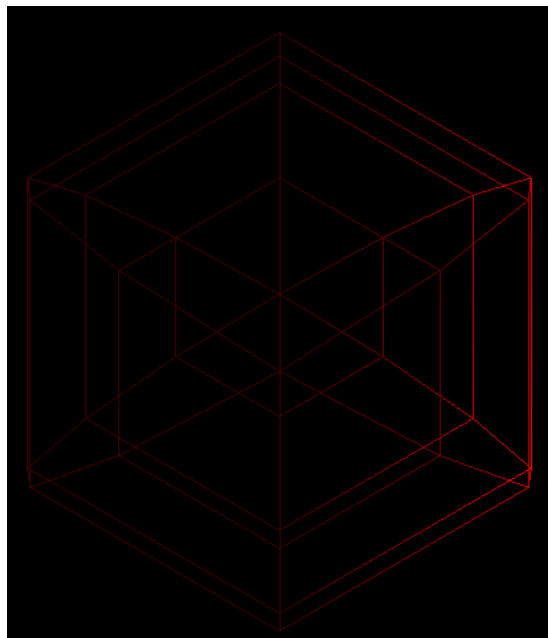


Figure 12 – 6 facettes



## 4.9 Scène lumière

Cette scène a été ajoutée durant le travail de Bachelor. Elle est composée d'un cube, d'une sphère et d'un cône. Elle contient 3 fichiers SHT ayant des effets différents mais travaillant tous avec des effets de lumière :

- 3lights : Ce fichier a pour but de créer un effet de lumière avec 3 lumières différentes.
- diffuse : Ce fichier a pour but de créer un éclairage avec lumière diffuse.
- specular : Ce fichier a pour but de créer un éclairage avec lumière spéculaire.

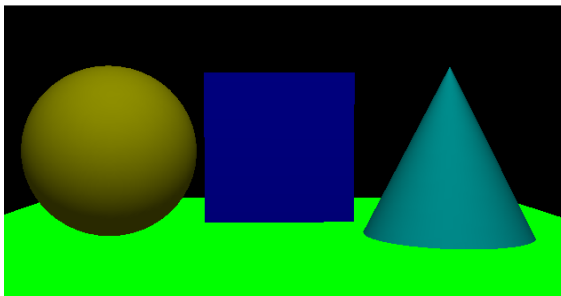


Figure 13 - Sans effet de shader

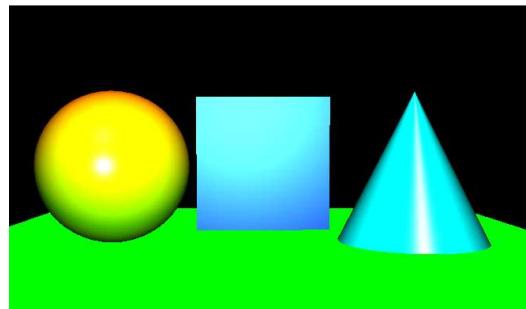


Figure 14 - Effet 3lights

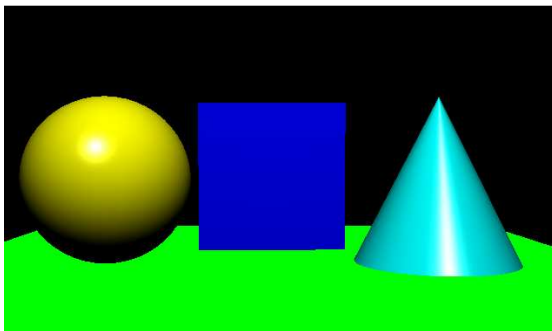


Figure 15 - effet specular

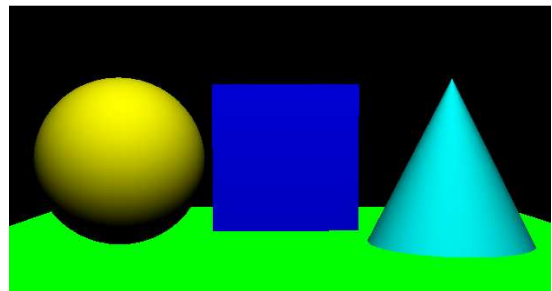


Figure 16 - Effet diffuse

## 4.10 Récupération de la position de la lumière

Les scènes « Drapeaux » et « Lumière » reçoivent, depuis le programme OpenGL, le qualifieur de type uniform *LightPosition*, qui contient les coordonnées de position de la Lumière. Après rotations et déplacements dans la scène, la position de la lumière ne correspond plus à la position réelle. Dans la version précédente du programme, la lumière ne se déplaçait pas avec la scène lors de la rotation de celle-ci, mais restait à la même position. Cela n'était vraiment pas joli et c'est pour cette raison que j'ai décidé d'y remédier.

Dans un premier temps je suis parti sur une fonction calculant la position absolue de la lumière, mais elle ne fonctionnait pas et renvoyait des valeurs quelconques.

Je me suis donc penché sur une autre méthode qui consiste à travailler avec une matrice de sauvegarde.

Il faut donc initialiser cette matrice :

```
widgetopengl.cpp

657 void WidgetOpenGL::initMatricePlan()
658 {
659     glMatrixMode(GL_MODELVIEW);
660     glPushMatrix();
661     glLoadIdentity();
662     glGetFloatv(GL_MODELVIEW_MATRIX, matrice);
663     glPopMatrix();
664 }
```

Il faut ensuite y copier la position réelle :

```
widgetopengl.cpp

416 glPushMatrix();
417     glRotatef(lightRotationAngle,0,1,0);
418     glGetFloatv(GL_MODELVIEW_MATRIX, matrice);
419     lightPosition();
420 glPopMatrix();
```

Pour finir, grâce à la fonction *getAbsoluteLightPos()*, on retourne la position de la lumière.

```
widgetopengl.cpp

258 GLfloat *WidgetOpenGL::getAbsoluteLightPos()
259 {
260     GLfloat *lightAbsPosTmp = new GLfloat[4];
261     lightAbsPosTmp[0] = matrice[0]*lightPos[0]+matrice[4]
        *lightPos[1]+matrice[8]*lightPos[2]+ matrice[12];
262     lightAbsPosTmp[1] = matrice[1]*lightPos[0]+matrice[5]
        *lightPos[1]+matrice[9]*lightPos[2]+matrice[13];
263     lightAbsPosTmp[2] = matrice[2]*lightPos[0]+matrice[6]
        *lightPos[1]+matrice[10]*lightPos[2]+matrice[14];
264
265
266     lightAbsPosTmp[3] = lightPos[3];
267     return lightAbsPosTmp;
268 }
```

## 4.11 Le didacticiel

Le but était de pouvoir modifier des variable se trouvant dans les programmes de shader de façon didactique et de pouvoir en ajouter autant que désiré et de façon automatique. Ce chapitre permet de comprendre les étapes afin d'y arriver. Seules les variables de type int et float sont gérées par l'application.

### 4.11.1 Parser le fichier

Pour qu'une variable puisse être gérée par l'application, elle doit impérativement, dans le fichier de shader, être initialisée et assignée sur la même ligne et entre les commentaires ci-dessous :

```
// Variables_obligatoires_debut  
  
float maVar = 0.1;  
  
// Variables_obligatoires_fin
```

Le fichier va donc être parsé une première fois pour récupérer le texte se trouvant entre les commentaires de début et de fin, grâce à l'expression régulière ci-dessous :

```
dockwidgetshaderprogram.cpp  
  
156     QRegExp regExp("\\bVariables_obligatoires_debut\\b(.*)  
        \\bVariables_obligatoires_fin\\b(.*)");
```

Il est ensuite parser une seconde fois pour récupérer les variables, grâce à une nouvelle expression régulière.

```
dockwidgetshaderprogram.cpp  
  
165     regExp.setPattern("((float|int)\\s+(\\S+)\\s*=\\s*(\\d+(\\.\\d+)?);)");
```

Les types, noms, et valeurs des variables sont alors récupérés et retournés à la classe DockWidgetShaderModification qui ajoutera toutes les variables au dockWidget en question. Les variables sont retournées grâce à une QListe contenant une autre QList qui elle contient le type, le nom et la valeur. Donc chaque variable se trouve dans une QList différente.

```
dockwidgetshaderprogram.cpp  
  
169     while(pos >= 0)  
170     {  
171         ....  
177         variableParam.append(regExp.cap(2)); // Type de la variable  
178         variableParam.append(regExp.cap(3)); // Nom de la variable  
179         variableParam.append(regExp.cap(4)); // Valeur de la variable  
180  
181         listVariables.append(variableParam);  
182         ....  
183     }
```

Pour modifier la valeur d'une variable dans le QTextEdit, lorsque celle-ci est modifiée de manière didactique depuis le didacticiel, la classe DockWidgetShaderModification appelle la fonction `setVariableValue(QString _name, QString _value)` de la classe `DockWidgetShaderProgram`.

Il faut donc à nouveau parser le fichier pour récupérer l'endroit où la modification de valeur aura lieu. Il est à nouveau parsé une première fois (comme pour la récupération des variables expliquée ci-dessus), à l'aide d'une expression régulière, pour récupérer le texte se trouvant entre les commentaires de début et de fin :

```
dockwidgetshaderprogram.cpp  
  
220     QRegExp regExp("\\bVariables_obligatoires_debut\\b(.*)  
        \\bVariables_obligatoires_fin\\b(.*)");
```

On parse ensuite le texte récupéré pour trouver la variable à modifier à l'aide du nom de celle-ci :

```
dockwidgetshaderprogram.cpp  
  
231     regExp.setPattern("(((float|int)\\s"+_name+"\\s*=\\s*)  
        (\\d+(\\.\\d+)?)?);)");
```

Pour finir on modifie le texte se trouvant dans le QTextEdit en question :

```
dockwidgetshaderprogram.cpp  
  
247     shaderText.insert(pos1+pos2, _value);  
248  
249     listTextEdit->at(i)->setText(shaderText);
```

## 5 Problèmes rencontrés

### 5.1 Shader pour les ombres.

Je n'ai malheureusement pas eu le temps de finir ce shader qui n'a donc pas été ajouté à la version finale du projet. Cette scène fonctionnait parfaitement lors de l'exécution de l'effet sans shader.

### 5.2 Scène « Drapeaux » avec shader

J'ai ajouté les effets de lumière à ce programme qui n'en possédait uniquement lors de l'exécution des effets sans shader. Une fonction permettant de calculer la normale a donc été ajoutée à chaque shader. Les effets de lumière fonctionnent correctement, mais les deux faces réagissent de la même façon. Malgré mes recherches, je n'ai pas trouvé de fonction GLSL ayant la même fonctionnalité que `glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE)` d'OpenGL qui permet de séparer le calcul des faces avant et arrière.

### 5.3 Récupération de la position de la lumière

La récupération de la position de la lumière n'a pas été facile. Comme décrit précédemment, j'ai essayé d'y parvenir avec une autre méthode avant d'utiliser la méthode actuelle et de parvenir à un résultat. Cela m'a pris du temps mais je suis quand même parvenu au résultat désiré.

### 5.4 Polymorphisme

J'ai pris comme mauvaise habitude de faire tous les includes dans les fichiers .h pour des raisons d'esthétique. Cela m'a vraiment posé de gros problème lors du développement de la classe Scene qui récupère son parent. En effet, la ligne ci-dessous se trouvait dans le fichier scene.h :

```
#include "widgetopengl.h"
```

Le programme ne compilait plus et le message d'erreur suivant apparaissait:

```
expected class-name before '{' token
```

Il m'a fallu quelque jour avant de comprendre que cette include ne devait pas se trouver dans le fichier scene.h mais dans le fichier scene.cpp.

```
scene.cpp  
  
2     #include "widgetopengl.h"
```

J'ai eu le même souci avec la classe FormModification qui était développée en parallèle.

### 5.5 Scène « Earth »

J'ai voulu réaliser un shader permettant de travailler en multitexturing. Le but était d'avoir une texture de la terre jour et une autre de nuit, de les superposer et d'avoir un effet avec changement de texture par rapport à la position du soleil.

En développant ce shader, j'ai réalisé que le shader travaillant avec une seule texture, réalisé précédemment, ne recevait jamais de texture et qu'il travaillait avec la texture gérée par OpenGL.

J'ai donc ajouté la classe MultiTextCoordARB qui gère le multitexturing, mais le programme de shader n'a jamais reçu de texture. J'ai quand même laissé cette scène à la version finale du faite que le shader travaillant avec une seule texture fonctionne quand même mais en travaillant avec la texture OpenGL et non celle qui lui est envoyée. Je n'ai pas ajouté le shader travaillant avec le multitexturing qui ne fonctionnait pas.

## 6 Améliorations possibles

### 6.1 Ajout d'autres scènes

Il serait intéressant d'enrichir le didacticiel avec d'autres scènes et avec des effets de shader différents. En effet, il existe de nombreux effets possibles grâce aux programmes de shader, tels que ceux listés au chapitre « 3.1 Liste des effets possibles ».

### 6.2 Ajouter d'autres types de variables gérées par le didacticiel

Le didacticiel gère actuellement les variables de type float et int. Il serait intéressant et pratique d'ajouter d'autres types de variables, tels que `vec3` et `vec4`, qui sont très souvent utilisées dans le domaine de la programmation 3D. Il faudrait avant tout créer une scène contenant un programme de shader pour les ombres comme demandé dans le cahier des charges.

### 6.3 La scène « Drapeaux »

Les shaders de la scène « Drapeaux » ont été modifiés de façon à ce qu'il y ait des effets de lumière sur les drapeaux, lors de l'exécution de ceux-ci. Cependant, les deux faces réagissent de la même manière. J'entends par là, que si la lumière se trouve devant le drapeau, les deux faces seront éclairées et si la lumière se trouve derrière, aucune des faces ne sera éclairée. Il faudrait modifier cela soit en trouvant une fonction équivalente à `glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE)` d'OpenGL, qui sépare le calcul des faces avant et arrière, soit d'apporter du volume aux drapeaux.

## 7 Conclusion

Malheureusement, le cahier des charges qui m'a été attribué n'a pas pu être totalement accompli dû notamment au fait que la structure complète du programme a été refaite de A à Z. Cependant, et même si cela ne m'a pas été demandé, il était vraiment nécessaire de restructurer le programme pour faciliter l'ajout de nouvelles scènes et faciliter une éventuelle prise en main d'un étudiant qui souhaiterait continuer le développement du didacticiel. Le programme est nettement mieux structuré que ce qu'il l'était par le passé.

Un programme de shader pour les ombres aurait dû être ajouté, mais je n'ai hélas pas eu le temps de le terminer.

Ma plus grande déception est de ne pas avoir pu finir le programme de shader sur les ombres, mais je reste quand même sur un point positif. J'ai beaucoup appris et me suis beaucoup amélioré en programmation. J'ai pu utiliser mes connaissances « théoriques » acquises dans le cadre d'autres cours et ai pu les mettre en pratique.

Le sujet m'a vraiment beaucoup plu et c'est pour cette raison que j'ai décidé de continuer sur ce thème pour mes projets de printemps et de bachelor afin de pouvoir encore élargir mes connaissances et surtout d'améliorer l'application sur laquelle j'avais déjà beaucoup travaillé.

## 8 Références

Sites internet :

- <http://fr.wikipedia.org/wiki/Shader>
- <http://www.siteduzero.com/tutoriel-3-8894-les-shaders-en-glsl.html>
- <http://garzul.tonsite.biz/Forum/viewtopic.php?f=28&t=426>
- <http://www.shadows.fr/tutoriaux-3D/shaders-3d-temps-reel>
- <http://www.01net.com/editorial/348715/directx-10-voici-pourquoi-vous-allez-changer-votre-carte-graphique/>

Documents :

- RapportFinal\_Corrigé.docx + codes sources (HES d'été)
- Rapport du projet d'automne
- Rapport du projet de printemps
- OpenGL SuperBible, 4th Edition (Addison Wesley, 2007).pdf
- GLSL\_Full.1.20.8.pdf (non imprimé du au nombre de page)
- GLSL\_Full.4.0.pdf (non imprimé du au nombre de page)
- docglsl.pdf (non imprimé du au nombre de page)

## 9 Liste des figures

Figure 1 - Nouveau document SHT .....	24
Figure 2 - Interface du didacticiel.....	25
Figure 3 - Architecture .....	28
Figure 4 - Lumière appliquée à une sphère.....	29
Figure 5 - Lumière appliquée à un drapeau exécuté sans shader.....	29
Figure 6 - Effet sans shaders.....	30
Figure 7 - Fils de fer - Drapeau.....	31
Figure 8 - Fils de fer - Earth.....	31
Figure 9 - Fils de fer - Sphère.....	31
Figure 10 – 50 facettes .....	32
Figure 11 – 25 facettes .....	32
Figure 12 – 6 facettes .....	32
Figure 13 - Sans effet de shader .....	33
Figure 14 - Effet 3lights .....	33
Figure 15 - effet specular.....	33
Figure 16 - Effet diffuse .....	33

## 10 Annexes

- Manuel d'utilisation
- Cahier des charges

Christophe Magri