Réalité Virtuelle

Leo Donati

OpenGL

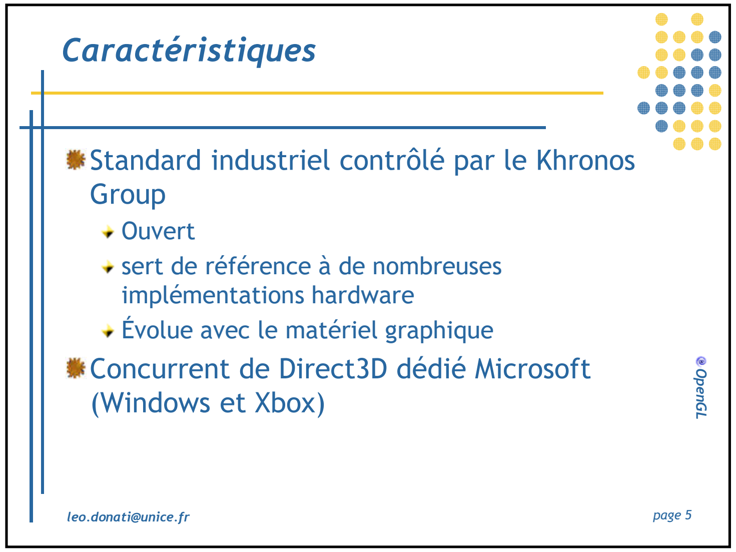
Université Nice Sophia Antipolis

OpenGL

1. Introduction à OpenGL

1.1. Présentation

- OpenGL® (Open Graphic Library) est une interface logicielle pour le hardware graphique qui permet aux programmeurs de produire des images de haute qualité d'objets 3D
  - Un seul objectif = le RENDU !
- Portabilité
  - systèmes : Windows, Unix et X-Windows, Mac OS, OS/2...
  - matériels : Intel, IBM, Power PC, Dec Alpha, Iris...
  - OpenGL ES pour systèmes embarqués : iOS, Android, Symbian
  - Consoles de jeux : PS3, Wii

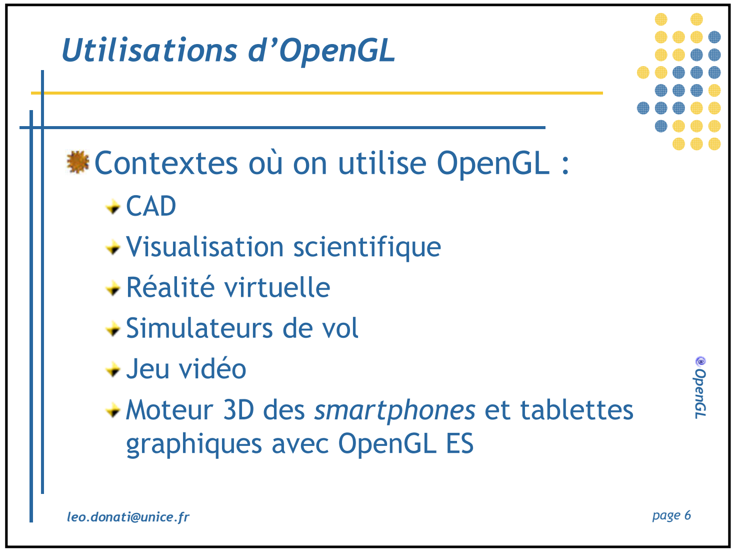


Caractéristiques

- Standard industriel contrôlé par le Khronos Group
  - Ouvert
  - sert de référence à de nombreuses implémentations hardware
  - Évolue avec le matériel graphique
- Concurrent de Direct3D dédié Microsoft (Windows et Xbox)

OpenGL

leo.donati@unice.fr *page 5*

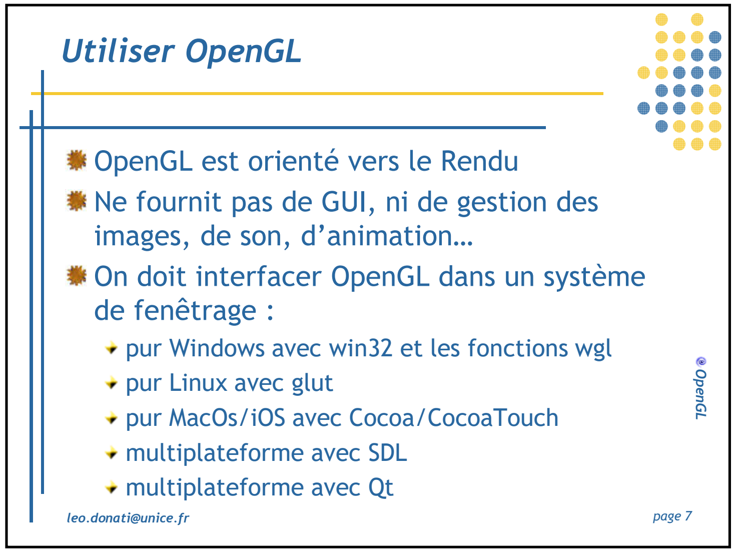


Utilisations d'OpenGL

- Contextes où on utilise OpenGL :
  - CAD
  - Visualisation scientifique
  - Réalité virtuelle
  - Simulateurs de vol
  - Jeu vidéo
  - Moteur 3D des *smartphones* et tablettes graphiques avec OpenGL ES

OpenGL

leo.donati@unice.fr *page 6*

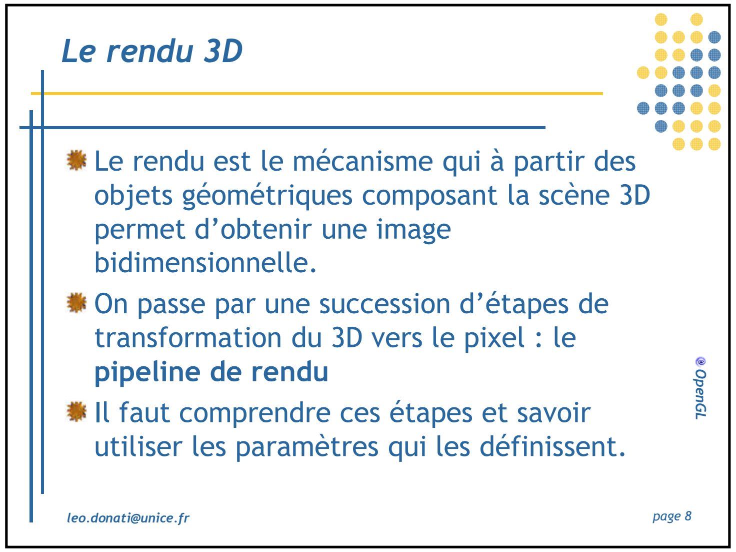


Utiliser OpenGL

- OpenGL est orienté vers le Rendu
- Ne fournit pas de GUI, ni de gestion des images, de son, d'animation…
- On doit interfacer OpenGL dans un système de fenêtrage :
  - pur Windows avec win32 et les fonctions wgl
  - pur Linux avec glut
  - pur MacOs/iOS avec Cocoa/CocoaTouch
  - multiplateforme avec SDL
  - multiplateforme avec Qt

OpenGL

leo.donati@unice.fr *page 7*

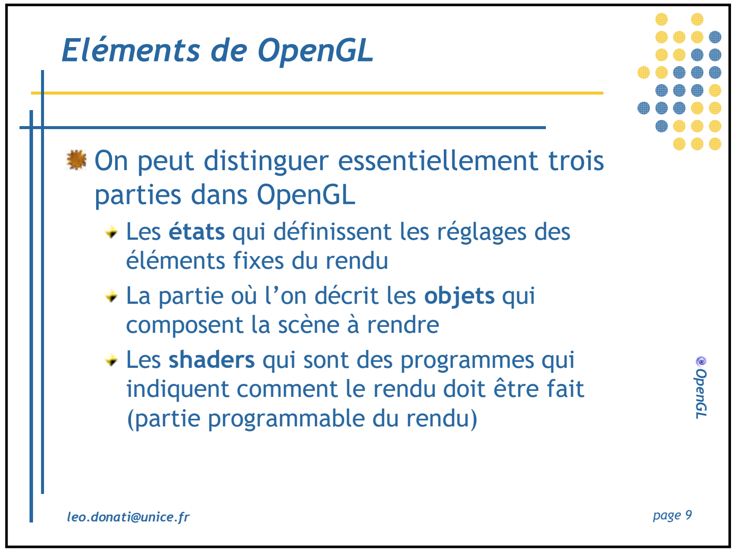


Le rendu 3D

- Le rendu est le mécanisme qui à partir des objets géométriques composant la scène 3D permet d'obtenir une image bidimensionnelle.
- On passe par une succession d'étapes de transformation du 3D vers le pixel : le **pipeline de rendu**
- Il faut comprendre ces étapes et savoir utiliser les paramètres qui les définissent.

OpenGL

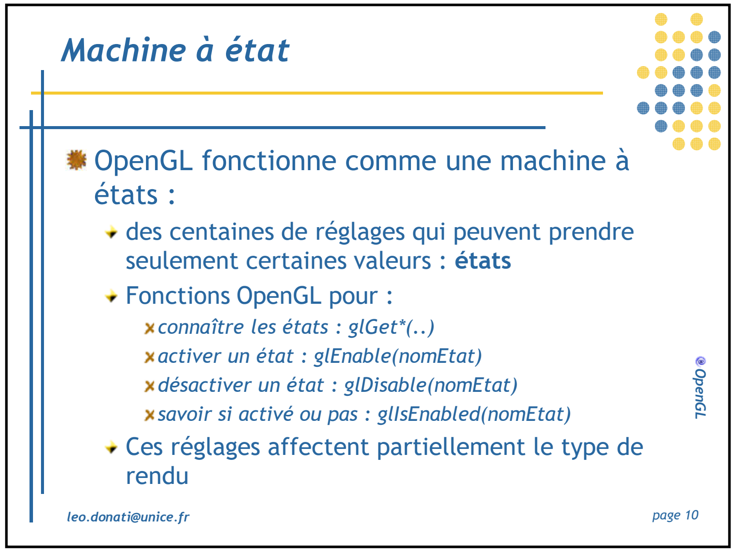
leo.donati@unice.fr *page 8*



Eléments de OpenGL

- On peut distinguer essentiellement trois parties dans OpenGL
  - Les **états** qui définissent les réglages des éléments fixes du rendu
  - La partie où l'on décrit les **objets** qui composent la scène à rendre
  - Les **shaders** qui sont des programmes qui indiquent comment le rendu doit être fait (partie programmable du rendu)

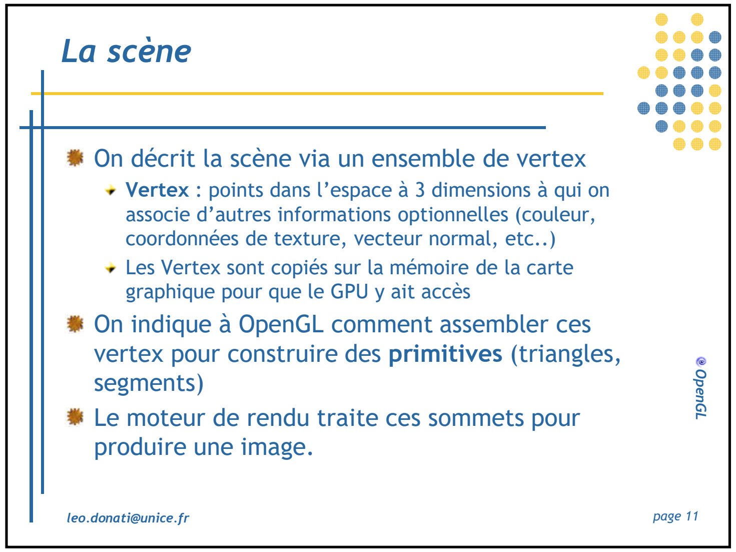
OpenGL



Machine à état

- OpenGL fonctionne comme une machine à états :
  - des centaines de réglages qui peuvent prendre seulement certaines valeurs : **états**
  - Fonctions OpenGL pour :
    - *connaître les états : glGet*(..)*
    - *activer un état : glEnable(nomEtat)*
    - *désactiver un état : glDisable(nomEtat)*
    - *savoir si activé ou pas : glIsEnabled(nomEtat)*
  - Ces réglages affectent partiellement le type de rendu

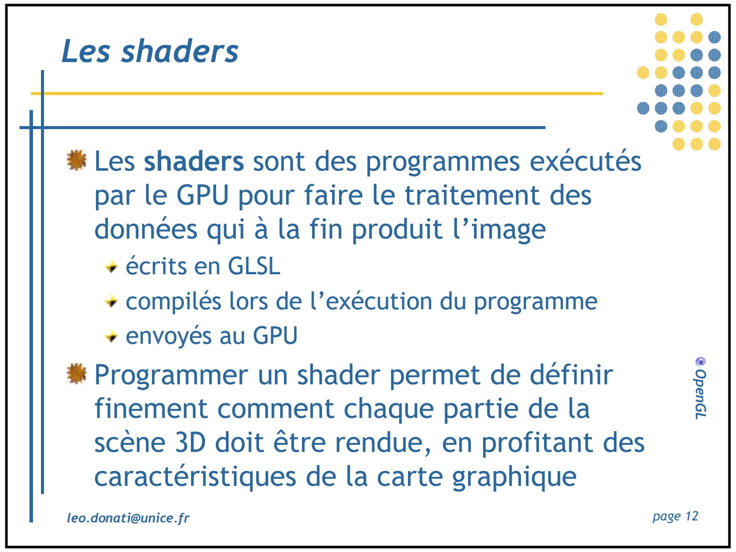
OpenGL



La scène

- On décrit la scène via un ensemble de vertex
  - **Vertex** : points dans l'espace à 3 dimensions à qui on associe d'autres informations optionnelles (couleur, coordonnées de texture, vecteur normal, etc..)
  - Les Vertex sont copiés sur la mémoire de la carte graphique pour que le GPU y ait accès
- On indique à OpenGL comment assembler ces vertex pour construire des **primitives** (triangles, segments)
- Le moteur de rendu traite ces sommets pour produire une image.

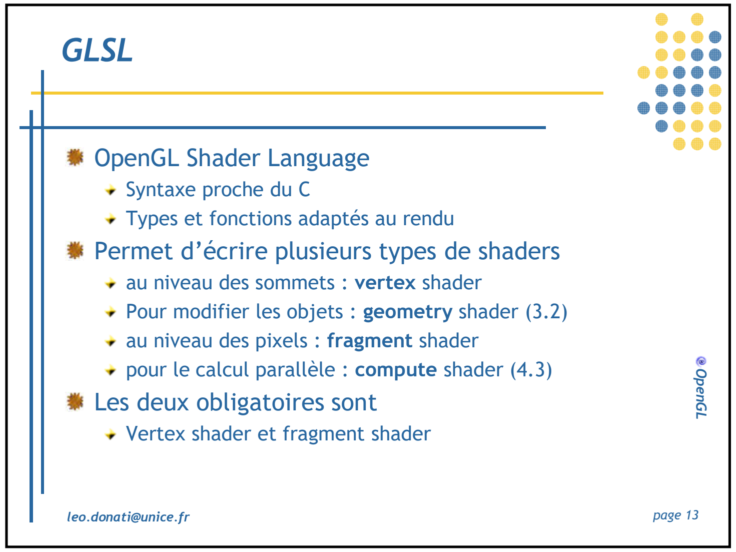
OpenGL



Les shaders

- Les **shaders** sont des programmes exécutés par le GPU pour faire le traitement des données qui à la fin produit l'image
  - écrits en GLSL
  - compilés lors de l'exécution du programme
  - envoyés au GPU
- Programmer un shader permet de définir finement comment chaque partie de la scène 3D doit être rendue, en profitant des caractéristiques de la carte graphique

OpenGL

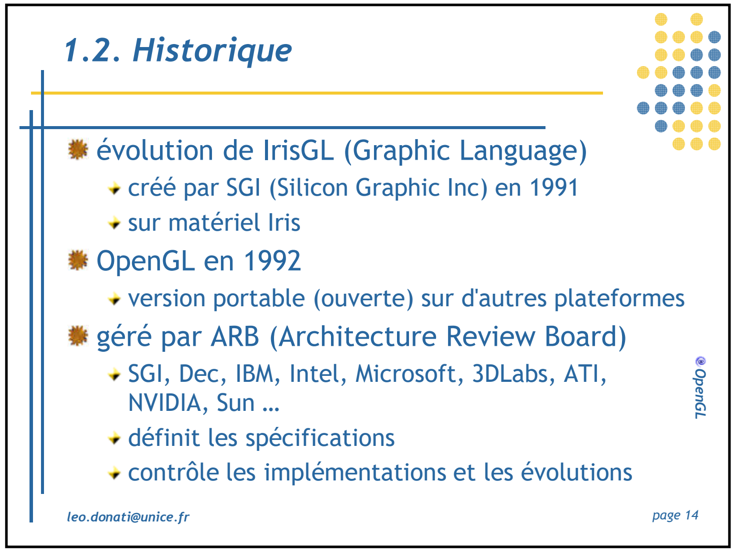


GLSL

- OpenGL Shader Language
  - Syntaxe proche du C
  - Types et fonctions adaptés au rendu
- Permet d'écrire plusieurs types de shaders
  - au niveau des sommets : **vertex** shader
  - Pour modifier les objets : **geometry** shader (3.2)
  - au niveau des pixels : **fragment** shader
  - pour le calcul parallèle : **compute** shader (4.3)
- Les deux obligatoires sont
  - Vertex shader et fragment shader

© OpenGL

leo.donati@unice.fr *page 13*

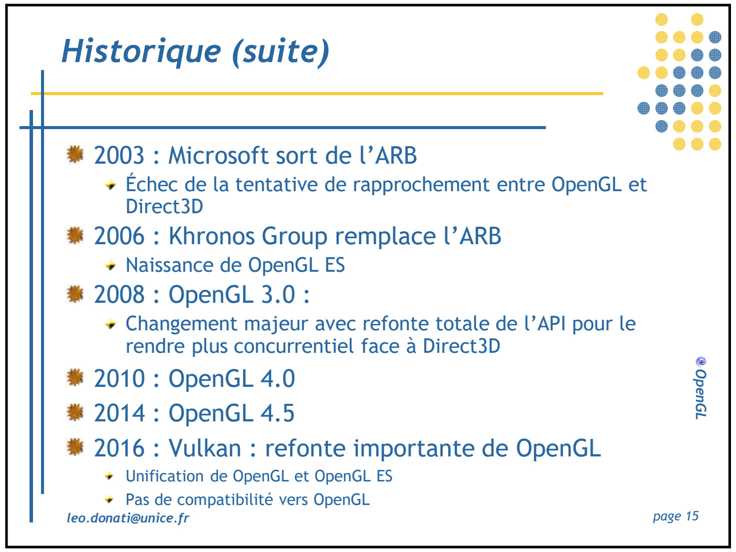


1.2. Historique

- évolution de IrisGL (Graphic Language)
  - créé par SGI (Silicon Graphic Inc) en 1991
  - sur matériel Iris
- OpenGL en 1992
  - version portable (ouverte) sur d'autres plateformes
- géré par ARB (Architecture Review Board)
  - SGI, Dec, IBM, Intel, Microsoft, 3DLabs, ATI, NVIDIA, Sun …
  - définit les spécifications
  - contrôle les implémentations et les évolutions

© OpenGL

leo.donati@unice.fr *page 14*



Historique (suite)

- 2003 : Microsoft sort de l'ARB
  - Échec de la tentative de rapprochement entre OpenGL et Direct3D
- 2006 : Khronos Group remplace l'ARB
  - Naissance de OpenGL ES
- 2008 : OpenGL 3.0 :
  - Changement majeur avec refonte totale de l'API pour le rendre plus concurrentiel face à Direct3D
- 2010 : OpenGL 4.0
- 2014 : OpenGL 4.5
- 2016 : Vulkan : refonte importante de OpenGL
  - Unification de OpenGL et OpenGL ES
  - Pas de compatibilité vers OpenGL

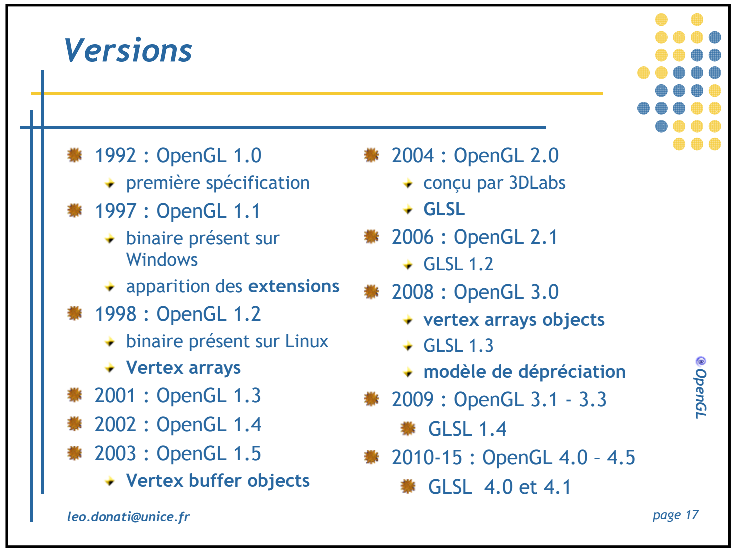
© OpenGL

leo.donati@unice.fr *page 15*

Khronos Group

© OpenGL

leo.donati@unice.fr *page 16*

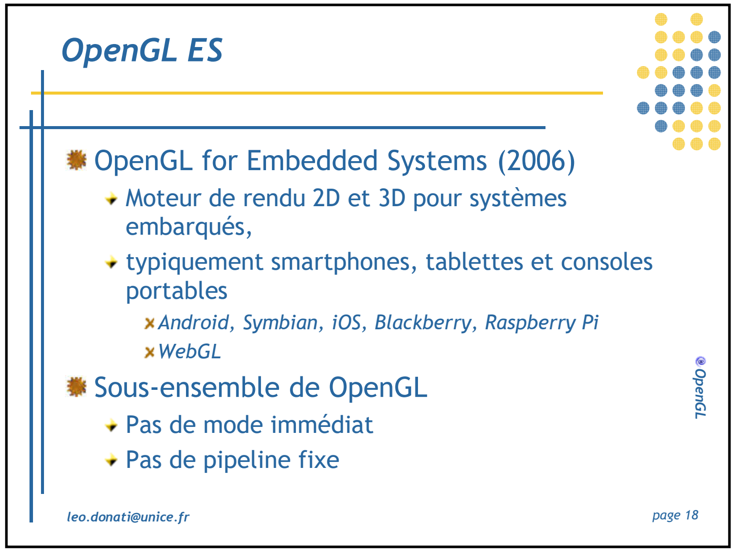


Versions

- 1992 : OpenGL 1.0
  - première spécification
- 1997 : OpenGL 1.1
  - binaire présent sur Windows
  - apparition des **extensions**
- 1998 : OpenGL 1.2
  - binaire présent sur Linux
  - **Vertex arrays**
- 2001 : OpenGL 1.3
- 2002 : OpenGL 1.4
- 2003 : OpenGL 1.5
  - **Vertex buffer objects**
- 2004 : OpenGL 2.0
  - conçu par 3DLabs
  - **GLSL**
- 2006 : OpenGL 2.1
  - GLSL 1.2
- 2008 : OpenGL 3.0
  - **vertex arrays objects**
  - GLSL 1.3
  - **modèle de dépréciation**
- 2009 : OpenGL 3.1 - 3.3
  - GLSL 1.4
- 2010-15 : OpenGL 4.0 - 4.5
  - GLSL 4.0 et 4.1

© OpenGL

leo.donati@unice.fr page 17

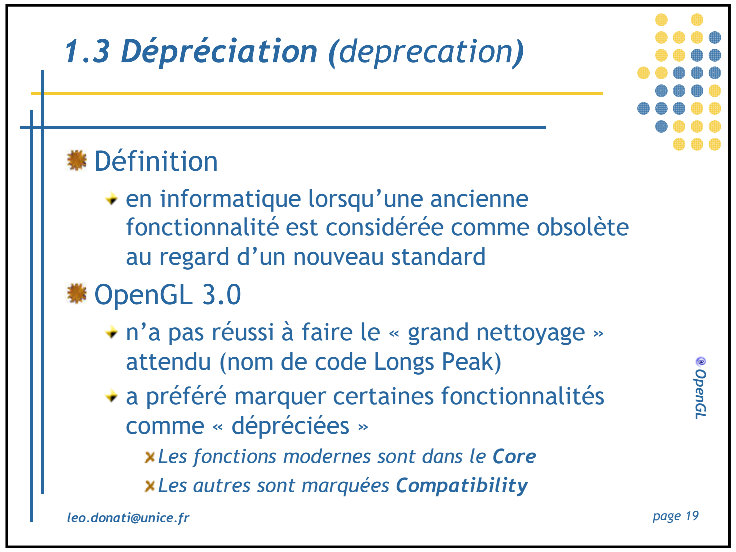


OpenGL ES

- OpenGL for Embedded Systems (2006)
  - Moteur de rendu 2D et 3D pour systèmes embarqués,
  - typiquement smartphones, tablettes et consoles portables
    - *Android, Symbian, iOS, Blackberry, Raspberry Pi*
    - *WebGL*
- Sous-ensemble de OpenGL
  - Pas de mode immédiat
  - Pas de pipeline fixe

© OpenGL

leo.donati@unice.fr page 18

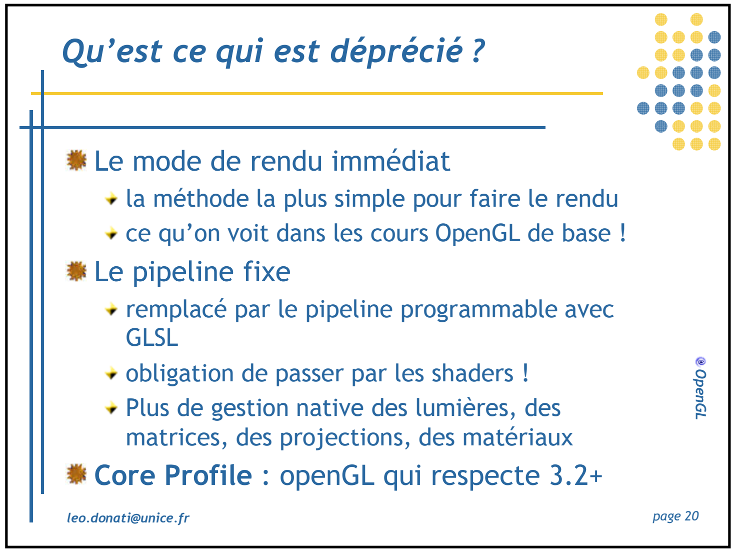


1.3 Dépréciation (deprecation)

- Définition
  - en informatique lorsqu'une ancienne fonctionnalité est considérée comme obsolète au regard d'un nouveau standard
- OpenGL 3.0
  - n'a pas réussi à faire le « grand nettoyage » attendu (nom de code Longs Peak)
  - a préféré marquer certaines fonctionnalités comme « dépréciées »
    - *Les fonctions modernes sont dans le **Core***
    - *Les autres sont marquées **Compatibility***

© OpenGL

leo.donati@unice.fr page 19



Qu'est ce qui est déprécié ?

- Le mode de rendu immédiat
  - la méthode la plus simple pour faire le rendu
  - ce qu'on voit dans les cours OpenGL de base !
- Le pipeline fixe
  - remplacé par le pipeline programmable avec GLSL
  - obligation de passer par les shaders !
  - Plus de gestion native des lumières, des matrices, des projections, des matériaux
- **Core Profile** : openGL qui respecte 3.2+

© OpenGL

leo.donati@unice.fr page 20

1.4 Extensions

- Les extensions sont un mécanisme OpenGL pour utiliser des capacités non prévues dans la version OpenGL installée
  - parce que gl.h est arrêté à la version 1.1
    - *il faut utiliser les extensions pour les VBO*
  - pour utiliser des capacités de la carte qui sont non standard
    - *spécifiques à une carte ou à un constructeur*
  - pour utiliser une fonctionnalité qui est un futur standard

© OpenGL

leo.donati@unice.fr *page 21*

2. Vertices

- 2.1. Mode immédiat
- 2.2. Vertex Buffer Object (VBO)
- 2.3. Index Buffer Object (IBO)
- 2.4. Vertex Array Object (VAO)

© OpenGL

leo.donati@unice.fr *page 22*

Définition des vertex

- L'entrée du pipeline de rendu est un ensemble de vertex
- Les vertex sont les sommets des faces des objets qui composent la scène
  - Beaucoup
- Les vertex portent (en plus de leur coordonnées) des données optionnelles spécifiques à chaque sommet qui seront utilisées lors du rendu
  - Couleur, texture, transparence, luminosité

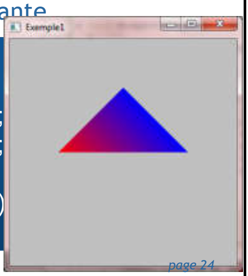
© OpenGL

leo.donati@unice.fr *page 23*

2.1 Mode immédiat - Déprécié

- Dans le mode immédiat :
  - chaque sommet est déclaré par un appel à **glVertex**
  - Le sommet porte la couleur courante

```
glBegin(GL_TRIANGLES);
                glColor3f (0.0f, 0.0f, 1.0f);
                glVertex3f(1.0f, 0.0f, 0.0f);
                glVertex3f(0.0f, 1.0f, 0.0f);
                glColor3f (1.0f, 0.0f, 0.0f);
                glVertex3f(-1.0f, 0.0f, 0.0f);
glEnd();
```



leo.donati@unice.fr *page 24*

Fonctions du mode immédiat

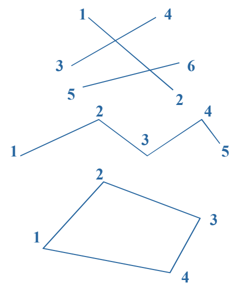
- Le bloc glBegin/glEnd définit le type de primitive à assembler
- Les fonctions openGL disponibles :
  - glVertex pour définir le sommet
  - glNormal pour définir le vecteur normal
  - glColor pour définir la couleur
  - glMaterial pour définir le matériau
  - glTexCoord pour définir les coordonnées textures

OpenGL

leo.donati@unice.fr page 25

Type de primitives

- Points
  - GL_POINTS
- Lignes
  - GL_LINES
  - GL_LINE_STRIP
  - GL_LINE_LOOP

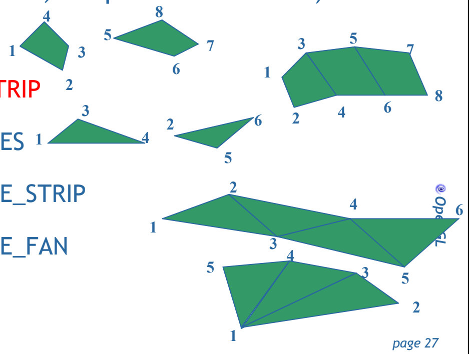


OpenGL

leo.donati@unice.fr page 26

Primitives (suite)

- Polygones (plans, simples et convexes)
  - GL_QUADS
  - GL_QUAD_STRIP
  - GL_TRIANGLES
  - GL_TRIANGLE_STRIP
  - GL_TRIANGLE_FAN



OpenGL

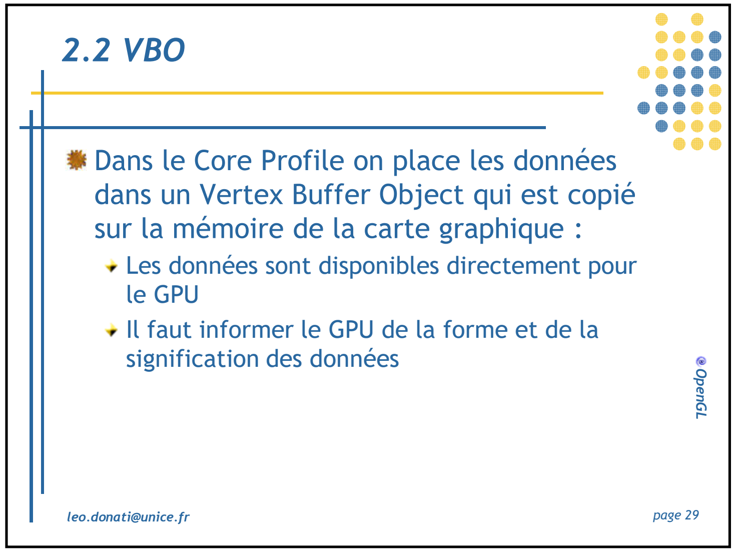
leo.donati@unice.fr page 27

Couleurs

- Dans OpenGL les couleurs sont spécifiées via une combinaison des trois couleurs primaires : R, G, B
  - OpenGL ajoute une quatrième composante appelée Alpha qui peut être utilisée pour la transparence ou d'autres effets.
- Chaque composante est exprimée par un GLfloat entre 0.0 et 1.0
  - 1.0 représente l'intensité maximale
  - 0.0 représente l'intensité nulle

OpenGL

leo.donati@unice.fr page 28

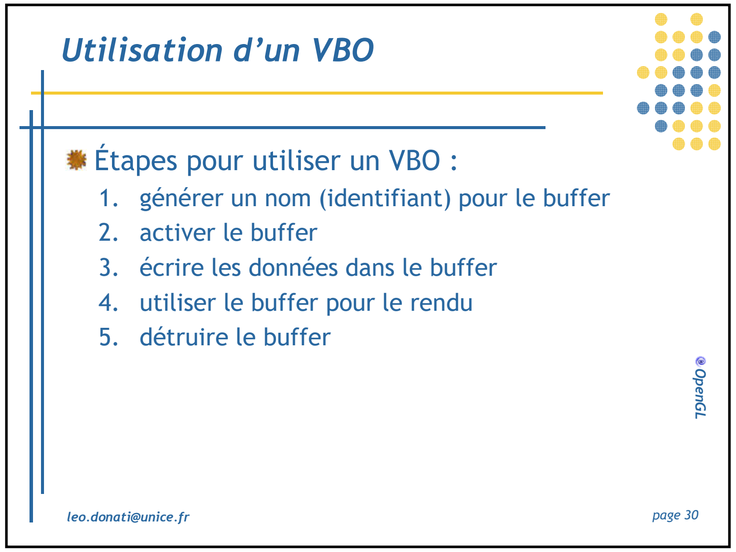


2.2 VBO

- Dans le Core Profile on place les données dans un Vertex Buffer Object qui est copié sur la mémoire de la carte graphique :
  - Les données sont disponibles directement pour le GPU
  - Il faut informer le GPU de la forme et de la signification des données

OpenGL

leo.donati@unice.fr page 29

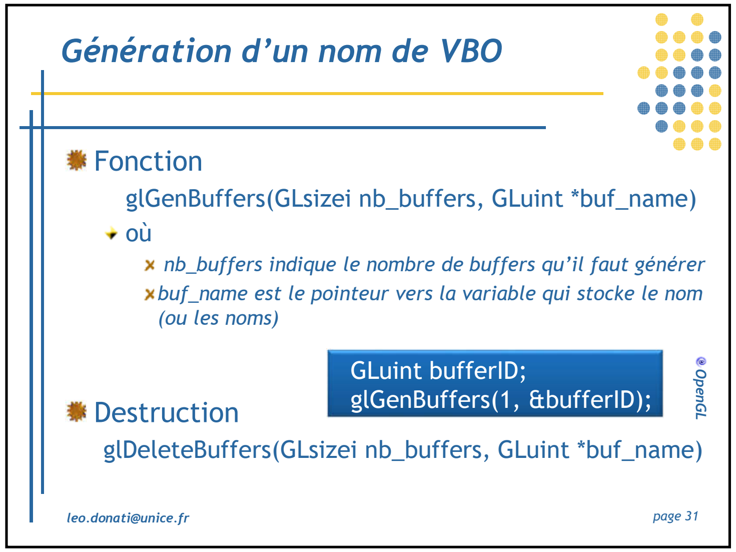


Utilisation d'un VBO

- Étapes pour utiliser un VBO :
  1. générer un nom (identifiant) pour le buffer
  2. activer le buffer
  3. écrire les données dans le buffer
  4. utiliser le buffer pour le rendu
  5. détruire le buffer

OpenGL

leo.donati@unice.fr page 30



Génération d'un nom de VBO

- Fonction

  glGenBuffers(GLsizei nb_buffers, GLuint *buf_name)

  - où
    - *nb_buffers indique le nombre de buffers qu'il faut générer*
    - *buf_name est le pointeur vers la variable qui stocke le nom (ou les noms)*

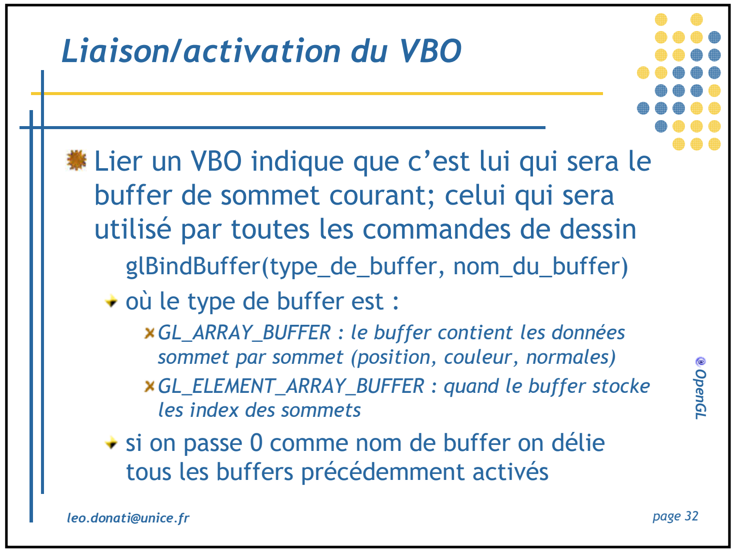
```
GLuint bufferID;
glGenBuffers(1, &bufferID);
```

- Destruction

  glDeleteBuffers(GLsizei nb_buffers, GLuint *buf_name)

OpenGL

leo.donati@unice.fr page 31



Liaison/activation du VBO

- Lier un VBO indique que c'est lui qui sera le buffer de sommet courant; celui qui sera utilisé par toutes les commandes de dessin

  glBindBuffer(type_de_buffer, nom_du_buffer)

  - où le type de buffer est :
    - *GL_ARRAY_BUFFER : le buffer contient les données sommet par sommet (position, couleur, normales)*
    - *GL_ELEMENT_ARRAY_BUFFER : quand le buffer stocke les index des sommets*
  - si on passe 0 comme nom de buffer on délie tous les buffers précédemment activés

OpenGL

leo.donati@unice.fr page 32

Remplissage du VBO

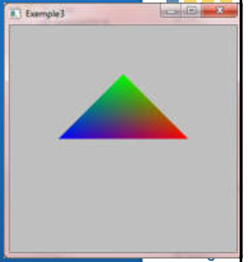
- Ecriture des données dans le VBO avec
  - glBufferData(type, taille, *data, usage)
  - où
    - *le type est le même que dans glBindBuffer*
    - *taille est la taille en octet du VBO*
    - *data est un pointeur sur les données*
    - *usage est ce qu'on veut faire du VBO (optimisation)*
      - *type d'accès*
        - *DRAW, READ, COPY*
      - *fréquence d'accès*
        - *STREAM (modifié une fois, lu peu de fois fois)*
        - *STATIC (modifié une fois, lu plusieurs fois)*
        - *DYNAMIC (souvent modifié et lu)*

OpenGL

leo.donati@unice.fr *page 33*

Exemple de VBO

```
GLfloat vertex [] = {-1.0f, -0.5f, -2.0f, //sommet 1
                      1.0f, -0.5f, -2.0f, //sommet 2
                      0.0f,  0.5f, -2.0f, //sommet 3
                      1.0f, 0.0f, 0.0f, //couleur 1
                      0.0f, 1.0f, 0.0f, //couleur 2
                      0.0f, 0.0f, 1.0f}; //couleur 3

//m_vbo est de type GLuint
glGenBuffers(1, &m_vbo);
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

glBufferData(GL_ARRAY_BUFFER,
             sizeof(GLfloat)*18,
             vertex,
             GL_STATIC_DRAW);
```



leo.donati@unice.fr *page 34*

Exemple d'extension

- Pour utiliser les VBO on a besoin de trois fonctions qui sont des extensions :
  - glGenBuffers
  - glBindBuffer
  - glBufferData

```
#include "glext.h"

PFNGLGENBUFFERSARBPROC glGenBuffers = NULL;
PFNGLBINDBUFFERPROC glBindBuffer = NULL;
PFNGLBUFFERDATAPROC glBufferData = NULL;
glGenBuffers = (PFNGLGENBUFFERSARBPROC)wglGetProcAddress("glGenBuffers");
glBindBuffer = (PFNGLBINDBUFFERPROC)wglGetProcAddress("glBindBuffer");
glBufferData = (PFNGLBUFFERDATAPROC)wglGetProcAddress("glBufferData");
```

leo.donati@unice.fr *page 35*

2.3 Index Buffer Object

- L'Index Buffer (IBO) est un tableau d'entiers qui stocke les indices des vertex à utiliser pour construire les primitives :
  - Au lieu de lire les sommets séquentiellement dans le VBO, on peut utiliser le IBO pour donner l'ordre de lecture des vertex
  - Ainsi le même vertex va être utilisé dans la construction de plusieurs primitives

OpenGL

leo.donati@unice.fr *page 36*

Utilisation des IBO

- La procédure est semblable à celle des VBO
  - Création d'un identifiant
  - Liaison de l'IBO
  - Remplissage de l'IBO
- Si on utilise un IBO on utilise la commande de **rendu indexé**

  glDrawElements(   )

  à la place de

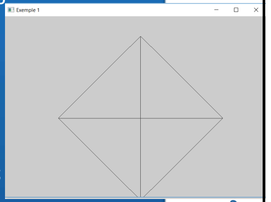
  glDrawArrays(   )

OpenGL

leo.donati@unice.fr page 37

Exemple de IBO

```
GLfloat tetraedre [] = {1.0f, 0.0f, 0.0f, //sommet 0
                        -1.0f, 0.0f, 0.0f, //sommet 1
                        0.0f,  0.0f, 1.0f, //sommet 2
                        0.0f, 2.0f, 0.0f}; //sommet 3
glGenBuffers(1, &m_vbo);
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(GLfloat)*12,
             tetraedre,
             GL_STATIC_DRAW);

GLint indices [] = { 0, 1, 3, 0, 1, 2, 0, 3, 2, 1, 2, 3};
glGenBuffers(1, &m_ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(GLint)*12,
             indices,
             GL_STATIC_DRAW);
```



OpenGL

leo.donati@unice.fr page 38

2.4 Vertex Array Object (VAO)

- Les VAO permettent de rassembler en un seul «objet» OpenGL :
  - Plusieurs VBO
  - Plusieurs Index buffers
  - Les liens entre les attributs des programmes et les différents VBO
- Ainsi si le Vao est bien préparé, lors du rendu
  - On lie (bind) le VAO
  - On lance la commadne de rendu
  - On libère le VAO

OpenGL

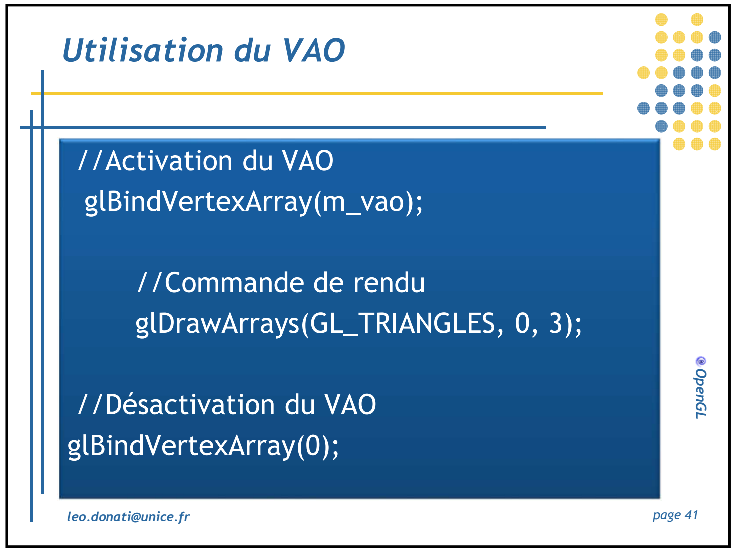
leo.donati@unice.fr page 39

Définition du VAO

```
glGenBuffers(1, &m_vbo);
 glGenVertexArrays(1, &m_vao);
 glBindVertexArray(m_vao);
 glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_data), vertex_data,
                                GL_STATIC_DRAW);

  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
                         BUFFER_OFFSET(9*sizeof(GLfloat)));

  glEnableVertexAttribArray(0);
  glEnableVertexAttribArray(1);

 glBindBuffer(GL_ARRAY_BUFFER, 0);
 glBindVertexArray(0);
```

OpenGL

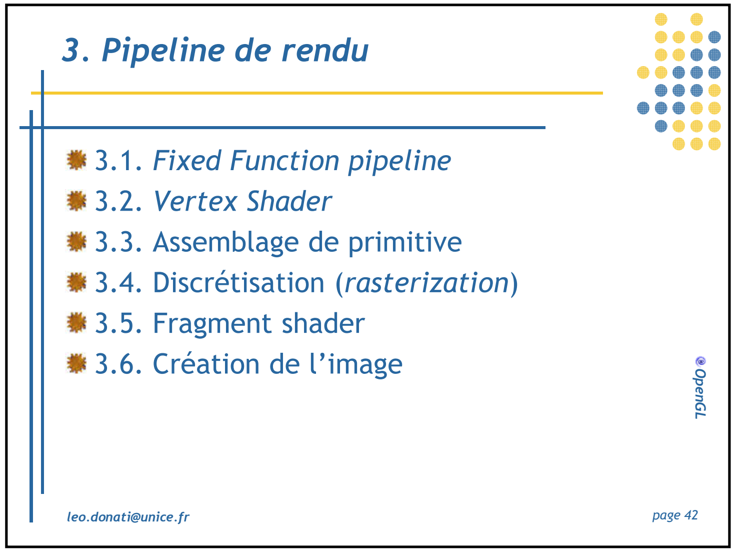
leo.donati@unice.fr page 40



Utilisation du VAO

```
 //Activation du VAO
  glBindVertexArray(m_vao);

        //Commande de rendu
        glDrawArrays(GL_TRIANGLES, 0, 3);

 //Désactivation du VAO
glBindVertexArray(0);
```

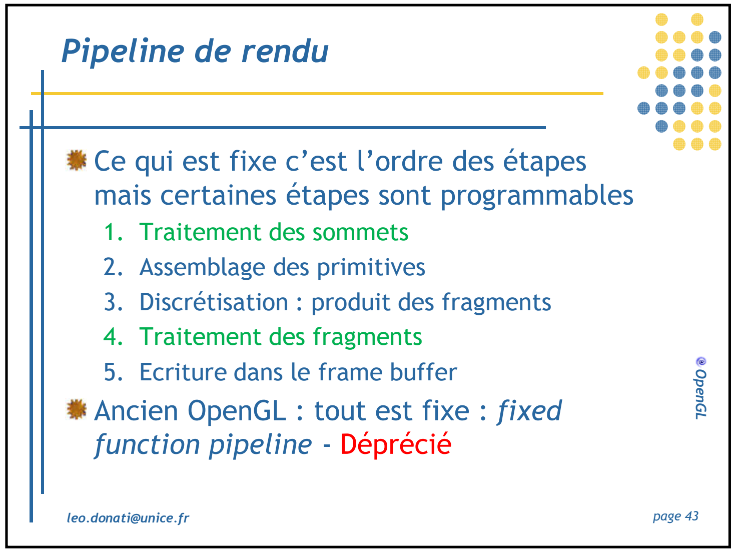
OpenGL



3. Pipeline de rendu

- 3.1. *Fixed Function pipeline*
- 3.2. *Vertex Shader*
- 3.3. Assemblage de primitive
- 3.4. Discrétisation (*rasterization*)
- 3.5. Fragment shader
- 3.6. Création de l'image

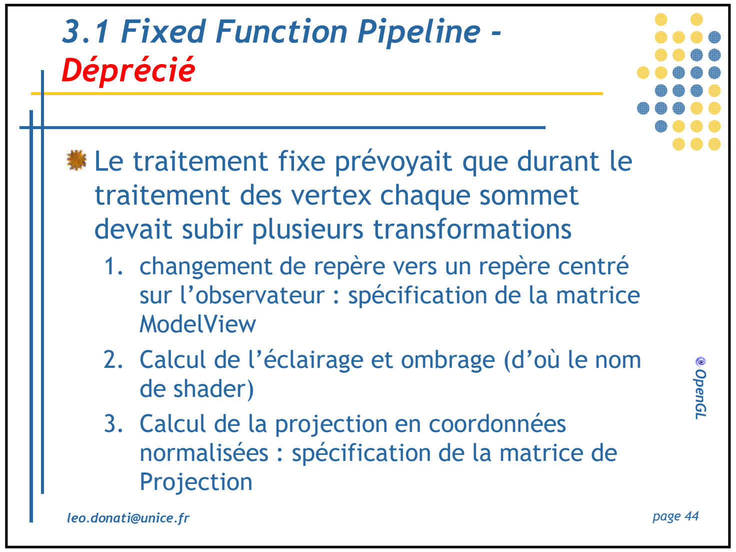
OpenGL



Pipeline de rendu

- Ce qui est fixe c'est l'ordre des étapes mais certaines étapes sont programmables
  1. Traitement des sommets
  2. Assemblage des primitives
  3. Discrétisation : produit des fragments
  4. Traitement des fragments
  5. Ecriture dans le frame buffer
- Ancien OpenGL : tout est fixe : *fixed function pipeline* - Déprécié

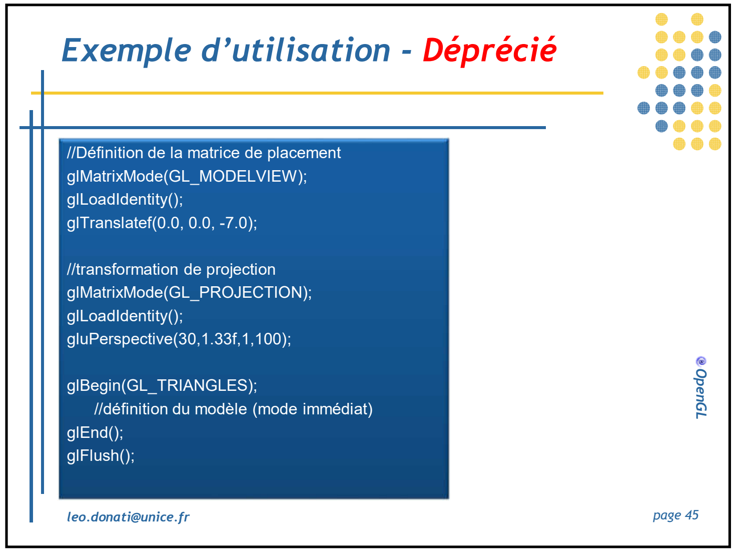
OpenGL



3.1 Fixed Function Pipeline - Déprécié

- Le traitement fixe prévoyait que durant le traitement des vertex chaque sommet devait subir plusieurs transformations
  1. changement de repère vers un repère centré sur l'observateur : spécification de la matrice ModelView
  2. Calcul de l'éclairage et ombrage (d'où le nom de shader)
  3. Calcul de la projection en coordonnées normalisées : spécification de la matrice de Projection

OpenGL

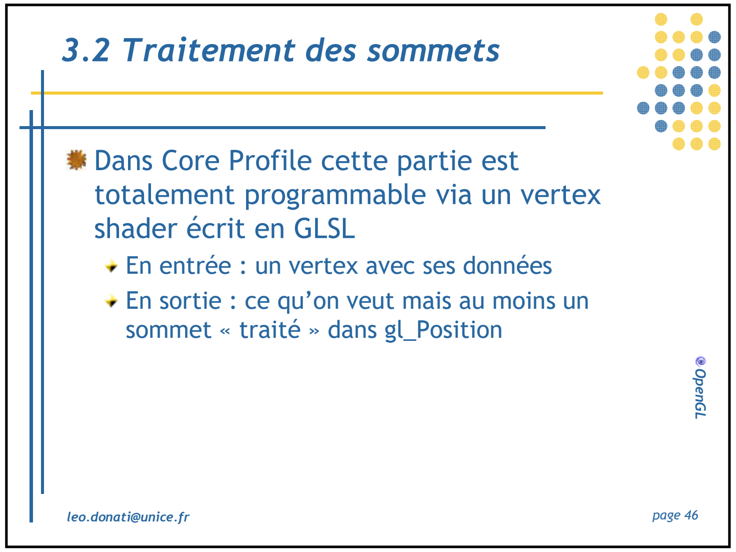


Exemple d'utilisation - Déprécié

```
//Définition de la matrice de placement
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -7.0);

//transformation de projection
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(30,1.33f,1,100);

glBegin(GL_TRIANGLES);
    //définition du modèle (mode immédiat)
glEnd();
glFlush();
```

© OpenGL

leo.donati@unice.fr page 45

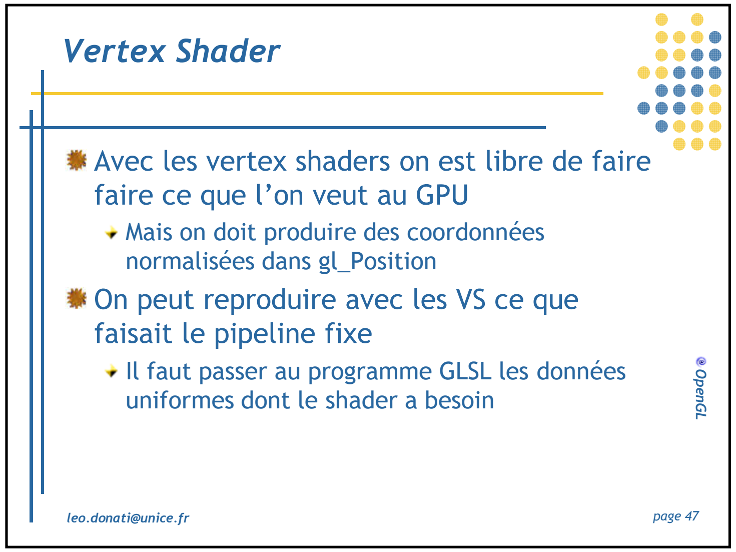


3.2 Traitement des sommets

- Dans Core Profile cette partie est totalement programmable via un vertex shader écrit en GLSL
  - En entrée : un vertex avec ses données
  - En sortie : ce qu'on veut mais au moins un sommet « traité » dans gl_Position

© OpenGL

leo.donati@unice.fr page 46

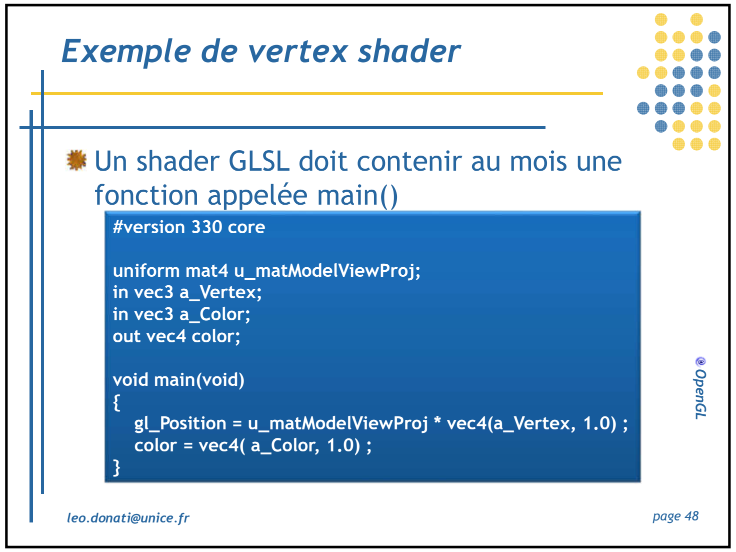


Vertex Shader

- Avec les vertex shaders on est libre de faire faire ce que l'on veut au GPU
  - Mais on doit produire des coordonnées normalisées dans gl_Position
- On peut reproduire avec les VS ce que faisait le pipeline fixe
  - Il faut passer au programme GLSL les données uniformes dont le shader a besoin

© OpenGL

leo.donati@unice.fr page 47



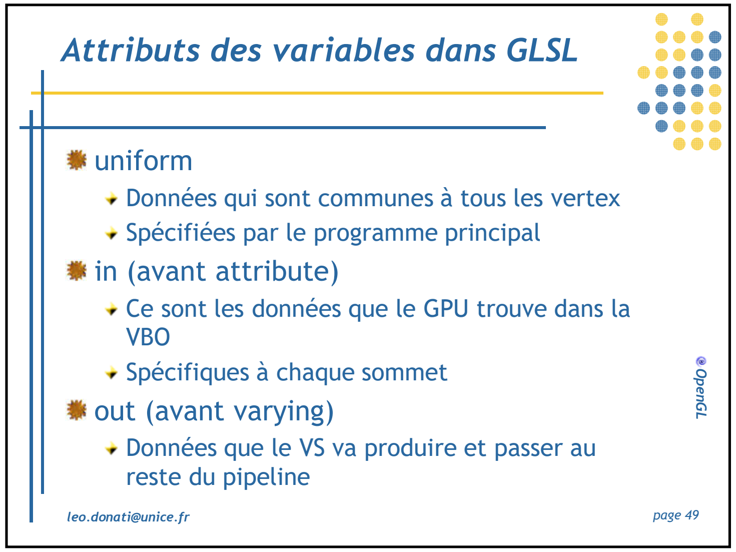
Exemple de vertex shader

- Un shader GLSL doit contenir au mois une fonction appelée main()

```
#version 330 core

uniform mat4 u_matModelViewProj;
in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;

void main(void)
{
   gl_Position = u_matModelViewProj * vec4(a_Vertex, 1.0) ;
   color = vec4( a_Color, 1.0) ;
}
```

© OpenGL

leo.donati@unice.fr page 48

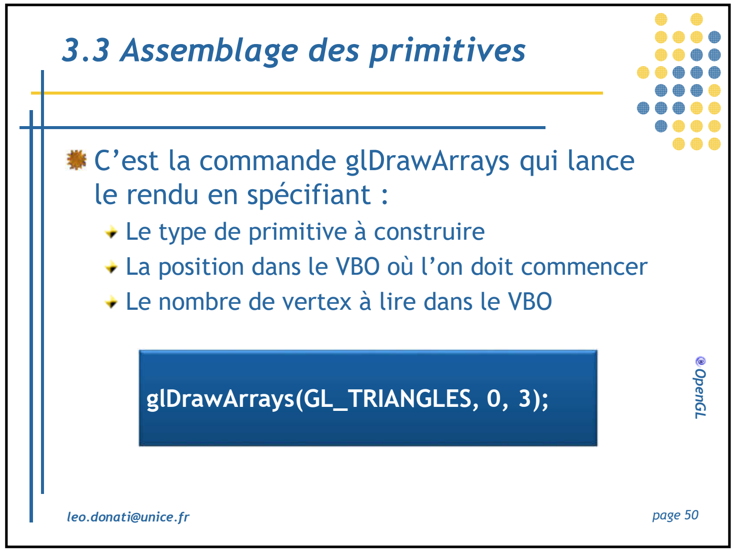


Attributs des variables dans GLSL

- uniform
  - Données qui sont communes à tous les vertex
  - Spécifiées par le programme principal
- in (avant attribute)
  - Ce sont les données que le GPU trouve dans la VBO
  - Spécifiques à chaque sommet
- out (avant varying)
  - Données que le VS va produire et passer au reste du pipeline

© OpenGL

leo.donati@unice.fr page 49



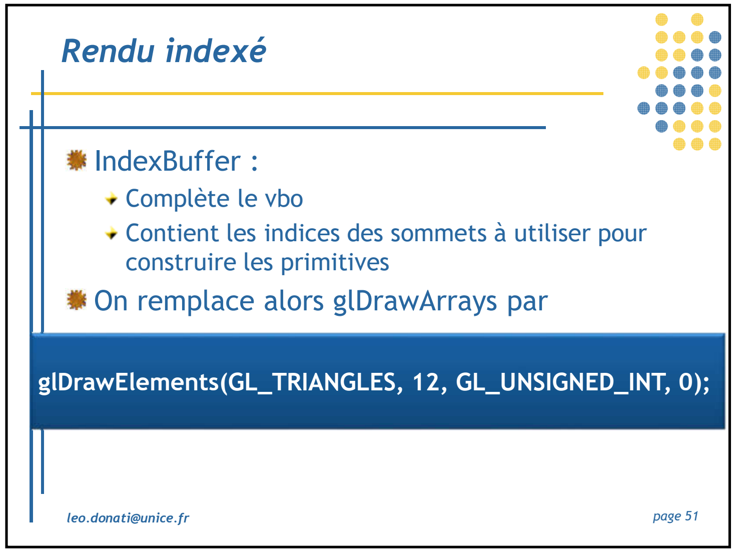
3.3 Assemblage des primitives

- C'est la commande glDrawArrays qui lance le rendu en spécifiant :
  - Le type de primitive à construire
  - La position dans le VBO où l'on doit commencer
  - Le nombre de vertex à lire dans le VBO

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

© OpenGL

leo.donati@unice.fr page 50

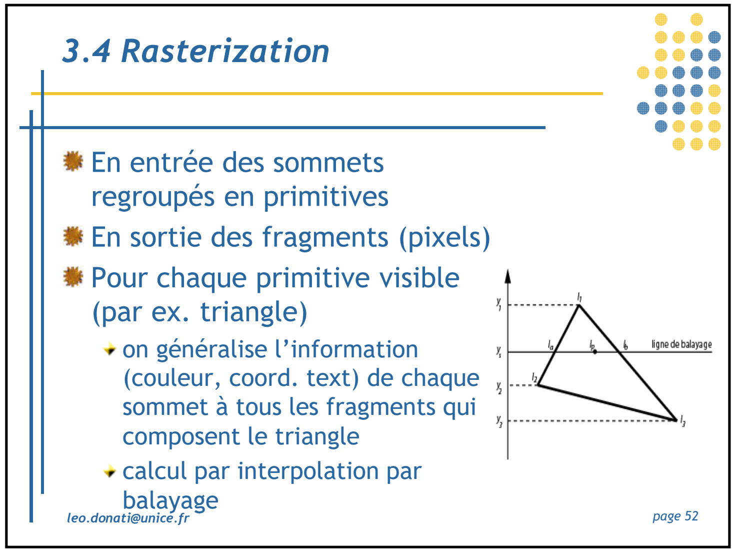


Rendu indexé

- IndexBuffer :
  - Complète le vbo
  - Contient les indices des sommets à utiliser pour construire les primitives
- On remplace alors glDrawArrays par

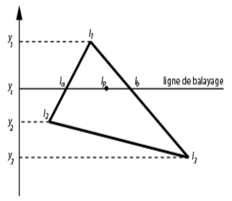
```
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);
```

leo.donati@unice.fr page 51



3.4 Rasterization

- En entrée des sommets regroupés en primitives
- En sortie des fragments (pixels)
- Pour chaque primitive visible (par ex. triangle)
  - on généralise l'information (couleur, coord. text) de chaque sommet à tous les fragments qui composent le triangle
  - calcul par interpolation par balayage



leo.donati@unice.fr page 52

Résultat de la discrétisation

- Grâce à l'interpolation, l'information qui était présente sur les sommets va être généralisée à chaque fragment
- On obtient un très grand nombre de fragments :
  - Pour une face triangulaire définie par trois vertex, on peut avoir des milliers de fragments

OpenGL

leo.donati@unice.fr *page 53*

3.5 Fragment shader

- Le fragment shader est un programme exécuté par le GPU qui :
  - reçoit les fragments issus de la rasterization
  - est responsable de la production de la couleur finale du fragment dans **fragColor**
  - Cela peut passer par
    - *application des textures*
    - *calcul de la couleur pixel par pixel*
    - *calcul du brouillard*
    - *calcul de l'éclairage pixel par pixel*
    - *Autre …*

OpenGL

leo.donati@unice.fr *page 54*

Exemple de fragment shader

- Fragment shader minimal qui ne fait aucun traitement sur la couleur des pixels

```
in vec4 color;
out vec4 fragColor

void main(void)
{
    fragColor = color ;
}
```

OpenGL

leo.donati@unice.fr *page 55*

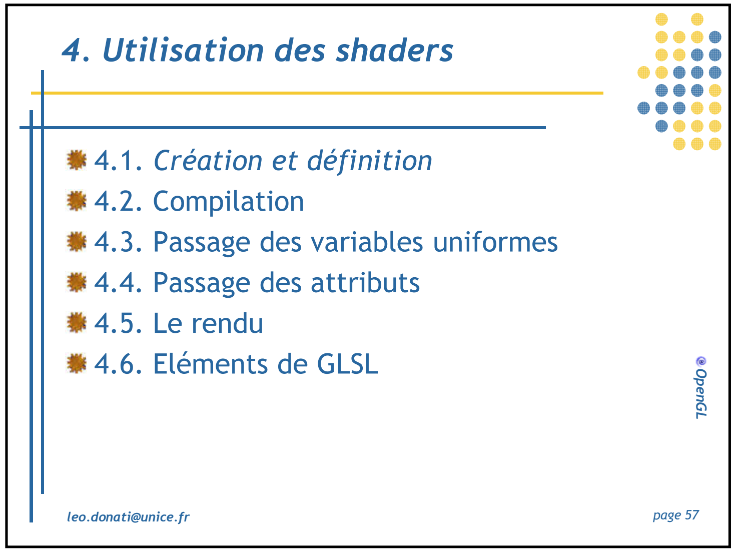
3.6 Création de l'image

- Dans le buffer d'image OpenGL les fragments ont des coordonnées normalisées qu'il faut ensuite convertir en pixels à afficher.
  - Il faut donner la transformation vers les coordonnées de la fenêtre :

glViewport(posX, posY, largeur, hauteur)

OpenGL

leo.donati@unice.fr *page 56*

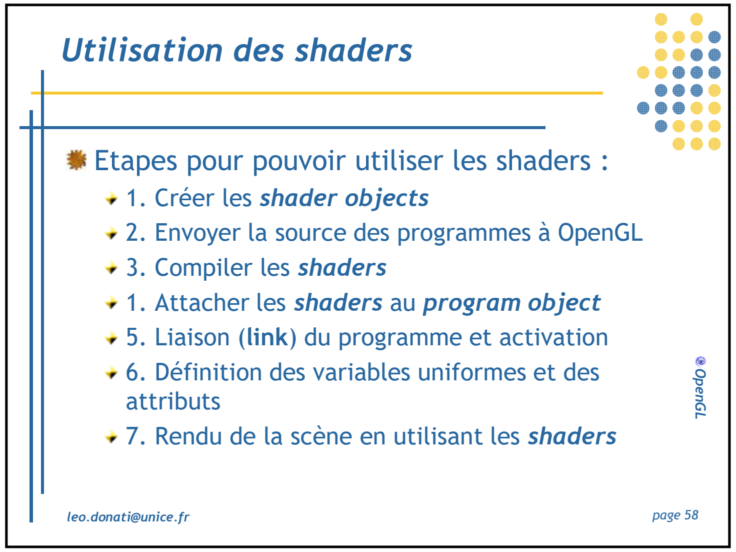


4. Utilisation des shaders

- 4.1. *Création et définition*
- 4.2. Compilation
- 4.3. Passage des variables uniformes
- 4.4. Passage des attributs
- 4.5. Le rendu
- 4.6. Eléments de GLSL

OpenGL

leo.donati@unice.fr page 57

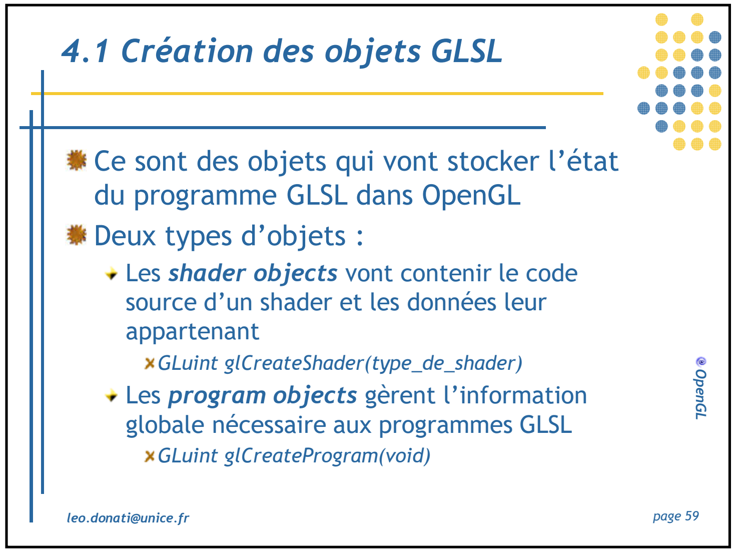


Utilisation des shaders

- Etapes pour pouvoir utiliser les shaders :
  - 1. Créer les ***shader objects***
  - 2. Envoyer la source des programmes à OpenGL
  - 3. Compiler les ***shaders***
  - 1. Attacher les ***shaders*** au ***program object***
  - 5. Liaison (**link**) du programme et activation
  - 6. Définition des variables uniformes et des attributs
  - 7. Rendu de la scène en utilisant les ***shaders***

OpenGL

leo.donati@unice.fr page 58

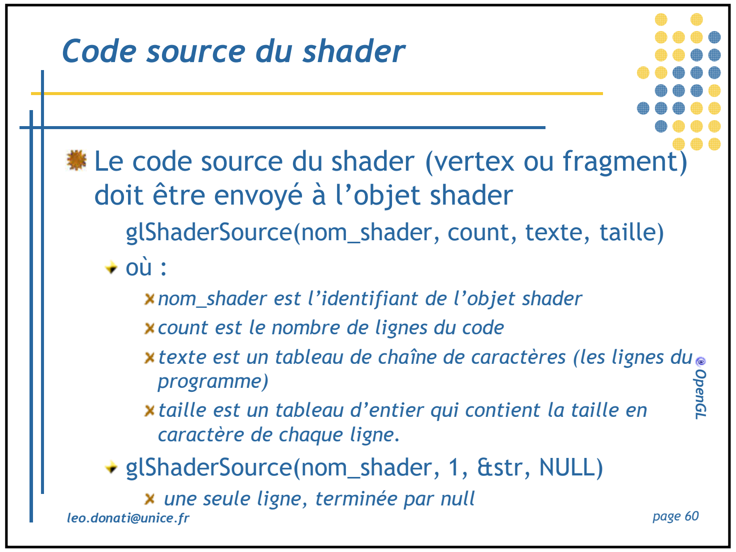


4.1 Création des objets GLSL

- Ce sont des objets qui vont stocker l'état du programme GLSL dans OpenGL
- Deux types d'objets :
  - Les ***shader objects*** vont contenir le code source d'un shader et les données leur appartenant
    - *GLuint glCreateShader(type_de_shader)*
  - Les ***program objects*** gèrent l'information globale nécessaire aux programmes GLSL
    - *GLuint glCreateProgram(void)*

OpenGL

leo.donati@unice.fr page 59



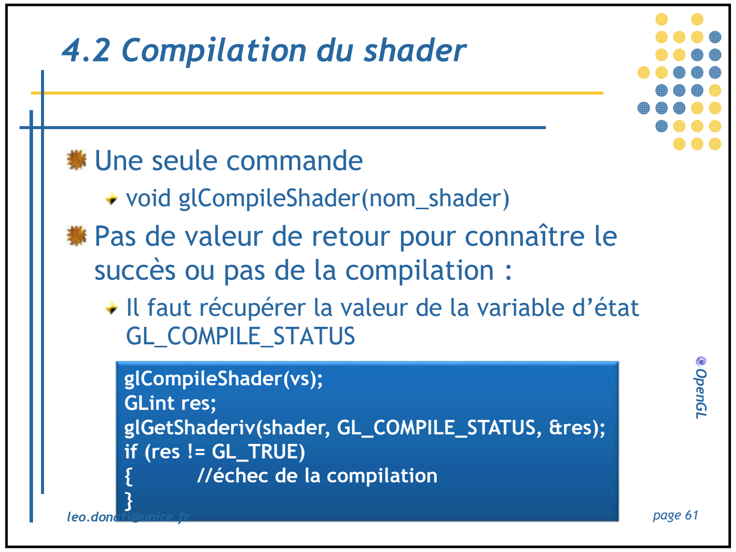
Code source du shader

- Le code source du shader (vertex ou fragment) doit être envoyé à l'objet shader

  glShaderSource(nom_shader, count, texte, taille)
  - où :
    - *nom_shader est l'identifiant de l'objet shader*
    - *count est le nombre de lignes du code*
    - *texte est un tableau de chaîne de caractères (les lignes du programme)*
    - *taille est un tableau d'entier qui contient la taille en caractère de chaque ligne.*
  - glShaderSource(nom_shader, 1, &str, NULL)
    - *une seule ligne, terminée par null*

OpenGL

leo.donati@unice.fr page 60



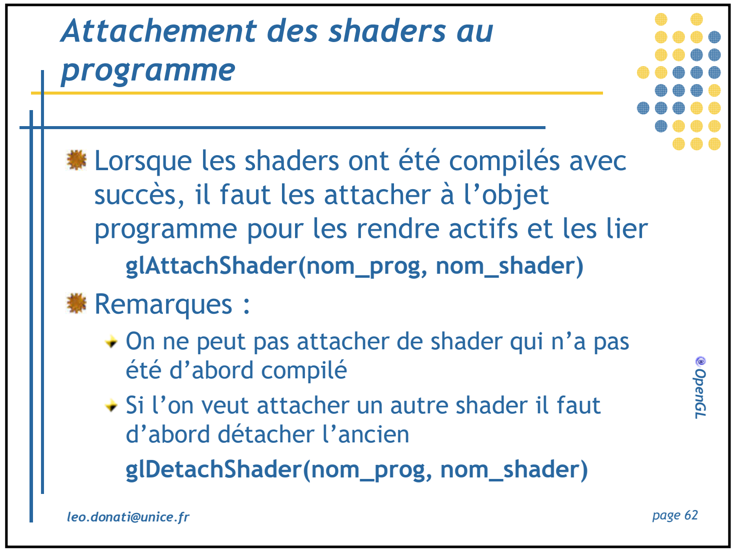
4.2 Compilation du shader

- Une seule commande
  - void glCompileShader(nom_shader)
- Pas de valeur de retour pour connaître le succès ou pas de la compilation :
  - Il faut récupérer la valeur de la variable d'état GL_COMPILE_STATUS

```
glCompileShader(vs);
GLint res;
glGetShaderiv(shader, GL_COMPILE_STATUS, &res);
if (res != GL_TRUE)
{          //échec de la compilation
}
```

© OpenGL

leo.donati@unice.fr page 61

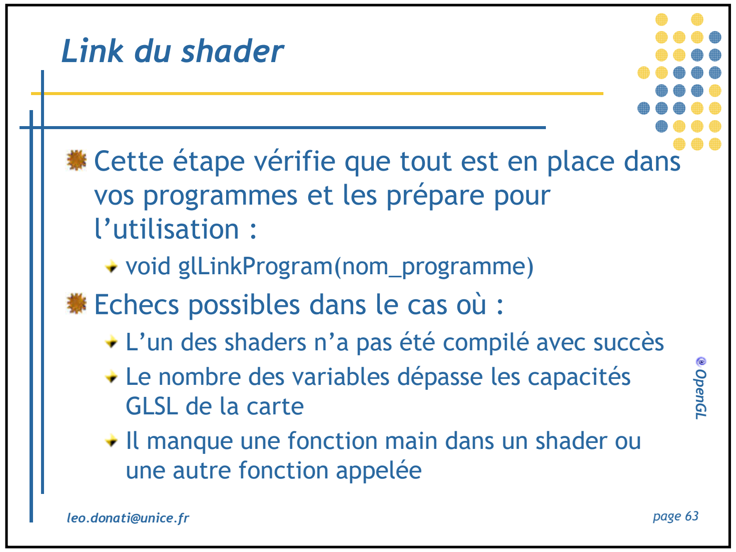


Attachement des shaders au programme

- Lorsque les shaders ont été compilés avec succès, il faut les attacher à l'objet programme pour les rendre actifs et les lier
  glAttachShader(nom_prog, nom_shader)
- Remarques :
  - On ne peut pas attacher de shader qui n'a pas été d'abord compilé
  - Si l'on veut attacher un autre shader il faut d'abord détacher l'ancien
  glDetachShader(nom_prog, nom_shader)

© OpenGL

leo.donati@unice.fr page 62

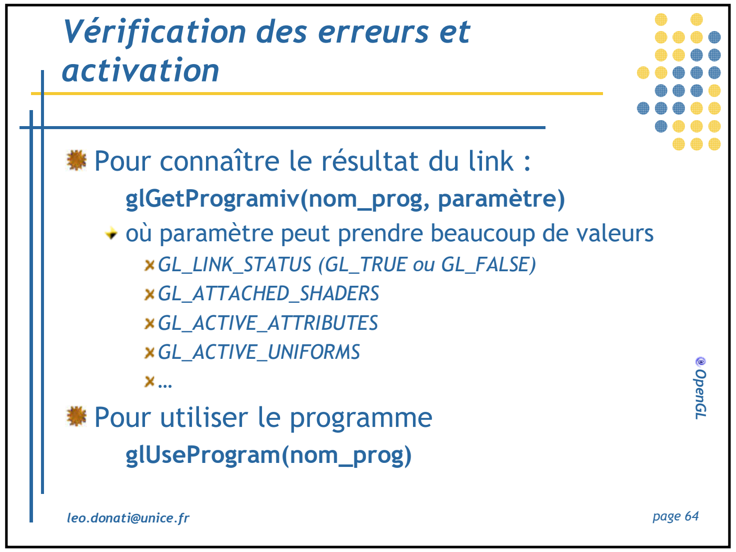


Link du shader

- Cette étape vérifie que tout est en place dans vos programmes et les prépare pour l'utilisation :
  - void glLinkProgram(nom_programme)
- Echecs possibles dans le cas où :
  - L'un des shaders n'a pas été compilé avec succès
  - Le nombre des variables dépasse les capacités GLSL de la carte
  - Il manque une fonction main dans un shader ou une autre fonction appelée

© OpenGL

leo.donati@unice.fr page 63

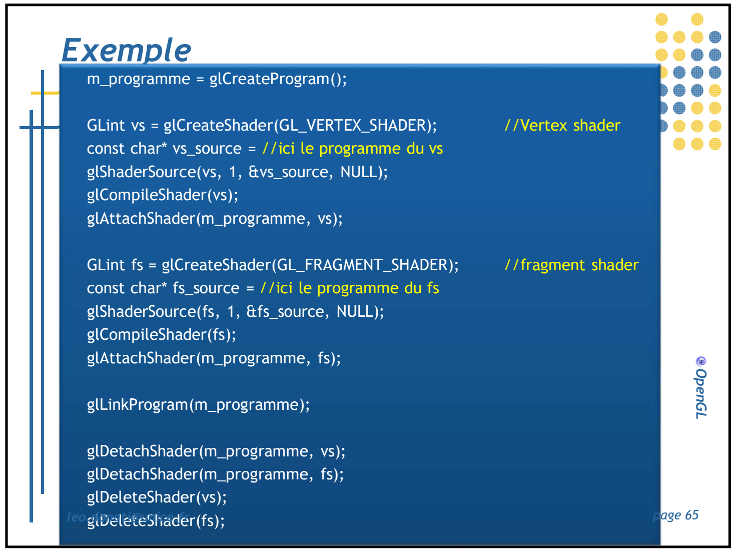


Vérification des erreurs et activation

- Pour connaître le résultat du link :
  glGetProgramiv(nom_prog, paramètre)
  - où paramètre peut prendre beaucoup de valeurs
    - *GL_LINK_STATUS (GL_TRUE ou GL_FALSE)*
    - *GL_ATTACHED_SHADERS*
    - *GL_ACTIVE_ATTRIBUTES*
    - *GL_ACTIVE_UNIFORMS*
    - …
- Pour utiliser le programme
  glUseProgram(nom_prog)

© OpenGL

leo.donati@unice.fr page 64

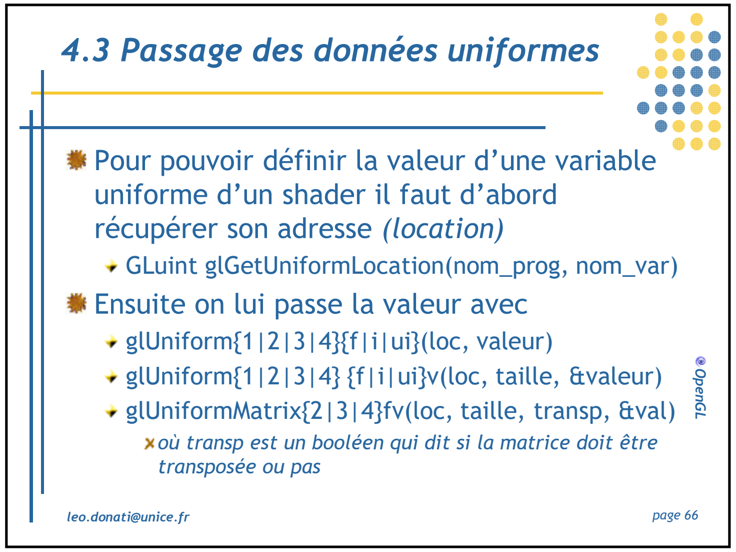


Exemple

```
m_programme = glCreateProgram();

GLint vs = glCreateShader(GL_VERTEX_SHADER);          //Vertex shader
const char* vs_source = //ici le programme du vs
glShaderSource(vs, 1, &vs_source, NULL);
glCompileShader(vs);
glAttachShader(m_programme, vs);

GLint fs = glCreateShader(GL_FRAGMENT_SHADER);        //fragment shader
const char* fs_source = //ici le programme du fs
glShaderSource(fs, 1, &fs_source, NULL);
glCompileShader(fs);
glAttachShader(m_programme, fs);

glLinkProgram(m_programme);

glDetachShader(m_programme, vs);
glDetachShader(m_programme, fs);
glDeleteShader(vs);
glDeleteShader(fs);
```

OpenGL

page 65

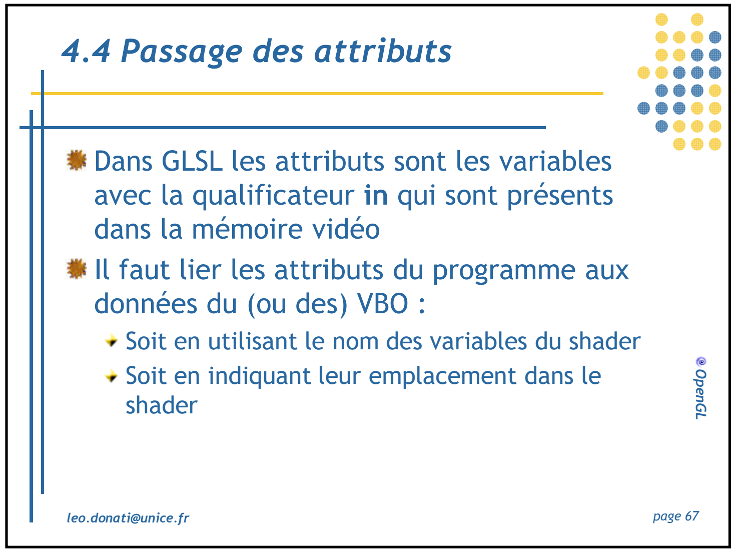


4.3 Passage des données uniformes

- Pour pouvoir définir la valeur d'une variable uniforme d'un shader il faut d'abord récupérer son adresse *(location)*
  - GLuint glGetUniformLocation(nom_prog, nom_var)
- Ensuite on lui passe la valeur avec
  - glUniform{1|2|3|4}{f|i|ui}(loc, valeur)
  - glUniform{1|2|3|4} {f|i|ui}v(loc, taille, &valeur)
  - glUniformMatrix{2|3|4}fv(loc, taille, transp, &val)
    - *où transp est un booléen qui dit si la matrice doit être transposée ou pas*

OpenGL

leo.donati@unice.fr page 66

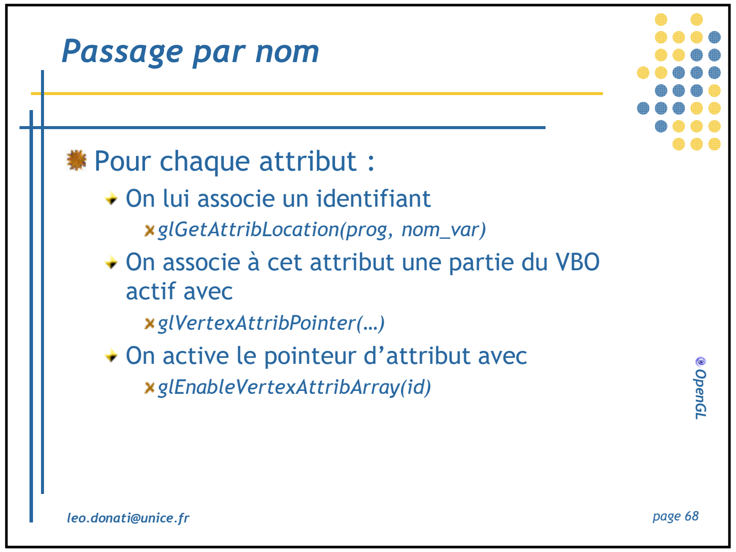


4.4 Passage des attributs

- Dans GLSL les attributs sont les variables avec la qualificateur **in** qui sont présents dans la mémoire vidéo
- Il faut lier les attributs du programme aux données du (ou des) VBO :
  - Soit en utilisant le nom des variables du shader
  - Soit en indiquant leur emplacement dans le shader

OpenGL

leo.donati@unice.fr page 67



Passage par nom

- Pour chaque attribut :
  - On lui associe un identifiant
    - *glGetAttribLocation(prog, nom_var)*
  - On associe à cet attribut une partie du VBO actif avec
    - *glVertexAttribPointer(…)*
  - On active le pointeur d'attribut avec
    - *glEnableVertexAttribArray(id)*

OpenGL

leo.donati@unice.fr page 68

Passage par emplacement

- Il faut dans le ***shader*** associer un emplacement à chaque attribut

```
layout(location=0) in vec3 pos;
layout(location=1) in vec3 col;
```

- On utilise l'emplacement dans
  - *glVertexAttribPointer*
  - *glEnableVertexAttrib*
- Plus commode avec les VAO

OpenGL

leo.donati@unice.fr page 69

glVertexAttribPointer

- Ses arguments sont :
  - *L'identifiant ou l'indice de l'attribut que l'on veut fixer*
  - *La taille des composants par sommet (entre 2 et 4)*
  - *Le type : GL_FLOAT, GL_DOUBLE, GL_BYTE, etc..*
  - *Un booléen normalized indique si l'on veut que les floats soient normalisés entre -1 et 1 ou entre 0 et 1*
  - *stride : indique le décalage de l'attribut entre sommets consécutifs (0 par défaut)*
  - *Pointeur sur les données (en fait un OFFSET du VBO)*

OpenGL

leo.donati@unice.fr page 70

4.5 Le rendu

- Pour donner la commande de rendu qui lance le moteur OpenGL il faut
  - Les données
    - *Soit un VBO*
    - *Soit un couple VBO et IBO*
    - *Soit un VAO qui encapsule tous les détails*
  - Un glProgram compilé et linké avec succès
- Il faut tout mettre ensemble et lancer la commande de rendu qui lance le GPU

OpenGL

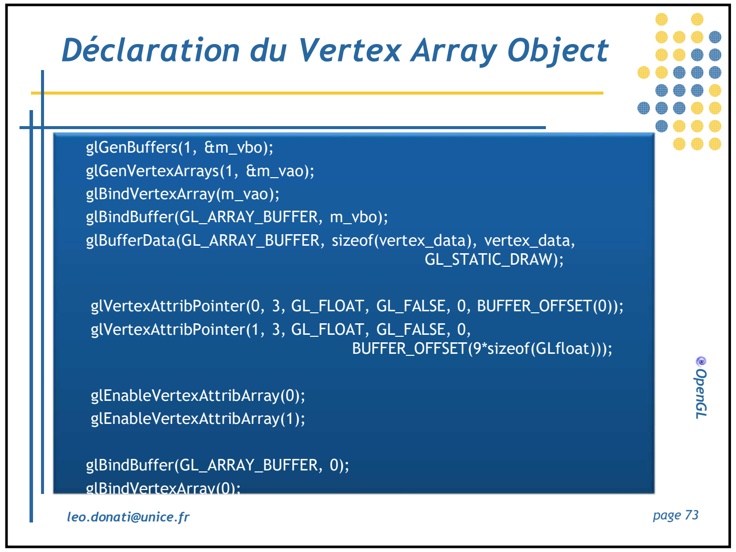
leo.donati@unice.fr page 71

Programme de rendu (sans VAO)

```
glUseProgram(m_programme);

GLuint position = getAttribLocation(m_programme, "a_Vertex");
GLuint couleur = getAttribLocation(m_programme, "a_Color");
GLuint matrice = getUnifromLocation(m_programme, "u_matModelView");

//On passe la variable uniforme au programme
glUniformMatrix4fv(matrice, 1, GL_FALSE, m_matrix);

glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
glEnableVertexAttribArray(position);
glEnableVertexAttribArray(couleur);

//Lien entre le VBO et les attributs
glVertexAttribPointer(position, 3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glVertexAttribPointer(couleur,3,  GL_FLOAT, 0, BUFFER_OFFSET(9*sizeof(GL_FLOAT)));

//Commande de dessin
glDrawArrays(GL_TRIANGLES, 0, 3);

glDisableVertexAttribArray(position);
glDisableVertexAttribArray(couleur);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

OpenGL

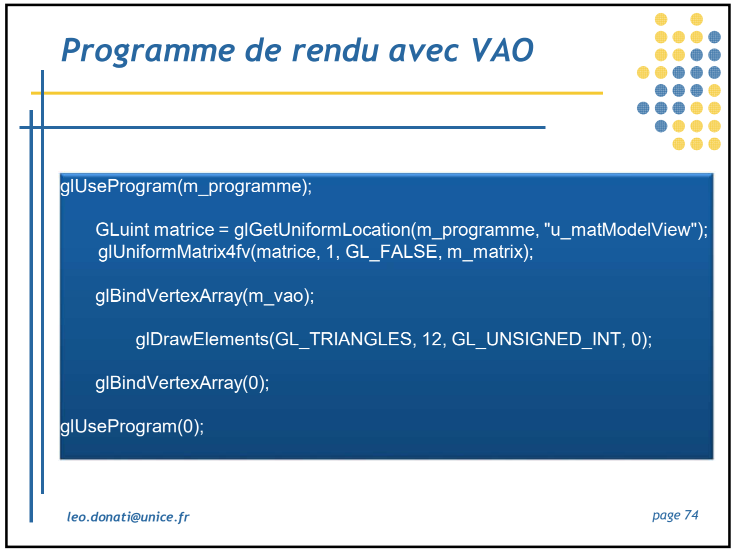
leo.donati@unice.fr page 72



Déclaration du Vertex Array Object

```
glGenBuffers(1, &m_vbo);
glGenVertexArrays(1, &m_vao);
glBindVertexArray(m_vao);
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_data), vertex_data,
                                          GL_STATIC_DRAW);

 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
                               BUFFER_OFFSET(9*sizeof(GLfloat)));

 glEnableVertexAttribArray(0);
 glEnableVertexAttribArray(1);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

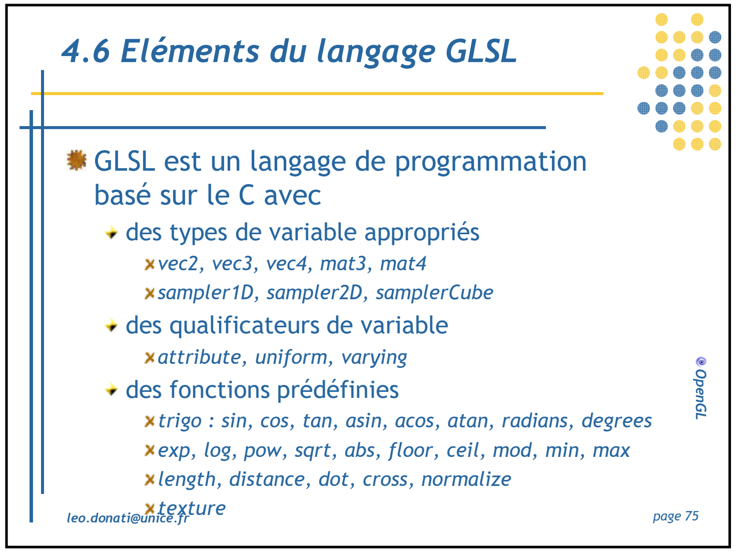
leo.donati@unice.fr page 73



Programme de rendu avec VAO

```
glUseProgram(m_programme);

    GLuint matrice = glGetUniformLocation(m_programme, "u_matModelView");
    glUniformMatrix4fv(matrice, 1, GL_FALSE, m_matrix);

    glBindVertexArray(m_vao);

        glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);

    glBindVertexArray(0);

glUseProgram(0);
```

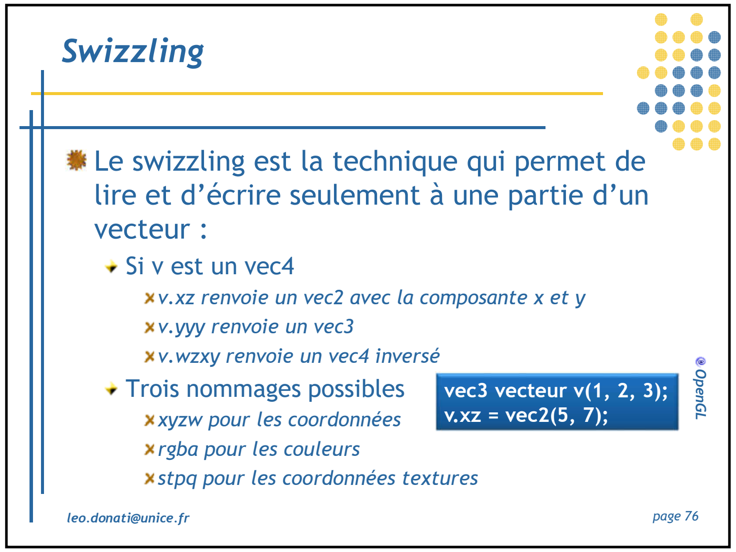
leo.donati@unice.fr page 74



4.6 Eléments du langage GLSL

- GLSL est un langage de programmation basé sur le C avec
  - des types de variable appropriés
    - *vec2, vec3, vec4, mat3, mat4*
    - *sampler1D, sampler2D, samplerCube*
  - des qualificateurs de variable
    - *attribute, uniform, varying*
  - des fonctions prédéfinies
    - *trigo : sin, cos, tan, asin, acos, atan, radians, degrees*
    - *exp, log, pow, sqrt, abs, floor, ceil, mod, min, max*
    - *length, distance, dot, cross, normalize*
    - *texture*

leo.donati@unice.fr page 75

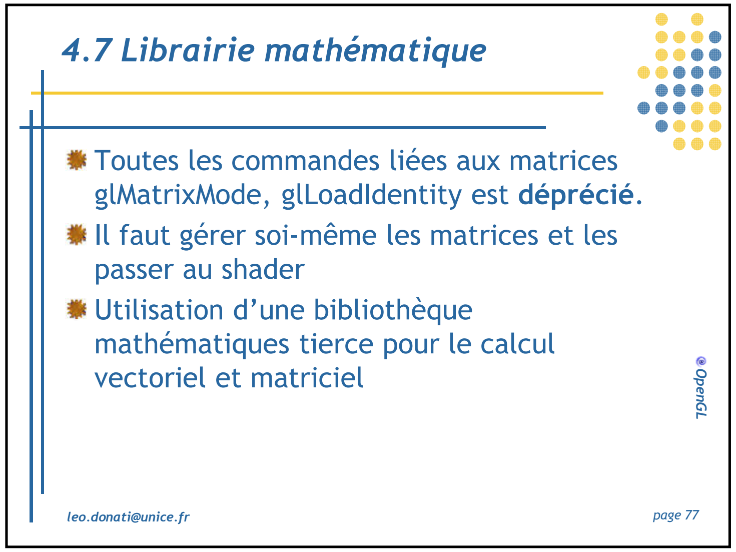


Swizzling

- Le swizzling est la technique qui permet de lire et d'écrire seulement à une partie d'un vecteur :
  - Si v est un vec4
    - *v.xz renvoie un vec2 avec la composante x et y*
    - *v.yyy renvoie un vec3*
    - *v.wzxy renvoie un vec4 inversé*
  - Trois nommages possibles
    - *xyzw pour les coordonnées*
    - *rgba pour les couleurs*
    - *stpq pour les coordonnées textures*

```
vec3 vecteur v(1, 2, 3);
v.xz = vec2(5, 7);
```

leo.donati@unice.fr page 76

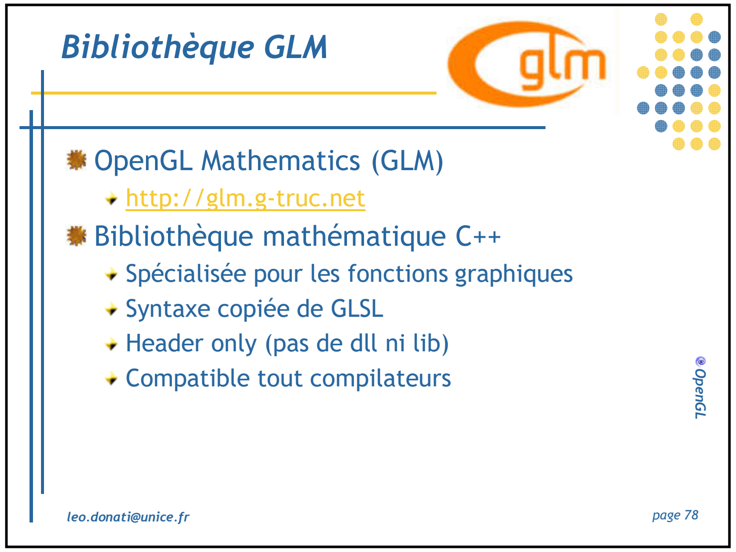


4.7 Librairie mathématique

- Toutes les commandes liées aux matrices glMatrixMode, glLoadIdentity est **déprécié**.
- Il faut gérer soi-même les matrices et les passer au shader
- Utilisation d'une bibliothèque mathématiques tierce pour le calcul vectoriel et matriciel

OpenGL

leo.donati@unice.fr *page 77*

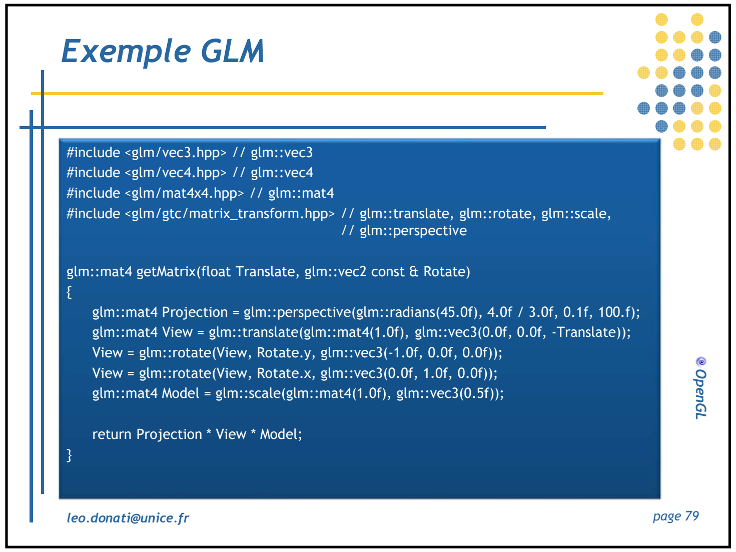


Bibliothèque GLM

- OpenGL Mathematics (GLM)
  - http://glm.g-truc.net
- Bibliothèque mathématique C++
  - Spécialisée pour les fonctions graphiques
  - Syntaxe copiée de GLSL
  - Header only (pas de dll ni lib)
  - Compatible tout compilateurs

OpenGL

leo.donati@unice.fr *page 78*

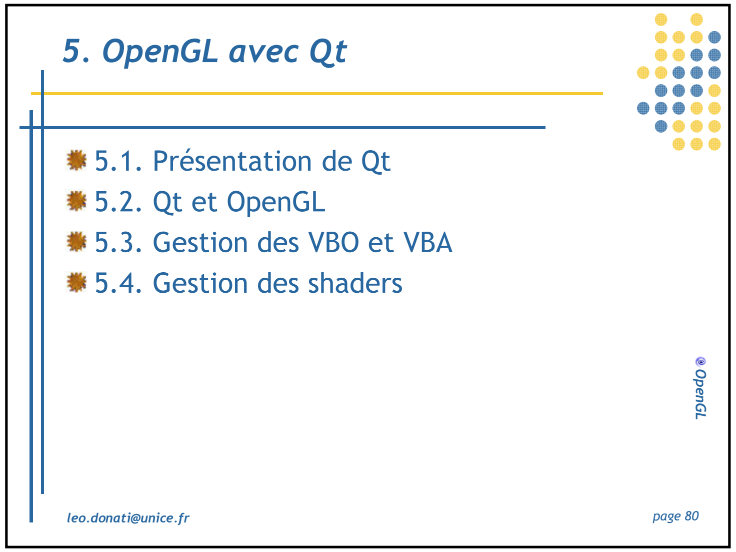


Exemple GLM

```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale,
                                        // glm::perspective

glm::mat4 getMatrix(float Translate, glm::vec2 const & Rotate)
{
    glm::mat4 Projection = glm::perspective(glm::radians(45.0f), 4.0f / 3.0f, 0.1f, 100.f);
    glm::mat4 View = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -Translate));
    View = glm::rotate(View, Rotate.y, glm::vec3(-1.0f, 0.0f, 0.0f));
    View = glm::rotate(View, Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));

    return Projection * View * Model;
}
```

OpenGL

leo.donati@unice.fr *page 79*



5. OpenGL avec Qt

- 5.1. Présentation de Qt
- 5.2. Qt et OpenGL
- 5.3. Gestion des VBO et VBA
- 5.4. Gestion des shaders

OpenGL

leo.donati@unice.fr *page 80*

5.1. Présentation de Qt

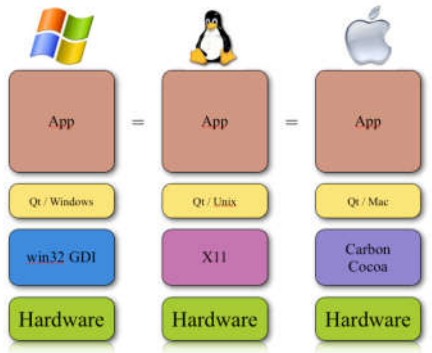
- Qt (*prononcer* ***cute***) est une bibliothèque logicielle développée en C++ qui offre des composants d'interface graphique (widgets)
- Qt est portable par simple recompilation du code source sur
  - Unix (dont linux) avec X Window System
  - Mac Os X
  - Microsoft Windows

© OpenGL

leo.donati@unice.fr

page 81

Portabilité

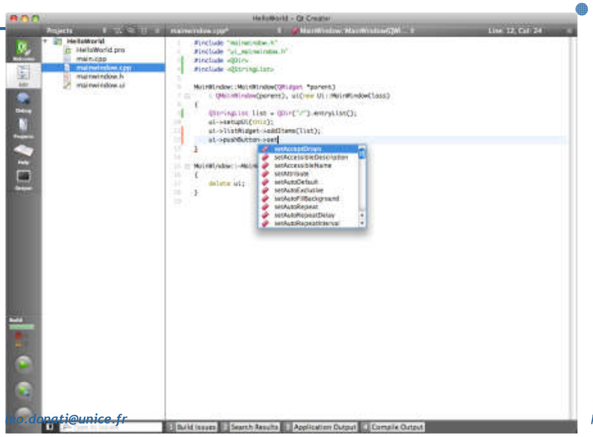


© OpenGL

leo.donati@unice.fr

page 82

ide : QtCreator



```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDir>
#include <QStringList>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), ui(new Ui::MainWindowClass)
{
    QStringList list = QDir("/").entryList();
    ui->setupUi(this);
    ui->listWidget->addItems(list);
    ui->pushButton->set
}

MainWindow::~Mai
{
    delete ui;
}
```

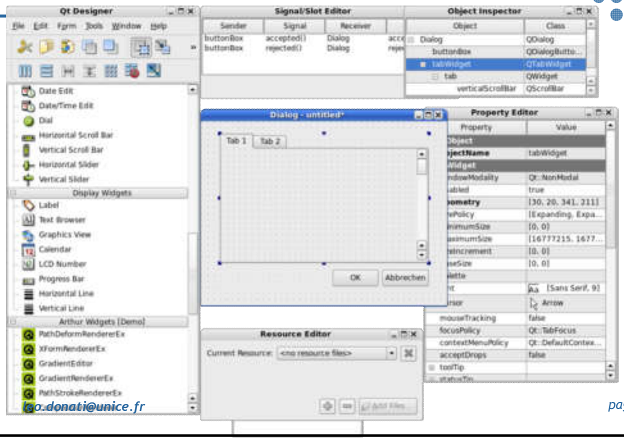
```
setAcceptDrops
setAccessibleDescription
setAccessibleName
setAttribute
setAutoDefault
setAutoExclusive
setAutoFillBackground
setAutoRepeat
setAutoRepeatDelay
setAutoRepeatInterval
```

© OpenGL

leo.donati@unice.fr

page 83

ihm : QtDesigner



Sender	Signal	Receiver
buttonBox	accepted()	Dialog
buttonBox	rejected()	Dialog

Object	Class
Dialog	QDialog
buttonBox	QDialogButto...
tabWidget	QTabWidget
tab	QWidget
verticalScrollBar	QScrollBar

Property	Value
objectName	tabWidget
windowModality	Qt::NonModal
enabled	true
geometry	[30, 20, 341, 211]
sizePolicy	[Expanding, Expa...
minimumSize	[0, 0]
maximumSize	[16777215, 1677...
sizeIncrement	[0, 0]
baseSize	[0, 0]
font	[Sans Serif, 9]
cursor	Arrow
mouseTracking	false
focusPolicy	Qt::TabFocus
contextMenuPolicy	Qt::DefaultContex...
acceptDrops	false
toolTip	

Current Resource: <no resource files>

© OpenGL

leo.donati@unice.fr

page 84

Signaux et slots

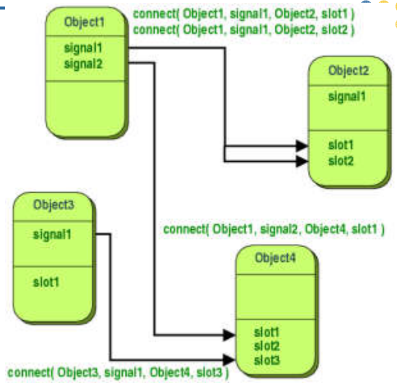
- Signaux et slots forment le mécanisme central de Qt qui permet aux objets Qt de communiquer entre eux
  - Boutons, zones de texte, fenêtres, etc…
- Le mot clé **signal** indique qu'un objet émet un signal (mais ne sait pas vers qui)
- Le mot clé **slot** indique qu'une méthode d'une classe peut être connecté à un signal
- La fonction **connect** lie le signal d'un objet au slot d'un autre

© OpenGL

leo.donati@unice.fr page 85

Couplage faible

- Un signal peut être envoyé vers plusieurs destinataires.
- Un slot peut être déclenché par plusieurs signaux.


connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object1
signal1
signal2
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

© OpenGL

leo.donati@unice.fr page 86

Metacompilation

- Un programme Qt contient des mots-clés qui ne sont pas du C++
  - signals, slots, Q_Object
- On ne peut pas compiler simplement un programme Qt avec un compilateur C++
- Pré-processeur : **moc** (meta-object compiler)
  - Pour générer les commandes C++ associées aux commandes Qt
  - pour mettre en place le mécanisme des signaux et des slots
- **qmake** permet d'automatise ces étapes

© OpenGL

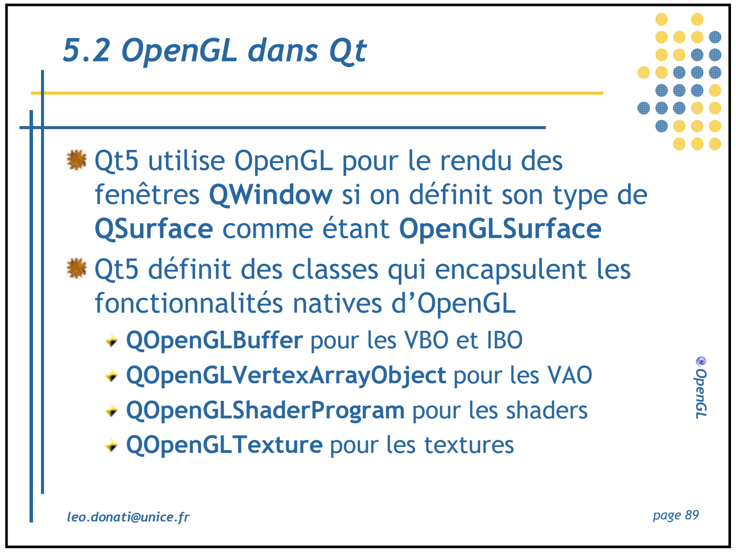
leo.donati@unice.fr page 87

Fichier projet .pro

- Décrit le projet
  - Fichiers (h et cpp)
  - Modules Qt à inclure
  - Fichier ui pour IHM
  - Bibliothèques
- Compilé par qmake
- Produit un makefile

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = nom_programme

TEMPLATE = app

SOURCES += main.cpp\
           mainwindow.cpp \
           widgetopengl.cpp

HEADERS  += mainwindow.h \
            widgetopengl.h

FORMS    += mainwindow.ui

INCLUDEPATH += $$PWD/../glm

windows:LIBS += -lopengl32
```

leo.donati@unice.fr page 88

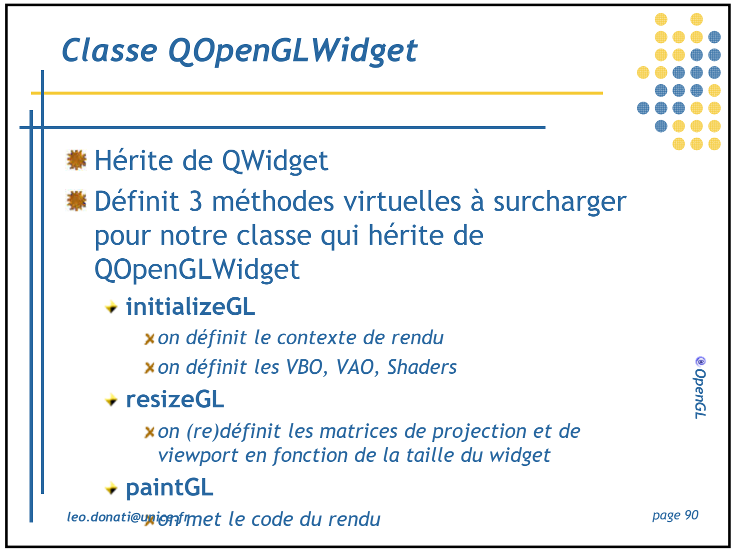


5.2 OpenGL dans Qt

- Qt5 utilise OpenGL pour le rendu des fenêtres **QWindow** si on définit son type de **QSurface** comme étant **OpenGLSurface**
- Qt5 définit des classes qui encapsulent les fonctionnalités natives d'OpenGL
  - **QOpenGLBuffer** pour les VBO et IBO
  - **QOpenGLVertexArrayObject** pour les VAO
  - **QOpenGLShaderProgram** pour les shaders
  - **QOpenGLTexture** pour les textures

© OpenGL

leo.donati@unice.fr *page 89*

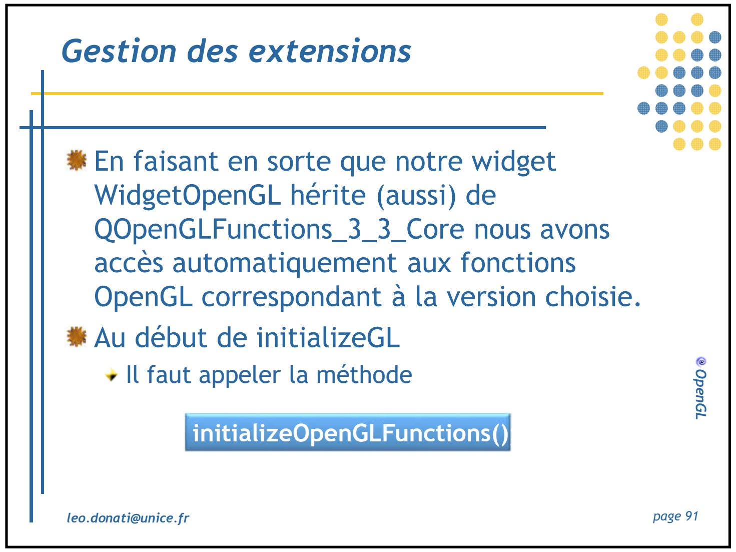


Classe QOpenGLWidget

- Hérite de QWidget
- Définit 3 méthodes virtuelles à surcharger pour notre classe qui hérite de QOpenGLWidget
  - **initializeGL**
    - *on définit le contexte de rendu*
    - *on définit les VBO, VAO, Shaders*
  - **resizeGL**
    - *on (re)définit les matrices de projection et de viewport en fonction de la taille du widget*
  - **paintGL**
    - *on met le code du rendu*

© OpenGL

leo.donati@unice.fr *page 90*



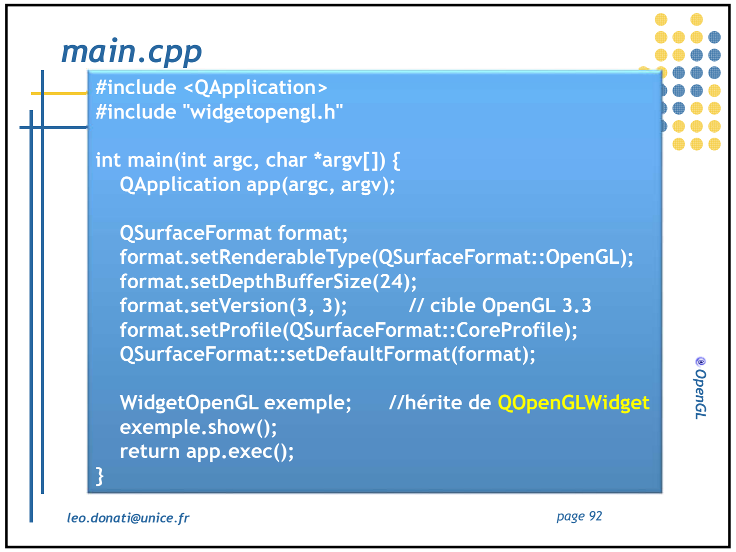
Gestion des extensions

- En faisant en sorte que notre widget WidgetOpenGL hérite (aussi) de QOpenGLFunctions_3_3_Core nous avons accès automatiquement aux fonctions OpenGL correspondant à la version choisie.
- Au début de initializeGL
  - Il faut appeler la méthode

`initializeOpenGLFunctions()`

© OpenGL

leo.donati@unice.fr *page 91*

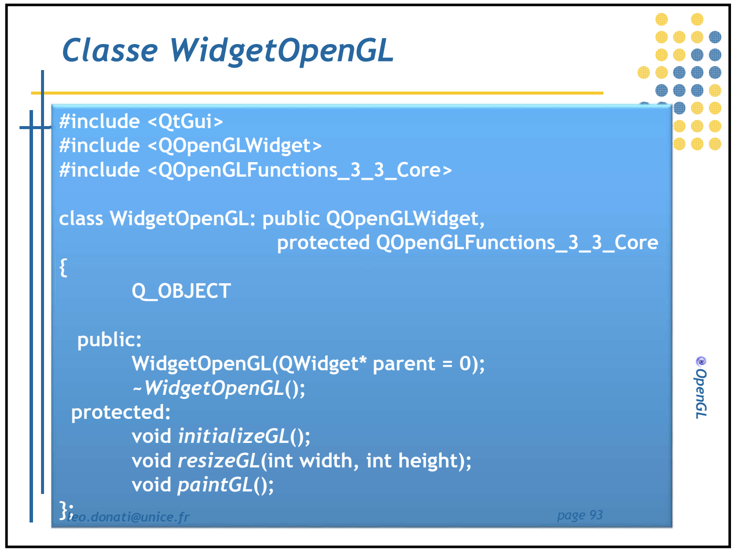


main.cpp

```
#include <QApplication>
#include "widgetopengl.h"

int main(int argc, char *argv[]) {
   QApplication app(argc, argv);

   QSurfaceFormat format;
   format.setRenderableType(QSurfaceFormat::OpenGL);
   format.setDepthBufferSize(24);
   format.setVersion(3, 3);         // cible OpenGL 3.3
   format.setProfile(QSurfaceFormat::CoreProfile);
   QSurfaceFormat::setDefaultFormat(format);

   WidgetOpenGL exemple;      //hérite de QOpenGLWidget
   exemple.show();
   return app.exec();
}
```

© OpenGL

leo.donati@unice.fr *page 92*

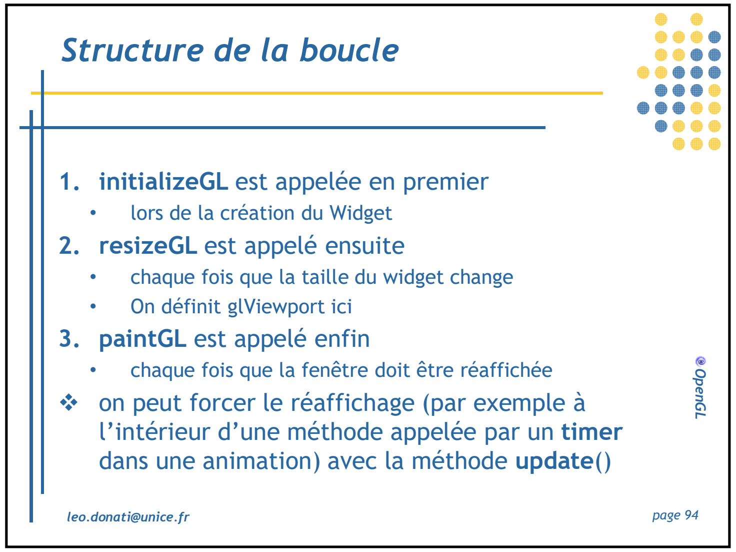


Classe WidgetOpenGL

```
#include <QtGui>
#include <QOpenGLWidget>
#include <QOpenGLFunctions_3_3_Core>

class WidgetOpenGL: public QOpenGLWidget,
                        protected QOpenGLFunctions_3_3_Core
{
        Q_OBJECT

  public:
        WidgetOpenGL(QWidget* parent = 0);
        ~WidgetOpenGL();
 protected:
        void initializeGL();
        void resizeGL(int width, int height);
        void paintGL();
};
```

leo.donati@unice.fr page 93

OpenGL



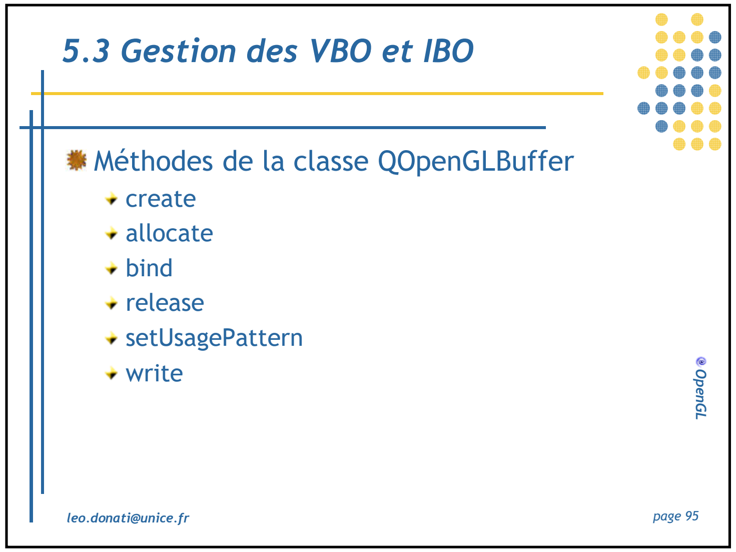
Structure de la boucle

1. **initializeGL** est appelée en premier
   - lors de la création du Widget
2. **resizeGL** est appelé ensuite
   - chaque fois que la taille du widget change
   - On définit glViewport ici
3. **paintGL** est appelé enfin
   - chaque fois que la fenêtre doit être réaffichée

❖ on peut forcer le réaffichage (par exemple à l'intérieur d'une méthode appelée par un **timer** dans une animation) avec la méthode **update**()

leo.donati@unice.fr page 94

OpenGL

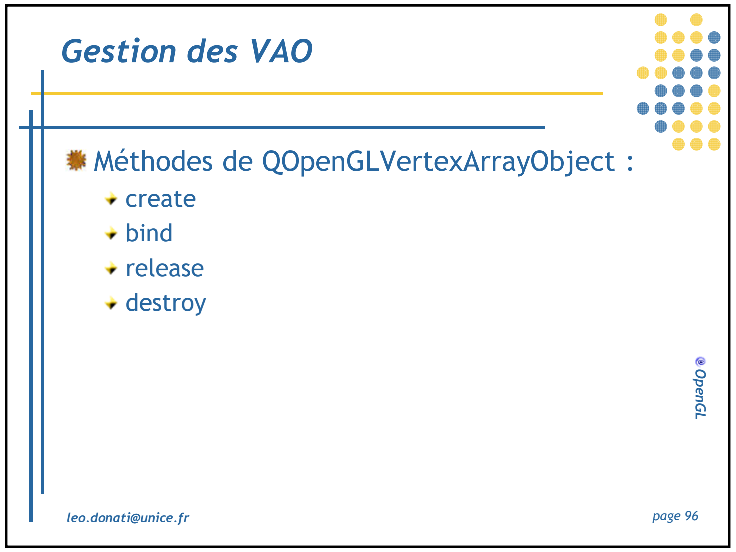


5.3 Gestion des VBO et IBO

- Méthodes de la classe QOpenGLBuffer
  - create
  - allocate
  - bind
  - release
  - setUsagePattern
  - write

leo.donati@unice.fr page 95

OpenGL

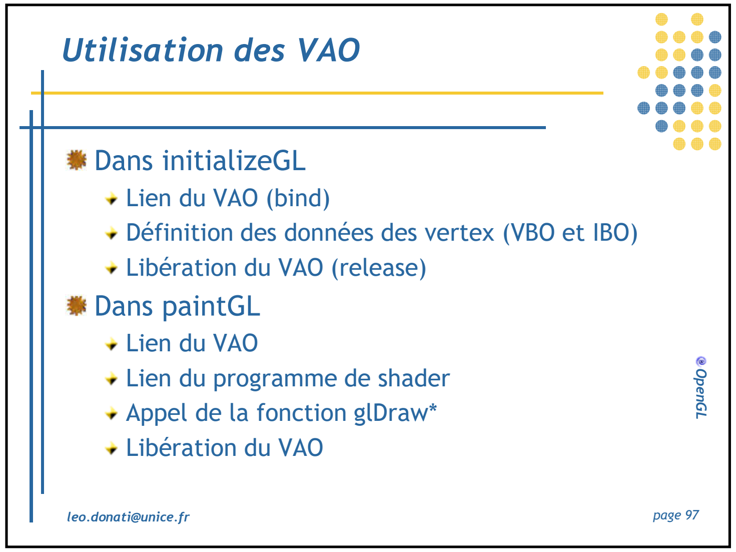


Gestion des VAO

- Méthodes de QOpenGLVertexArrayObject :
  - create
  - bind
  - release
  - destroy

leo.donati@unice.fr page 96

OpenGL

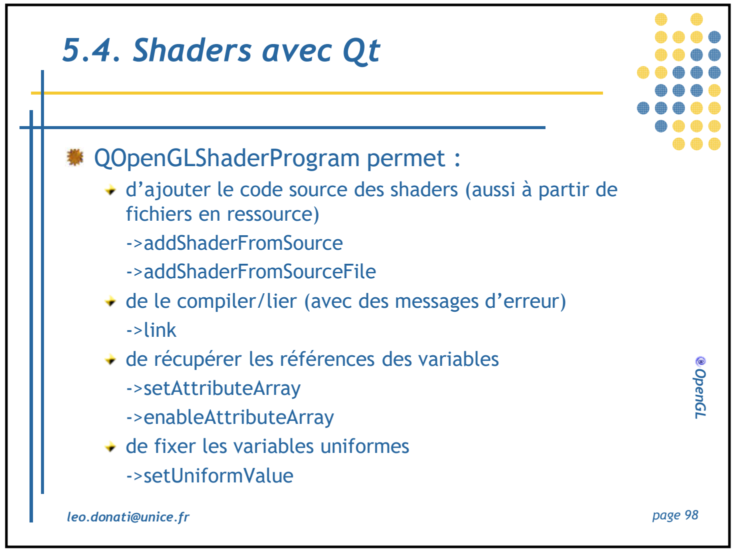


Utilisation des VAO

- Dans initializeGL
  - Lien du VAO (bind)
  - Définition des données des vertex (VBO et IBO)
  - Libération du VAO (release)
- Dans paintGL
  - Lien du VAO
  - Lien du programme de shader
  - Appel de la fonction glDraw*
  - Libération du VAO

OpenGL

leo.donati@unice.fr page 97

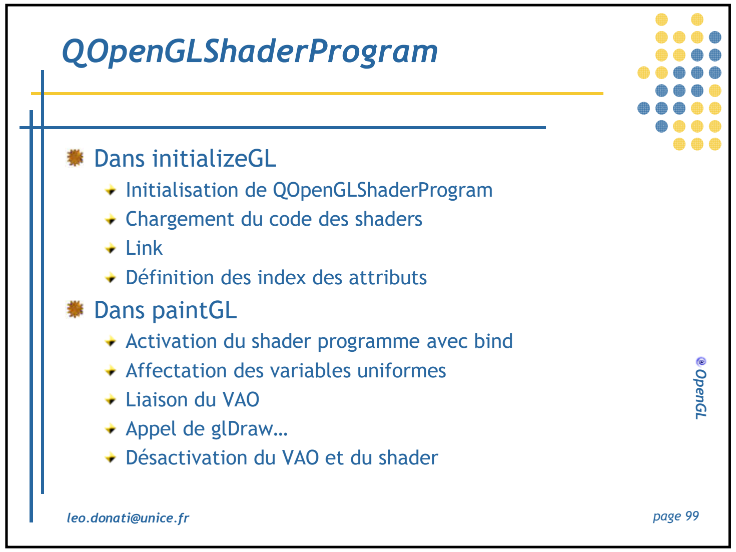


5.4. Shaders avec Qt

- QOpenGLShaderProgram permet :
  - d'ajouter le code source des shaders (aussi à partir de fichiers en ressource)
    ->addShaderFromSource
    ->addShaderFromSourceFile
  - de le compiler/lier (avec des messages d'erreur)
    ->link
  - de récupérer les références des variables
    ->setAttributeArray
    ->enableAttributeArray
  - de fixer les variables uniformes
    ->setUniformValue

OpenGL

leo.donati@unice.fr page 98

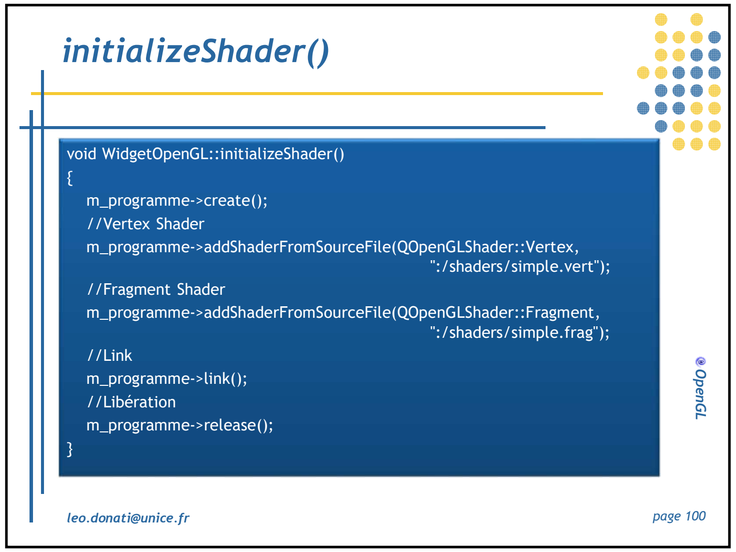


QOpenGLShaderProgram

- Dans initializeGL
  - Initialisation de QOpenGLShaderProgram
  - Chargement du code des shaders
  - Link
  - Définition des index des attributs
- Dans paintGL
  - Activation du shader programme avec bind
  - Affectation des variables uniformes
  - Liaison du VAO
  - Appel de glDraw…
  - Désactivation du VAO et du shader

OpenGL

leo.donati@unice.fr page 99

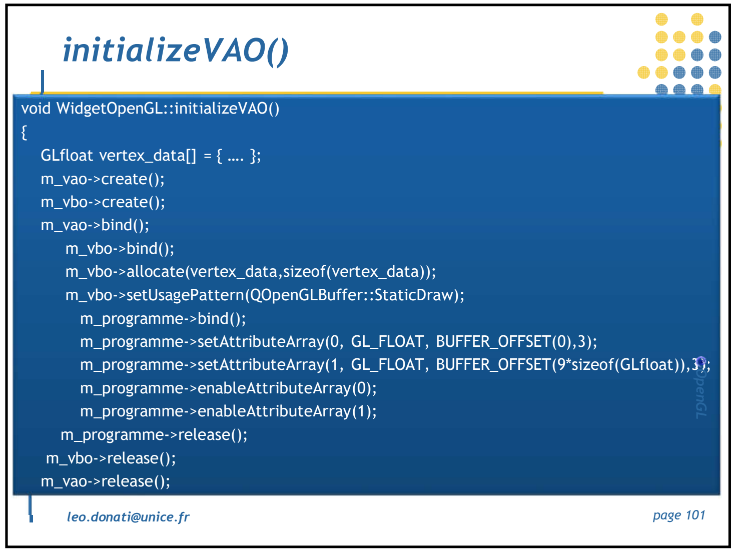


initializeShader()

```
void WidgetOpenGL::initializeShader()
{
   m_programme->create();
   //Vertex Shader
   m_programme->addShaderFromSourceFile(QOpenGLShader::Vertex,
                                         ":/shaders/simple.vert");
   //Fragment Shader
   m_programme->addShaderFromSourceFile(QOpenGLShader::Fragment,
                                         ":/shaders/simple.frag");
   //Link
   m_programme->link();
   //Libération
   m_programme->release();
}
```

OpenGL

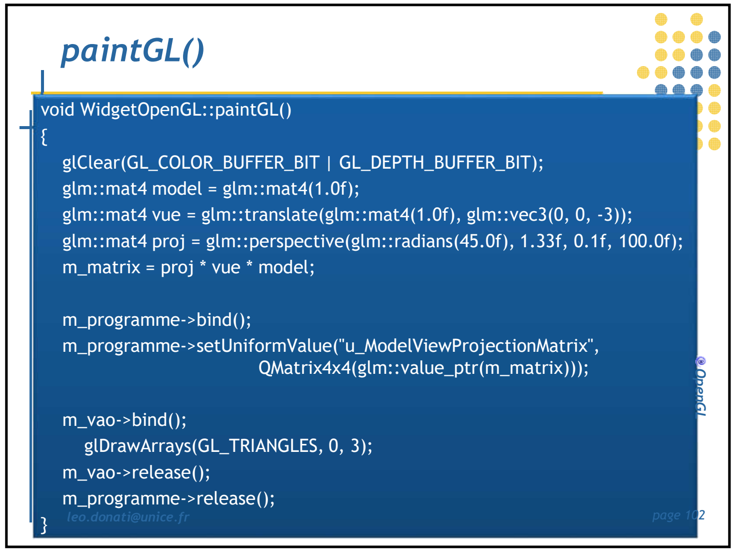
leo.donati@unice.fr page 100



initializeVAO()

```
void WidgetOpenGL::initializeVAO()
{
    GLfloat vertex_data[] = { .... };
    m_vao->create();
    m_vbo->create();
    m_vao->bind();
         m_vbo->bind();
         m_vbo->allocate(vertex_data,sizeof(vertex_data));
         m_vbo->setUsagePattern(QOpenGLBuffer::StaticDraw);
            m_programme->bind();
            m_programme->setAttributeArray(0, GL_FLOAT, BUFFER_OFFSET(0),3);
            m_programme->setAttributeArray(1, GL_FLOAT, BUFFER_OFFSET(9*sizeof(GLfloat)),3);
            m_programme->enableAttributeArray(0);
            m_programme->enableAttributeArray(1);
        m_programme->release();
     m_vbo->release();
    m_vao->release();
```

leo.donati@unice.fr page 101



paintGL()

```
void WidgetOpenGL::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glm::mat4 model = glm::mat4(1.0f);
    glm::mat4 vue = glm::translate(glm::mat4(1.0f), glm::vec3(0, 0, -3));
    glm::mat4 proj = glm::perspective(glm::radians(45.0f), 1.33f, 0.1f, 100.0f);
    m_matrix = proj * vue * model;

    m_programme->bind();
    m_programme->setUniformValue("u_ModelViewProjectionMatrix",
                                 QMatrix4x4(glm::value_ptr(m_matrix)));

    m_vao->bind();
        glDrawArrays(GL_TRIANGLES, 0, 3);
    m_vao->release();
    m_programme->release();
}
```

leo.donati@unice.fr page 102