

Laboratori de IDI

Índex de les Transparències

Professorat IDI

22 de maig de 2021

Índex

1	Preliminars	3
1.1	Organització de projecte i/o fitxers[S1.1][T:5–6,25]	3
1.2	Compilació i Execució [S1.1][T:6–7]	6
1.3	Mètodes que cal implementar[S1.1][T:9]	8
2	Models	9
2.1	VAOs i VBOs[S1.1][T:11–34]	9
2.1.1	Creació[S1.1][T:12–17]	10
2.1.2	Càrrega[S1.1][T:18–21]	16
2.1.3	Pintar[S1.1][T:22–24]	20
2.2	Exemple complet[S1.1][T: 26–34]	23
2.3	Càrrega de models[S2.1][T: 11–15]	31
3	Shaders	36
3.1	Vertex Shader[S1.2][T:4–8]	36
3.2	Fragment Shader[S1.2][T:9]	41
3.3	Exemples Vertex i Fragment Shaders[S1.2][T:14,23–24][T:14–16,23–24]	42
3.4	GLSL[S1.2][T:10–20]	47
3.4.1	Tipus de Dades[S1.2][T:17]	50
3.4.2	Operacions[S1.2][T:18]	51
3.4.3	Funcions[S1.2][T:19–20]	52
3.4.4	Funcions[S1.2][T:19–20]	53
3.5	Variables Predefinides[S1.2][T:21]	54
3.6	Discard[S1.2][T:22]	55
3.7	Gestió de Shaders[S1.2][T:28–39]	56
3.7.1	Procès de càrrega[S1.2][T:28–37]	56
3.7.2	Altres Exemples[S1.2][T:39]	66
3.8	Uniforms[S1.3][T:3–9]	67

4	Interacció	74
4.1	Baix Nivell[S1.3][T:10–15]	74
4.2	Qt-Widgets	80
4.2.1	Layouts[Qt1][T:7]	80
4.2.2	Signals i Slots[Qt1][T:8–11]	81
4.2.3	Disseny[Qt1][T:12]	85
4.2.4	Assistant[Qt1][T:13]	86
4.2.5	Compilació[Fitxers per Compilar]	87
4.3	Qt-Custom Widgets[Qt2]	95
4.4	Qt-MyGLWidget[Qt2]	100
5	Transformacions Geomètriques	101
5.1	Matrius de Transformació[S1.3][T:16–18]	101
5.2	glm[S1.3][T:19–20]	104
6	Càmera	106
6.1	View Matrix	106
6.1.1	LookAt[S2.1][T:7–10]	106
6.1.2	Euler[S2.3][T:2]	109
6.2	Project Matrix	110
6.2.1	Perspectiva[S2.1][T:3–6]	110
6.2.2	Ortogonal[S2.2][T:11–12]	113
6.3	Tercera Persona[S2.2][T:2–4],[S2.3][T:3–4]	115
6.4	Resize[S2.2][T:5–7,13]	119
6.5	Zoom[S2.3][T:5]	123
7	Realisme	124
7.1	Z-Buffer i Culling[S2.1][T:16]	124
7.2	Il·luminació	125
7.2.1	Models Empírics[S3.1][T:2–5]	125
7.2.2	Models Empírics: Lambert[S3.1][T:7]	129
7.2.3	Models Empírics: Phong[S3.1][T:10]	130
7.2.4	Normal Matrix[S3.1][T:4–5]	131
7.2.5	Focus (càmera, escena, objecte)[S3.2][T:2–3]	133
7.2.6	Al Fragment Shader[S3.2][T:6]	135

1 Preliminars

1.1 Organització de projecte i/o fitxers[S1.1][T:5–6,25]

Projecte Qt

main.cpp

```
#include <QApplication>
```

```
#include "MyForm.h"
```

```
int main (int argc, char **argv)
```

```
{
```

```
→   QApplication a(argc, argv);
```

```
    MyForm myf;
```

```
    myf.show ();
```

```
→   return a.exec ();
```

```
}
```

Projecte Qt

- Crear un fitxer `.pro` que conté la descripció del projecte que estem programant
- Utilitzar les comandes `qmake` i `make`.
 - `qmake` genera el `Makefile` a partir del `.pro`
 - `make` compila i enllaça.

Exemple complet

- Exemple que teniu a /assig/idi/blocs/bloc-1

Defineix els components de l'aplicació

Bloc1_exemple.pro

Disseny de la interfície

MyForm.ui

Programa principal

main.cpp

Classe que hereta de QOpenGLWidget
Implementa tot el procés de pintat

Classe que engloba la interfície

MyForm.h

MyForm.cpp

MyGLWidget.h

MyGLWidget.cpp

Projecte Qt

- Crear un fitxer `.pro` que conté la descripció del projecte que estem programant
- Utilitzar les comandes `qmake` i `make`.
 - `qmake` genera el `Makefile` a partir del `.pro`
 - `make` compila i enllaça.

Compilar i enllaçar

- Crear un fitxer “*helloQt.pro*”
 - `TEMPLATE = app`
 - `QT += widgets`
 - `DEPENDPATH += .`
 - `INCLUDEPATH += .`
 - `#Input`
 - `SOURCES += main.cpp`
- Compilem i enllacem
 - `qmake` (al laboratori cal fer `qmake-qt5`)
 - `make`
- Executable anomenat *helloQt* en el directori on estiguem.
- Executar-lo amb:
 - `./helloQt`

1.3 Mètodes que cal implementar[S1.1][T:9]

OpenGL amb Qt

Per usar OpenGL amb Qt cal derivar una classe de QOpenGLWidget.

Mètodes virtuals que cal implementar:

- *initializeGL ()*

- Codi d'inicialització d'OpenGL.
- Qt la cridarà abans de la 1ª crida a *resizeGL*.

- *paintGL ()*

- Codi per redibuixar l'escena.
- Qt la cridarà cada cop que calgui el repintat. El *swapBuffers()* és automàtic per defecte.

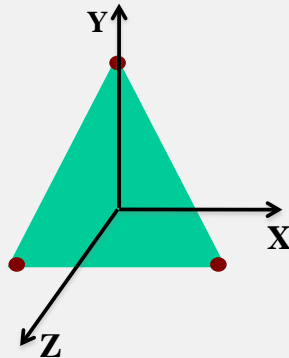
- *resizeGL ()*

- Codi que cal fer quan es redimensiona la finestra.
- Qt la cridarà quan es creï la finestra, i cada cop que es modifiqui la mida de la finestra.

2 Models

2.1 VAOs i VBOs[S1.1][T:11–34]

Informació del model



Possible informació associada a un vèrtex:

- Posició (coordenades)
- Color (rgb/rgba)
- Vector normal (coordenades)
- ...

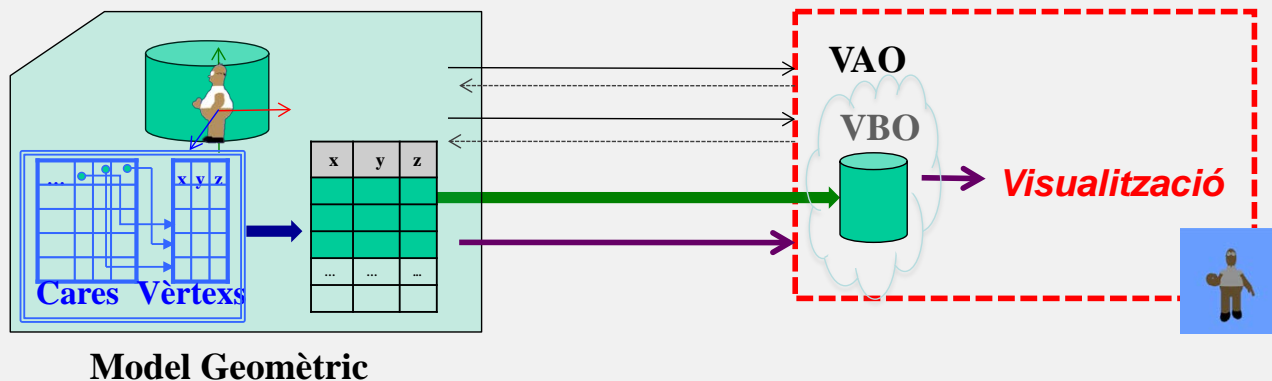
Per a cada model cal generar un Vertex Array Object (VAO).

Les dades dels vèrtexs s'han de passar a la tarja gràfica guardats en Vertex Buffer Object (VBO).

Pintarem els VAOs.

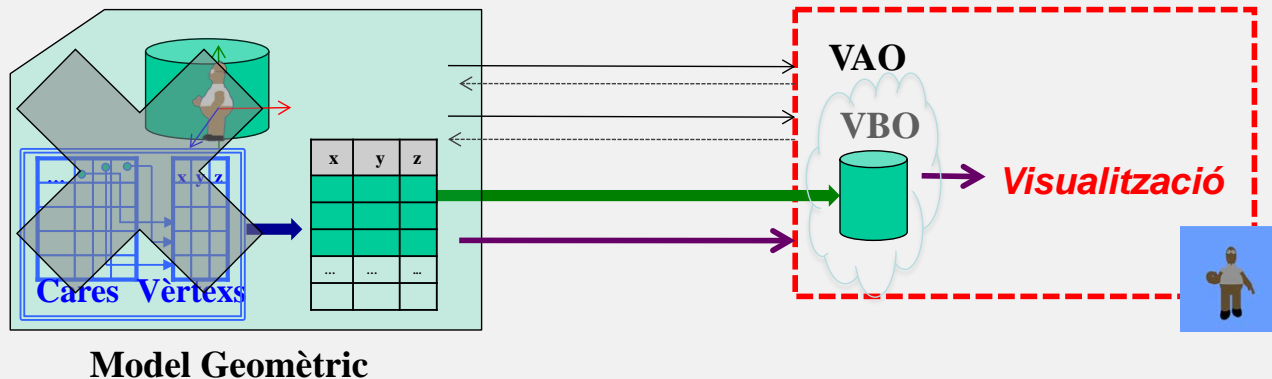
Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un **VAO** que encapsularà dades del model. Crear **VBO** que guardarà les coordenades dels vèrtexs (potser cal altres per normal, color,...)
2. Guardar llista de vèrtexs (amb repetició) en el **VBO** (i si cal, color i normal en els seus **VBO**)
3. Cada cop que es requereix pintar, indicar el **VAO** a pintar i dir que es pinti: *glDrawArrays(...)*. Acció *pinta_model()* a teoria.



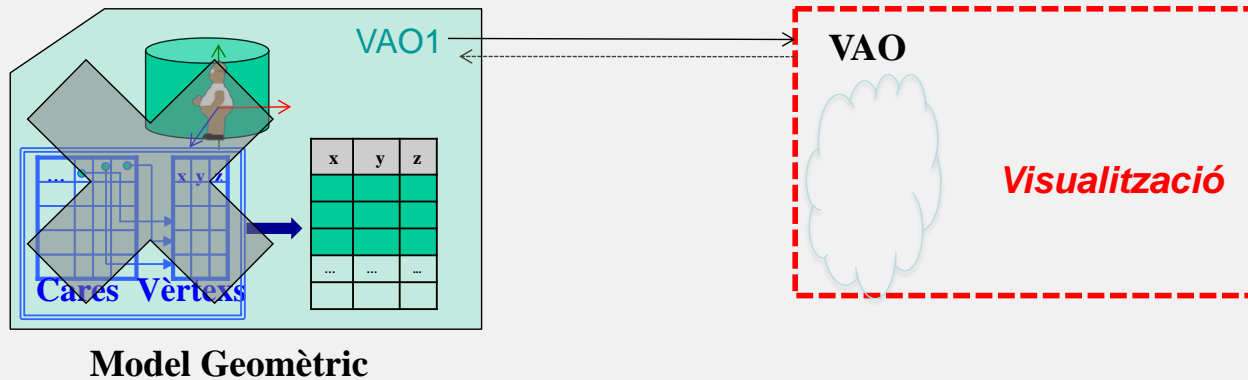
Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un *VAO* que encapsularà dades del model. Crear *VBO* que guardarà les coordenades dels vèrtexs (potser cal altres per normal, color,...)
2. Guardar llista de vèrtexs (amb repetició) en el *VBO* (i si cal, color i normal en els seus *VBO*)
3. Cada cop que es requereix pintar, indicar el *VAO* a pintar i dir que es pinti: *glDrawArrays(...)*. Acció *pinta_model()* a teoria.



Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un VAO



`GLuint VAO1; // variable on guardarem l'identificador del VAO`

`glGenVertexArrays (1, &VAO1); // generació de l'identificador`

`glBindVertexArray (VAO1); // activació del VAO`

Informació del model

Per a generar un VAO, descripció de les crides:

```
void glGenVertexArrays (GLsizei n, GLuint *arrays);
```

Genera *n* identificadors per a VAOs i els retorna a *arrays*

n : nombre de VAOs a generar

arrays : vector de GLuint on els noms dels VAO generats es retornen

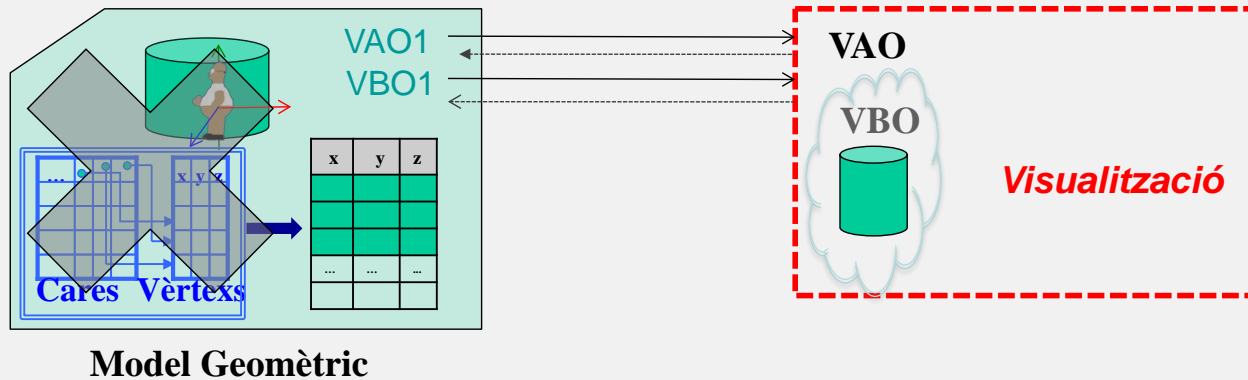
```
void glBindVertexArray (GLuint array);
```

Activa el VAO identificat per *array*

array : nom del VAO a activar

Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un *VAO*. Crear *VBO*.



`GLuint VBO1; // variable on guardarem l'identificador del VBO`

`glGenBuffers (1, &VBO1); // generació de l'identificador`

`glBindBuffer (GL_ARRAY_BUFFER, VBO1); // activació del VBO`

Informació del model

Per a generar un VBO, descripció de les crides:

```
void glGenBuffers (GLsizei n, GLuint *buffers);
```

Genera *n* identificadors per a VBOs i els retorna a *buffers*

n : nombre de VBOs a generar

buffers : vector de GLuint on els noms dels VBO generats es retornen

```
void glBindBuffer (GLenum target, GLuint buffer);
```

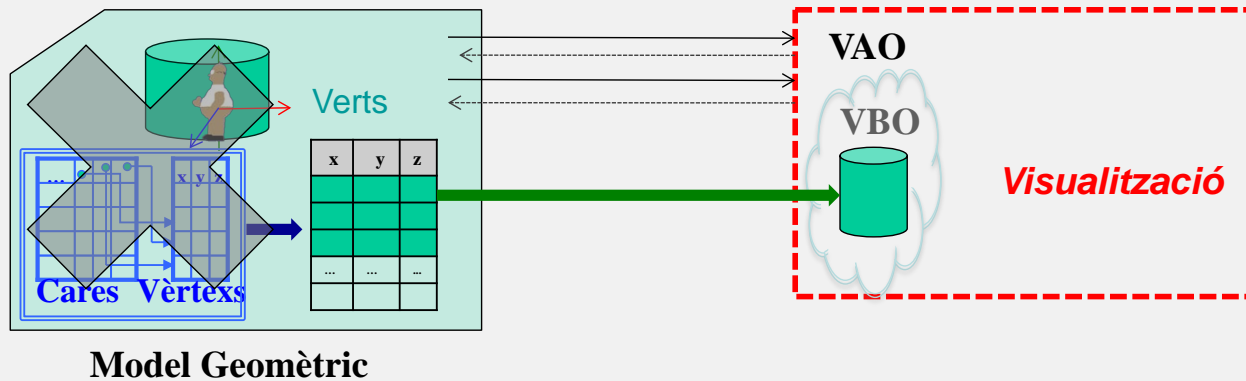
Activa el VBO identificat per *buffer*

target : tipus de buffer de la GPU que s'usarà (GL_ARRAY_BUFFER, ...)

buffer : nom del VBO a activar

Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un *VAO*. Crear *VBO*.
2. Guardar llista de vèrtexs (amb repetició) en el *VBO*



```
glBufferData (GL_ARRAY_BUFFER, sizeof (Verts), Verts, GL_STATIC_DRAW);
```


Informació del model

Per a omplir les dades d'un VBO:

```
void glBufferData (GLenum target, GLsizeiptr size,  
                  const GLvoid *data, GLenum usage);
```

Envia les dades que es troben en *data* per a què siguin emmagatzemades a la GPU

target : tipus de buffer de la GPU que s'usarà (GL_ARRAY_BUFFER, ...)

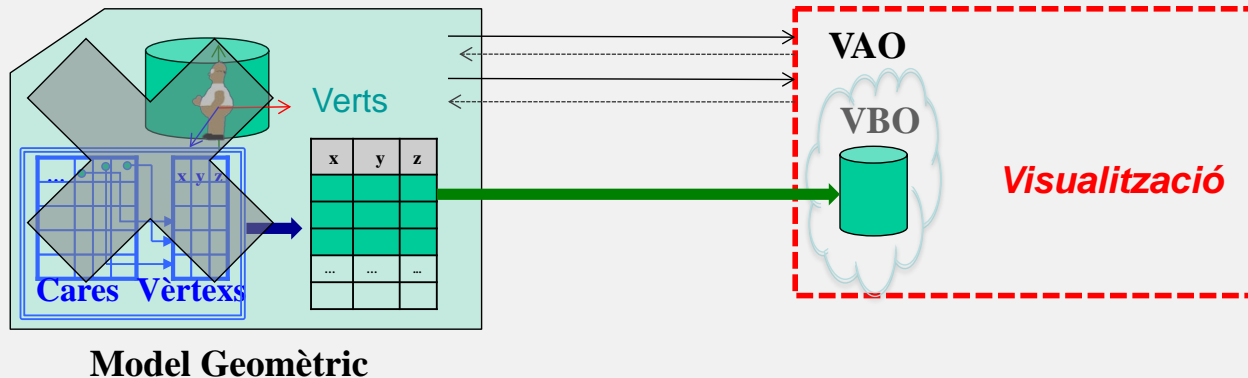
size : mida en bytes de les dades

data : apuntador a les dades

usage : patró d'ús esperat per a aquestes dades (GL_STATIC_DRAW, GL_DYNAMIC_DRAW, ...)

Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un *VAO*. Crear *VBO*.
2. Guardar llista de vèrtexs (amb repetició) en el *VBO*



```
// Cal indicar a la GPU com ha d'interpretar les dades que li hem passat (Verts)
glVertexAttribPointer (vertexLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray (vertexLoc);
```

Informació del model

Per a indicar a la GPU l'atribut dels vèrtexs a tenir en compte:

```
void glVertexAttribPointer (GLuint index, GLint size, GLenum type,  
                             GLboolean normalized, GLsizei stride, const GLvoid *pointer);
```

Indica les característiques de l'atribut del vèrtex identificat per *index*

index : nom de l'atribut

size : nombre de components que componen l'atribut

type : tipus de cada component (GL_FLOAT, GL_INT, ...)

normalized : indica si els valors de cada component s'han de normalitzar

stride : offset en bytes entre dos atributs consecutius (normalment 0)

pointer : offset del primer component del primer atribut respecte al buffer (normalment 0)

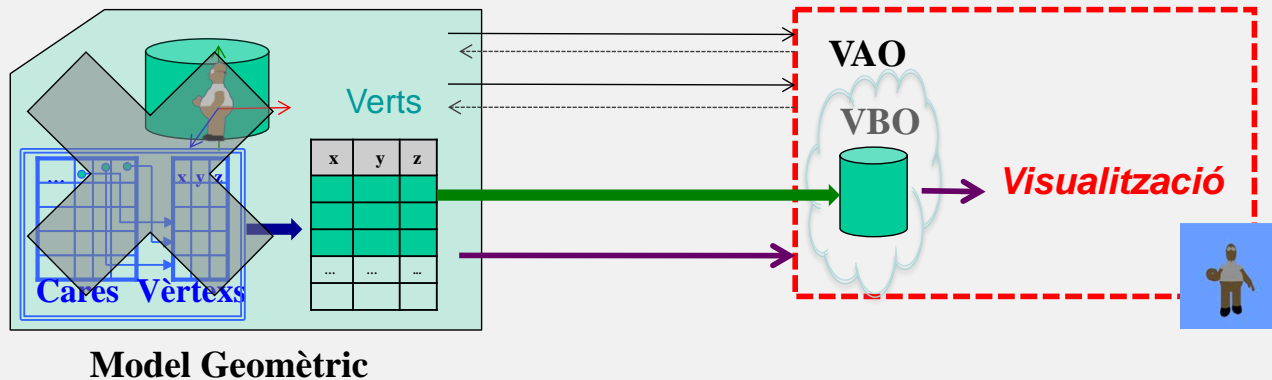
```
void glEnableVertexAttribArray (GLuint index);
```

Activa l'atribut del vèrtex identificat per *index*

index : nom de l'atribut a activar

Pintar en OpenGL 3.3: “core” mode

1. Crear en GPU/OpenGL un *VAO*. Crear *VBO*.
2. Guardar llista de vèrtexs (amb repetició) en el *VBO*
3. Per pintar: Indicar el *VAO* a pintar i dir que es pinti



```
glBindVertexArray (VAO1);  
glDrawArrays (GL_TRIANGLES, 0, 3);
```

Pintar un VAO

Per a pintar un VAO:

- 1) Activar el VAO amb `glBindVertexArray (GLuint array);`
- 2) Pintar el VAO:

`void glDrawArrays (GLenum mode, GLint first, GLsizei count);`

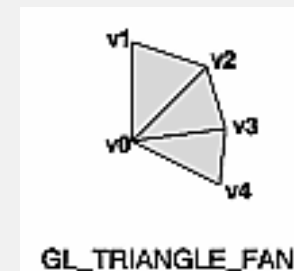
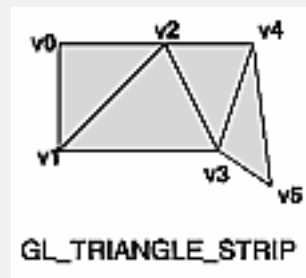
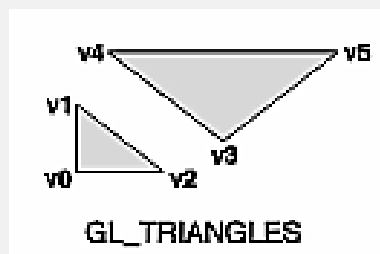
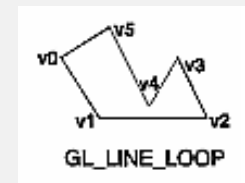
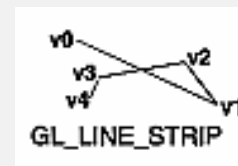
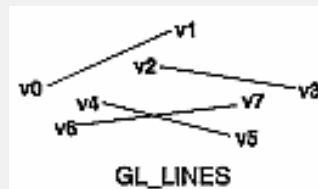
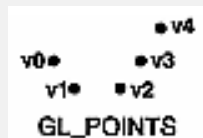
mode : tipus de primitiva a pintar (GL_TRIANGLES, ...)

first : índex del primer element de l'array

count : nombre d'elements a tenir en compte de l'array

Primitives en OpenGL

- Totes les primitives s'especificuen mitjançant vèrtexs:



2.2 Exemple complet[S1.1][T: 26–34]

Exemple complet: Bloc1_exemple.pro

```
TEMPLATE = app
```

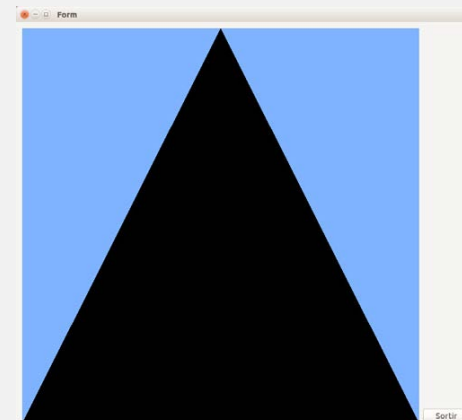
```
QT += opengl
```

```
INCLUDEPATH += /usr/include/glm
```

```
FORMS += MyForm.ui
```

```
HEADERS += MyForm.h MyGLWidget.h
```

```
SOURCES += main.cpp \  
          MyForm.cpp MyGLWidget.cpp
```



Exemple complet: main.cpp

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

int main (int argc, char **argv)
{
    QApplication a(argc, argv);

    → [ QSurfaceFormat f;
        f.setVersion (3, 3);
        f.setProfile (QSurfaceFormat::CoreProfile);
        QSurfaceFormat::setDefaultFormat (f);

        MyForm myf;
        myf.show ();

        return a.exec ();
    }
```


Exemple complet:

MyGLWidget.h

```
#include <QOpenGLFunctions_3_3_Core>
#include <QOpenGLWidget>
..... // ho explicarem el proper dia
#include "glm/glm.hpp"

class MyGLWidget : public QOpenGLWidget, protected QOpenGLFunctions_3_3_Core
{
    Q_OBJECT
public:
    MyGLWidget (QWidget *parent=0);
    ~MyGLWidget ();
protected:
    virtual void initializeGL (); // Inicialitzacions del contexte gràfic
    virtual void paintGL (); // Mètode de pintat
    virtual void resizeGL (int width, int height); // Es crida quan canvia dimensió finestra
private:
    void creaBuffers ();
    ..... // ho explicarem el proper dia
    GLuint VAO1, vertexLoc;
    GLint ample, alt;
};
```



Exemple complet:

MyGLWidget.h

```
#include <QOpenGLFunctions_3_3_Core>
#include <QOpenGLWidget>
..... // ho explicarem el proper dia
#include "glm/glm.hpp"
```

```
→ class MyGLWidget : public QOpenGLWidget, protected QOpenGLFunctions_3_3_Core
{
    Q_OBJECT
public:
    MyGLWidget (QWidget *parent=0);
    ~MyGLWidget ();
protected:
    → { virtual void initializeGL (); // Inicialitzacions del contexte gràfic
        virtual void paintGL (); // Mètode de pintat
        virtual void resizeGL (int width, int height); // Es crida quan canvia dimensió finestra
    private:
        void creaBuffers ();
        ..... // ho explicarem el proper dia
        GLuint VAO1, vertexLoc;
        GLint ample, alt;
};
```

Exemple complet:

MyGLWidget.h

```
#include <QOpenGLFunctions_3_3_Core>
#include <QOpenGLWidget>
..... // ho explicarem el proper dia
#include "glm/glm.hpp"

class MyGLWidget : public QOpenGLWidget, protected QOpenGLFunctions_3_3_Core
{
    Q_OBJECT
public:
    MyGLWidget (QWidget *parent=0);
    ~MyGLWidget ();
protected:
    virtual void initializeGL (); // Inicialitzacions del contexte gràfic
    virtual void paintGL (); // Mètode de pintat
    virtual void resizeGL (int width, int height); // Es crida quan canvia dimensió finestra
private:
    void creaBuffers ();
    ..... // ho explicarem el proper dia
    GLuint VAO1, vertexLoc;
    GLint ample, alt;
};
```

Exemple complet:

MyGLWidget.cpp (1)

```
#include "MyGLWidget.h"

MyGLWidget::MyGLWidget (QWidget* parent) : QOpenGLWidget (parent), program(NULL)
{
    setFocusPolicy(Qt::StrongFocus); // per rebre events de teclat
}
MyGLWidget::~MyGLWidget ()
{
    if (program != NULL) delete program;
}

void MyGLWidget::initializeGL ()
{
    // cal inicialitzar l'ús de les funcions d'OpenGL
    initializeOpenGLFunctions ();

    glClearColor (0.5, 0.7, 1.0, 1.0); // defineix color de fons (d'esborrat)
    ..... // ho explicarem el proper dia
    creaBuffers();
}
```

Exemple complet:

MyGLWidget.cpp (2)

```
void MyGLWidget::creaBuffers ()
{
    glm::vec3 Vertices[3]; // Tres vèrtexs amb X, Y i Z
    Vertices[0] = glm::vec3(-1.0, -1.0, 0.0);
    Vertices[1] = glm::vec3(1.0, -1.0, 0.0);
    Vertices[2] = glm::vec3(0.0, 1.0, 0.0);
    // Creació del Vertex Array Object (VAO) que usarem per pintar
    glGenVertexArrays(1, &VAO1);
    glBindVertexArray(VAO1);
    // Creació del buffer amb les dades dels vèrtexs
    GLuint VBO1;
    glGenBuffers(1, &VBO1);
    glBindBuffer(GL_ARRAY_BUFFER, VBO1);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);
    // Activem l'atribut que farem servir per vèrtex
    glVertexAttribPointer(vertexLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(vertexLoc);
    // Desactivem el VAO
    glBindVertexArray(0);
}
```

Exemple complet:

MyGLWidget.cpp (3)

```
void MyGLWidget::paintGL ()
{
    glClear (GL_COLOR_BUFFER_BIT); // Esborrem el frame-buffer

    // glViewport (0, 0, ample, alt); // Aquesta crida només cal si paràmetres diferents

    // Activem l'Array a pintar
    glBindVertexArray(VAO1);
    // Pintem l'escena
    glDrawArrays(GL_TRIANGLES, 0, 3);
    // Desactivem el VAO
    glBindVertexArray(0);
}

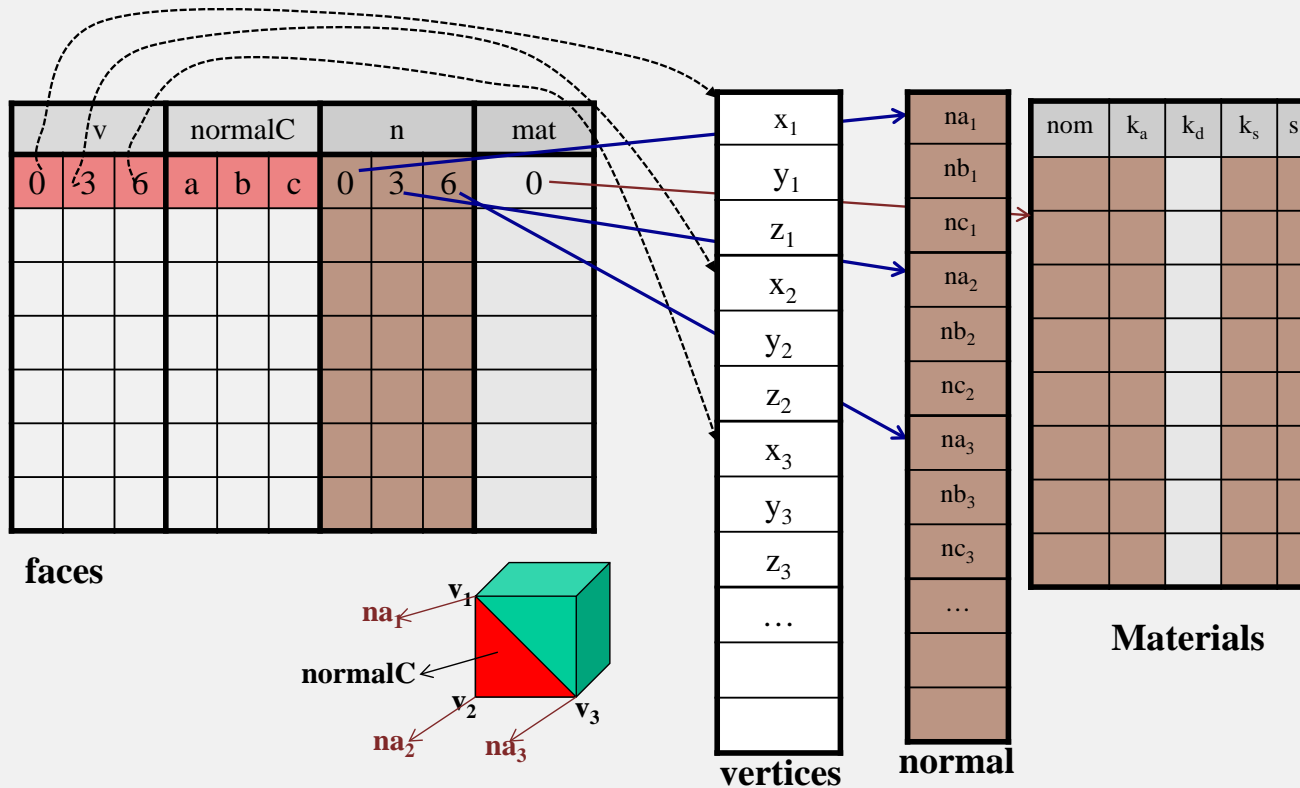
void MyGLWidget::resizeGL (int w, int h)
{
    ample = w;
    alt = h;
}
```

Càrrega de models OBJ (exercici 4)

- Classe Model: permet carregar *objecte.obj*
 - `/assig/idi/Model` (copieu-vos la carpeta en un directori vostre)
 - Analitzeu el `model.h` (classe Model)
 - Mètode `Model::load(std::string filename)`
Inicialitza les estructures de dades a partir d'un model en format OBJ-Wavefront en disc
- Modifiqueu el fitxer `.pro` afegint

```
INCLUDEPATH += <el-vostre-directori>/Model
SOURCES += <el-vostre-directori>/Model /model.cpp
```
- En `/assig/idi/models` trobareu models d'objectes.
 - Si els copieu a un directori local, per cada `.obj` copieu també (si existeix) el `.mtl` → definició dels materials corresponents.
 - Fins la propera sessió usarem el **HomerProves**
- Més models els podeu trobar a la xarxa.

Representació classe Model



Analitzeu l'arxiu **model.h**

Compte!! amb el nom dels camps de Material que en l'esquema són simbòlics; p.e. **k_d** és **float diffuse[4]**

Representació auxiliar de la classe Model

x_1	nx_1	r_1	r_1	r_1	sh_1
y_1	ny_1	g_1	g_1	g_1	sh_2
z_1	nz_1	b_1	b_1	b_1	sh_3
x_2	nx_2	r_2	r_2	r_2	...
y_2	ny_2	g_2	g_2	g_2	
z_2	nz_2	b_2	b_2	b_2	
x_3	nx_3	r_3	r_3	r_3	
y_3	ny_3	g_3	g_3	g_3	
z_3	nz_3	b_3	b_3	b_3	
...	

VBO_vertices **VBO_normals** **VBO_matamb** **VBO_matdiff** **VBO_matspec** **VBO_matshin**

Ús de la classe Model (exercici 4)

- Construcció d'un objecte de tipus Model (declaració)

```
Model m; // un únic model
```

```
Model vectorModels[3]; // array de 3 models
```

```
vector<Model> models; // vector stl de models
```

- Càrrega d'un arxiu (model) .obj

```
m.load ("../models/HomerProves.obj");
```

- Accés als seus VBOs (els genera la propia classe Model)

```
glBufferData (... , m.VBO_vertices (), GL_STATIC_DRAW); // posició
```

```
glBufferData (... , m.VBO_matdiff (), GL_STATIC_DRAW); // color
```

- Per a saber el nombre de cares (totes les cares són triangles)

```
m.faces().size()
```

```
sizeof(GLfloat) * m.faces ().size () * 3 * 3 // nombre de bytes dels buffers
```

Exemples

- Pas de dades del buffer de posicions cap a la GPU

```
glBufferData (GL_ARRAY_BUFFER,  
             sizeof(GLfloat) * m.faces ().size () * 3 * 3,  
             m.VBO_vertices (), GL_STATIC_DRAW);
```

- Pintar l'objecte

```
glDrawArrays (GL_TRIANGLES, 0, m.faces ().size () * 3);
```

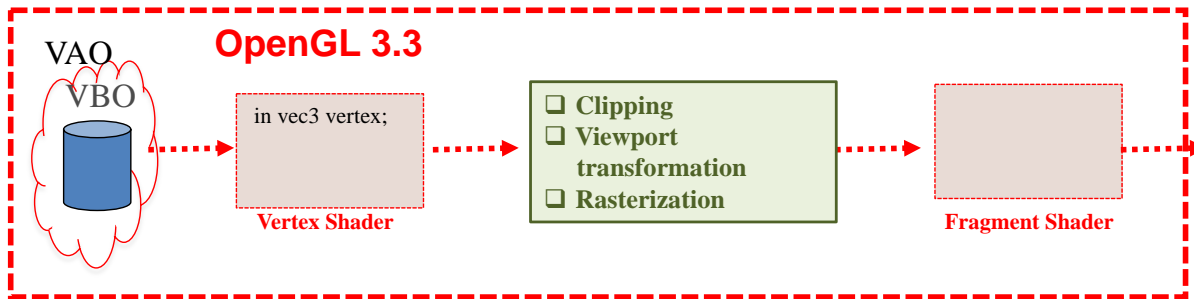
- Recorregut de la taula de vèrtexs

```
for (unsigned int i = 0; i < m.vertices().size(); i+=3) {  
    // escric per pantalla les coordenades del vèrtex  
    std::cout << "(x, y, z) = (" << m.vertices()[i] << " , "  
                << m.vertices()[i+1] << " , "  
                << m.vertices()[i+2] << ")" << std::endl;  
}
```

3 Shaders

3.1 Vertex Shader[S1.2][T:4–8]

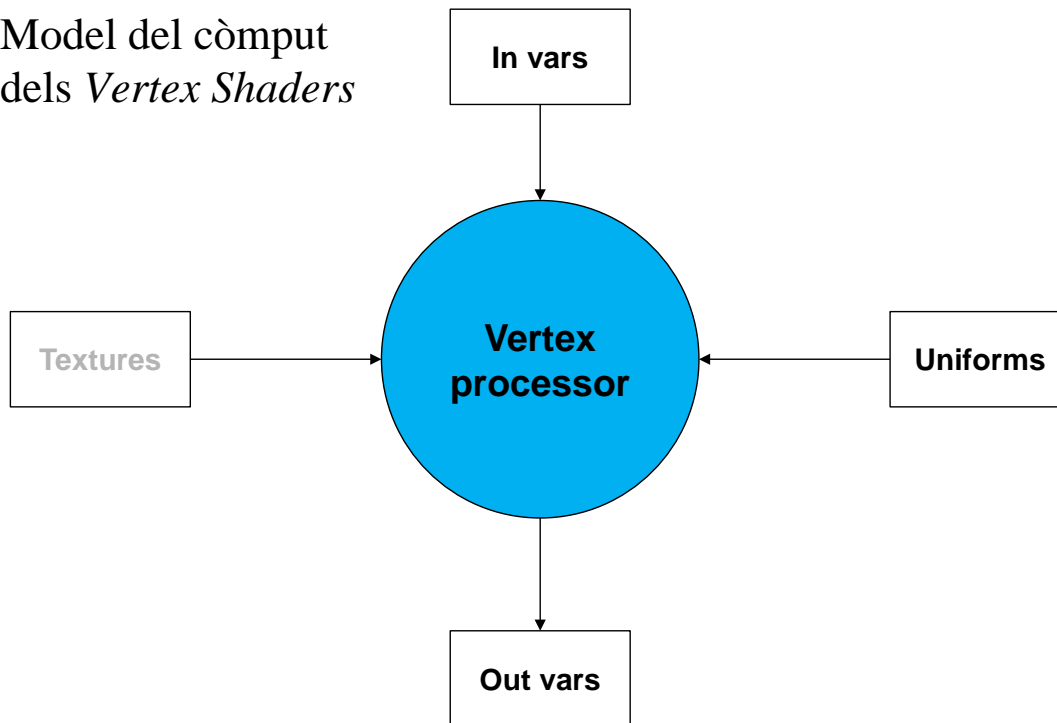
Paradigma projectiu bàsic amb OpenGL 3.3



- Per a cada vèrtex s'executa el Vertex Shader.
- OpenGL després *retalla* la primitiva, *passa a coordenades de dispositiu* el vèrtex i *rasteritza*, produint els fragments.
- Per a cada fragment s'executa el Fragment Shader.

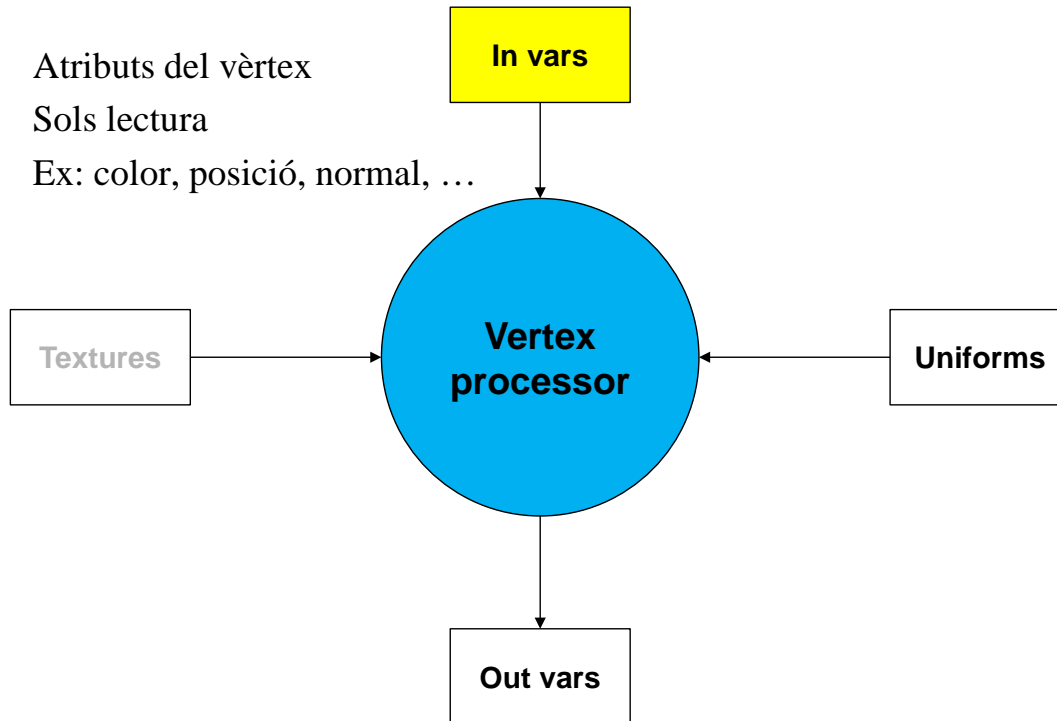
Vertex processor (1)

- Model del còmput dels *Vertex Shaders*

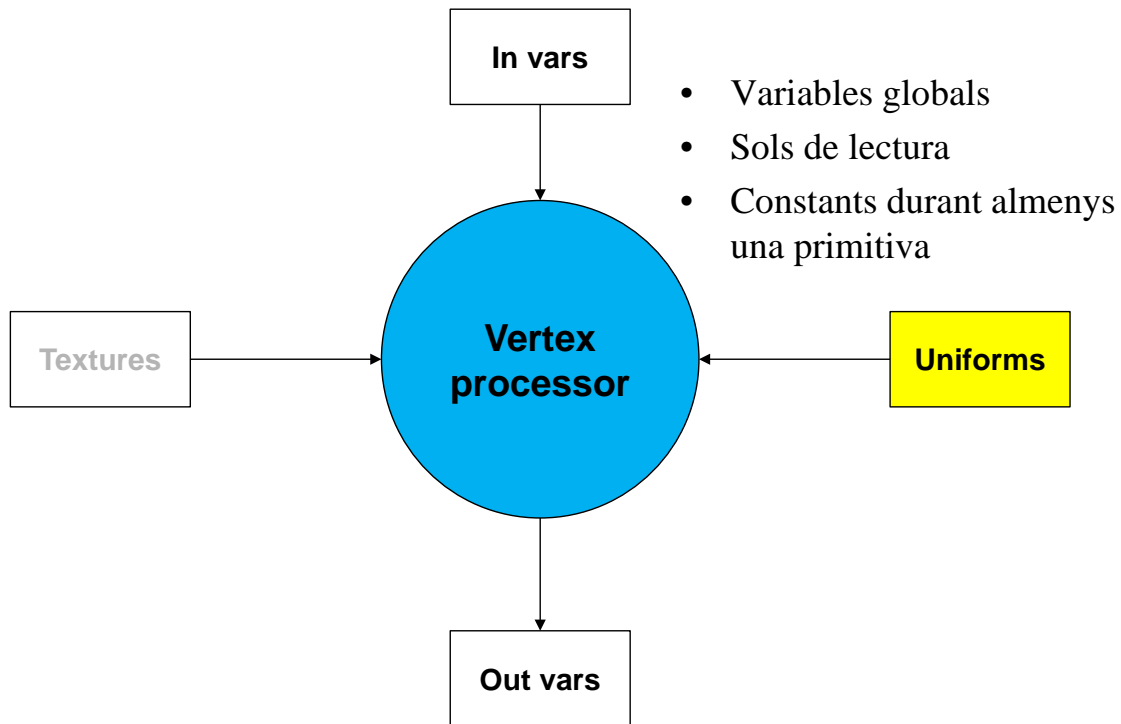


Vertex processor (2)

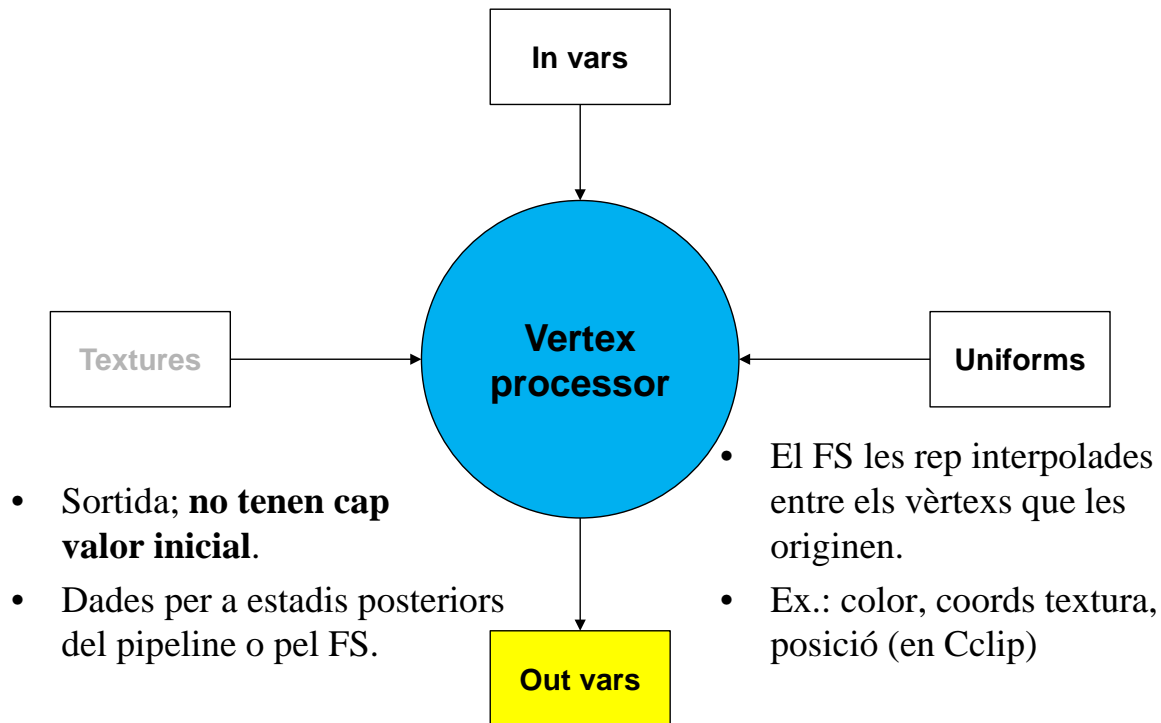
- Atributs del vèrtex
- Sols lectura
- Ex: color, posició, normal, ...



Vertex processor (3)



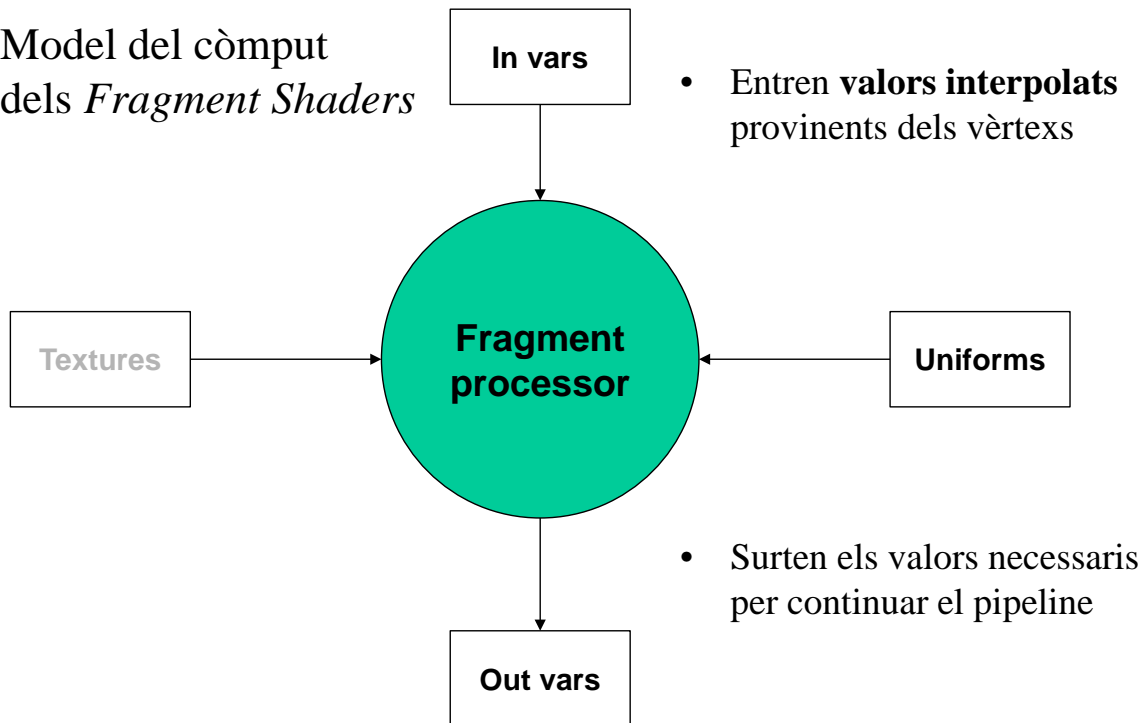
Vertex processor (4)



3.2 Fragment Shader[S1.2][T:9]

Fragment processor

- Model del còmput dels *Fragment Shaders*



3.3 Exemples Vertex i Fragment Shaders[S1.2][T:14,23–24][T:14–16,23–24]

Exemple de vertex shader

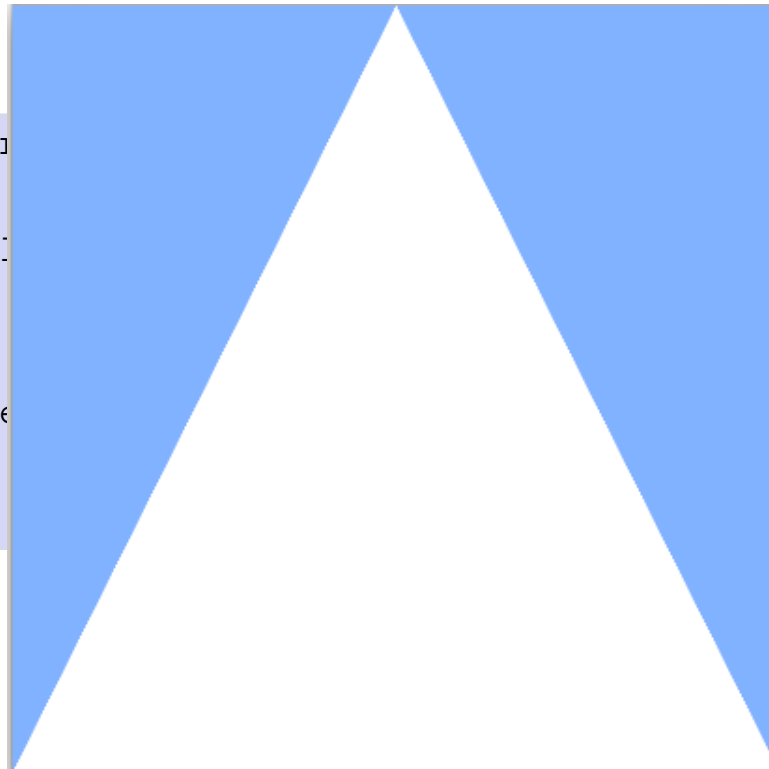
```
1. #version 330 core
2.
3. in vec3 vertex;
4.
5. void main() {
6.     gl_Position = vec4(vertex, 1.0);
7. }
```

Exemple de fragment shader

```
1. #version 330 core
2.
3. out vec4 FragColor;
4.
5. void main() {
6.     FragColor = vec4(1.);
7. }
```

Exemple de fragment shader

```
1. #version 330 core
2.
3. out vec4 FragColor;
4.
5. void main() {
6.     FragColor = vec4(1.0, 1.0, 1.0, 1.0);
7. }
```



Un autre exemple de VS + FS

```
1. #version 330 core
2. in vec3 vertex;
3. void main() {
4.     gl_Position = vec4(vertex, 1.0);
5. }
```

**Vertex
Shader**

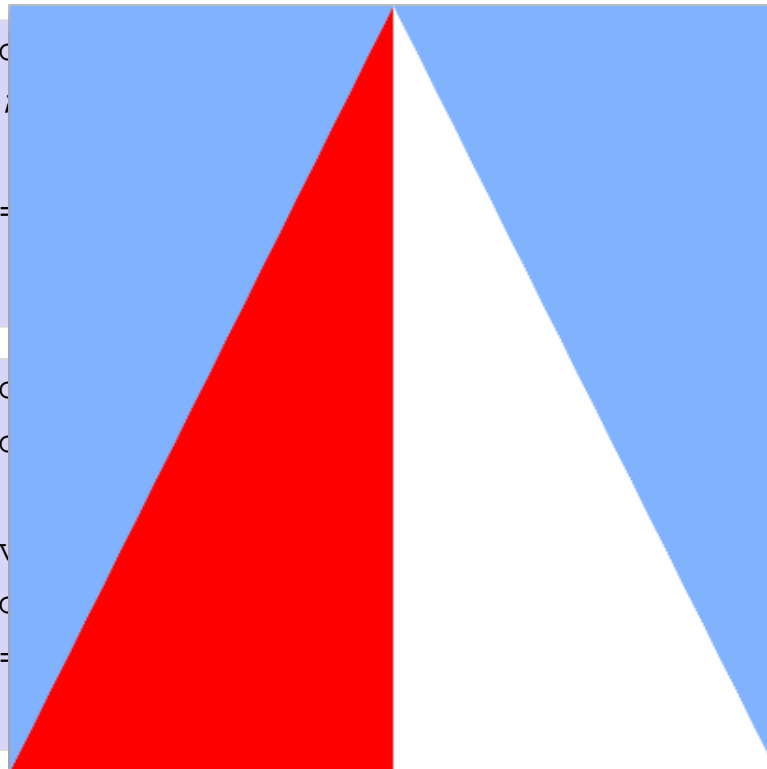
```
1. #version 330 core
2. out vec4 FragColor;
3. void main() {
4.     FragColor = vec4(1.);
5.     if (gl_FragCoord.x < 354.)
6.         FragColor = vec4(1.,0.,0.,1);
7. }
```

**Fragment
Shader**

Un autre exemple de VS + FS

```
1. #version 330 core
2. in vec3 vertexColor;
3. void main() {
4.     gl_Position =
5. }
```

```
1. #version 330 core
2. out vec4 FragColor;
3. void main() {
4.     FragColor =
5.     if (gl_FragCoord
6.         FragColor =
7. }
```




GLSL

- OpenGL Shading Language (GLSL) és el llenguatge de programació d'alt nivell per programar shaders.
- Sintaxi basada en C.
- GLSL permet:
 - Compatibilitat multiplataforma.
 - Escriure shaders que es poden usar en qualsevol tarjeta gràfica de qualsevol fabricant que suporti GLSL.

Evolució GLSL

Versió GLSL	Versió OpenGL	Data	Incorpora
1.10	2.0	Abril 2004	Vertex i fragment shaders
1.20	2.1	Setembre 2006	
1.30	3.0	Agost 2008	Core and compatibility profiles in, out, inout
1.40	3.1	Març 2009	
1.50	3.2	Agost 2009	Geometry shaders
	3.3	Febrer 2010	
	4.0	Març 2010	Tessellation shaders
	
	4.6	Juliol 2017	

Evolució GLSL

Versió GLSL	Versió OpenGL	Data	Incorpora
1.10	2.0	Abril 2004	Vertex i fragment shaders
1.20	2.1	Setembre 2006	
1.30	3.0	Agost 2008	Core and compatibility profiles in, out, inout
1.40	3.1	Març 2009	
1.50	3.2	Agost 2009	Geometry shaders
	3.3	Febrer 2010	
	4.0	Març 2010	Tessellation shaders
	...		
	4.6	Juliol 2017	

GLSL: Tipus de dades

- Tipus bàsics
 - **Escalars:** void, int, uint, float, bool
 - **Vectorials:** vec2, vec3, vec4, mat2, mat3, mat4, mat2x3, ..., ivec3, bvec4, uvec2, ...
- Altres
 - **Arrays:** mat2 mat[3];
 - **Structs:**

```
1. struct light {  
2.     vec3 color;  
3.     vec3 pos;  
4. };
```

Els structs defineixen implícitament constructors:

```
light l1(col, p);
```

GLSL: Operacions

- Inicialitzacions variables
 - **tipus bàsics:** `float b = 2.6;`
 - **vectorials:** `vec3 p(1.,1.5,2.);`
`p = vec3(3.,2.5,1.);`
 - **matrius:** per columnes `mat2 m=mat2(1.,2.,3.,4.);`
`m = [1., 3.]`
`[2., 4.]`
- Accés als elements de vectors
 - **swizzling:** `vec4 v;`
`v.xyzw; / v.rgba; // vec4`
`v.xy; / v.rg; // vec2`
`v.zyx; // canvia ordre!`
- Operacions vectors - multiplicació component a component

GLSL: Funcions predefinides

- Moltes funcions predefinides, especialment en àrees que poden interessar quan tractem geometria o volem dibuixar:
 - **trigonomètriques**
`radians()`, `degrees()`, `sin()`, `cos()`, `tan()`, `asin()`,
`acos()`, `atan()` (amb 1 o 2 paràmetres)
 - **numèriques** (poden operar sobre vectors component a comp.)
`pow()`, `log()`, `exp()`, `abs()`, `sign()`, `floor()`, `min()`,
`max()`
 - **sobre vectors i punts**
`length()`, `distance()`, `dot()`, `cross()`, `normalize()`

GLSL: Crear noves funcions

- Es poden definir noves funcions usant una sintaxi similar a C.
 - COMPTE amb l'eficiència!! Els paràmetres es copien.
 - Hi ha tres tipus de paràmetres: **in** (default), **out**, **inout**

```
1.  vec4 exemple(in vec4 a, float b) { ... }  
2.  
3.  float[6] exemple(out vec3 inds) { ... }  
4.  
5.  void altreExemple(in float a, inout bool flag)  
6.  { ... }
```

3.5 Variables Predefinides[S1.2][T:21]

GLSL: Variables pre-definides

- **No cal declarar-les!**
- Vertex shader

```
1. out vec4 gl_Position;
```

- Fragment shader

```
1. in vec4 gl_FragCoord;  
2. out float gl_FragDepth;
```

3.6 Discard[S1.2][T:22]

GLSL: discard

- `discard` és una instrucció especial pels **fragment shaders**.
- Aquesta instrucció descarta el fragment (i conclou l'execució).

```
1. discard;
```

3.7 Gestió de Shaders[S1.2][T:28–39]

3.7.1 Procès de càrrega[S1.2][T:28–37]

Classes Qt per gestionar shaders

- Els shaders que usará el nostre programa s'han d'indicar en la classe `MyGLWidget`.
- Usarem les següents classes Qt per fer-ho:
 - **QOpenGLShader**: Ofereix un embolcall per a cadascun dels shaders del nostre programa, i gestiona la seva definició, compilació i vinculació a un *Shader Program*.
 - **QOpenGLShaderProgram**: Permet agrupar uns shaders dissenyats per a funcionar conjuntament, i muntar un Shader Program.

Carrega shaders (1)

```
1. void MyGLWidget::carregaShaders() {  
2.     QOpenGLShader fs (QOpenGLShader::Fragment, this);  
3.     QOpenGLShader vs (QOpenGLShader::Vertex, this);  
  
1.     fs.compileSourceFile("shaders/fragshad.frag");  
2.     vs.compileSourceFile("shaders/vertshad.vert");  
  
3.     program = new QOpenGLShaderProgram(this);  
  
4.     program->addShader(&fs);  
5.     program->addShader(&vs);  
  
6.     program->link();  
  
7.     program->bind();  
8.     ...  
9. }
```

**Creem els shaders pel
fragment shader i el
vertex shader**

Carrega shaders (2)

```
1. void MyGLWidget::carregaShaders() {  
2.     QOpenGLShader fs (QOpenGLShader::Fragment, this);  
3.     QOpenGLShader vs (QOpenGLShader::Vertex, this);  
  
4.     fs.compileSourceFile("shaders/fragshad.frag");  
5.     vs.compileSourceFile("shaders/vertshad.vert");  
  
6.     program = new QOpenGLShaderProgram(this);  
  
7.     program->addShader(&fs);  
8.     program->addShader(&vs);  
  
9.     program->link();  
  
10.    program->bind();  
11.    ...  
12. }
```

**Carreguem el codi dels
fitxers i els compilem**

Carrega shaders (3)

```
1. void MyGLWidget::carregaShaders() {  
2.     QOpenGLShader fs (QOpenGLShader::Fragment, this);  
3.     QOpenGLShader vs (QOpenGLShader::Vertex, this);  
  
4.     fs.compileSourceFile("shaders/fragshad.frag");  
5.     vs.compileSourceFile("shaders/vertshad.vert");  
  
6.     program = new QOpenGLShaderProgram(this);  
  
7.     program->addShader(&fs);  
8.     program->addShader(&vs);  
  
9.     program->link();  
  
10.    program->bind();  
11.    ...  
12. }
```

Creem el program



Carrega shaders (4)

```
1. void MyGLWidget::carregaShaders() {  
2.     QOpenGLShader fs (QOpenGLShader::Fragment, this);  
3.     QOpenGLShader vs (QOpenGLShader::Vertex, this);  
  
4.     fs.compileSourceFile("shaders/fragshad.frag");  
5.     vs.compileSourceFile("shaders/vertshad.vert");  
  
6.     program = new QOpenGLShaderProgram(this);  
  
7.     program->addShader(&fs);  
8.     program->addShader(&vs);  
  
9.     program->link();  
  
10.    program->bind();  
11.    ...  
12. }
```

**Afegim al program els
shaders creats abans**

Carrega shaders (5)

```
1. void MyGLWidget::carregaShaders() {  
2.     QOpenGLShader fs (QOpenGLShader::Fragment, this);  
3.     QOpenGLShader vs (QOpenGLShader::Vertex, this);  
  
4.     fs.compileSourceFile("shaders/fragshad.frag");  
5.     vs.compileSourceFile("shaders/vertshad.vert");  
  
6.     program = new QOpenGLShaderProgram(this);  
  
7.     program->addShader(&fs);  
8.     program->addShader(&vs);  
  
9.     program->link();  
10.    program->bind();  
11.    ...  
12. }
```

Linkem el program



Carrega shaders (6)

```
1. void MyGLWidget::carregaShaders() {  
2.     QOpenGLShader fs (QOpenGLShader::Fragment, this);  
3.     QOpenGLShader vs (QOpenGLShader::Vertex, this);  
  
4.     fs.compileSourceFile("shaders/fragshad.frag");  
5.     vs.compileSourceFile("shaders/vertshad.vert");  
  
6.     program = new QOpenGLShaderProgram(this);  
  
7.     program->addShader(&fs);  
8.     program->addShader(&vs);  
  
9.     program->link();  
  
10.    program->bind();  
11.    ...  
12. }
```

**Indiquem que aquest és el
program que volem usar**

Comunicar informació

CPU → shader

- Cal indicar en el nostre programa com passar informació al shader.
- Cal enllaçar els atributs d'entrada del shader a la nostra classe C++, és a dir, obtenir la posició de l'atribut a través del seu nom.
- Això es fa mitjançant un **attrib location** per a cada atribut d'entrada del shader. Per exemple:

```
1. vertexLoc = glGetUniformLocation (program->programId(),  
    "vertex" );
```

- **Aquest pas només cal fer-lo un cop per a cada atribut d'entrada.**

Detalls del mètode

GLint glGetAttribLocation (GLuint *program*, const GLchar **name*);

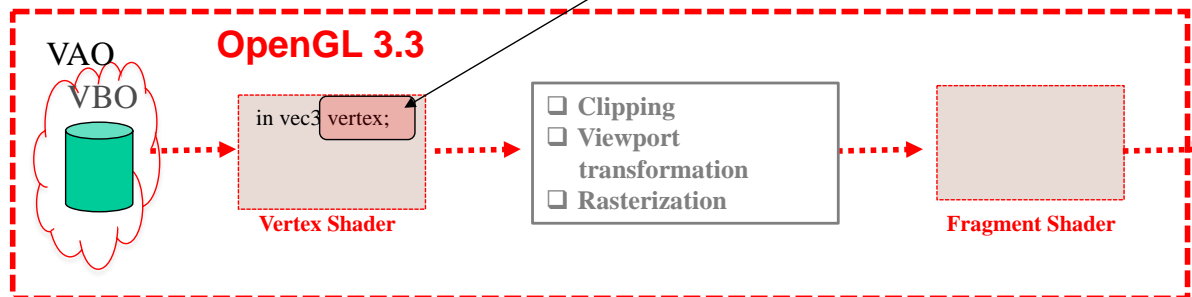
Retorna l'identificador que lliga amb l'atribut definit en el Vertex Shader

program : identificador del program

name : nom de l'atribut en el Vertex Shader

```
1. vertexLoc = glGetAttribLocation (program->programId(),  
    "vertex");
```

Lligam entre els dos



Comunicar informació

CPU → shader

- Un cop tenim l'identificador de l'atribut hem de lligar l'atrib location amb el buffer corresponent:

```
1. glVertexAttribPointer(vertexLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

- I a continuació activar-lo:

```
1. glEnableVertexAttribArray(vertexLoc);
```

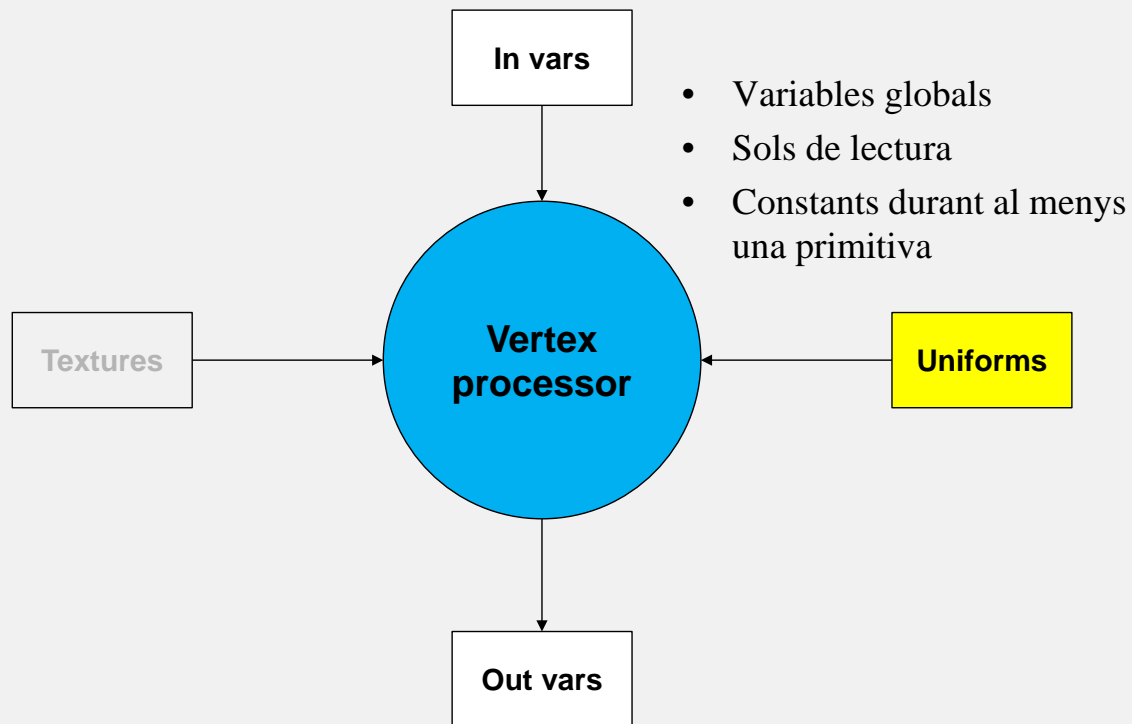
Exemple de vertex shader amb dos atributs d'entrada

```
1. #version 330 core
2. in vec3 vertex;
3. in vec3 color;
4. out vec3 fcolor;
5.
6. void main() {
7.     fcolor = color;
8.     gl_Position = vec4(vertex, 1.0);
9. }
```

- Penseu vosaltres com ha de ser el codi corresponent al Fragment Shader

3.8 Uniforms[S1.3][T:3–9]

Uniforms



Uniforms

- All vertex shader:

```
#version 330 core
```

```
in vec3 vertex;
```

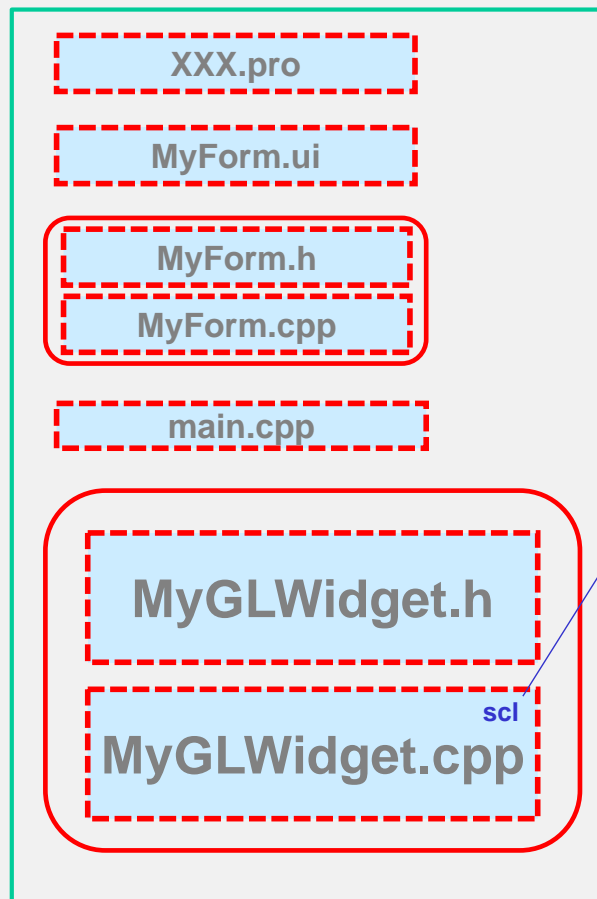
```
uniform float val;
```

```
void main ()
```

```
{
```

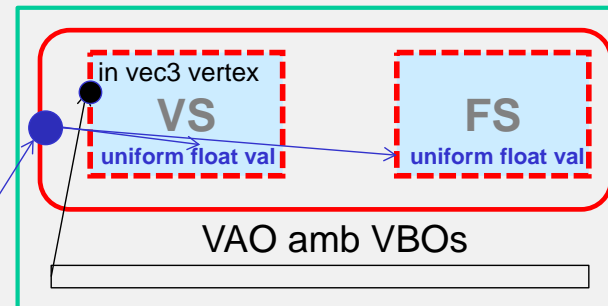
```
    gl_Position = vec4 (vertex * val, 1.0);
```

```
}
```



Compilació: qmake-qt5
make

Execució: QOpenGLShader
QOpenGLShaderProgram



varLoc= **glGetUniformLocation**
(program → programId(), "val");

glUniform1f (varLoc, scl);

Uniforms

- Al codi cpp de MyGLWidget:
 - Associar identificador al shader (només cal fer-ho un cop)
`varLoc = glGetUniformLocation (program->programId (), "val");`
 - Donar valor al uniform (cal fer-ho cada cop que es vulgui canviar el valor del paràmetre *scl*)
`glUniform1f (varLoc, scl);`
`// scl variable que conté el valor que es vol per "val"`

Funcions OpenGL per a uniforms

`GLint glGetUniformLocation (GLuint program, const GLchar *name);`

Obté la posició d'un uniform declarat al shader amb nom *name*

program : program al que està lligat el shader que conté el uniform

name : nom que identifica al uniform en el shader

`void glUniform1f (GLint location, GLfloat value);`

Especifica el valor *value* per al uniform identificat per *location*

location : identificador del uniform aconseguit amb glGetUniformLocation

value : valor que es passa cap al shader

Funcions OpenGL per a uniforms

Altres crides possibles:

`glUniform{1|2|3|4}{f|i|ui} // nombre de paràmetres depenent de 1|2|3|4`

1 – tipus float (f), int (i), unsigned int (ui), bool (f|i|ui)

2 – tipus vec2 (f), ivec2 (i), uvec2 (ui), bvec2 (f|i|ui)

3 – tipus vec3 (f), ivec3 (i), uvec3 (ui), bvec3 (f|i|ui)

4 – tipus vec4 (f), ivec4 (i), uvec4 (ui), bvec4 (f|i|ui)

`glUniform{1|2|3|4}{f|i|ui}v (GLint loc, GLsizei count, const Type *value);`

{1|2|3|4} i {f|i|ui} – igual que crida anterior

count – nombre d'elements de l'array *value*, 1: un sol valor; >=1 array de valors

`glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv`

`(GLint loc, GLsizei count, GLboolean transpose, const GLfloat *value);`

{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3} – defineix les dimensions de la matriu

count – nombre de matrius de l'array *value*

transpose – si la matriu s'ha de transposar

Funcions OpenGL per a uniforms

Les que més usarem:

`glUniform1{f|i|ui}` // per a passar un únic valor

Exemple:

```
float scl = 0.5;  
glUniform1f (varLoc, scl);
```

`glUniform3fv` // per a passar vectors de 3 components

Exemple:

```
glm::vec3 posLlum = glm::vec3 (1.0, 5.0, 0.0);  
glUniform3fv (posLlumLoc, 1, &posLlum[0]);
```

`glUniformMatrix4fv` // per a passar les matrius de transformació

Exemple:

```
glm::mat4 TG = glm::mat4(1.0);  
glUniformMatrix4fv (transLoc, 1, GL_FALSE, &TG[0][0])
```

4 Interacció

4.1 Baix Nivell[S1.3][T:10–15]

Interacció directa amb Qt

- Per tal de tractar events de baix nivell en una aplicació OpenGL amb Qt cal re-implementar els mètodes virtuals corresponents (a la classe `MyGLWidget`):

```
virtual void mousePressEvent ( QMouseEvent * e )
```

```
virtual void mouseReleaseEvent ( QMouseEvent * e )
```

```
virtual void mouseMoveEvent ( QMouseEvent * e )
```

```
virtual void keyPressEvent ( QKeyEvent * e )
```

Interacció directa amb Qt

- Per tal de tractar events de baix nivell en una aplicació OpenGL amb Qt cal re-implementar els mètodes virtuals corresponents (a la classe `MyGLWidget`):

```
virtual void mousePressEvent ( QMouseEvent * e )
```

```
virtual void mouseReleaseEvent ( QMouseEvent * e )
```

```
virtual void mouseMoveEvent ( QMouseEvent * e )
```

```
virtual void keyPressEvent ( QKeyEvent * e )
```

Interacció directa amb Qt

- Exemple d'implementació:

```
void MyGLWidget::keyPressEvent (QKeyEvent *e) {  
    makeCurrent ();  
    switch ( e->key() ) {  
        case Qt::Key_S :  
            scl += 0.1;  
            glUniform1f (varLoc, scl);  
            break;  
        case Qt::Key_D :  
            scl -= 0.1;  
            glUniform1f (varLoc, scl);  
            break;  
        default: e->ignore (); // propagar al pare  
    }  
    update ();  
}
```

En MyGLWidget.h caldrà afegir:

```
#include <QKeyEvent>
```

i declarar el mètode virtual

```
virtual void keyPressEvent (QKeyEvent *e);
```

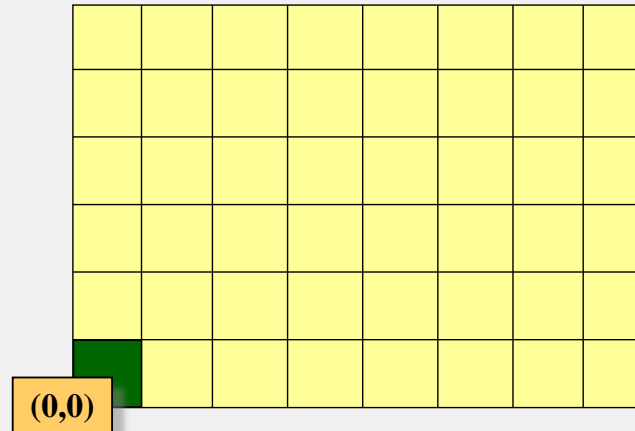
Interacció directa amb Qt

- Exemple d'implementació:

```
void MyGLWidget::keyPressEvent (QKeyEvent *e) {  
→ makeCurrent ();    // fa actiu el nostre context d'OpenGL  
  switch ( e->key() ) {  
    case Qt::Key_S :  
      scl += 0.1;  
      glUniform1f (varLoc, scl);  
      break;  
    case Qt::Key_D :  
      scl -= 0.1;  
      glUniform1f (varLoc, scl);  
      break;  
    default: e->ignore (); // propagar al pare  
  }  
→ update ();    // provoca que es torni a pintar l'escena  
}
```

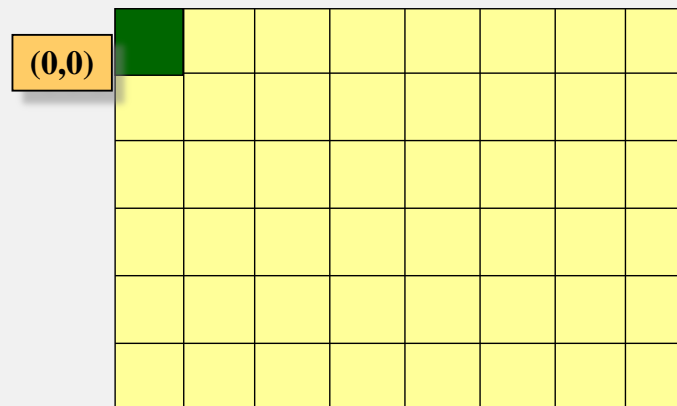
Consideració important

- OpenGL considera l'origen del SC de dispositiu a la cantonada inferior esquerra de la finestra gràfica.



Consideració important

- Qt considera l'origen del SC de dispositiu a la cantonada superior esquerra de la finestra gràfica.



4.2 Qt-Widgets

4.2.1 Layouts[Qt1][T:7]

Els Layouts

Els **layout** (disposicions) permeten organitzar els components visuals dintre de formularis i quadres de diàleg.



Horitzontal
(QHBoxLayout)



Vertical
(QVBoxLayout)



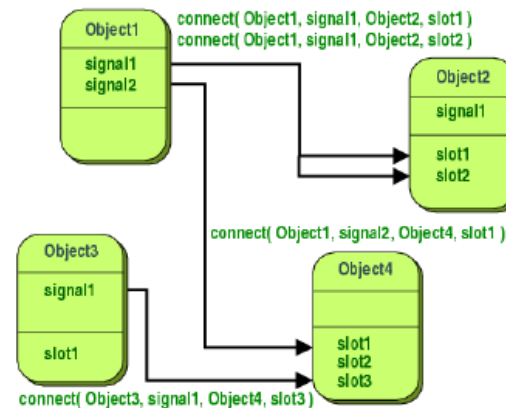
En graella
(QGridLayout)

Signals i slots

- Per tal de connectar la interfície gràfica que dissenyem amb la nostra aplicació, caldrà connectar els elements gràfics Qt al nostre codi C++.
 - Les connexions poden ser:
 - Alt nivell: associades als components
 - Baix nivell: events bàsics del computador
 - **Signal:** Esdeveniment que succeeix durant l'execució de l'aplicació.
 - Ex: Clic sobre un widget...
 - **Slot:** mètode especial d'una classe que es pot connectar amb signals.
-

Signals i slots

Els signals i els slots **són mètodes** que s'usen per a la comunicació entre objectes. Qualsevol classe que hereti de QObject (o de les seves subclasses), pot contenir signals i slots. Per tant, qualsevol classe de Qt conté signals i slots.



Els **signals** es *llancen* quan es produeix un esdeveniment en l'aplicació (pex. clicar botó).

Els **slots** s'executen quan es produeix un signal.

Signals i slots

- La informació que circula entre signals i slots viatja a través dels **paràmetres**.
 - Els slots tenen paràmetres que venen carregats de dades, les que envia el signal.
 - Pot haver **més d'un slot connectat a un mateix signal**, de manera que quan s'emeti un signal, s'executaran tots els seus slots; no podem saber, però, en quin ordre.
-

Signals i slots

- En el directori

`/assig/idi/Qt/S1-IntroQt`

trobareu un fitxer `lab0.pro` i un `lab0.cpp`

- `lab0.pro` serveix per a descriure com és el vostre projecte: els fitxers que el componen, les llibreries que cal enllaçar...
 - Podeu executar-lo fent `./lab0`
 - Mireu el codi i intenteu entendre el que fa
-

4.2.3 Disseny[Qt1][T:12]

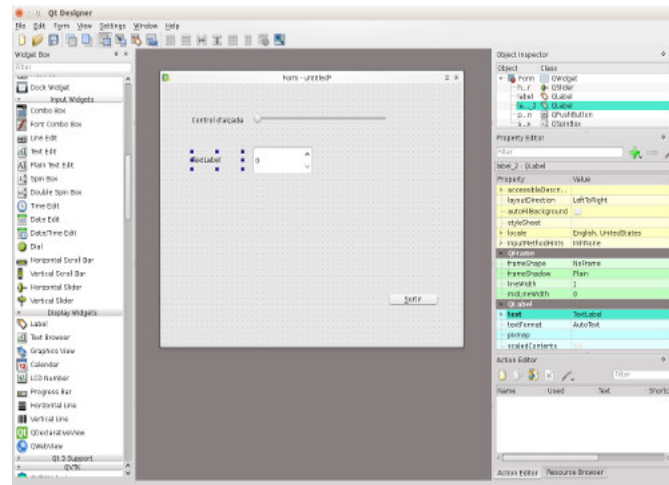
Disseny interfície

- Eina gràfica per a dissenyar aplicacions amb Qt:

designer

(al laboratori

designer-qt5)

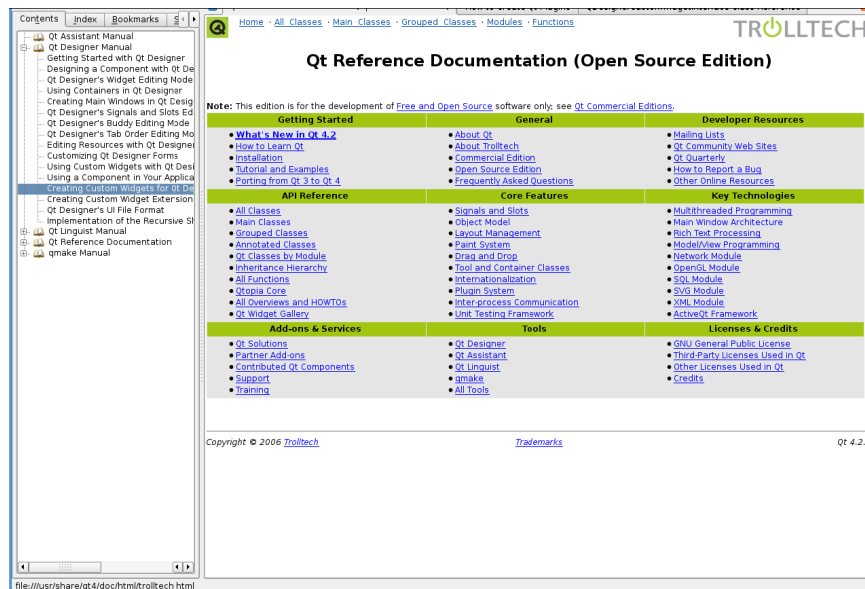


- Demo explicació
- Fitxers necessaris per a compilació (altres transpes)
- Mireu-vos apartats 1 i 2 del document (Apunts-Qt.pdf)

4.2.4 Assistant[Qt1][T:13]

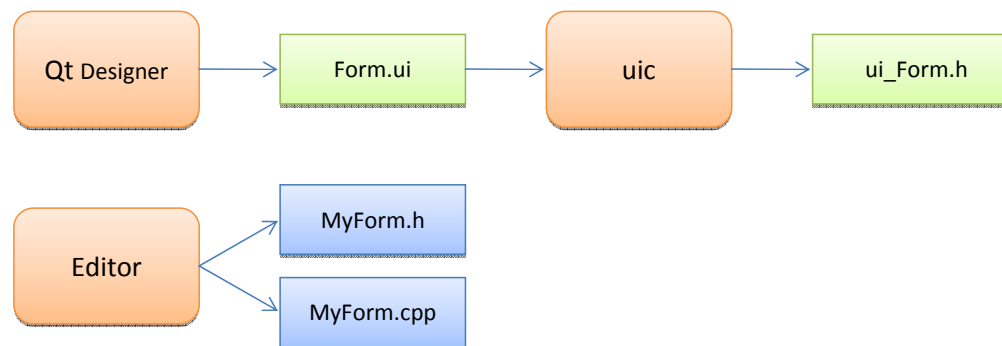
Ajuda

- Més informació usant la comanda:
assistant& (al laboratori assistant-qt5)

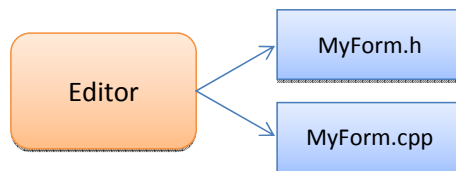
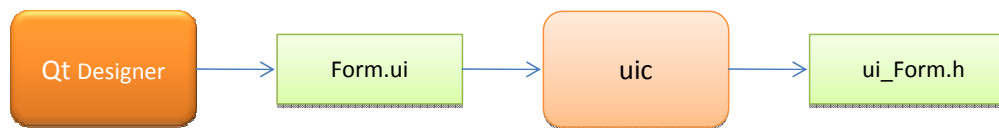


4.2.5 Compilació[Fitxers per Compilar]

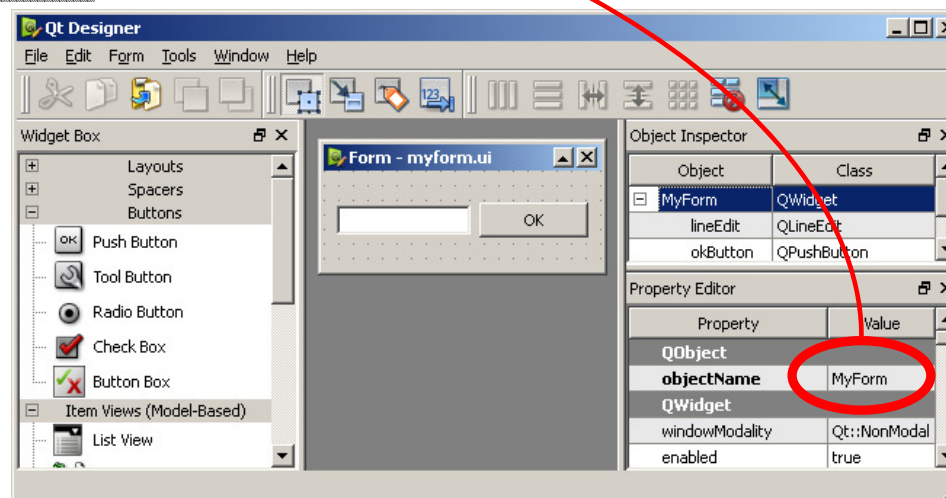
Fitxers per a compilar interfícies del Designer



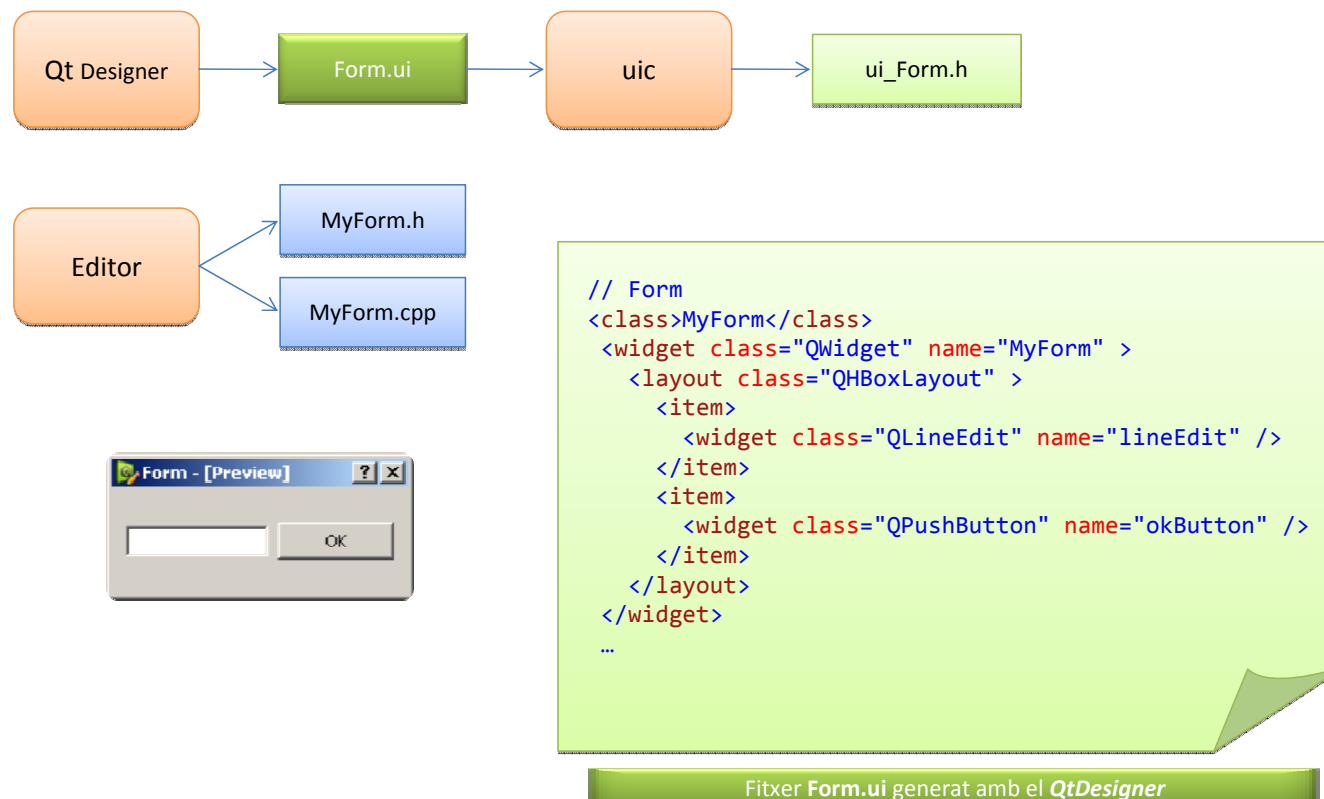
Fitxers per a compilar interfícies del Designer



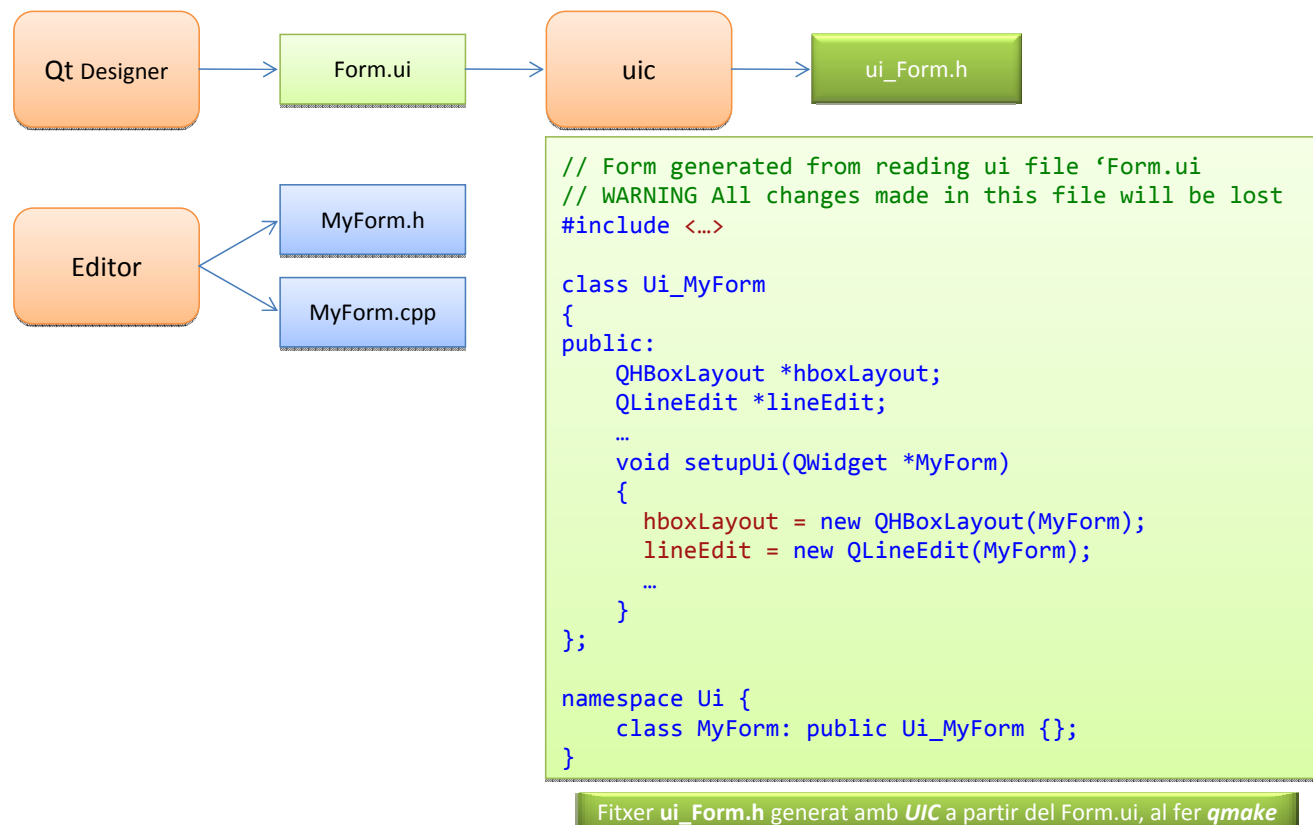
El nom del formulari es farà servir a MyForm.h



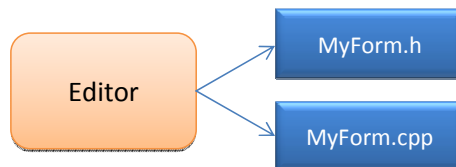
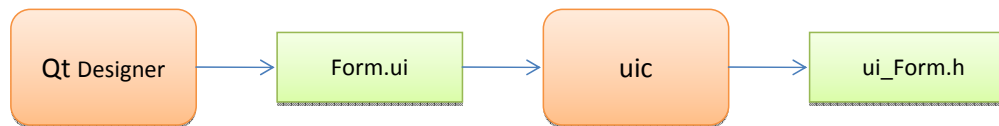
Fitxers per a compilar interfícies del Designer



Fitxers per a compilar interfícies del Designer



Usant agregació

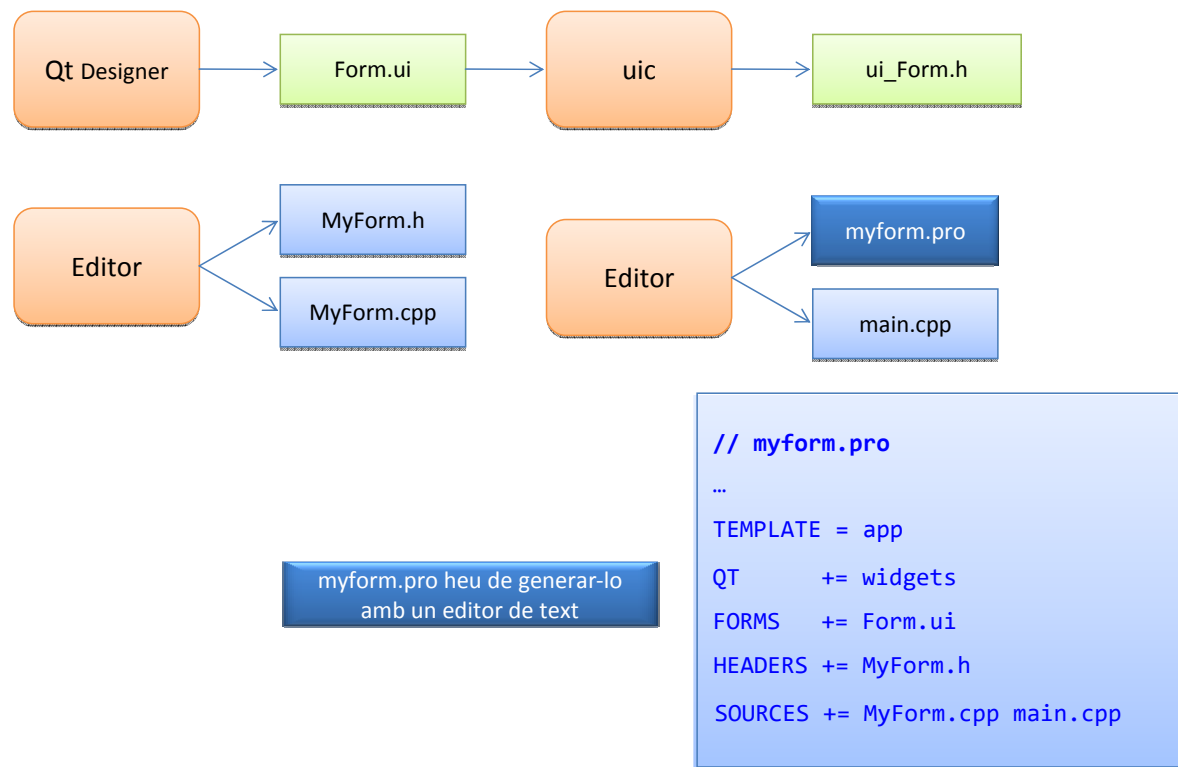


```
// MyForm.h – Ui::MyForm com a membre  
#include "ui_Form.h"  
  
class MyForm : public QWidget  
{  
    Q_OBJECT  
  
public:  
    MyForm(QWidget *parent = 0);  
  
private:  
    Ui::MyForm ui;  
};
```

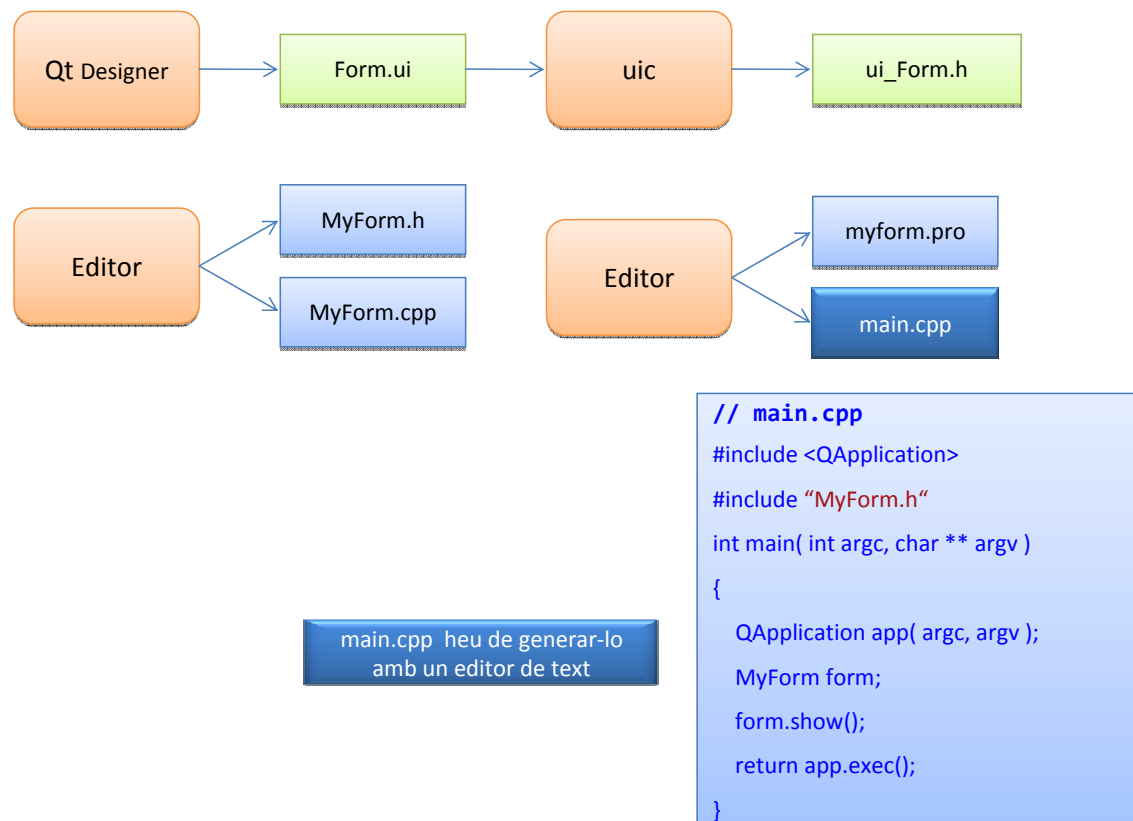
```
// MyForm.cpp  
#include "MyForm.h"  
  
MyForm::MyForm(QWidget* parent): QWidget(parent)  
{  
    ui.setupUi(this);  
}
```

Fixeu-vos que el nom d'aquesta variable ha de coincidir amb el nom del formulari al Designer!

Fitxers per a compilar interfícies del Designer



Fitxers per a compilar interfícies del Designer



Compilant tot



```
> qmake  
> make
```

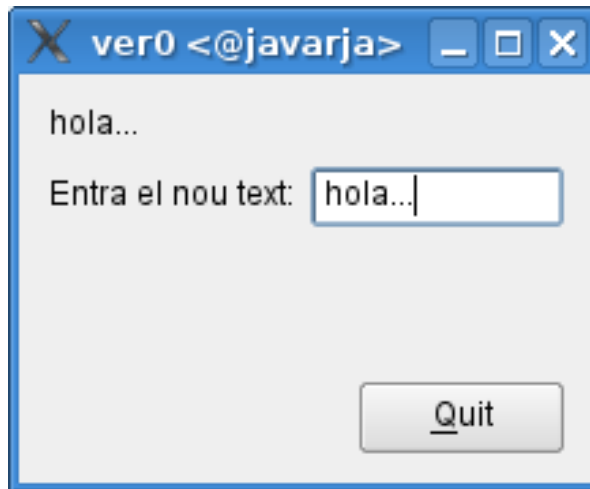
(al laboratori qmake-qt5)

Vigileu si copieu i pegueu!! les cometes dels #include "nomfitxer.h" poden donar problemes al compilar! Assegureu-vos de que utilitzeu les cometes que hi ha a la tecla del numero 2

4.3 Qt-Custom Widgets[Qt2]

Llibreria Qt: No tot es pot fer directe

- Si volem que només copiï el text a l'etiqueta quan es fa *<return>*...



Signals QLineEdit:

- returnPressed ()
- textChanged (QString)

Slots QLabel:

- setText (QString)

NO ES POT FER!

Llibreria Qt: Classes pròpies

- En algunes ocasions ens caldrà crear les nostres pròpies classes derivades de les de Qt per a programar els slots i afegir els signals que calguin. Podem derivar de:
 - `QObject` (per a objectes no gràfics)
 - `QWidget` o qualsevol de les seves derivades (per a dissenyar nous components gràfics amb noves funcionalitats)
-

Exemple: MyLineEdit.h

```
#include <QLineEdit>

class MyLineEdit: public QLineEdit
{
    Q_OBJECT    ←----- IMPORTANT
public:
    MyLineEdit (QWidget *parent);
public slots:    ←----- IMPORTANT
    void tractaReturn ();
signals:        ←----- IMPORTANT
    void enviaText (const QString &);
};
```

Els slots els implementarem a
[MyLineEdit.cpp](#)

Els signals no els implementem
però es poden llençar en
qualsevol punt del codi cridant a
la funció:

emit nom_signal(paràmetres)

Exemple: MyLineEdit.cpp

```
#include "MyLineEdit.h"

// constructor
MyLineEdit::MyLineEdit(QWidget *parent)
    : QLineEdit(parent) {
    connect(this, SIGNAL(returnPressed()), this, SLOT(tractaReturn()));
    // Inicialització d'atributs si cal
}

// implementació slots
void MyLineEdit::tractaReturn() {
    // Implementació de tractaReturn
    emit enviaText (text());
}
```

El constructor ha de cridar al constructor de la classe base

La implementació del slot només ha de produir el nou signal enviant el text.

Llibreria Qt: Classes pròpies

Per a compilar la classe MyLineEdit

No és codi C++ → Necessita ser preprocessat amb el meta-object compiler (MOC):

Ho fa automàticament el Makefile si ho afegim al .pro

- Afegir MyLineEdit.h al HEADERS del .pro
- Afegir MyLineEdit.cpp al SOURCES del .pro

Per a usar un objecte d'aquesta nova classe al designer:

- promote...
-

Llibreria Qt: La classe MyGLWidget

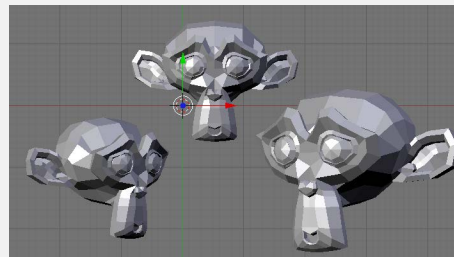
- Com podeu veure, la nostra classe d'OpenGL MyGLWidget, en realitat és una classe pròpia derivada de QOpenGLWidget de Qt...
 - Podeu veure que el .h inclou la macro Q_OBJECT
 - I que tenim el fitxer .h en el tag HEADERS del .pro
 - Per tant podem usar-la per a afegir comportament si volem que es pugui lligar amb altres components de Qt (és a dir, podem afegir-li signals i slots)
 - Recordeu afegir el “makeCurrent ()” al principi de qualsevol slot que hagi d'usar codi OpenGL.
-

5 Transformacions Geomètriques

5.1 Matrius de Transformació[S1.3][T:16–18]

Matrius de transformació

- Hem de poder transformar els vèrtexs (pex, amb transformacions de model):



$M = I,$
 TG_1, TG_2



V_M

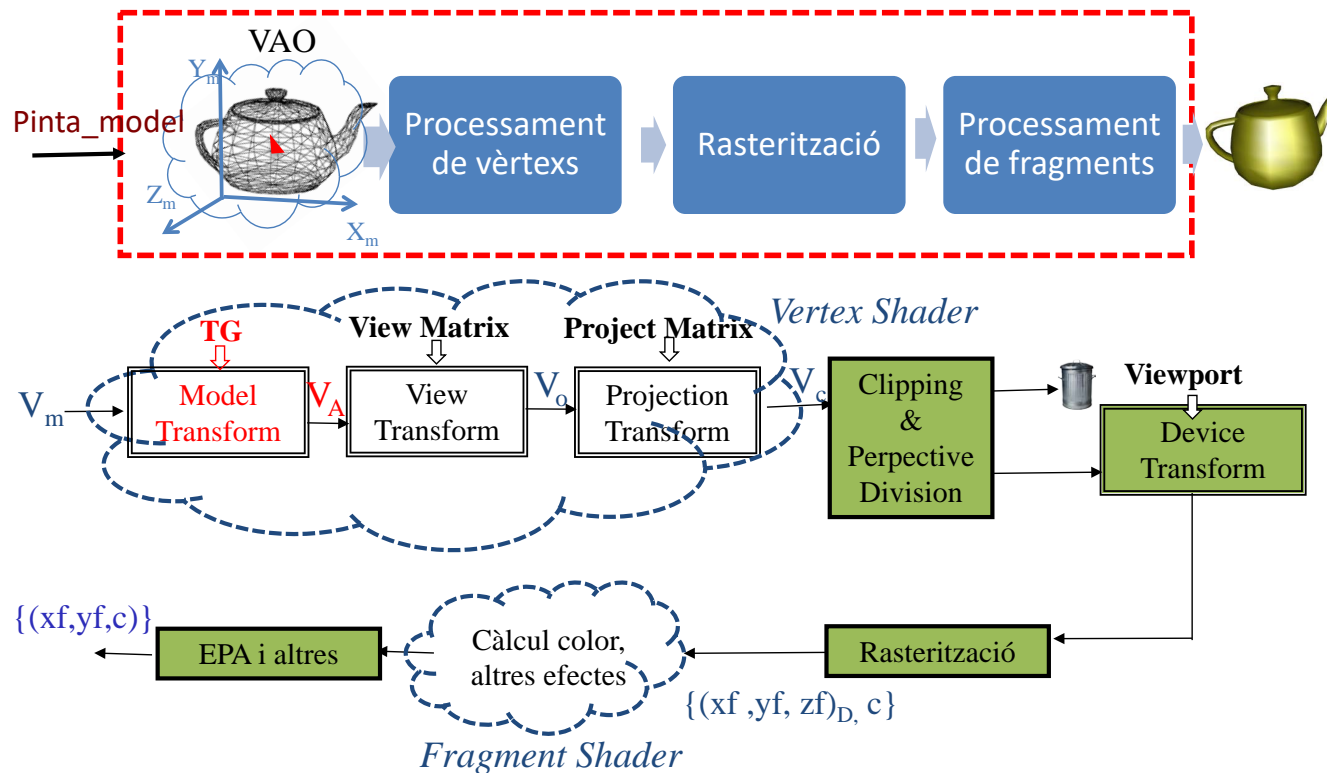
TG



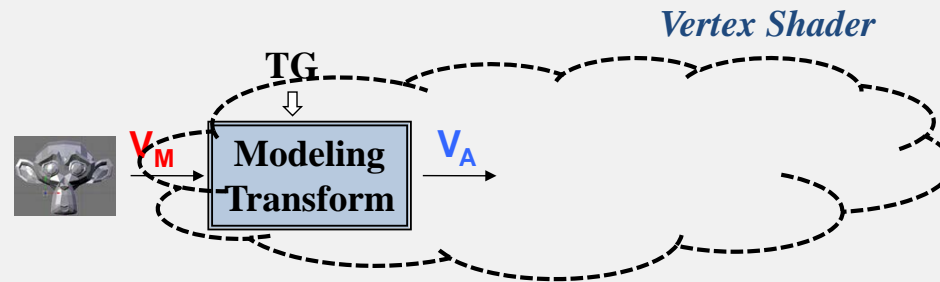
Modeling
Transform

V_A

Paradigma projectiu bàsic amb OpenGL 3.3



Matrius de transformació



- Cal passar la matriu al Vertex Shader (com a uniform):

```
in vec 3 vertex;  
uniform mat4 TG;  
void main () {  
    gl_Position = TG * vec4 (vertex, 1.0);  
}
```
- I lligar el uniform al nostre programa (en MyGLWidget):

```
GLuint transLoc;  
transLoc = glGetUniformLocation (program->programId(), "TG");
```

Matrius de transformació

- Usarem glm per construir la matriu de transformació:
- Exemple:

```
void MyGLWidget::modelTransform () {  
    glm::mat4 TG (1.0); // Matriu de transformació, inicialment identitat  
    TG = glm::translate (TG, glm::vec3 (-0.5, 0.5, 0.0));  
    glUniformMatrix4fv (transLoc, 1, GL_FALSE, &TG[0][0]);  
}
```

- I completant l'exemple, per pintar farem:

```
void MyGLWidget::paintGL () {  
    glClear (GL_COLOR_BUFFER_BIT); // Esborrem el frame-buffer  
  
    modelTransform ();  
    glBindVertexArray (VAO1);  
    glDrawArrays (GL_TRIANGLES, 0, 3);  
    glBindVertexArray (0);  
}
```


Matrius de transformació

- Mètodes de transformacions geomètriques de la glm:

```
translate (glm::mat4 m_ant, glm::vec3 vec_trans);
```

```
// retorna el producte de m_ant per una matriu que fa una
```

```
// translació pel vector vec_trans
```

```
scale (glm::mat4 m_ant, glm::vec3 vec_scale);
```

```
// retorna el producte de m_ant per una matriu que fa un
```

```
// escalat en cada direcció segons els factors vec_scale
```

```
rotate (glm::mat4 m_ant, float angle, glm::vec3 vec_axe);
```

```
// retorna el producte de m_ant per una matriu que fa una
```

```
// rotació de angle radians al voltant de l'eix vec_axe
```

- Per a poder incloure aquestes funcions de la glm:

```
#include "glm/gtc/matrix_transform.hpp"
```

- Per a que els angles a usar a la rotació siguin en radians ens cal afegir al nostre codi (al fitxer MyGLWidget.h, abans includes de glm) el següent:

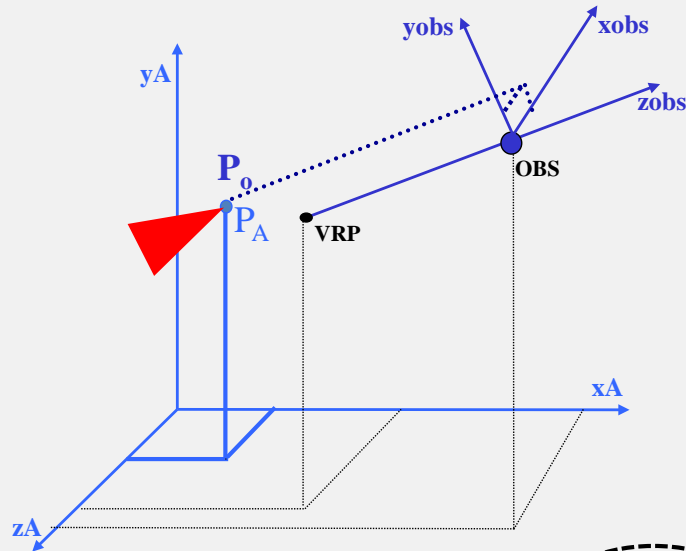
```
#define GLM_FORCE_RADIANS
```

6 Càmera

6.1 View Matrix

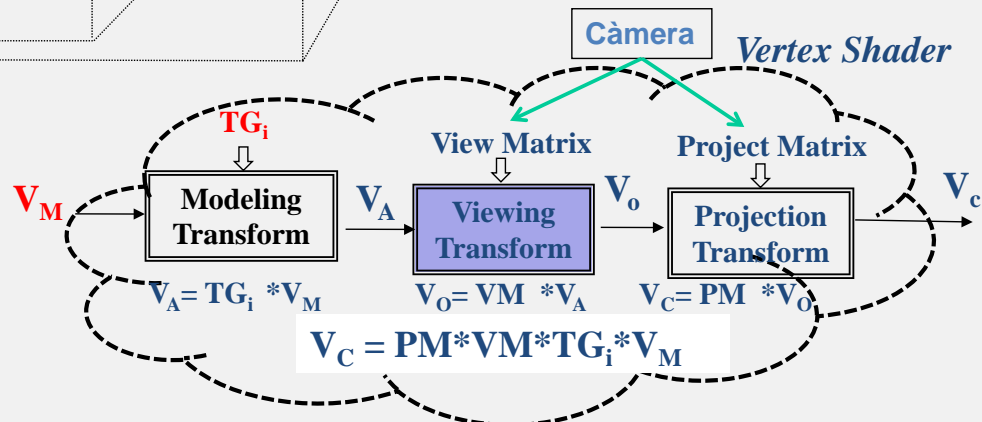
6.1.1 LookAt[S2.1][T:7-10]

Transformació de punt de vista (view)



OBS , VRP, up

```
VM = lookAt (OBS, VRP, up);  
viewMatrix (VM);
```



Transformació de punt de vista (view)

(exercici 2)

- Al codi cpp de MyGLWidget:
 - Demanem un uniform location per al uniform de la matriu
 - Definim un mètode que ens calculi la transformació de punt de vista (view) i envii el uniform amb la matriu cap al vertex shader

```
viewLoc = glGetUniformLocation (program->programId(), "view")
```

```
void MyGLWidget::viewTransform () {  
    // glm::lookAt (OBS, VRP, UP)  
    glm::mat4 View = glm::lookAt (glm::vec3(0,0,1),  
                                   glm::vec3(0,0,0), glm::vec3(0,1,0));  
    glUniformMatrix4fv (viewLoc, 1, GL_FALSE, &View[0][0]);  
}
```

Transformació de punt de vista (view)

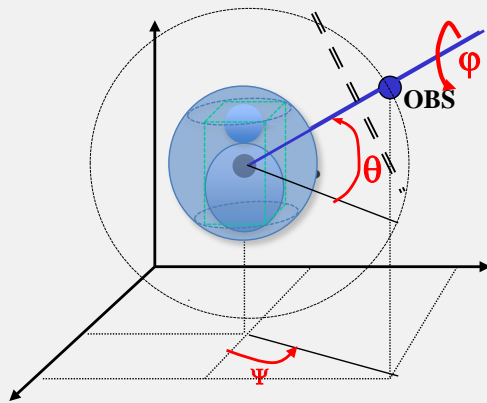
(exercici 2)

- Al vertex shader (afegir):

```
...  
uniform mat4 view;  
...  
void main () {  
    ...  
    gl_Position = proj * view * ... * vec4 (vertex, 1.0);  
}
```

Transf. *view* amb angles d'Euler

(exercici 1)



```
VM=Translate (0.,0.,-d)
VM=VM*Rotate(-\phi,0,0,1)
VM= VM*Rotate (\theta,1,0.,0.)
VM= VM*Rotate(-\psi.,0.,1.,0.)
VM= VM*Translate(-VRP.x,-VRP.y,-VRP.z)
viewMatrix(VM)
```

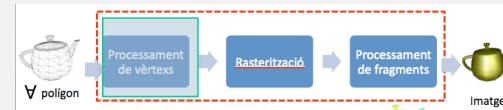
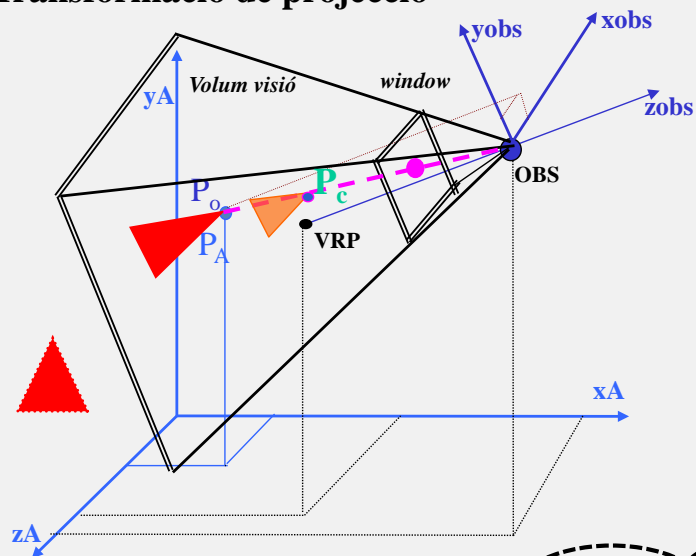
Atenció a l'ordre!

Compte amb signes:

- Si s'ha calculat ψ positiu quan càmera gira cap a la dreta, serà un gir anti-horari respecte eix Y de la càmera, per tant, matemàticament positiu; com girem els objectes en sentit contrari, cal posar $-\psi$ en el codi.
- Si s'ha calculat θ positiu quan pugem la càmera, serà un gir horari; per tant, matemàticament un gir negatiu; com objecte girarà en sentit contrari (anti-horari), ja és correcte deixar signe positiu.

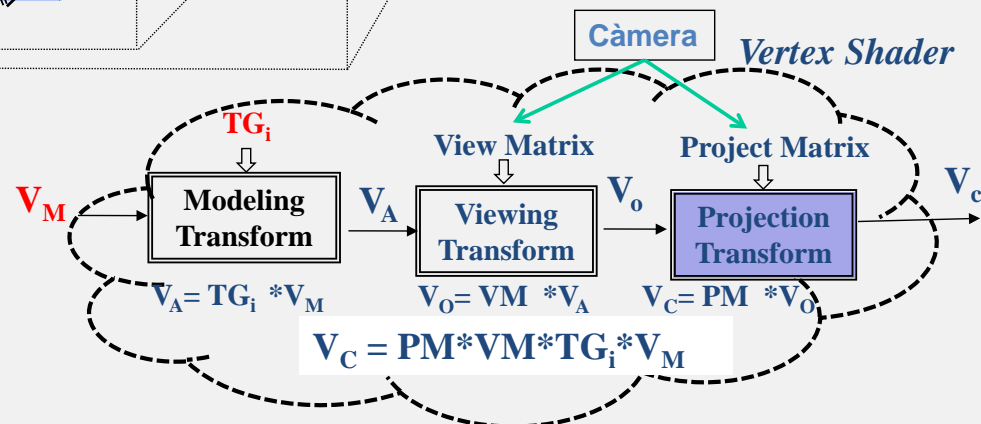
6.2.1 Perspectiva[S2.1][T:3–6]

Transformació de projecció



FOV, zNear, zFar, ra_w

```
PM = perspective (FOV, raw, zN, zF)
projectMatrix(PM);
```



Transformació de projecció

(exercici 1)

- Al codi cpp de MyGLWidget:
 - Demanem un uniform location per al uniform de la matriu
- Definim un mètode que ens calculi la transformació de projecció i envii el uniform amb la matriu cap al vertex shader (cal que els paràmetres siguin floats)

```
void MyGLWidget::projectTransform () {  
    // glm::perspective (FOV en radians, ra window, znear, zfar)  
    glm::mat4 Proj = glm::perspective (float(M_PI)/2.0f, 1.0f, 0.4f, 3.0f);  
    glUniformMatrix4fv (projLoc, 1, GL_FALSE, &Proj[0][0]);  
}
```

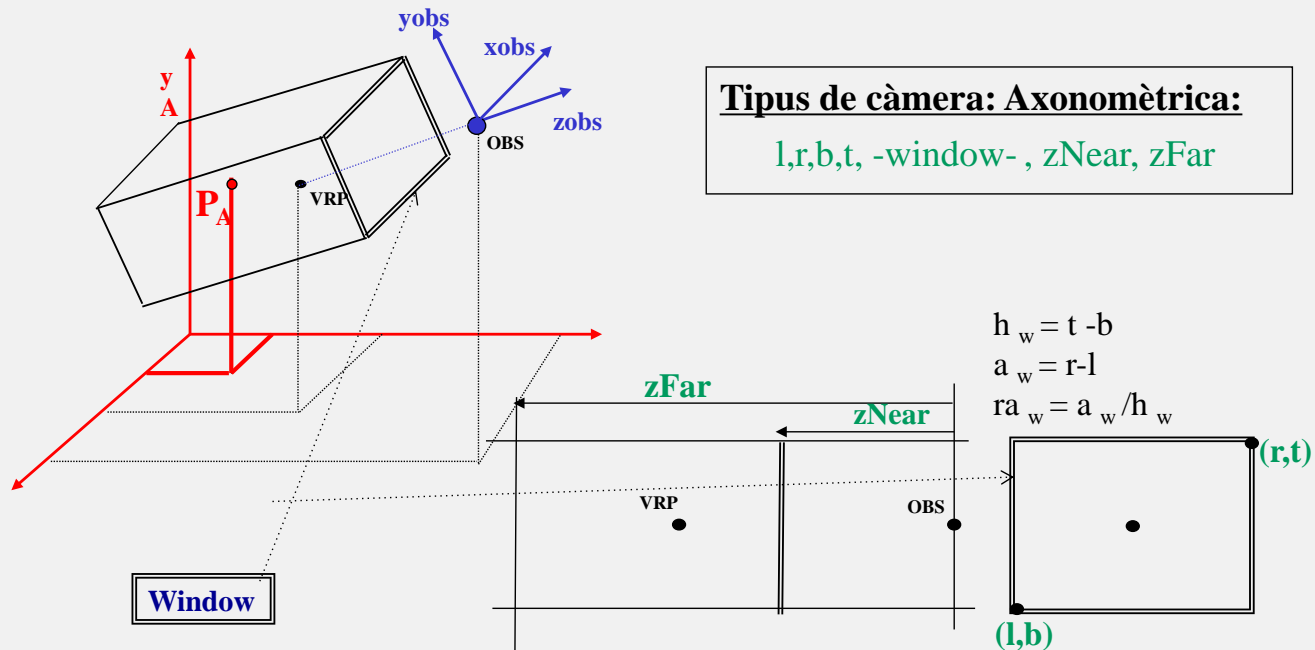
Transformació de projecció

(exercici 1)

- Al vertex shader (afegir):

```
...  
uniform mat4 proj;  
...  
void main () {  
    ...  
    gl_Position = proj * ... * vec4 (vertex, 1.0);  
}
```


Càmera ortogonal (exercici 5)



Càmera ortogonal (exercici 5)

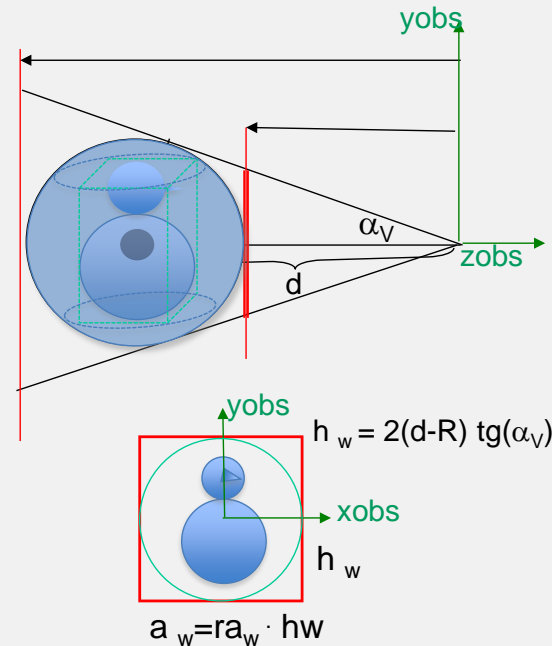
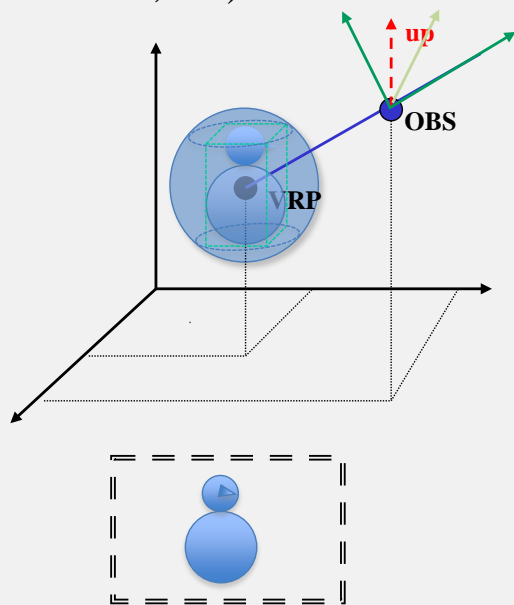
- Càlcul matriu de projecció (òptica de la càmera) amb la crida:

```
glm::mat4 Proj = glm::ortho (left, right, bottom, top, ZNear, ZFar)
```

- Afegir la possibilitat de tenir les dues òptiques possibles i decidibles amb la tecla 'O':
 - Inicialment tenim òptica perspectiva i canviarem d'òptica cada cop que l'usuari premi la tecla 'O'

Càmera en 3^a persona (exercicis 1 i 2)

- Considerar la capsa (i esfera) mínima contenidora de l'escena
- Càlculer els paràmetres de posició i orientació (OBS,VRP,Up)
- Calcular els paràmetres de l'òptica perspectiva (FOV, raw, ZN, ZF)

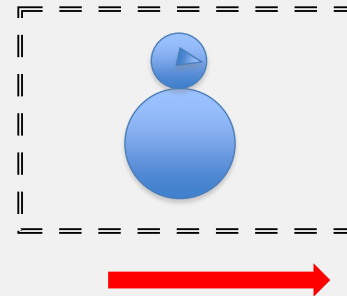
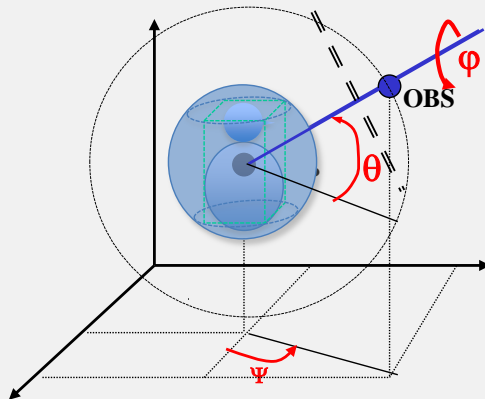


Càmera en tercera persona

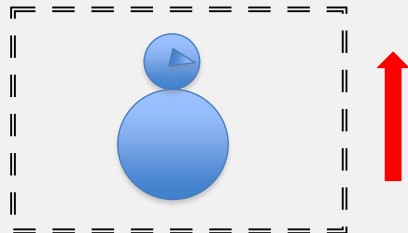
- Mètode per a calcular centre i radi d'escena: (exercici 1)
 - Donats punt mínim i màxim de la caixa contenidora
coneguts en la majoria de casos
- Usar centre i radi escena per a posar paràmetres càmera en tercera persona: (exercici 2)
 - Que es vegi escena centrada, sencera, sense retallar i ocupant màxim del viewport.

Interacció amb angles d'Euler

(exercici 2)



Moviment del ratolí d'esquerra a dreta → increment angle Ψ



Moviment del ratolí de baix a dalt → increment angle θ

Interacció amb angles d'Euler

(exercici 2)

Es vol que el moviment de càmera es faci prement el **botó esquerre** del ratolí, i no qualsevol.

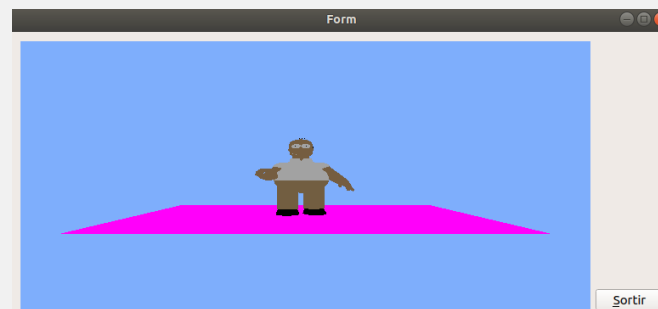
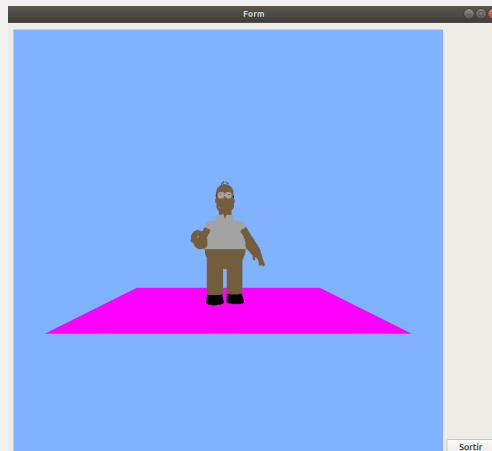
- Si volem controlar el botó del ratolí que s'usa:
`if (e->buttons() == Qt::LeftButton) // e és QMouseEvent`
- Si volem controlar que a més no s'ha usat cap modificador (Shift, Ctrl, Alt):

```
if ( e->buttons() == Qt::LeftButton &&  
    ! ( e->modifiers() &  
        ( Qt::ShiftModifier | Qt::AltModifier | Qt::ControlModifier ) ) )  
    // controla que s'ha premut botó esquerre i cap modificador
```

6.4 Resize[S2.2][T:5-7,13]

Redimensionat sense deformació ni retallat (exercici 3)

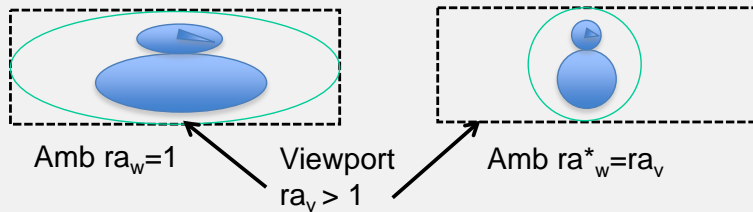
- Quan l'usuari redimensiona la finestra gràfica s'executa automàticament el mètode `resizeGL ()`
- Si aquest mètode no fa res:



Redimensionat sense deformació ni retallat

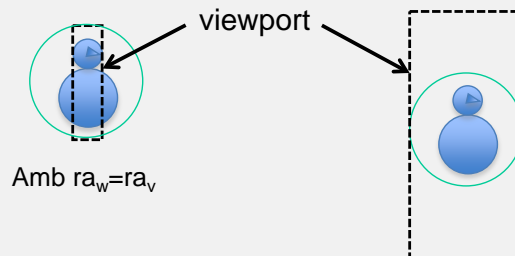
- La relació d'aspecte (ra) del window ha de ser igual que la del viewport:
 $ra_w = ra_v$
- Per tant si canvia la $ra_v \rightarrow$ ha de canviar la $ra_w \rightarrow$ refer perspective (...)

- Si $ra_v > 1$ i $ra_w = ra_v \Rightarrow$ la nova $a_w^* > a_w$ mínima requerida \Rightarrow No es retalla



no cal modificar α_v (FOV)

- Si $ra_v < 1 \Rightarrow ra_w^* < ra_w \Rightarrow a_w^* < a_w \Rightarrow$ retallarà; per evitar-ho cal incrementar l'angle d'obertura (quedarà espai lliure a dalt i a baix)



- Amb $ra_w = ra_v$ i nou FOV

- $FOV = 2 \alpha_v^*$ on $\alpha_v^* = \arctg(\tg(\alpha_v) / ra_v)$

- Sempre cal calcular el nou angle a partir de l'inicial (window quadrat).

Redimensionat sense deformació ni retallat

(exercici 3)

- El mètode `resizeGL` rep com a paràmetres l'amplada i alçada de la finestra gràfica
 - `void resizeGL (int w, int h);`
`// possible càlcul de la relació d'aspecte del viewport`
`float ra = float (w) / float (h);`
- Mètodes de `QOpenGLWidget` que ens poden ser útils:
 - `width ()` → retorna amplada de la finestra gràfica (int)
 - `height ()` → retorna alçada de la finestra gràfica (int)

Resize per a càmera ortogonal (exercici 6)

Afegir/modificar al mètode `resizeGL` el necessari per a que no deformi ni retalli tampoc amb aquesta òptica.

En un exemple on R és el radi de l'esfera tenim:

- Window mínim requerit (centrat) = $(-R, R, -R, R)$ \Rightarrow una $ra_w = 1$
- Si $ra_w \neq ra_v \Rightarrow$ deformació
 - Si $ra_v > 1 \Rightarrow$ cal incrementar la $ra_w \Rightarrow$ *modificar window*
com $ra_w = a_w/h_w \Rightarrow$ podem incrementar a_w o decrementar h_w (és retallaria esfera!!)
Per tant:
 $a_w^* = ra_v * h_w = ra_v * 2R \Rightarrow inc_a = a_w^* - a_w$
 $window = (- (R + inc_a/2), R + inc_a/2, -R, R) = (-R \cdot ra_v, R \cdot ra_v, -R, R)$
 - raonament similar per recalculer window quan $ra_v < 1$

Zoom

(exercici 3)

- Per a fer un zoom ho farem modificant l'angle d'obertura de la càmera (FOV)
 - Zoom-in → decrementar l'angle FOV (tecla 'Z')
 - Zoom-out → incrementar l'angle FOV (tecla 'X')
- Per a càmera ortogonal (opcional):
 - Modificar el window (left, right, bottom, top) mantenint ra

7 Realisme

7.1 Z-Buffer i Culling[S2.1][T:16]

Z-buffer (exercici 4)

- Algorisme de Z-buffer:
 - Activar el z-buffer (només cal fer-ho un cop!)
`glEnable (GL_DEPTH_TEST);`
 - Esborrar el buffer de profunditats a la vegada que el frame buffer
`glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

7.2 Il·luminació

7.2.1 Models Empírics[S3.1][T:2–5]

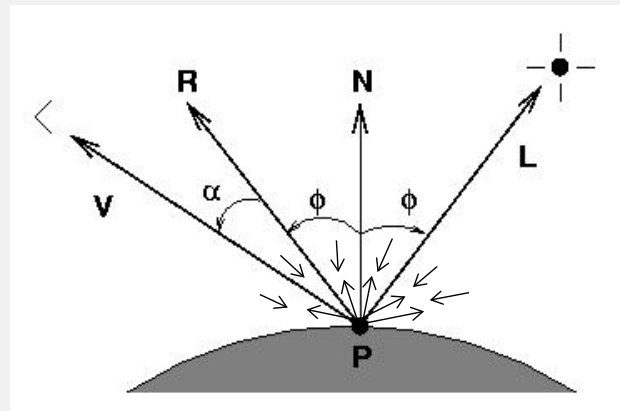
Càlcul color en un punt: models empírics

$$I_{\lambda}(P) = I_{a\lambda}k_{a\lambda} + \sum_i (I_{fi\lambda}k_{d\lambda} \cos(\Phi_i)) + \sum_i (I_{fi\lambda}k_{s\lambda} \cos^n(\alpha_i))$$

$$\cos(\Phi) \Rightarrow \text{dot}(L, N)$$

$$\cos(\alpha) \Rightarrow \text{dot}(R, v)$$

L, N, R i v normalitzats



Càlcul color en un punt: models empírics

Què necessitem?

- Propietats del material
- Vector normal
- Color de llum ambient

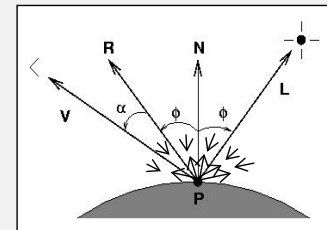


Per cada vertex (punt)

- Posició del focus de llum
- Color del focus de llum
- Posició observador – en SCO sabem que és (0,0,0) -



Per cada focus de llum



Càlcul color en un punt: models empírics

Primer farem el càlcul per cada vèrtex (al Vertex Shader)
I **el farem en SCO**, per tant:

- Cal passar la posició del vèrtex a SCO
 - multiplicant per (**view * TG**)
- Cal passar el vector normal a SCO
 - multiplicant per la matriu **inversa de la transposada de (view * TG)**
- li direm **NormalMatrix** -
- La posició del focus de llum també ha d'estar en SCO
 - Multiplicat per **view** (si no la tenim ja directament en SCO)

Càlcul color en un punt: models empírics

Calcular matriu inversa de la trasposada de $\text{view} * \text{TG}$

- Al vertex shader (en GLSL):

```
mat3 NormalMatrix = inverse (transpose (mat3 (view * TG)));
```

➤ es fa el càlcul de la matriu per a cada vèrtex

- Al programa (amb glm):

```
#include "glm/gtc/matrix_inverse.hpp"
```

```
glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(View*TG));
```

➤ cal tenir les matrius View i TG com a atributs de la classe

➤ i cal passar la NormalMatrix com a uniform al VS per cada objecte

Exercici 1

Càlcul color usant model Lambert:

```
vec3 Lambert (vec3 NormSCO, vec3 L)
{
    // Aquesta funció calcula la il·luminació amb Lambert assumint que els vectors
    // que rep com a paràmetres estan normalitzats

    vec3 colRes = llumAmbient * matamb; // Inicialitzem color a component ambient
    // Afegim component difusa, si n'hi ha
    if (dot (L, NormSCO) > 0)
        colRes = colRes + colFocus * matdiff * dot (L, NormSCO);
    return (colRes);
}
```

Cal calcular en *main*: L en SCO, Normal en SCO,
normalitzar vectors i cridar a Lambert

7.2.3 Models Empírics: Phong[S3.1][T:10]

Exercici 2

Càlcul color usant model Phong:

```
vec3 Phong (vec3 NormSCO, vec3 L, vec4 vertSCO)
{
    // Els vectors rebuts com a paràmetres (NormSCO i L) estan normalitzats
    vec3 colRes = Lambert (NormSCO, L); // Inicialitzem color a Lambert
    // Calculem R i V
    if (dot (NormSCO, L) < 0)
        return colRes; // no afecta la component especular
    vec3 R = reflect (-L, NormSCO); // equival a:: 2.0 * dot (NormSCO, L) * NormSCO - L;
    vec3 V = normalize (-vertSCO.xyz);
    if ((dot (R, V) < 0) || (matshin == 0))
        return colRes; // no afecta la component especular
    // Afegim la component especular
    float shine = pow (dot (R, V), matshin);
    return colRes + matspec * colFocus * shine;
}
```

Càlcul color en un punt: models empírics

Primer farem el càlcul per cada vèrtex (al Vertex Shader)
I **el farem en SCO**, per tant:

- Cal passar la posició del vèrtex a SCO
 - multiplicant per (**view * TG**)
- Cal passar el vector normal a SCO
 - multiplicant per la matriu **inversa de la transposada de (view * TG)**
- li direm **NormalMatrix** -
- La posició del focus de llum també ha d'estar en SCO
 - Multiplicat per **view** (si no la tenim ja directament en SCO)

Càlcul color en un punt: models empírics

Calcular matriu inversa de la trasposada de $\text{view} * \text{TG}$

- Al vertex shader (en GLSL):

```
mat3 NormalMatrix = inverse (transpose (mat3 (view * TG)));
```

➤ es fa el càlcul de la matriu per a cada vèrtex

- Al programa (amb glm):

```
#include "glm/gtc/matrix_inverse.hpp"
```

```
glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(View*TG));
```

➤ cal tenir les matrius View i TG com a atributs de la classe

➤ i cal passar la NormalMatrix com a uniform al VS per cada objecte

Posició del focus de llum

Relativa a:

- L'escena – la posició del focus en SCA
 - Posició fixa del focus respecte a l'escena
 - Multiplicar posFocus per view Matrix per a tenir-la en SCO
- La càmera – la posició del focus en SCO
 - Posició fixa respecte a la càmera
 - posFocus ja està en SCO directament
- Un model – la posició del focus en SCM
 - Posició fixa respecte al model d'un objecte
 - Multiplicar posFocus per (view * TG) igual que al model

Cal tenir en compte

Quan els càlculs es fan en el MyGLWidget:

- Cada cop que es **modifica la viewMatrix**:
 - **Recalcular posFocus** si va multiplicada per viewMatrix
 - **Recalcular NormalMatrix** si es té calculada en MyGLWidget
- Cada cop que es **modifica la modelMatrix (TG)**:
 - **Recalcular NormalMatrix** si es té calculada en MyGLWidget
 - **Recalcular posFocus** si va multiplicada per TG

Exercici 2

Càlcul color en el Fragment Shader:

- Passar les funcions Lambert i Phong al FS
- Fer que hi arribin les dades necessàries des del VS:
 - Posició del vèrtex en SCO
 - Normal al vèrtex en SCO
 - Propietats del material (matamb, matdiff, matspec, matshin)
- Uniforms amb les dades de llum ambient i focus de llum al FS

Els vectors normalitzats en el VS no arriben normalitzats al FS
(després de la interpolació)

Recordeu que els atributs no es poden modificar en el shader