

# Pipeline programable

Professors d'IDI

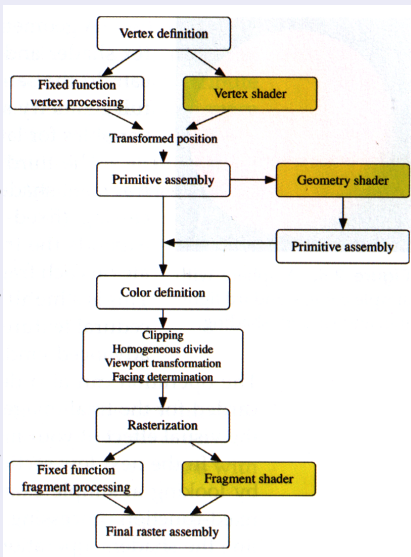
grup ViRVIG

Classes d'IDI, 1516Q1

# El pipeline programable

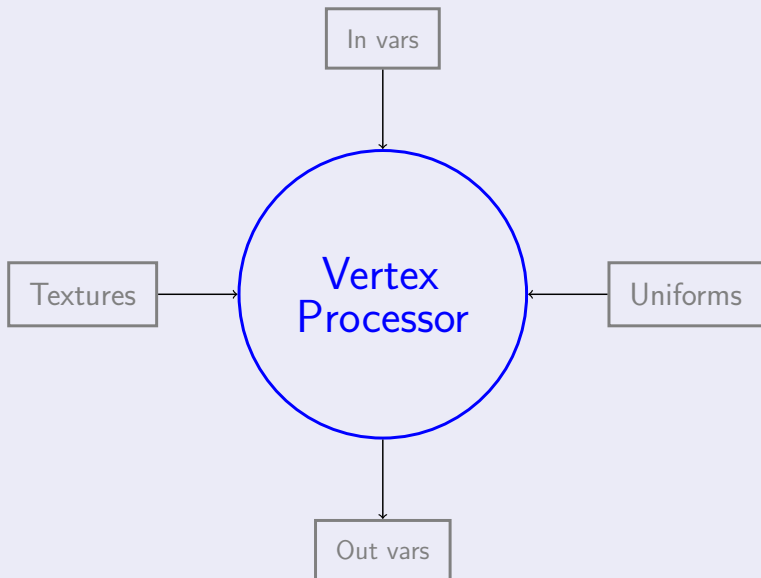
Funcionalitats substituïdes

extreta de *Graphic Shaders*, M. Bailey & S. Cunningham



# El pipeline programable

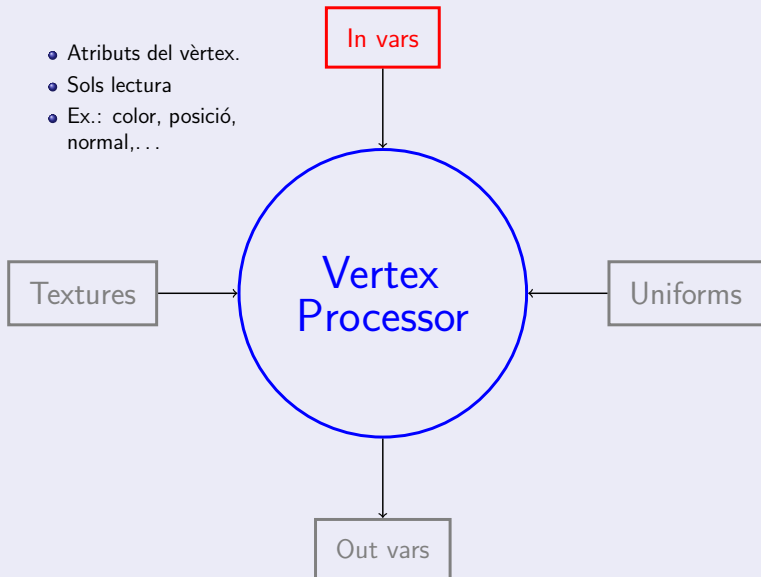
Model de còmput dels *Vertex Shaders*



# El pipeline programable

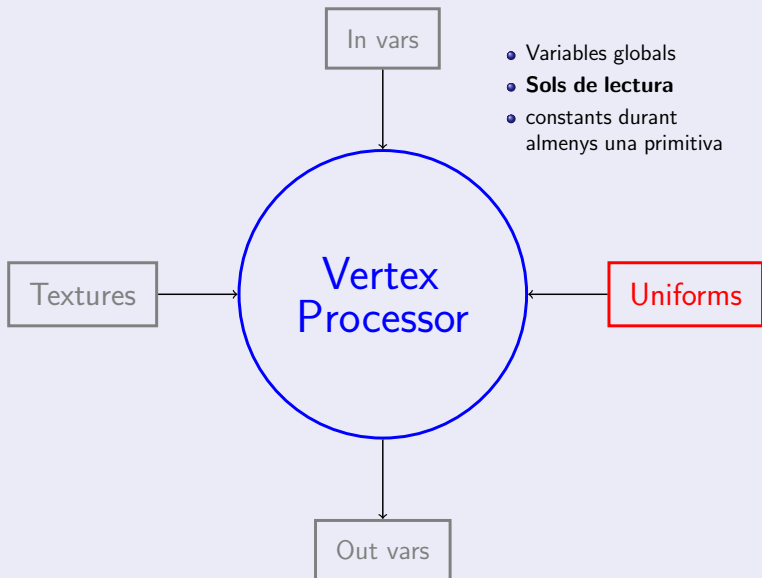
## Model de còmput dels *Vertex Shaders*

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



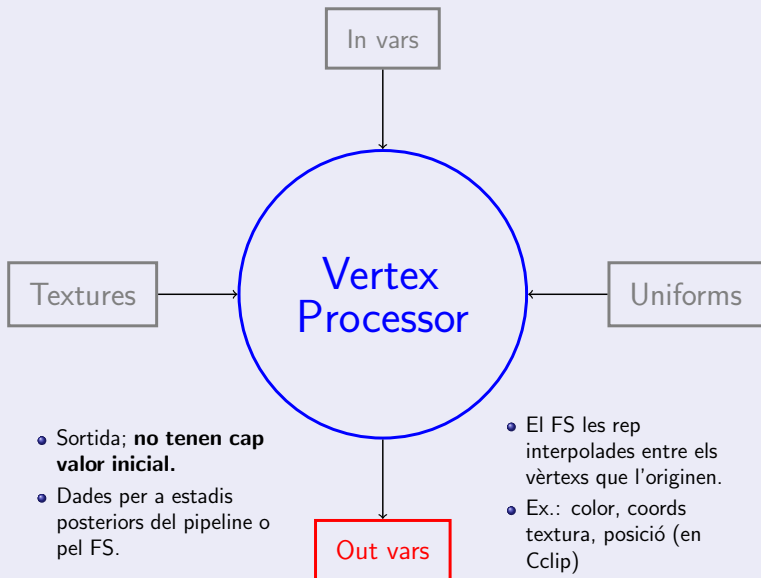
# El pipeline programable

## Model de còmput dels *Vertex Shaders*



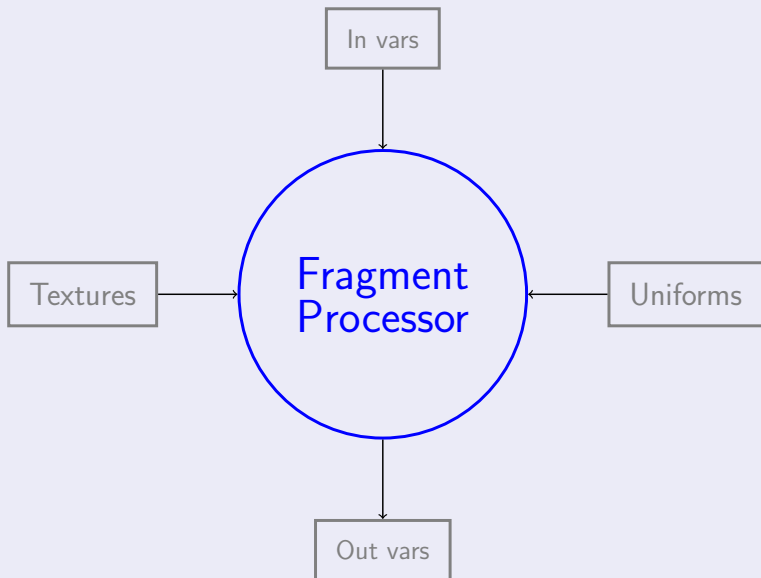
# El pipeline programable

## Model de còmput dels *Vertex Shaders*



# El pipeline programable

Model de còmput dels *Fragment Shaders*



# Evolució

## Versions

Versió	Vers. OGL	data	incorpora
1.10	2.0	2004	vertex i fragment shaders
1.20	2.1	2006	
1.30	3.0	2008	Core and Compatibility profiles, in, out, inout
1.40	3.1	2009	
1.50	3.2	2009	geometry shaders
	3.3	2010	
	4.0	2010	tessellation shaders
	...		
	4.3	2012	compute shaders



# Evolució

## Versions

Versió	Vers. OGL	data	incorpora
1.10	2.0	2004	vertex i fragment shaders
1.20	2.1	2006	
1.30	3.0	2008	Core and Compatibility profiles, in, out, inout
1.40	3.1	2009	
1.50	3.2	2009	geometry shaders
→	3.3	2010	
	4.0	2010	
	...		
	4.3	2012	compute shaders



# Introducció al GLSL

## Exemple de *vertex shader*

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

# Introducció al GLSL

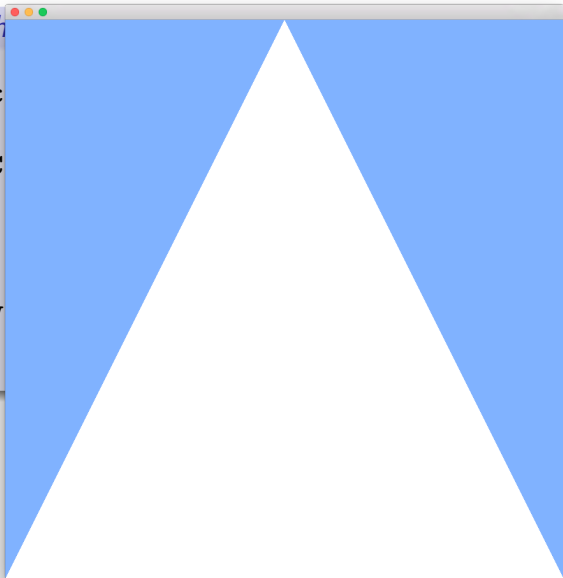
## Exemple de *fragment shader*

```
1  #version 330 core
2
3  out vec4 FragColor;
4
5  void main()
6  {
7      FragColor = vec4(1.);
8  }
```

# Introducció al GLSL

## Exemple de *fragment shader*

```
1  #version 330 c
2
3  out vec4 FragColor
4
5  void main()
6  {
7      FragColor = v
8  }
```



# Elements del llenguatge

## Tipus bàsics

### Escalars

`void, int, uint, float, bool`

### Vectorials

`vec2, vec3, vec4, mat2, mat3, mat4, mat2x3, ..., ivec3, bvec4, uvec2...`

### Constructors

Hi ha *arrays*: `mat2 mats[3];`  
i també *structs*:

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

que defineixen implícitament constructors: `light l1(col,p);`

# Elements del llenguatge

## Funcions

N'hi ha moltes, especialment en les àrees que poden interessar quan tractem geometria o volem dibuixar. Per exemple,

### trigonomètriques

`radians()`, `degrees()`, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()` (amb un o amb dos paràmetres)

### numèriques (poden operar sobre vectors comp. a comp.)

`pow()`, `log()`, `exp()`, `abs()`, `sign()`, `floor()`, `min()`, `max()`

### sobre vectors i punts

`length()`, `distance()`, `dot()`, `cross()`, `normalize()`,...

# Elements del llenguatge

## Funcions

Hom pot definir de noves, amb sintaxi semblant a C, C++ o Java, però...

### Els paràmetres es copien

```
1 vec4 exemple(in vec4 a, float b) { ... }  
2  
3 float[6] exemple(out ivec3 inds) { ... }  
4  
5 void altreExemple(in float a, inout bool flag) {
```

# Elements del llenguatge

## Variables especials (pre-definides)

### Vertex shader

```
1 out vec4 gl_Position;
```

### Fragment shader

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



# Elements del llenguatge

Una instrucció especial pels *fragment shaders*

Descarta el fragment (i conclou l'execució)

```
1  discard;
```

## Altre exemple de VS+FS:

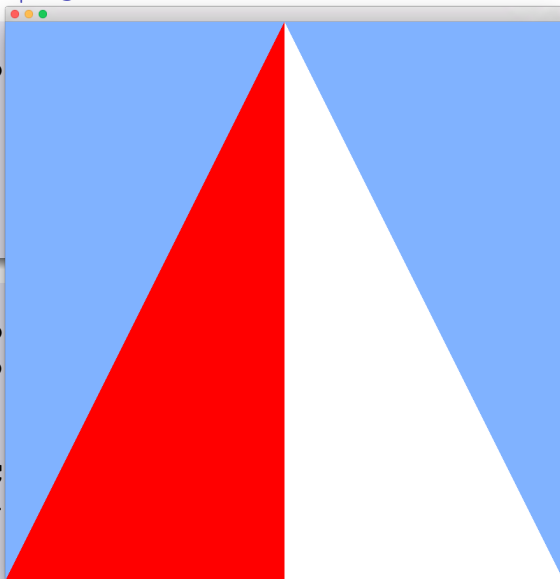
```
1 #version 330 core
2 in vec3 vertex;
3 void main() {
4     gl_Position = vec4(vertex, 1.0);
5 }
```

```
1 #version 330 core
2 out vec4 FragColor;
3 void main() {
4     FragColor = vec4(1.);
5     if (gl_FragCoord.x < 400.)
6         FragColor = vec4(1.,0.,0.,1.);
7 }
```

## Altre exemple de VS+FS:

```
1 #version 330 co
2 in vec3 vertex;
3 void main() {
4     gl_Position
5 }
```

```
1 #version 330 co
2 out vec4 FragCo
3 void main() {
4     FragColor =
5     if (gl_FragC
6         FragColor
7 }
```



# Classes a Qt per a ajudar amb els shaders

Per a gestionar els shaders, farem servir dues classes que ofereix Qt:

## QGLShader

Ofereix un embolcall per a cadascun dels shaders del nostre programa, i gestiona la seva definició, compilació i vinculació a un *Shader Program*. Per accedir a la seva definició caldrà afegir

```
1    #include <QGLShader >
```

## QGLShaderProgram

Permet agrupar uns shaders dissenyats per a funcionar conjuntament, i muntar un *Shader Program*. Per accedir a la seva definició caldrà afegir

```
1    #include <QGLShaderProgram >
```

# QGLShader

La fem servir per a carregar i compilar cada shader:

```
1 QGLShader fs(QGLShader::Fragment, this);  
2 fs.compileSourceFile("./fragshad.frag");  
3 QGLShader vs(QGLShader::Vertex, this);  
4 vs.compileSourceFile("./vertshad.vert");
```

# OpenGLShaderProgram

La fem servir per a construir un programa de shaders:

```
1 program = new OpenGLShaderProgram(this);  
2 program->addShader(&fs);  
3 program->addShader(&vs);  
4 program->link();
```

i per a activar un programa prèviament construït amb èxit:

```
1 program->bind();
```

# Més atributs per vèrtex

Obtenim la posició d'un atribut, per nom

```
1 pos = glGetAttribLocation(program->programId(),  
2                               "vertex");
```

Ara, amb el buffer per aquest atribut lligat a GL\_ARRAY\_BUFFER:

```
1 glVertexAttribPointer(pos, 3, GL_FLOAT,  
2                        GL_FALSE, 0, 0);  
3 glEnableVertexAttribArray(pos);
```

## Detall dels paràmetres

```
void glVertexAttribPointer(
```

<code>GLuint index,</code>	la posició de l'atribut en qüestió.
<code>GLint size,</code>	nombre de components de l'atribut
<code>GLenum type,</code>	tipus de cada component ( <code>GL_FLOAT</code> , <code>GL_INT</code> , ...)
<code>GLboolean normalized,</code>	indica si els floats cal normalitzar-los
<code>GLsizei stride,</code>	distància en bytes separant els atributs consecutius al buffer
<code>const GLvoid *pointer);</code>	offset des del començament del buffer al començament del primer atribut

```
glVertexAttribPointer(pos, 3, GL_FLOAT, GL_FALSE, 0, 0);
```



## Detall dels paràmetres

```
void glVertexAttribPointer(
```

<code>GLuint index,</code>	la posició de l'atribut en qüestió.
<code>GLint size,</code>	nombre de components de l'atribut
<code>GLenum type,</code>	tipus de cada component ( <code>GL_FLOAT</code> , <code>GL_INT</code> , ...)
<code>GLboolean normalized,</code>	indica si els floats cal normalitzar-los
<code>GLsizei stride,</code>	distància en bytes separant els atributs consecutius al buffer
<code>const GLvoid *pointer);</code>	offset des del començament del buffer al començament del primer atribut

```
glVertexAttribPointer(pos, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

## Exemple d'un VS 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*0.5, 1.0);
9 }
```

# Ús de *resources* de Qt

## Afegir al `.pro`

```
1  RESOURCES += shaders.qrc
```

## crear `shaders.qrc`

```
1  <!DOCTYPE RCC><RCC version="1.0">
2  <qresource prefix="/shaders">
3      <file>vertshad.vert</file>
4      <file>fragshad.frag</file>
5  </qresource>
6  </RCC>
```

## l ara podem referir-nos als arxius així:

```
1  fs.compileSourceFile(":/shaders/fragshad.frag");
```

# Resultats dels exercicis proposats (per exemple)

