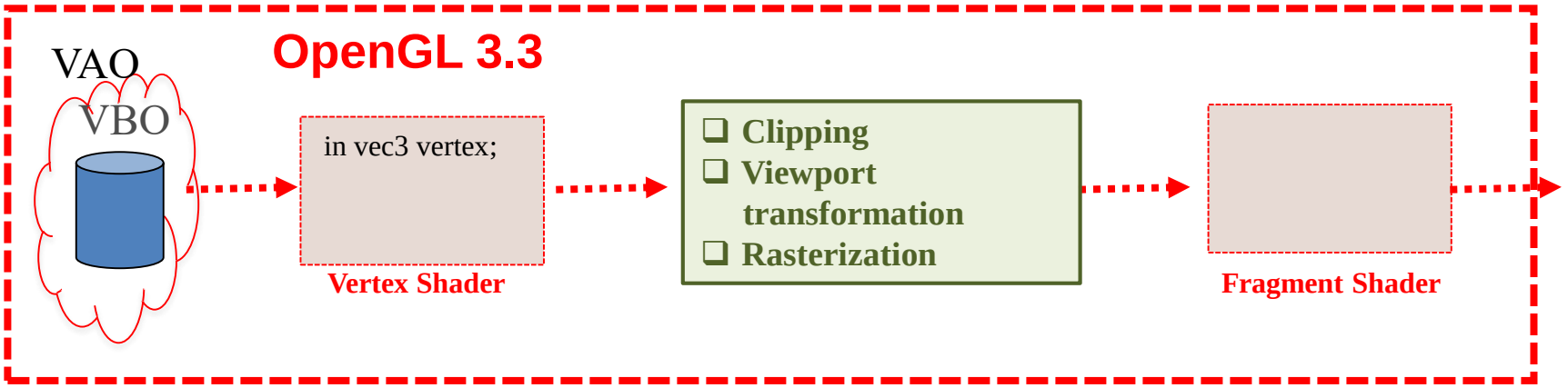
Laboratori IDI

Sessió 1.2

Taula de continguts

Taula de continguts

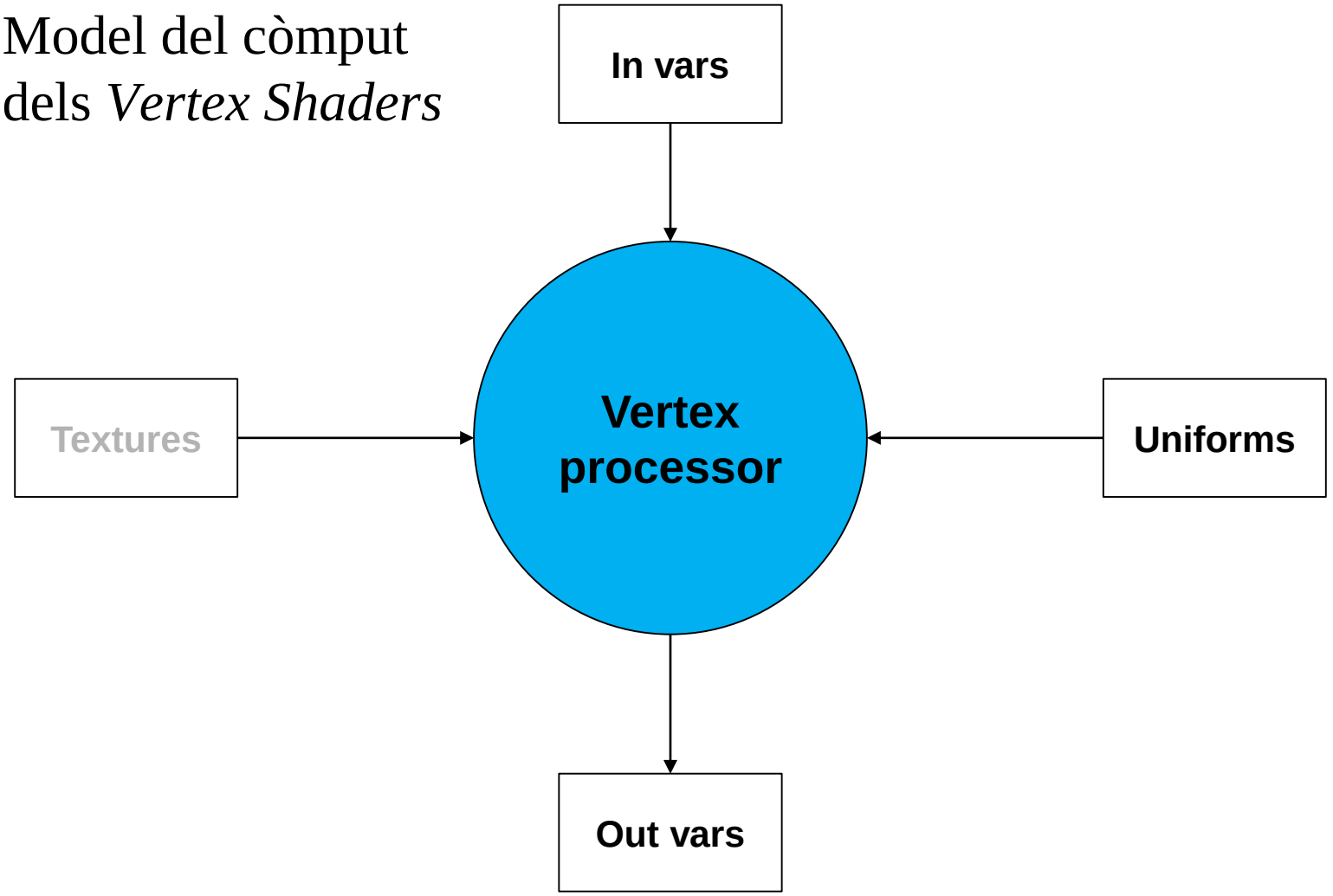
Paradigma projectiu bàsic amb OpenGL 3.3



- 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.

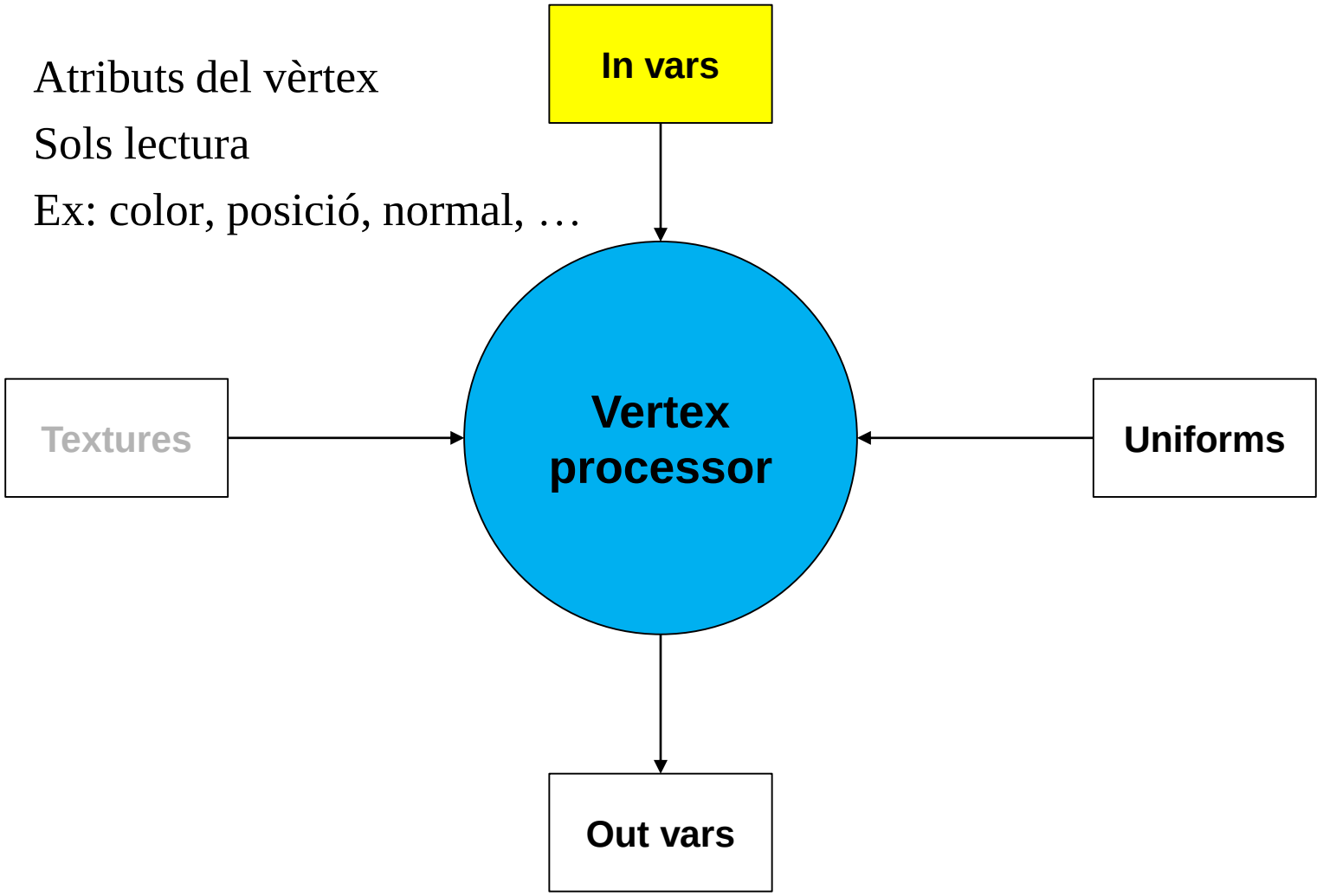
Vertex processor (1)

- Model del còmput dels *Vertex Shaders*

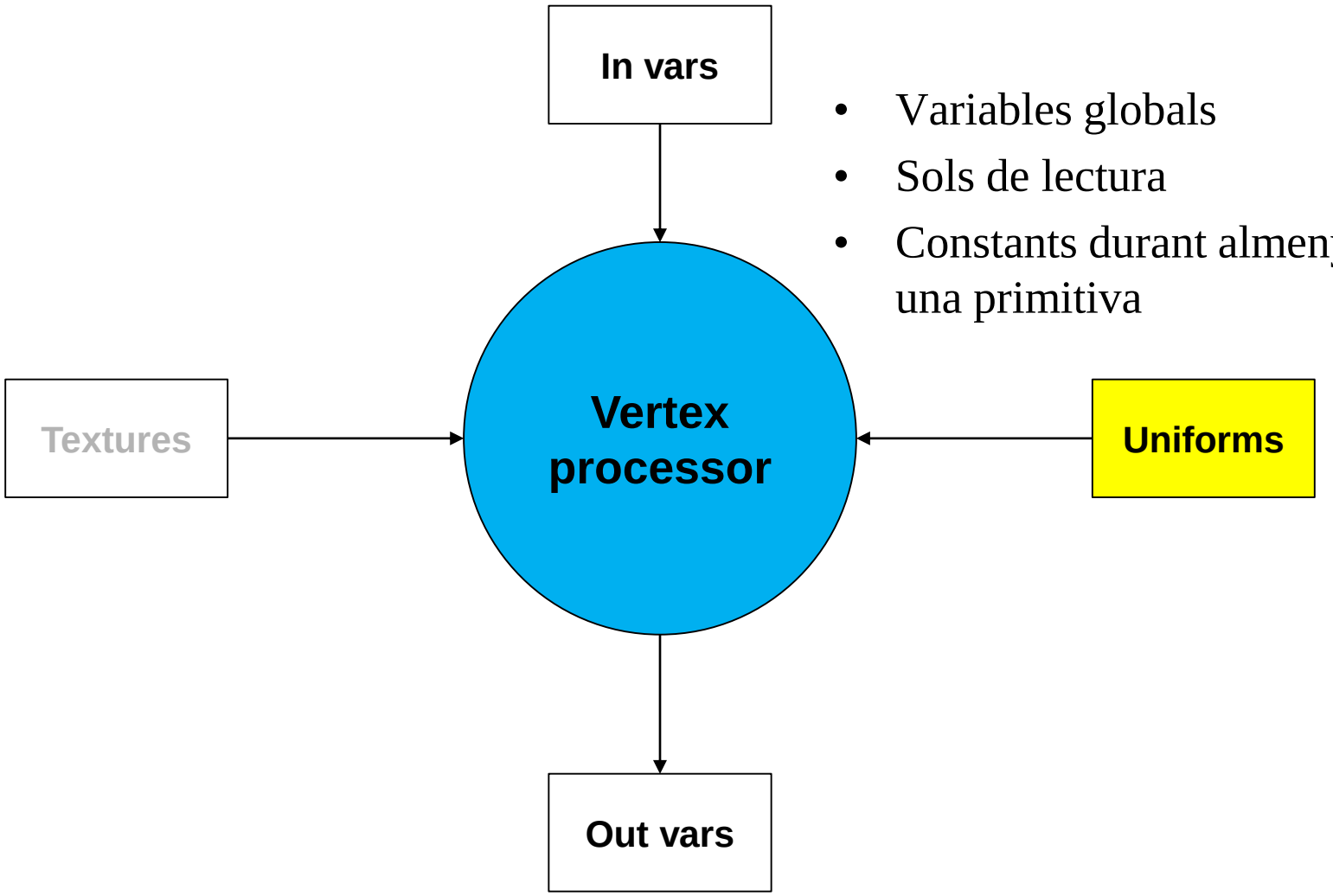


Vertex processor (2)

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

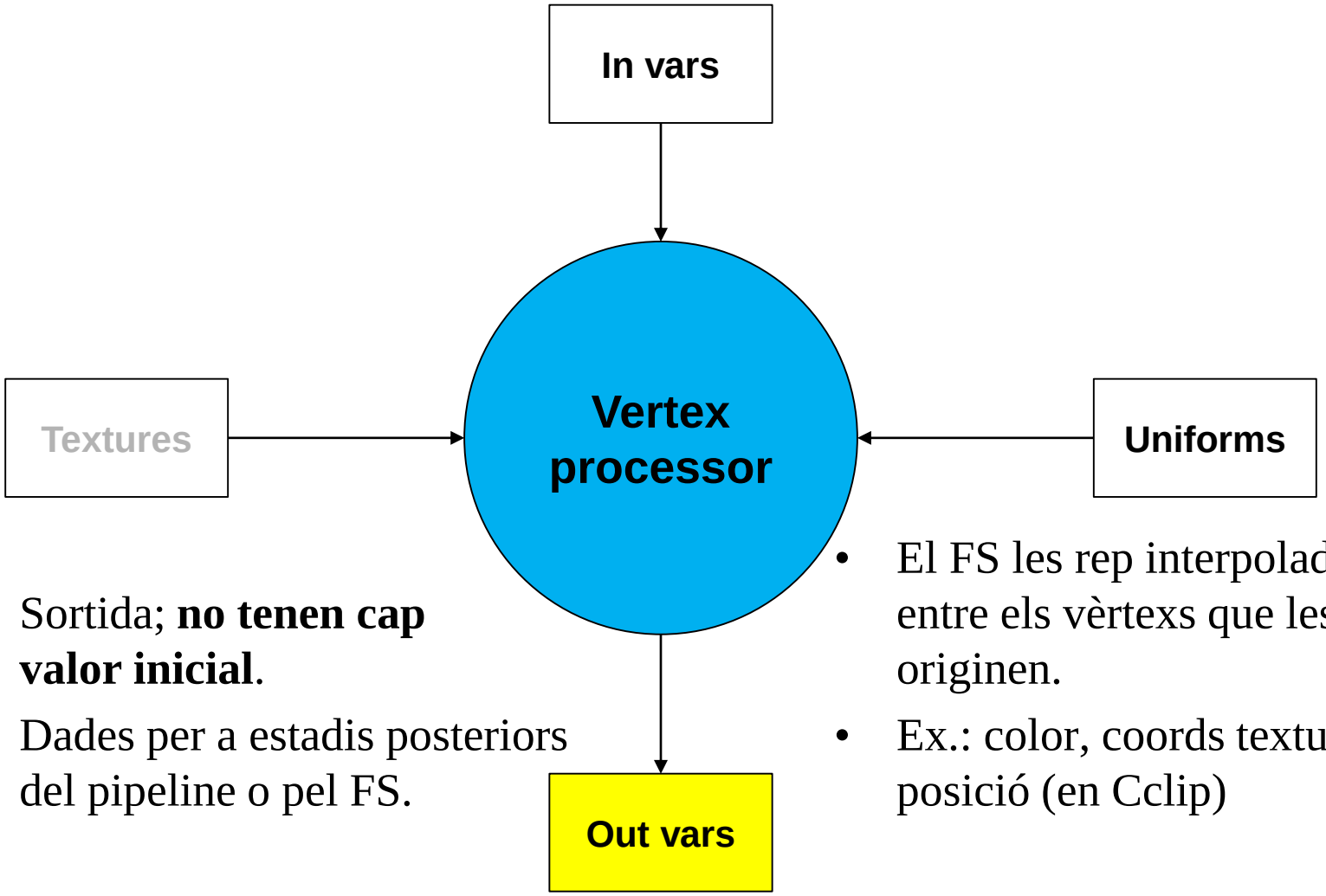


Vertex processor (3)



- Variables globals
- Sols de lectura
- Constants durant almenys una primitiva

Vertex processor (4)

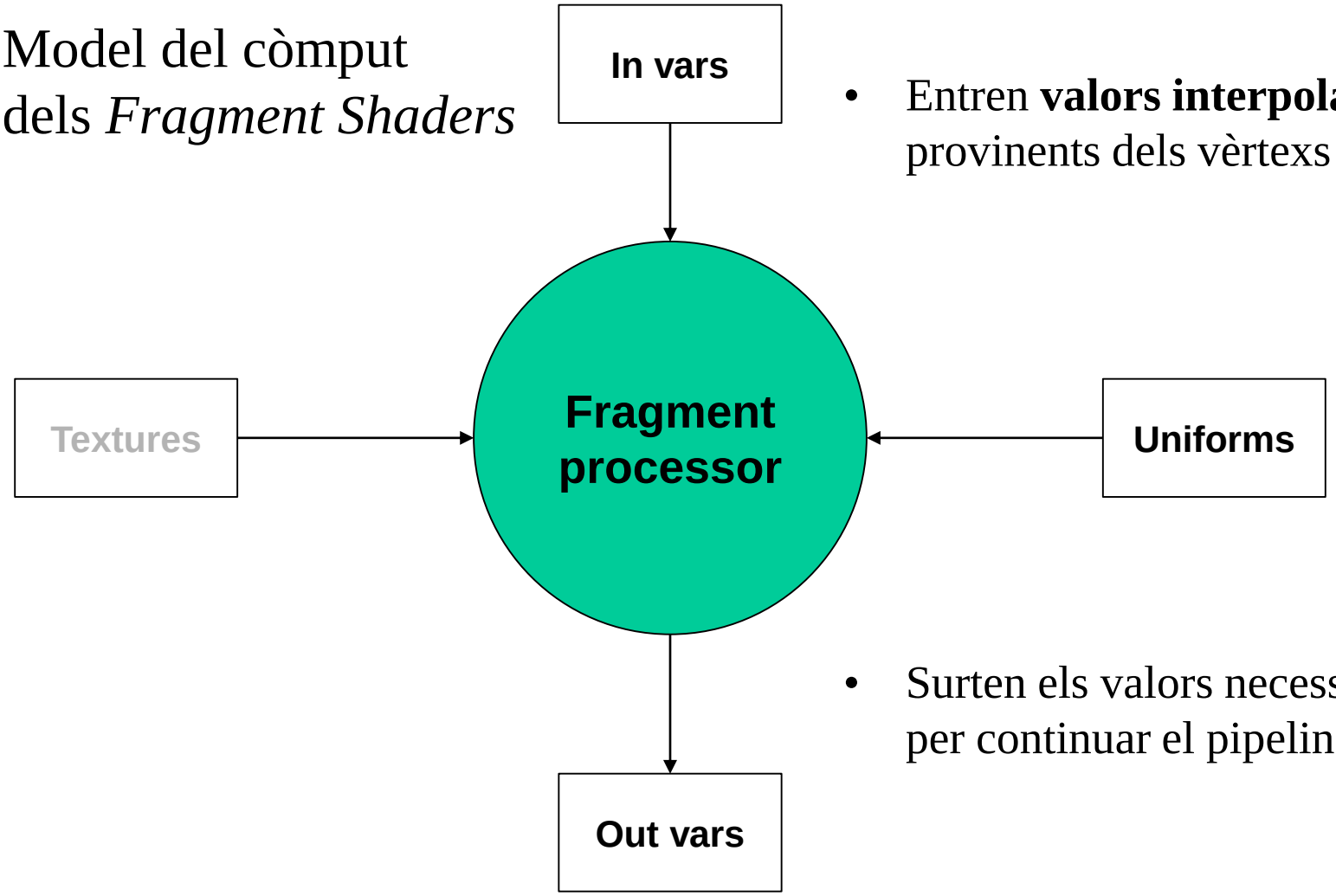


- Sortida; **no tenen cap valor inicial**.
- Dades per a estadis posteriors del pipeline o pel FS.

- El FS les rep interpolades entre els vèrtexs que les originen.
- Ex.: color, coords textura, posició (en Cclip)

Fragment processor

- Model del còmput dels *Fragment Shaders*



- Entren **valors interpolats** provinents dels vèrtexs

- Surten els valors necessaris per continuar el pipeline

Taula de continguts

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	Tesselation 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	Tesselation shaders
…			
4.6		Juliol 2017	Compute shaders

Exemple de vertex shader

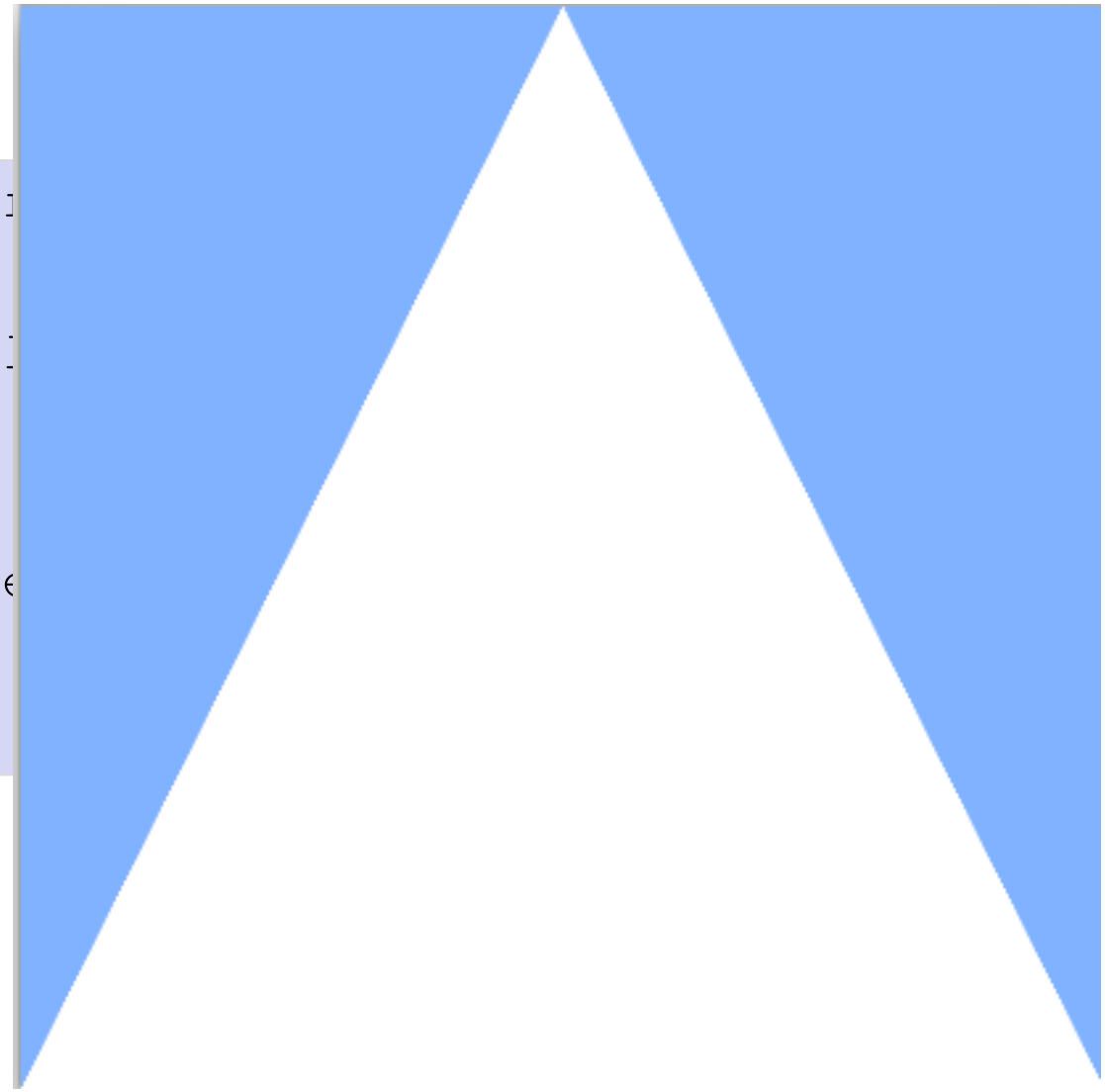
```
1.  #version 330 core
2.
3.  in vec3 vertex;
4.
5.  void main() {
6.    gl_Position = vec4(vertex, 1.0);
7.  }
```

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 co
2.
3.  out vec4 FragCo
4.
5.  void main() {
6.    FragColor = ve
7.  }
```



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.  { … }
```

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;
```

GLSL: discard

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

```
1.  discard;
```

Un altre exemple de VS + FS

Vertex Shader

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

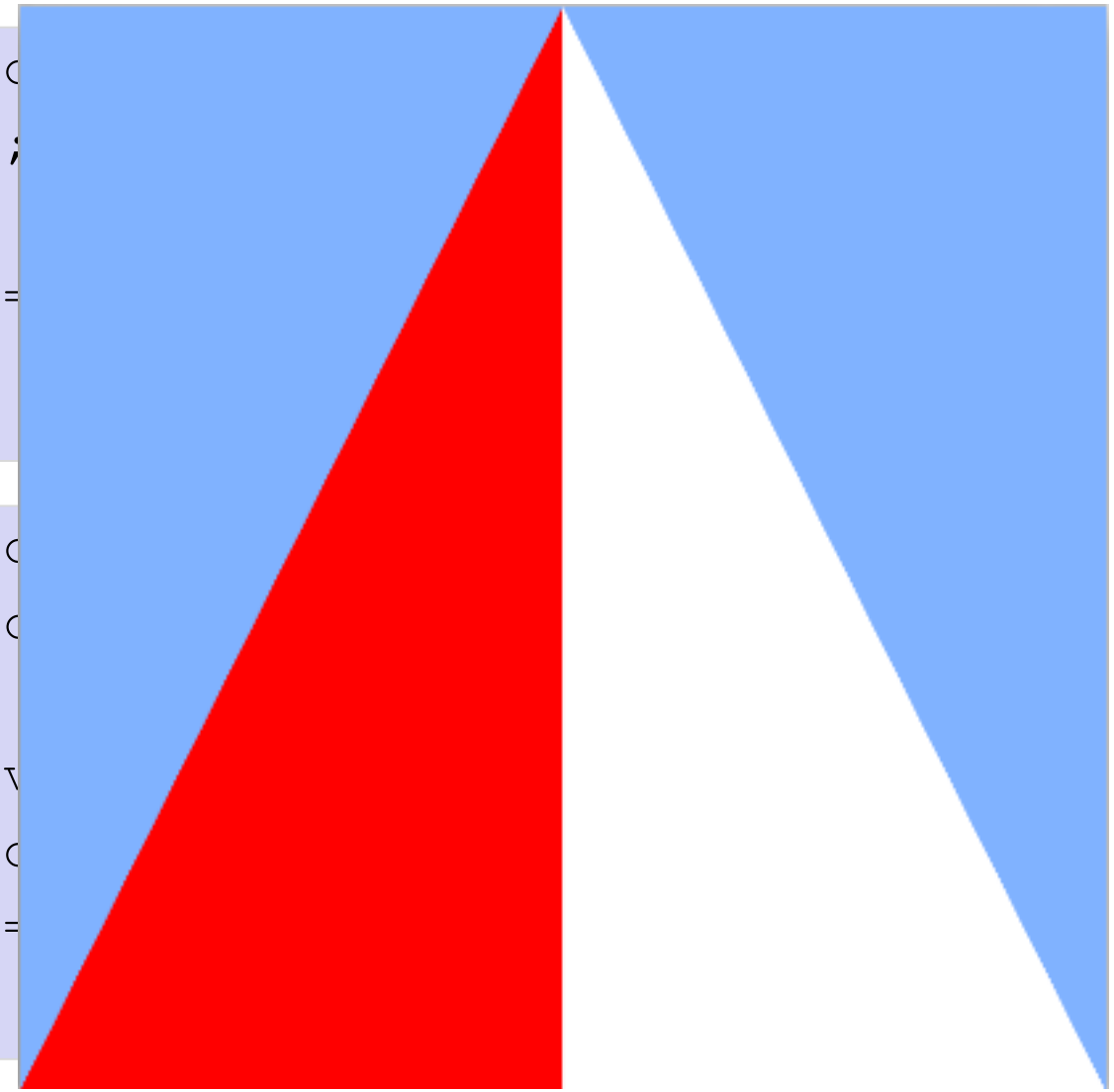
Fragment Shader

```
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.  }
```

Un 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 = v
5.    if (gl_FragCo
6.      FragColor =
7.  }
```



Taula de continguts

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

MyGLWidget.h

MyGLWidget.cpp

Classe que engloba la interfície

MyForm.h

MyForm.cpp

MyGLWidget.h

```
1.  ...
    #include <QOpenGLShader>
    #include <QOpenGLShaderProgram>
4.  class MyGLWidget : public QOpenGLWidget, protected
    QOpenGLFunctions_3_3_Core {
5.    Q_OBJECT
6.    ...
7.    private:
8.      ...
        void carregaShaders ();
10.
11.     GLuint vertexLoc;     // attribute locations
12.
        QOpenGLShaderProgram *program;     // Program
14. };
```

Classes Qt per gestionar shaders

- Els shaders que usarà el nostre programa s'han d'indicar en la classe MyGLWidget.
- Usarem les següents clases 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 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 per al fragment shader i el vertex shader

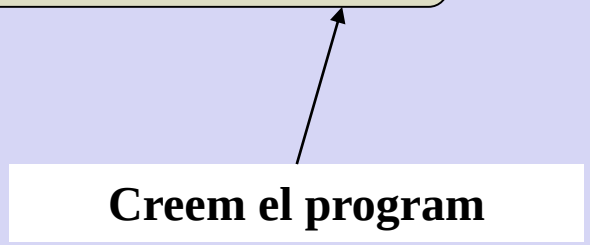
Carrega shaders (2)

```
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. }
```



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

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

Detalls del mètode

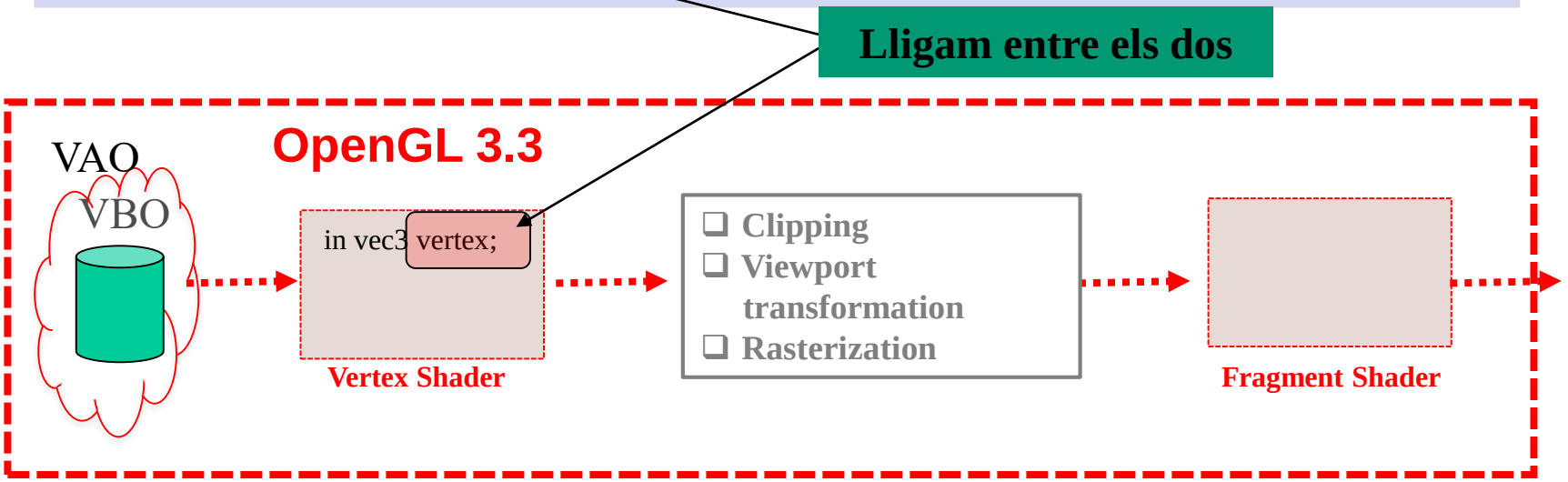
GLint glGetAttribLocation (GLuint *program,* const GLchar **name*);

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");
```



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);
```

Taula de continguts

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

Ús de resources de Qt (1)

- Els shaders no es compilen i es linken amb el codi C++.
- **Els shaders es compilen en temps d'execució**.
- Per tant, si volem passar el nostre programa a algú a més de passar l'executable hauríem de passar el codi dels shaders.
- Per evitar això es poden usar els resources de Qt per tal d'incloure els shaders dins l'executable.

Ús de resources de Qt (2)

1. Afegir al *.pro*

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

2. Crear el fitxer *shaders.qrc*

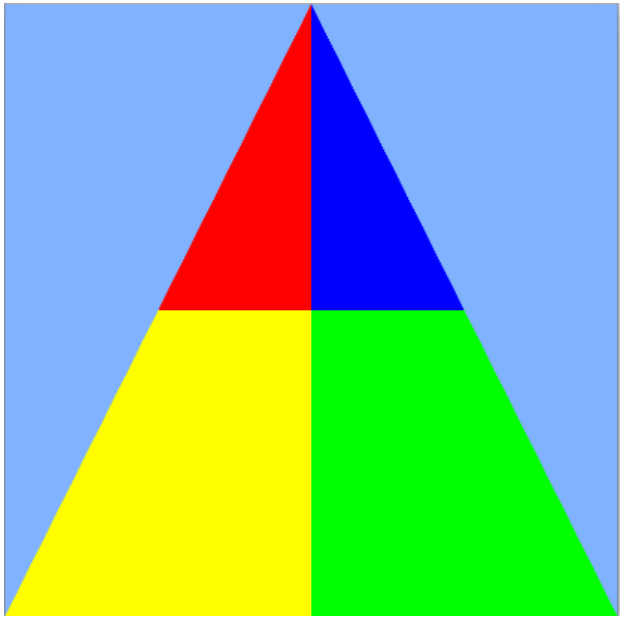
```
1.  <!DOCTYPE RCC>
2.  <RCC version="1.0">
3.  <qresource>
4.    <file>shaders/vertshad.vert</file>
5.    <file>shaders/fragshad.frag</file>
6.  </qresource >
7.  </RCC>
```

3. Canviar les referències als noms dels fitxers shaders:

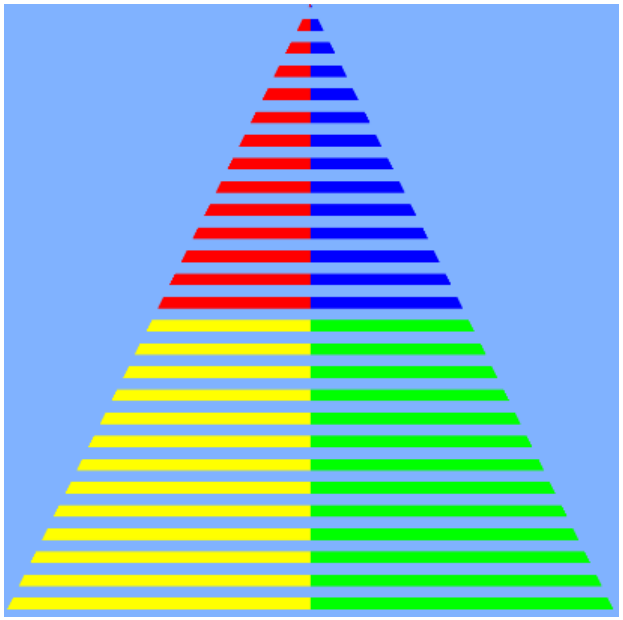
```
1.  fs.compileSourceFile(":/shaders/fragshad.frag");
```

Taula de continguts

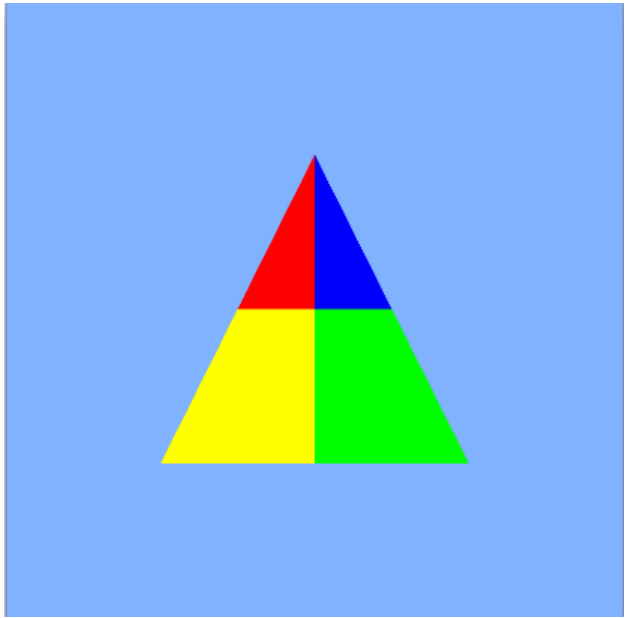
Exercici 1



Exercici 2



Exercici 3



Exercici 4