técnicas avanzadas de gráficos

ingeniería multimedia

Seminario 8

Programación de la GPU con GLSL

Departamento de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante

Programar la GPU

¿Qué es la GPU?

¿Qué es un *shader*?

¿Cómo se programa la GPU?

¿Qué etapas se pueden programar?

¿Qué es una GPU?

- La GPU (*Graphics Processing Unit* - Unidad de Procesamiento Gráfico) es un procesador especializado en gráficos
- Podemos aprovechar su alta capacidad de cálculo y el hecho de que cada ordenador tiene una GPU

Características de la GPU

- Billones de transistores
- Cientos de núcleos de procesamiento en paralelo
- Sistema de memoria distribuido de gran ancho de banda
- Pueden procesar un alto número de tareas en paralelo
- Rendimiento superior a un procesador multinúcleo
- Especializadas en el cálculo en coma flotante

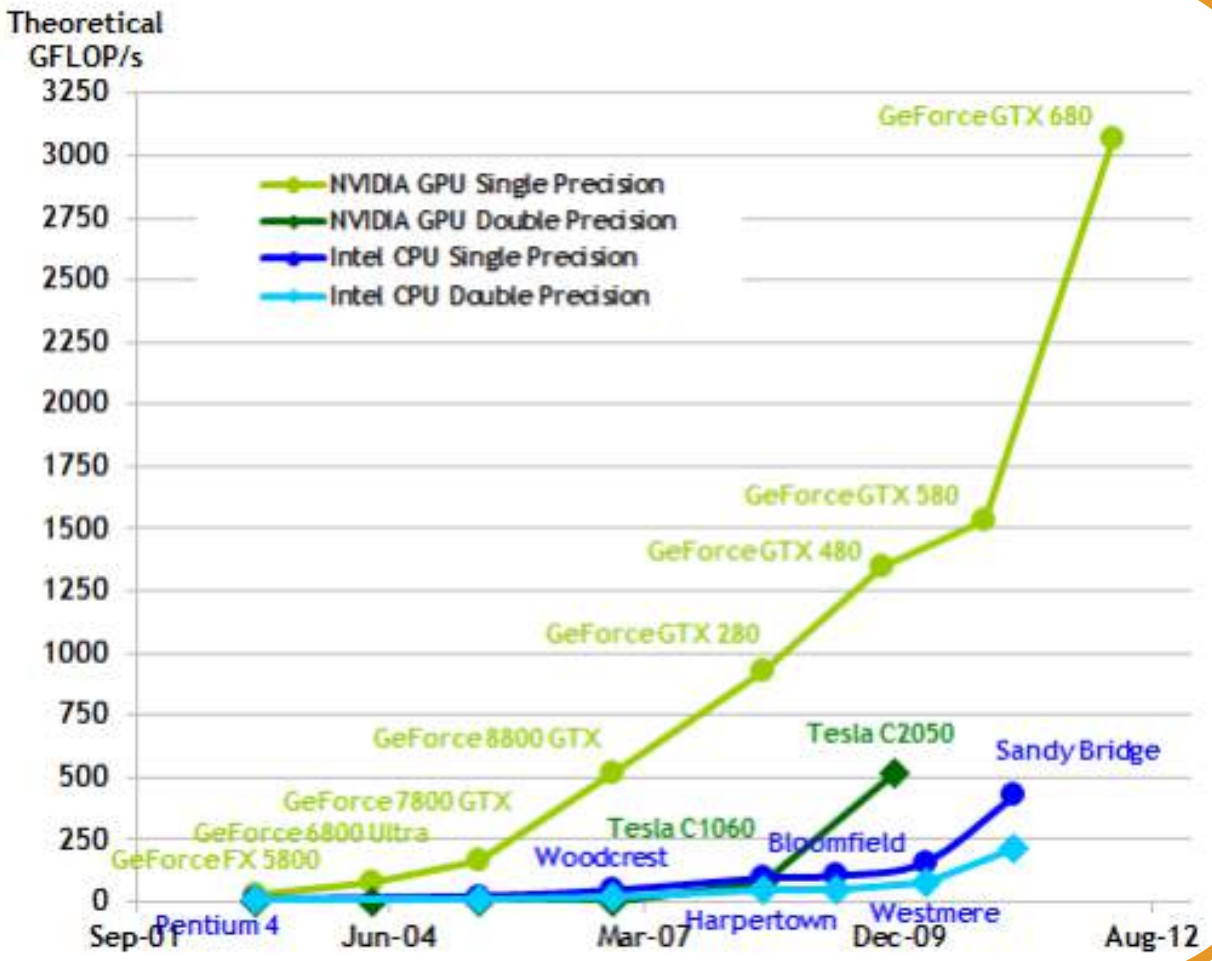
GPU vs. CPU

- Latencia ➔ Tiempo de respuesta
- Rendimiento ➔ Nº operaciones por unidad de tiempo

GPU	CPU
Latencia media	Latencia baja
Rendimiento altísimo	Rendimiento medio
⬇	⬇
Muchas tareas en un tiempo razonable	Una única tarea tan rápido como sea posible

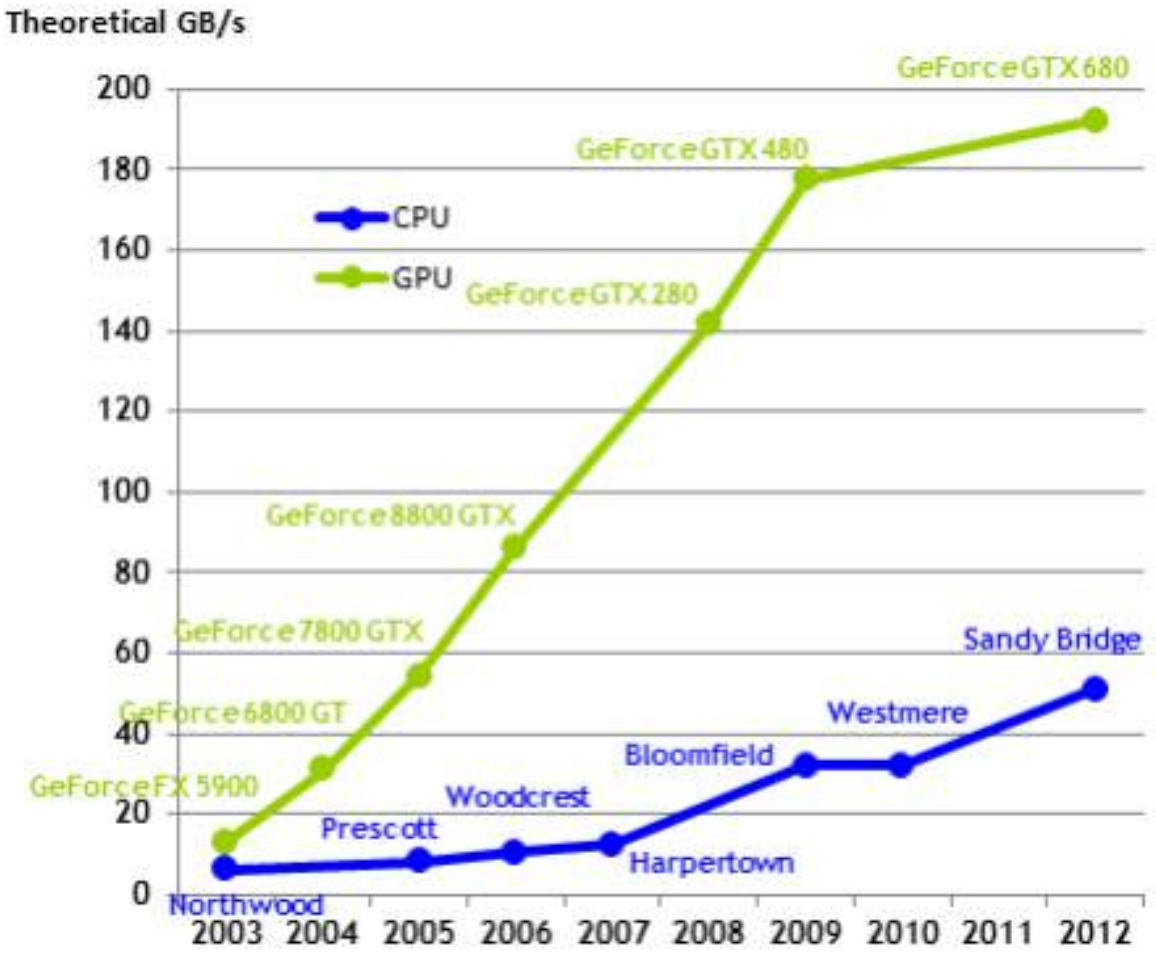
GPU vs. CPU

Operaciones en coma flotante por segundo



GPU vs. CPU

Ancho de banda de la memoria



Aplicaciones de la GPU

- Gráficos
  - Aplicaciones iniciales para las que se diseñó:
    - Transformaciones geométricas
    - Iluminación
    - Rasterización…
  - Millones de polígonos por segundo
  - Lenguajes de programación de *shaders*:
    - GLSL: OpenGL Shading Language
    - Cg: C for Graphics (NVIDIA)
    - HLSL: High Level Shanding Language (Microsoft)

Aplicaciones de la GPU

- Cualquier aplicación paralelizable
  - Nuevo paradigma de programación
  - GPGPU: General-Purpose Computing on Graphics Processing Units
  - Algunos lenguajes GPGPU
    - CUDA (Compute Unified Device Architecture) de NVIDIA, con implementaciones para C/C++, Python, Fortran, Java…
    - DirectCompute de Microsoft
    - OpenCL (Open Computing Language)
    - BrookGPU de la Universidad de Stanford
  - Algunos algoritmos pueden alcanzar hasta 100x en GPU sobre su versión en CPU

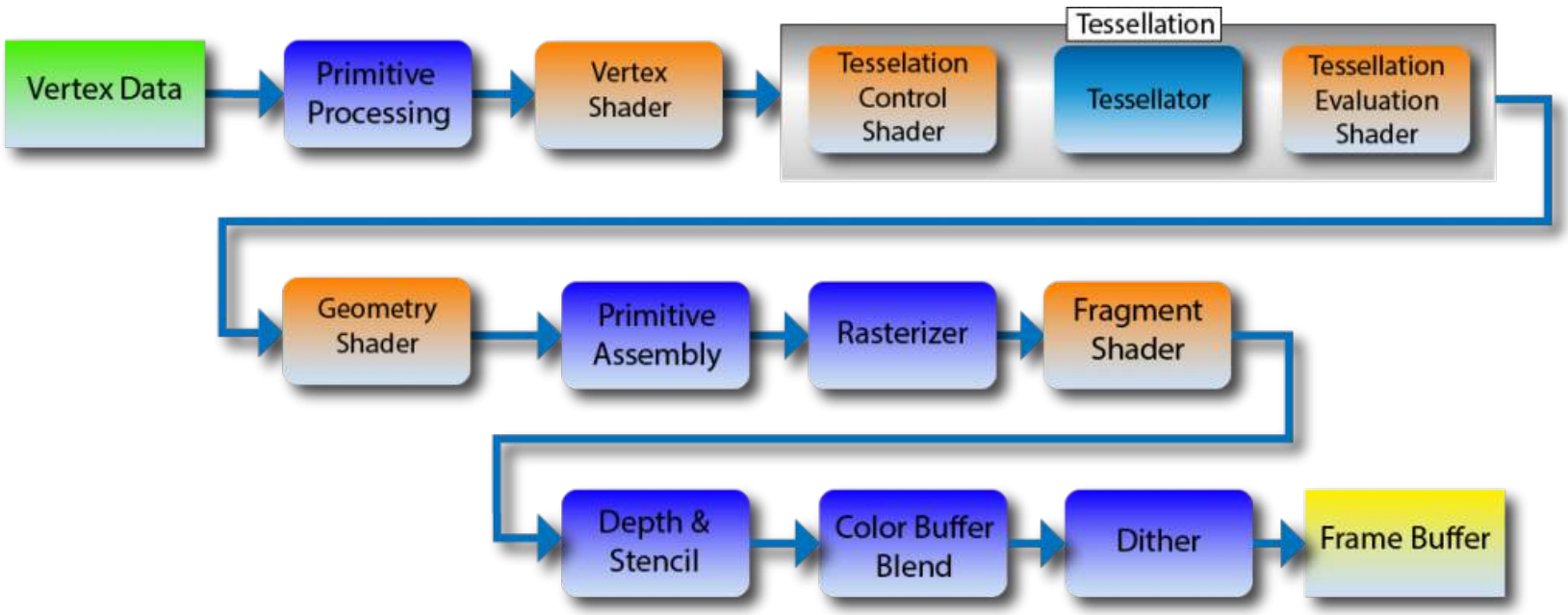
Aplicaciones de la GPU

- En la GPU, técnicamente se puede programar lo que sea
- La GPU no es tan flexible como la CPU
- CPU + GPU = combinación de flexibilidad y rendimiento

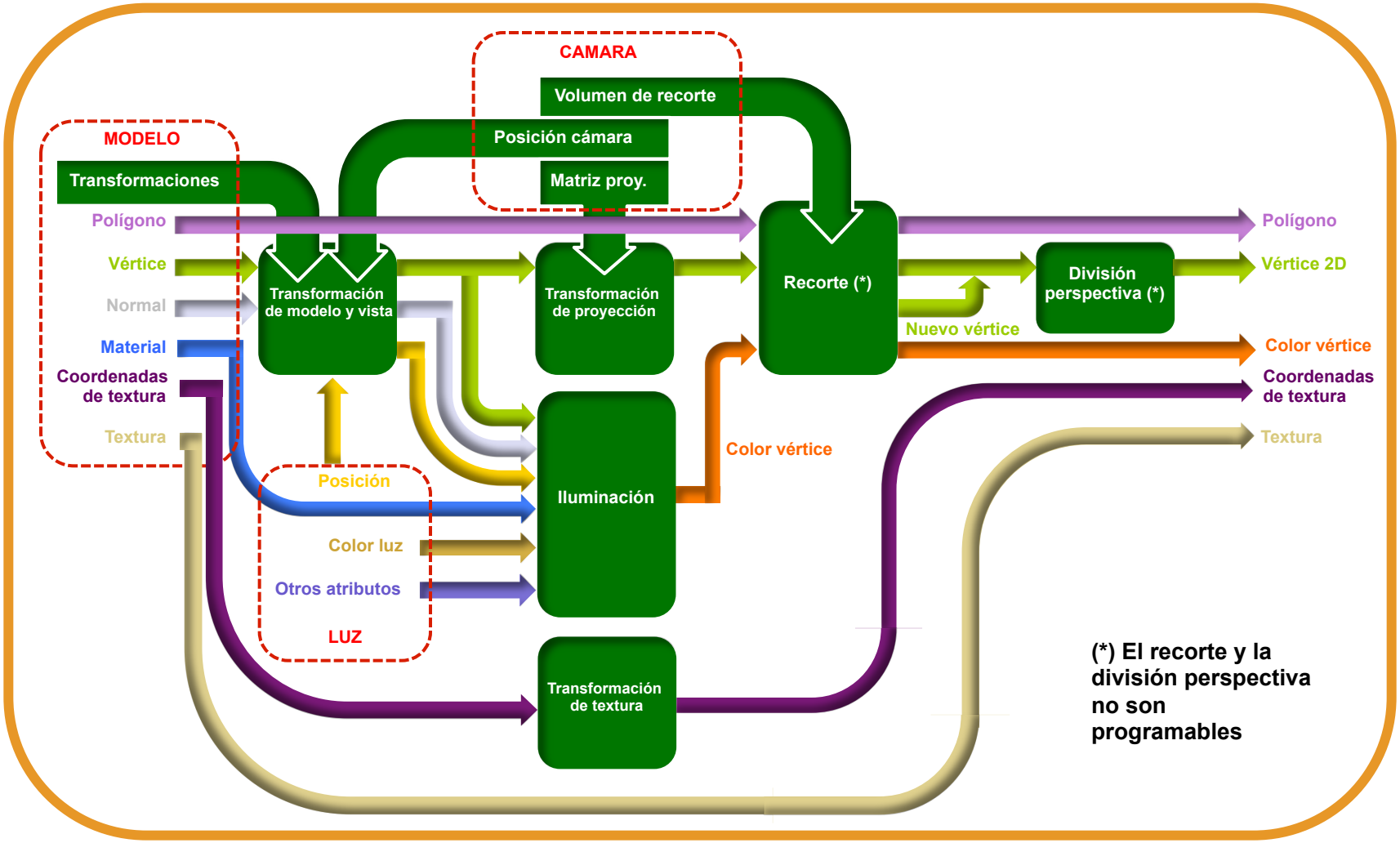
Programación de la GPU con GLSL (4.X)

- OpenGL *Shading Language*
- Veremos GLSL 4.X (versiones anteriores se consideran obsoletas*)*
- Basado en ANSI C
- Permite programar algunas etapas del pipeline, mediante programas denominados *shaders*
- 5 etapas programables:
  - Vertex shader
  - Tessellation control shader
  - Tessellation evaluation shader
  - Geometry shader
  - Fragment shader

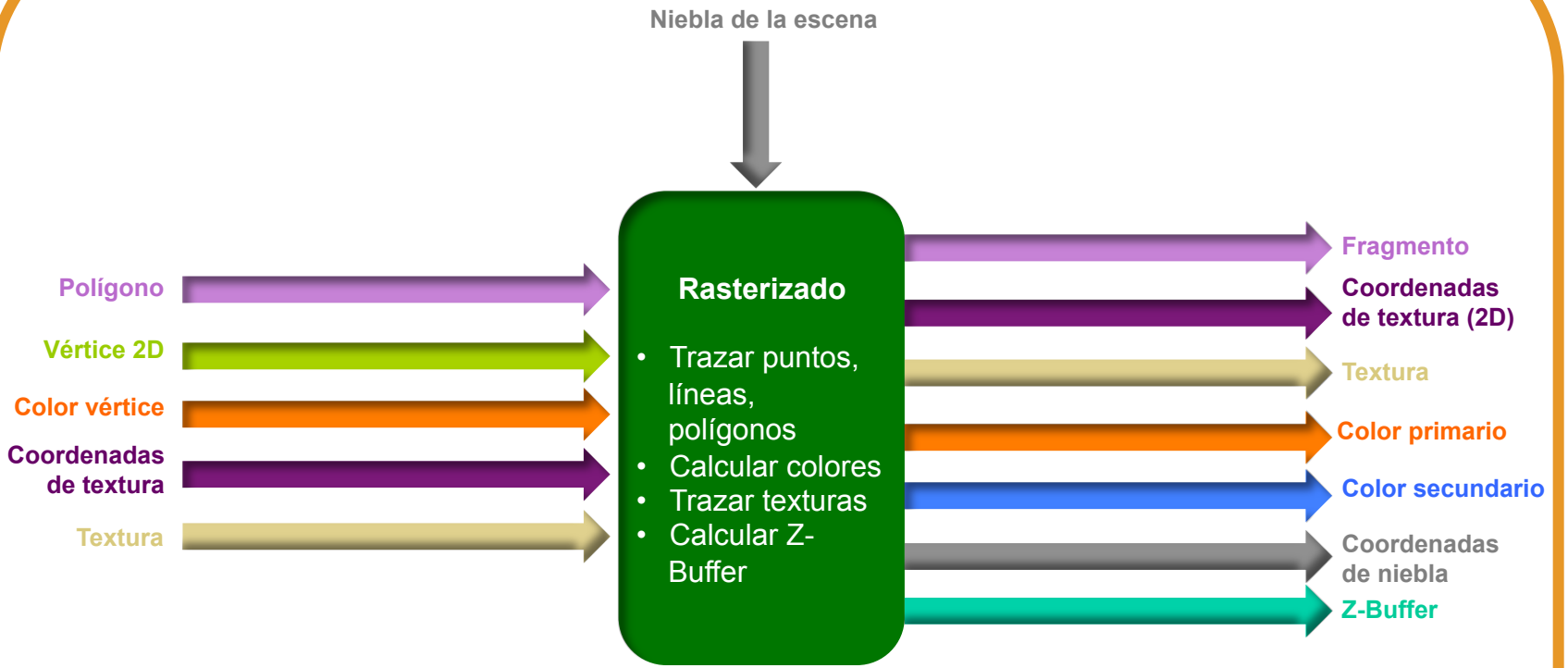
Pipeline OpenGL simplificado



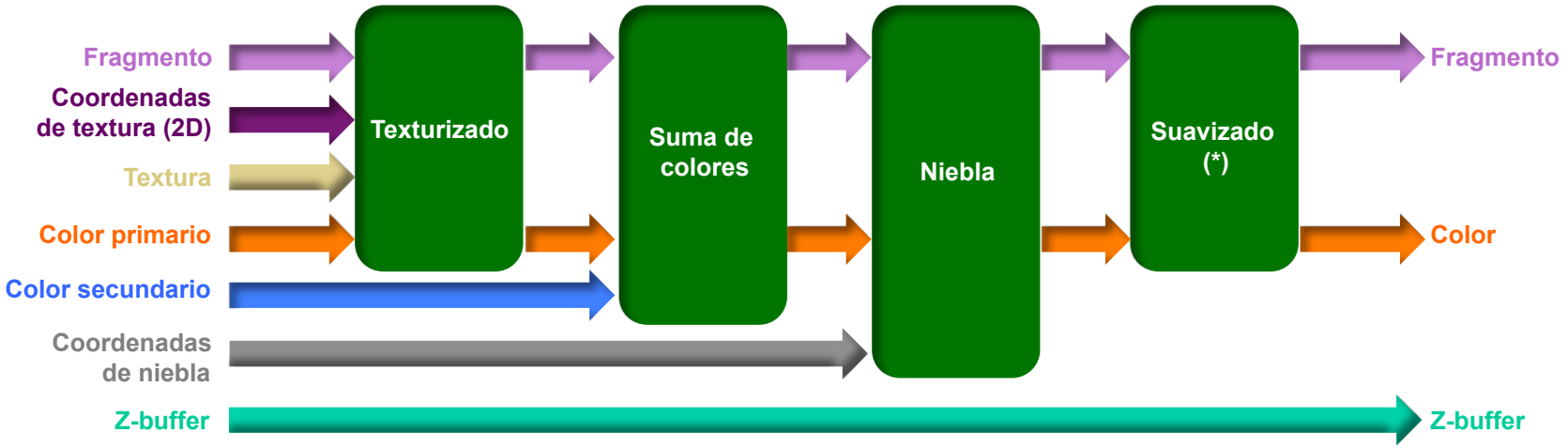
Vertex shader



Pipeline (etapas intermedias)

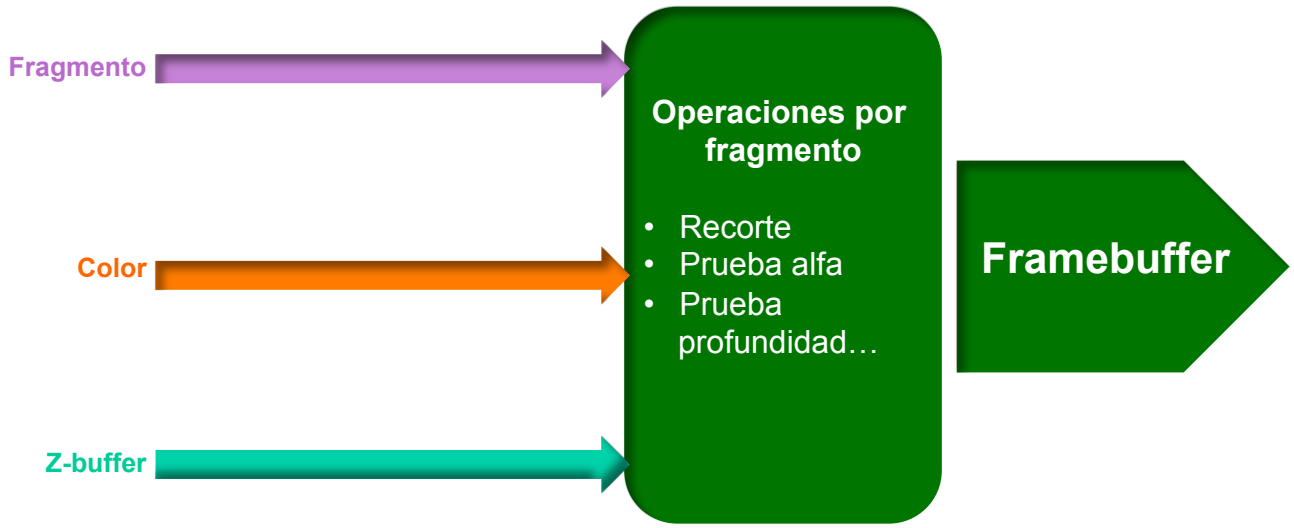


Fragment shader



(*) El suavizado no es programable

Pipeline parte final



Shaders

- *Vertex shader*: programa de usuario que realiza operaciones sobre los vértices:
  - Entrada: vértices y sus atributos
  - Salida: vértices transformados y otros atributos
- *Fragment shader*: programa de usuario que realiza operaciones sobre fragmentos:
  - Entrada: fragmentos y sus atributos
  - Salida: píxeles en el *framebuffer*

Tipos de datos de GLSL

Tipo de datos GLSL	Tipo de datos C	Descripción
bool	int	Booleano
int	int	Entero
float	float	Flotante
vec2	float [2]	Vector de 2 flotantes
vect3	float [3]	Vector de 3 flotantes
vec4	float [4]	Vector de 4 flotantes
bvec2	int [2]	Vector de 2 booleanos
bvec3	int [3]	Vector de 3 booleanos
bvec4	int [4]	Vector de 4 booleanos
ivec2	int [2]	Vector de 2 enteros
ivec3	int [3]	Vector de 3 enteros
ivec4	int [4]	Vector de 4 enteros
mat2	float [4]	Matriz de 2x2 flotantes
mat3	float [9]	Matriz de 3x3 flotantes
mat4	float [16]	Matriz de 4x4 flotantes
sampler1D	int	Puntero a una textura 1D
sampler2D	int	Puntero a una textura 2D
sampler3D	int	Puntero a una textura 3D
samplerCube	int	Puntero a una textura Cubemap
sampler1DShadow	int	Puntero a una textura de produndidad 1D con comparación
sampler2DShadow	int	Puntero a una textura de produndidad 2D con comparación

Vectores y matrices. Curiosidades

Acceso a vectores

[1]	[2]	[3]	[4]	Para hacer bucles
x	y	z	w	Para representar puntos
s	t	q	p	Para representar coordenadas de textura
r	g	b	a	Para representar colores

Ejemplos

```
vec2 p; vec4 v4;
p.x         // correcto
p.z         // incorrecto: pos es un vector de 2 componentes, no tiene componente z
v4.rgba;    // es un vec4, equivalente a utilizar v4
v4.rgb;     // es un vec3
v4.b;       // es un float
v4.xy;      // es un vec2
v4.xgba; // incorrecto: los nombres de componentes deben ser del mismo conjunto

vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);   // inicializador estilo C++
vec4 swiz = pos.wzyx;                  // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy;                   // dup = (1.0, 1.0, 2.0, 2.0)
pos.xw = vec2(5.0, 6.0);               // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0);               // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0);               // incorrecto: x definida dos veces

mat3 matriz = mat3 (2.0);        // inicializa una matriz con la diagonal a 2.0
mat4 transf = mat4 (pos, swiz, dup, vec4 (1.0, 2.0, 3.0, 4.0));  // otra inicialización
```

Operadores

- Operadores, como en C, pero **no existen**: `*` y `&` (punteros), `sizeof`, `<<`, `>>`, `^`, `|` (bits), `%`, `~` (not unario)
- Operadores para vectores y matrices

Operador	Vectores	Matrices
-x	Negación del vector	Negación de la matriz
x+y	Suma de vectores (igual dimensión)	Suma de matrices (igual dimensión)
x-y	Resta de vectores (igual dimensión)	Resta de matrices(igual dimensión)
x*y	Producto de vectores por componentes	Producto algebraico de matrices o vector-matriz (no por componentes)
x/y	Dividisión de vectores por componentes	División algebraica de matrices o vector-matriz (no por componentes)
dot(x,y)	Producto escalar de vectores (igual dimensión)	
cross(x,y)	Producto vectorial de vectores (sólo válida para vectores vec3)	
matrixCompMult(x,y)		Producto por componentes de matrices (igual dimensión)
normalize(x)	Normalización del vector	
reflect(t,n)	Reflejo el vector t según el vector n	

Tipos estructurados, funciones, estructuras de control

- Arrays y estructuras similar a C

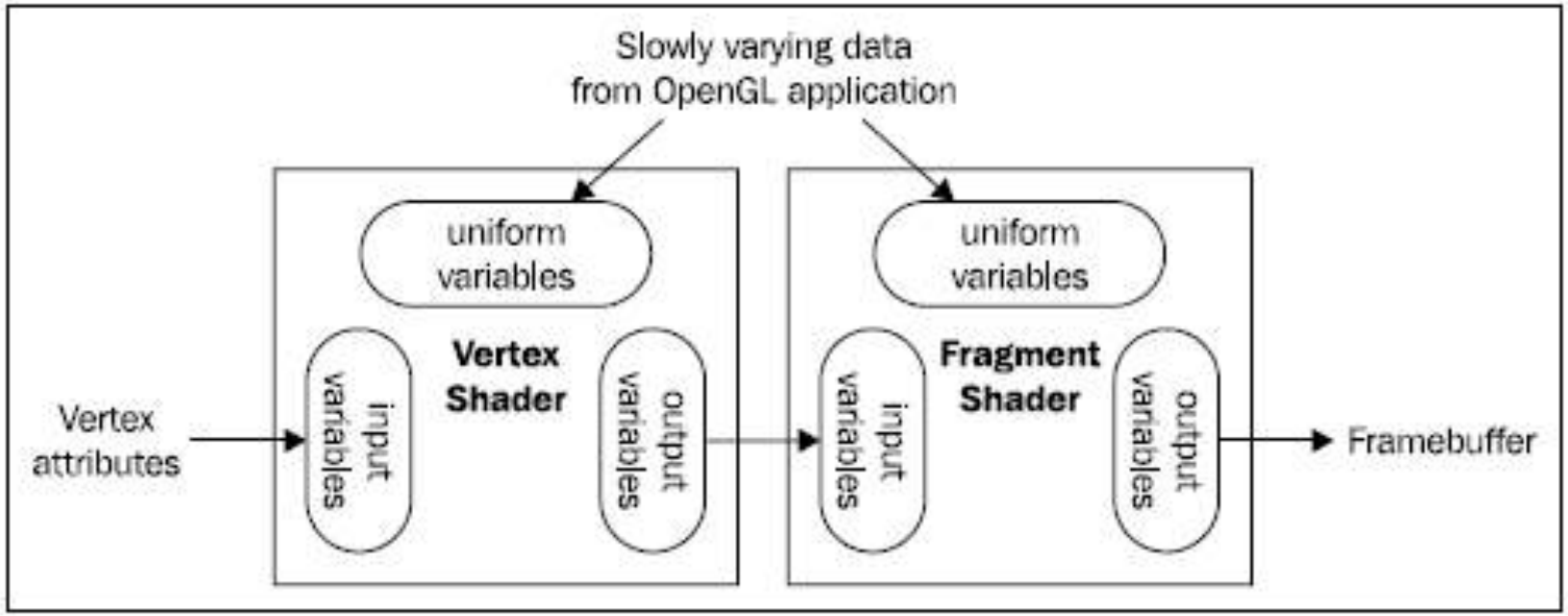
```
struct light {                  // light es el nombre del tipo de datos
          float intensity;
          vec3 position;
} lightVar;

float frequencies[3];
light lights[8];
```

- Funciones, como en C. Pueden sobrecargarse, como en C++. Los parámetros pueden ser constantes (**`const`**), de entrada (**`in`**), de salida (**`out`**) o de entrada y salida (**`inout`**).
- Estructuras de control, como en C. Sentencia **`discard`**: abandonar el procesamiento del fragmento actual (sólo para fragment shaders)

Comunicación entre la aplicación y los shaders

- Para comunicar la aplicación con los *shaders*, y los *shaders* entre ellos mediante variables, que pueden ser:
- **Variables de usuario**: las define el usuario a su conveniencia. Pueden ser:
  - ***In***: Variables de entrada que representan los atributos del vértice procesado: posición, color… Diferentes para cada llamada.
  - ***Out***: Variables de salida para pasar datos de un *shader* a otro posterior (por ejemplo, del *vertex shader* al *fragment shader*).
  - ***Uniform***: Variables que son comunes para toda la escena y no cambian de un vértice a otro: matrices, posición de la luz, …
- **Variables *built-in*** o incorporadas: ya vienen definidas y no hay que declararlas. Se utilizan para comunicar *shaders* entre sí

Comunicación entre la aplicación y los shaders

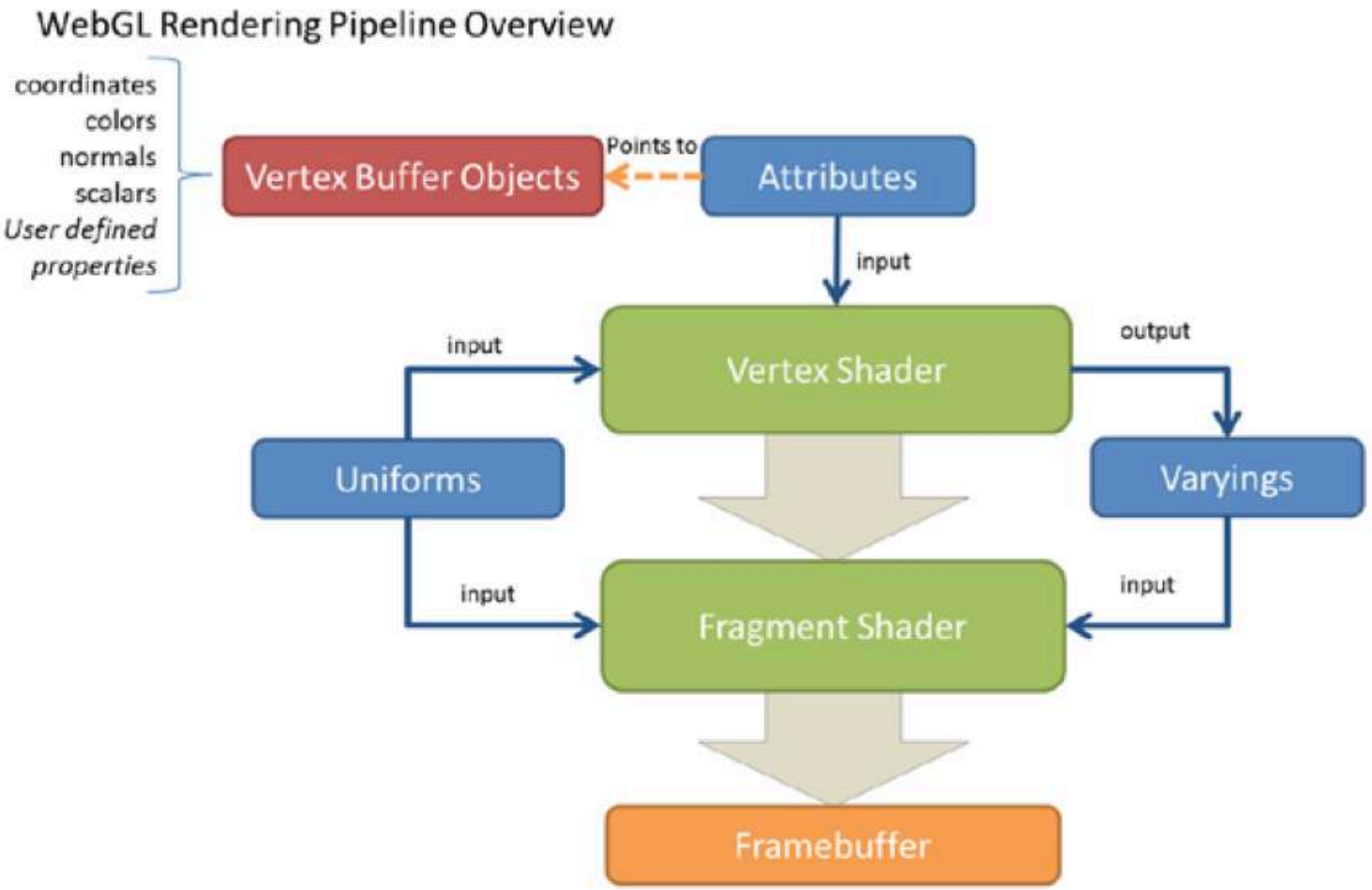


Variables de usuario

- El usuario puede definir variables locales o globales (para pasar datos propios entre módulos).
- Se utilizan calificadores para las variables, según uso.

Calificador	Descripción
<ninguno>	Variables locales y parámetros de funciones
const	Valor constante. El valor se fija durante la compilación y no puede cambiarse durante la ejecución. Al estilo de C++.
in	Variable de entrada, de solo lectura. Representan los atributos del vértice procesado (coordenadas, normal, color…), por lo tanto su valor es distinto en cada llamada del *vertex shader*.
out	Variable de salida, para la comunicación entre *shaders* (de salida para los *vertex shaders* y que sirven de entrada para los *fragment shaders*)
uniform	Variable de entrada. Representan atributos comunes para toda la escena (posición luces, niebla…), por lo tanto su valor es constante dentro de cada ciclo de dibujado.

Variables de usuario

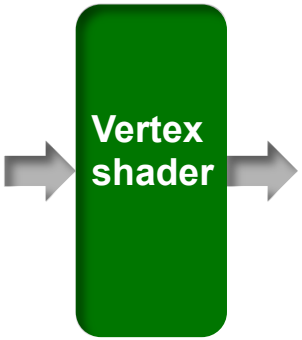


Variables incorporadas

- OpenGL se comunica con los *shaders* a través de variables *built-in* (ya definidas, no se declaran).
- Todas empiezan con el prefijo **`gl_`**.
- Son diferentes para los *vertex shaders* y para los *fragment shaders*.
- Pueden ser:
  - De entrada
  - De salida

Variables incorporadas. Vertex shaders

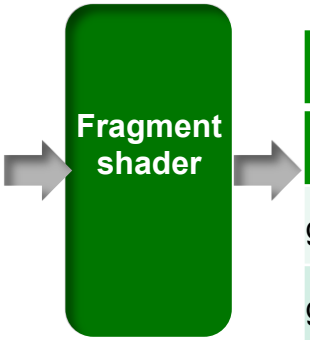
ENTRADA (in)		
Nombre	**Tipo**	**Descripción**
gl_vertexId	int	Índice vértice actual
gl_InstanceId	int	Índice instancia actual



SALIDA (out)		
Nombre	**Tipo**	**Descripción**
gl_Position	vec4	Coord. del vértice transformado
gl_PointSize	float	Tamaño del punto en píxeles
gl_ClipDistance	float	Distancia del vértice al plano de recorte

Variables incorporadas. Fragment shaders

ENTRADA (in)		
Nombre	**Tipo**	**Descripción**
gl_FragCoord	vec4	Coord. de fragmento
gl_FrontFacing	bool	Es frontal
gl_PointCoord	vec2	Coord. del punto que define la posición del fragmento
…	…	…



SALIDA (out)		
Nombre	**Tipo**	**Descripción**
gl_FragDepth	float	Profundidad del fragmento
gl_SampleMask	int	Máscara de muestreo (para antialiasing)

Otros elementos predefinidos

- Además de las variables incorporadas, hay otros elementos predefinidos:
  - Constantes: **`gl_MaxVertexAttribs`**, **`gl_MaxVertexOutputComponents`**, **`gl_MaxVertexUniformComponents`**, **`gl_MaxVertexTextureImageUnits`**…
  - Funciones: **`sin`**, **`cos`**, **`pow`**, **`log`**, **`min`**, **`max`**, **`length`**, **`dot`**, **`cross`**, **`normalize`**…

Vertex shaders

- El *vertex shader* se llama para cada vértice, normalmente en paralelo
- Este *shader* manipula los datos por vértice: coordenadas de vértice, normales, colores, coordenadas de textura…
- Estos datos provienen de la descripción de cada vértice y se representan a través de los atributos (variables con el calificador *attribute*)

Vertex shaders

- Desde el vertex shader no se puede acceder a:
  - Información de conectividad (caras)
  - Información sobre otros vértices
  - Información sobre el framebuffer
- Se puede acceder a:
  - Información sobre el propio vértice, a través de variables *attribute*: posición, normal, color, textura…
  - Estado de OpenGL, a través de variables *uniform*: matrices, luces…
  - Otra información a través de las variables *built-in*

Vertex shaders

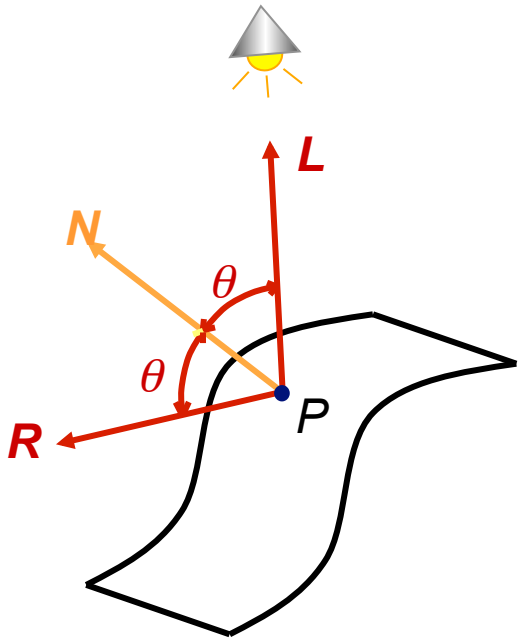
- Como mínimo, debe calcular la posición de los vértices, y asignarla a la variable *built-in* **`gl_Position`**
- Si queremos, podemos manipular en este *shader* otros aspectos como el color o las coordenadas de textura, y pasarlas al *fragment shader* mediante variables *varying*.
- No es posible pasar datos entre *vertex shaders*

Vertex shaders

sombreado por vértice, con una sola luz, difusa

- Componente difusa del modelo de Phong

$$I_{difusa} = I_d\, k_d(\lambda)\, \cos\theta = I_d\, k_d\,(\lambda)\, (L \cdot N) \qquad 0 \le \theta \le 2\pi$$



Vertex shaders

sombreado por vértice, con una sola luz, difusa

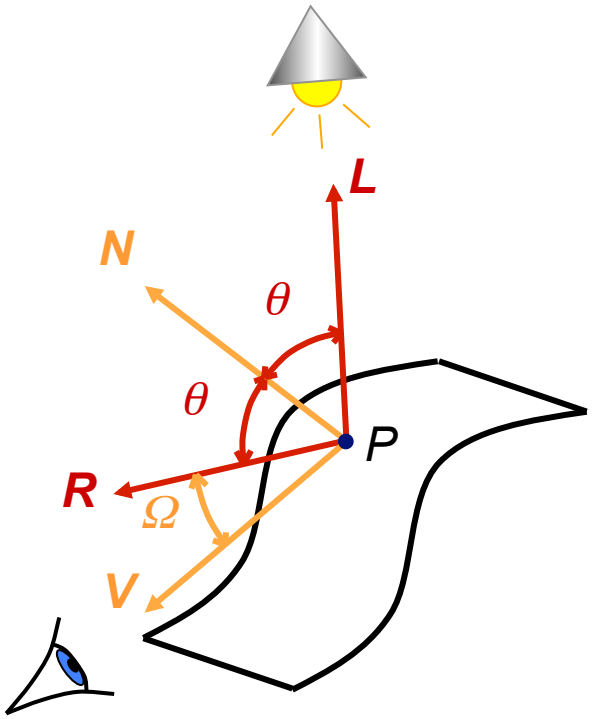
```
// ATRIBUTOS
layout (location = 0) in vec4 VertexPosition; //VERTICE EN COORDENADAS GLOBALES
layout (location = 1) in vec3 VertexNormal;   //NORMAL EN COORDENADAS GLOBALES

// SALIDA PARA COMUNICAR CON EL FRAGMENT
out vec3 LightIntensity;         //COLOR DEL VERTICE

// ESTADO DE OPENGL
uniform vec4 LightPosition;      //POSICIÓN DE LA LUZ EN COORDENADAS GLOBALES
uniform vec3 Ld;                 //INTENSIDAD DIFUSA DE LA LUZ
uniform vec3 Kd;                 //COMPONENTE DIFUSA DEL MATERIAL

uniform mat4 ModelMatrix;        //MATRIZ DE MODELO
uniform mat4 ViewMatrix;         //MATRIZ DE VISTA (INVERSA DE LA CAMARA)
uniform mat3 NormalMatrix;       //MATRIZ DE NORMALES
uniform mat4 ProjectionMatrix;   //MATRIZ DE PROYECCIÓN

void main(){

          // TRANSFORMAR VERTICE, NORMAL Y LUZ A COORDENADAS DE VISTA
          vec4 eyeVertex = ViewMatrix * ModelMatrix * VertexPosition;
          vec3 eyeNormal = normalize (NormalMatrix * VertexNormal;
          vec4 eyeLight  = ViewMatrix * LightPosition;

          // CALCULAR EL VECTOR QUE VA DEL VÉRTICE A LA LUZ
          vec3 lightVector = normalize( vec3(eyeLight - eyeVertex));

          // CALCULAR EL SOMBREADO DIFUSO
          LightIntensity = Ld * Kd * max (dot (LightVector, eyeNormal), 0.0);

          // TRANSFORMACION COMPLETA DEL VERTICE
          gl_Position = ProjectionMatrix * ViewMatrix * ModelMatrix * VertexPosition;
}
```

Vertex shaders

sombreado por vértice, con una sola luz, amb+dif+spe

- Componente difusa del modelo de Phong

$$I_{Phong} = I_a k_a(\lambda) + I_d k_d(\lambda)(\mathbf{L}\cdot\mathbf{N}) + I_e k_e(\lambda)(\mathbf{R}\cdot\mathbf{V})^n$$



Vertex shaders

sombreado por vértice, con una sola luz, amb+dif+spe

```
// ATRIBUTOS
layout (location = 0) in vec4 VertexPosition; //VERTICE EN COORDENADAS GLOBALES
layout (location = 1) in vec3 VertexNormal;   //NORMAL EN COORDENADAS GLOBALES

// SALIDA PARA COMUNICAR CON EL FRAGMENT
out vec3 LightIntensity;                      //COLOR DEL VERTICE

// ESTADO DE OPENGL
struct LightInfo {                            //ATRIBUTOS DE LA LUZ
  vec4 Position;                              //OJO!!: LUZ EN COORDENADAS DE VISTA
  vec3 La;
  vec3 Ld;
  vec3 Ls;
};
uniform LightInfo Light;

struct MaterialInfo {                         //ATRIBUTOS DEL MATERIAL
  vec3 Ka;
  vec3 Kd;
  vec3 Ks;
  float Shininess;
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;                 //MATRIZ DE MODELO Y VISTA (YA MULTIPLICADAS
uniform mat3 NormalMatrix;                    //MATRIZ DE NORMALES
uniform mat4 ProjectionMatrix;                //MATRIZ DE PROYECCIÓN
Uniform mat4 MVP;                             //MATRIZ MODELO*VISTA*PROYECCION
```

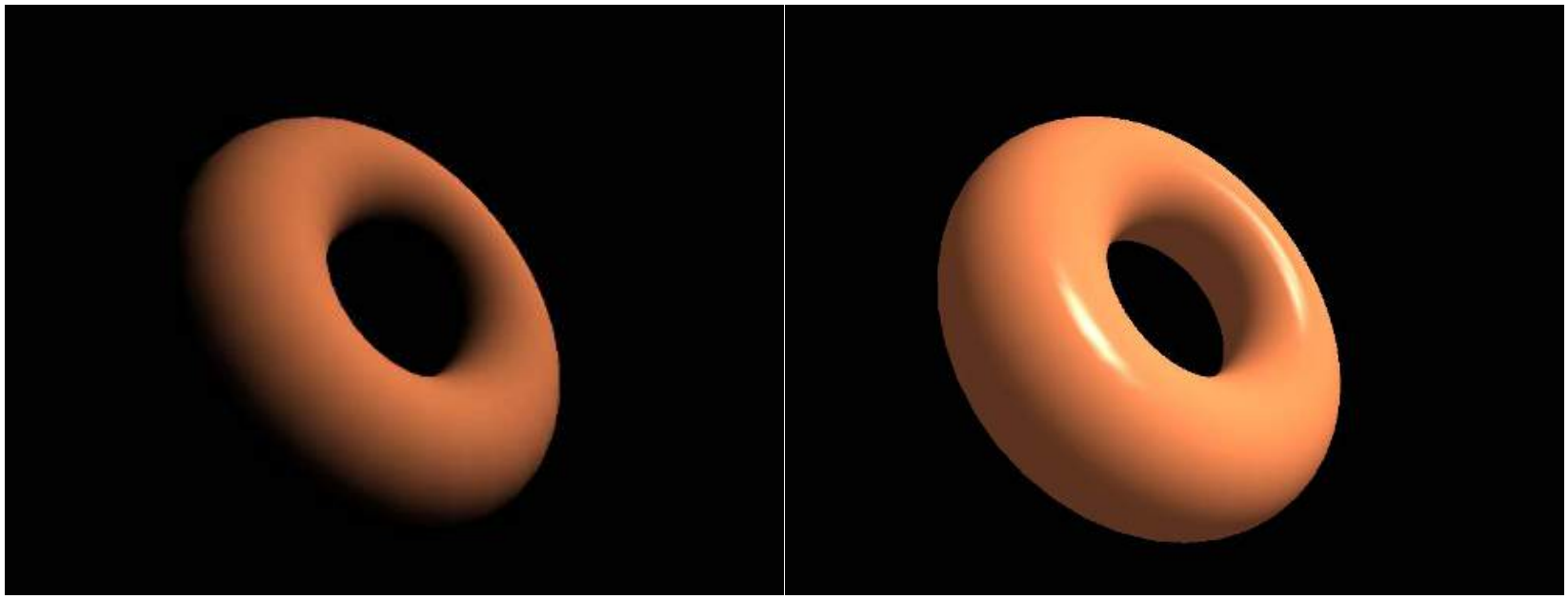
Vertex shaders

sombreado por vértice, con una sola luz, amb+dif+spe

```
void main(){

        // TRANSFORMAR EL VERTICE Y LA NORMAL A COORDENADAS DE VISTA
        vec4 eyeVertex = ModelViewMatrix * VertexPosition;
        vec3 eyeNormal = normalize (NormalMatrix * VertexNormal;

        // CALCULAR VECTORES VÉRTICE->LUZ, VÉRTICE->OBSERVADOR, REFLEXION y LUZ*NORMAL
        vec3 lightVector = normalize (vec3(Light.Position - eyeVertex));
        vec3 viewVector  = normalize (-eyeVertex.xyz);
        vec3 reflVector  = reflect (-lightVector, eyeNormal);
        float lightDOTnormal = max (dot (lightVector, eyeNormal), 0.0);

        // CALCULAR COMPONENTES DEL SOMBREADO
        vec3 ambient = Light.La * Material.Ka;
        vec3 diffuse = Light.Ld * Material.Kd * lightDOTnormal;
        vec3 specular = vec3 (0.0);
        if (lightDOTnormal>0.0) {
                float reflDOTview = max (dot (reflVector, viewVector), 0.0);
                specular = Light.Ls * Material.Ls * pow (reflDOTview, Material.Shininess);
        }

        // CALCULAR ILUMINACIÓN COMPLETA
        LightIntensity = ambient + difusse + specular;

        // TRANSFORMACIÓN COMPLETA DEL VÉRTICE
        gl_Position = MVP * VertexPosition;
}
```

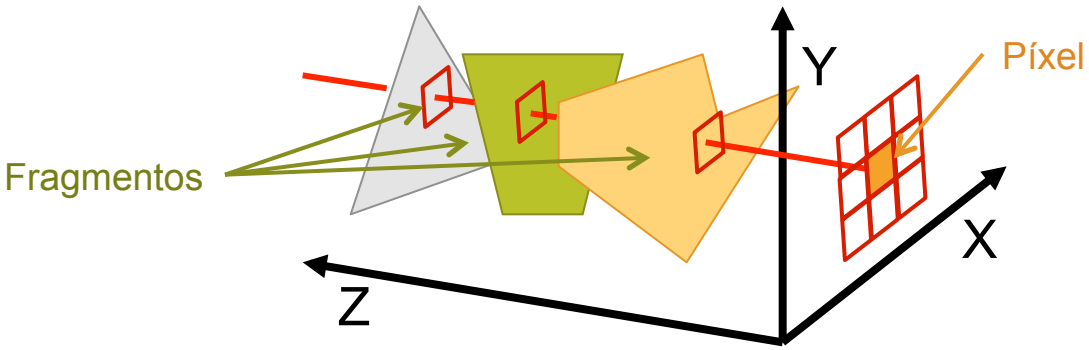
Comparación

componente difusa vs. componentes amb+dif+spe



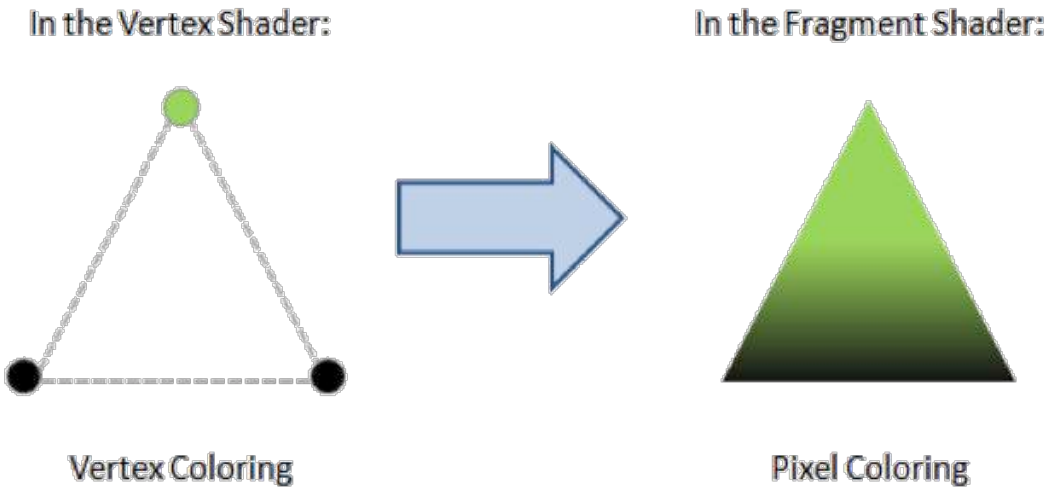
Fragment shaders

- El *fragment shader* se llama para cada fragmento
- Un fragmento es cada elemento de una superficie que se representará como un píxel, por eso a veces se habla incorrectamente de *pixel shader*
- Diferencia entre fragmento y pixel:



Fragment shaders

- El objetivo de este *shader* es calcular el color final de cada fragmento a partir de los datos de los vértices implicados que provienen del *vertex shader*



Fragment shaders

- Desde el *frament shader* no se puede acceder a información sobre otros fragmentos
- Se puede acceder a:
  - Información sobre el propio fragmento
  - Estado de OpenGL
  - Posición del fragmento (pero no se puede cambiar)

Fragment shaders

- Como mínimo, debemos calcular:
  - El color del fragmento.
  - Si queremos podemos cambiar la profundidad del fragmento.
- Otros cálculos:
  - No se puede cambiar la posición de los fragmentos
  - No es posible pasar datos entre *fragment shaders*

Fragment shaders

sombreado por fragmento, con una sola luz, amb+dif+spe

```
// VERTEX SHADER

// ATRIBUTOS
layout (location = 0) in vec4 VertexPosition; //VERTICE EN COORDENADAS GLOBALES
layout (location = 1) in vec3 VertexNormal;   //NORMAL EN COORDENADAS GLOBALES

// SALIDA PARA COMUNICAR CON EL FRAGMENT
out vec3 Position;                        //VERTICES EN COORDINADAS DE VISTA
out vec3 Normal;                          //NORMAL EN COORDENADAS DE VISTA

// ESTADO DE OPENGL
uniform mat4 ModelViewMatrix;             //MATRIZ DE MODELO Y VISTA (YA MULTIPLICADAS
uniform mat3 NormalMatrix;                //MATRIZ DE NORMALES
uniform mat4 ProjectionMatrix;            //MATRIZ DE PROYECCIÓN
Uniform mat4 MVP;                         //MATRIZ MODELO*VISTA*PROYECCION

void main(){

          // TRANSFORMAR EL VERTICE Y LA NORMAL A COORDENADAS DE VISTA
          Position = vec3 (ModelViewMatrix * VertexPosition);
          Normal = normalize (NormalMatrix * VertexNormal;

          // TRANSFORMACIÓN COMPLETA DEL VÉRTICE
          gl_Position = MVP * VertexPosition;
}
```

Fragment shaders

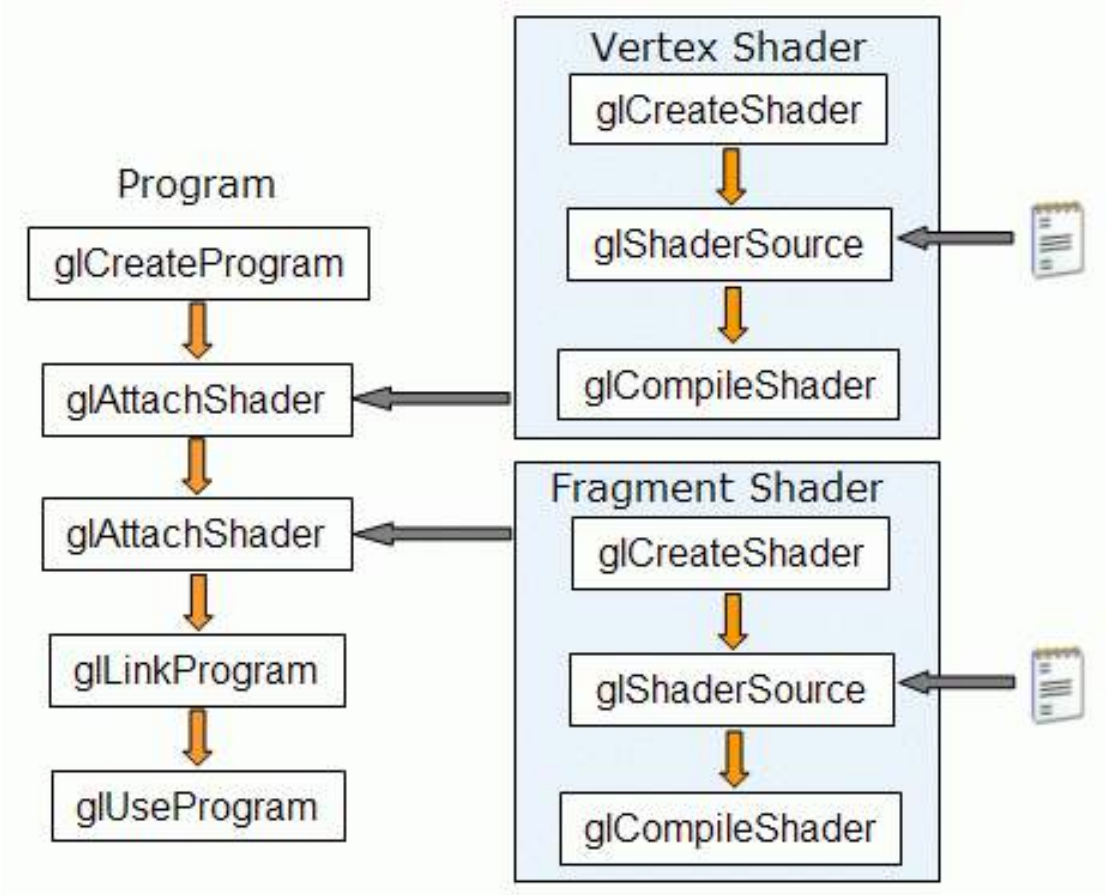
sombreado por fragmento, con una sola luz, amb+dif+spe

```
// FRAGMENT SHADER

// ENTRADA, PROVENIENTE DEL VERTEX SHADER
in vec3 Position;                          //VERTICES EN COORDINADAS DE VISTA
in vec3 Normal;                            //NORMAL EN COORDENADAS DE VISTA

// SALIDA PARA COMUNICAR CON EL RESTO DEL PIPELINE
Layout (location=0) out vec4 FragColor; //COLOR DEL FRAGMENTO

// ESTADO DE OPENGL
uniform vec4 LightPosition;      //POSICIÓN DE LA LUZ EN COORDENADAS DE VISTA
uniform vec3 LightIntensity;     //INTENSIDAD DE LA LUZ
uniform vec3 Kd;                 //COMPONENTE DIFUSA DEL MATERIAL
uniform vec3 Ka;                 //COMPONENTE AMBIENTAL DEL MATERIAL
uniform vec3 Ks;                 //COMPONENTE ESPECULAR DEL MATERIAL
Uniform float Shininess;         //FACTOR DE BRILLO DEL MATERIAL

// FUNCION QUE CALCULA EL MODELO DE PHONG
Vec3 Phong () {
          vec3 n = normalize (Normal);
          vec3 s = normalize (vec3 (LightPosition) - Position);
          vec3 v = normalize (vec3 (-Position));
          vec3 r = reflect (-s, n);
          vec3 light = LightIntensity * (Ka + Kd * max (dot (s, n), 0.0) + Ks *
                       pow (max (dot (r,v), 0.0), Shininess));
          return light;
}

void main () // CALCULAR EL COLOR DEL FRAGMENTO
{
          FragColor = vec4 (Phong (), 1.0);
}
```

Comparación

sombreado por vértice (Gouraud) vs. sombreado por fragmento (Phong)

Utilizar shaders desde OpenGL



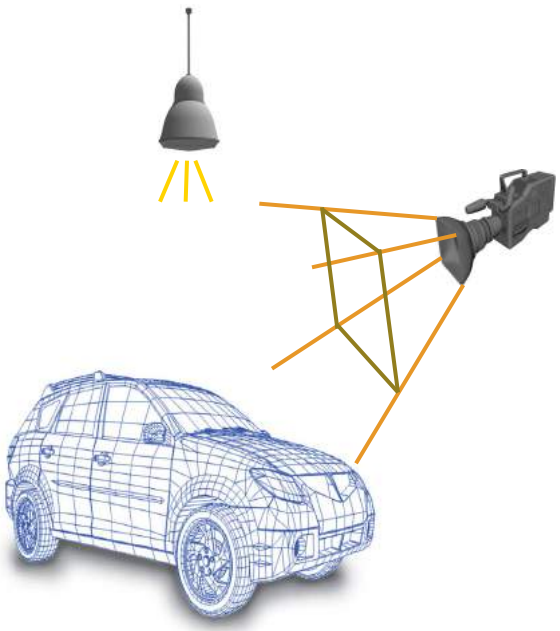
Ejemplo
versión básica

```
void setShaders() {
        GLuint vsHandle, fsHandle, programHandle;
        const GLchar *vsContent, *fsContent;

        // CREAR LOS SHADERS
        vsHandle = glCreateShader (GL_VERTEX_SHADER);
        fsHandel = glCreateShader (GL_FRAGMENT_SHADER);

        // LEER LOS FICHEROS Y DEFINIRLOS COMO FUENTES
        vsContent = MiFuncionParaLeerFicheros ("toon.vert");
        fsContent = MiFuncionParaLeerFicheros ("toon.frag");

        glShaderSource (vsHandle, 1, &vsContent, NULL);
        glShaderSource (fsHandle, 1, &fsContent, NULL);

        // COMPILAR LOS SHADERS
        glCompileShader (vsHandle);
        glCompileShader( fsHandle);

        // CREAR EL PROGRAMA, ASOCIARLES LOS SHADERS Y ENLAZARLO TODO
        programHandle = glCreateProgram();
        glAttachShader (programHandle , fsHandle);
        glAttachShader (programHandle , vsHandle);
        glLinkProgram (programHandle);

        ...

        // AHORA SE PUEDE USAR EN CUALQUIER MOMENTO
        glUseProgram (programHandle );
}
```

Crear / Eliminar shaders

- La lectura de fichero, compilación y enlazado de *shaders* es lenta, por eso conviene crear los *shaders*, compilarlos, y enlazarlos para crear el programa una sola vez.
- Cuando no se vayan a utilizar, pueden borrarse tanto los *shaders* como los programas para liberar memoria:
  - **`glDeleteShader`**
  - **`glDeleteProgram`** (borra el programa pero no los shaders asociados)

Comunicación con los shaders



Uniforms

Datos referidos a todos los vértices:

- Matriz de modelo y/o vista
- Matriz de proyección
- Posición de las luces
- …

Atributos (*in*)

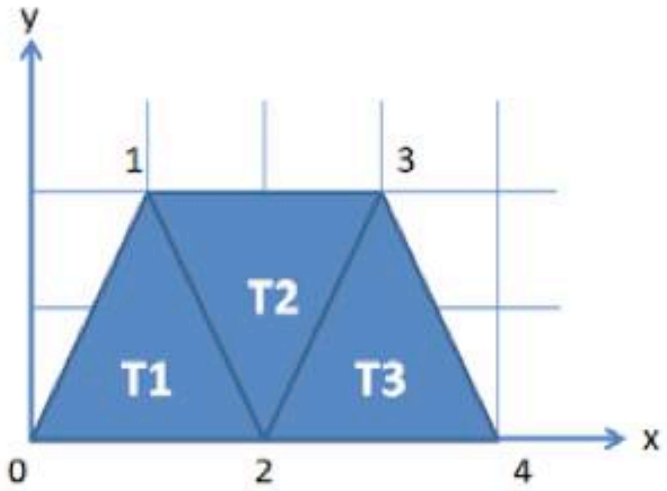
Datos distintos para cada vértice:

- Posición del vértice
- Normal
- Coordenada de textura
- …

Vertex shader

Comunicación con los shaders
Atributos. Usando glDrawArrays

Using drawArrays



On the x-y plane (discarding z-coordinate)

#	Vertex Coordinates
0	(0,0)
1	(10,20)
2	(20,0)
3	(30,20)
4	(40,0)

Vertex array = [0,0,10,20,20,0,10,20,20,0,30,20,20,0,30,20,40,0]

Triangle 1 Triangle 2 Triangle 3

`drawArrays` uses the vertex data in the order they are defined in the vertex array

Comunicación con los shaders
Atributos. Usando glDrawArrays

- En el shader:
  - Definir las variables como *in*

```
in vec3 VertexPosition;
in vec3 VertexColor;
```

- En el programa:
  - Supongamos los datos creados y los buffers rellenos de datos
  - Antes de enlazar (*link*) el *shader*, asignar una localización para las variables *in*.
  - Asignar un *handle* a cada buffer y asociarle los datos
  - Asociar cada buffer a una variable *in*
  - Dibujar los arrays

```
float positionData[] = {
     -0.8f, -0.8f, 0.0f,
      0.8f, -0.8f, 0.0f,
      0.0f,  0.8f, 0.0f
};

float colorData[] = {
      1.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f,
      0.0f, 0.0f, 1.0f
};
```

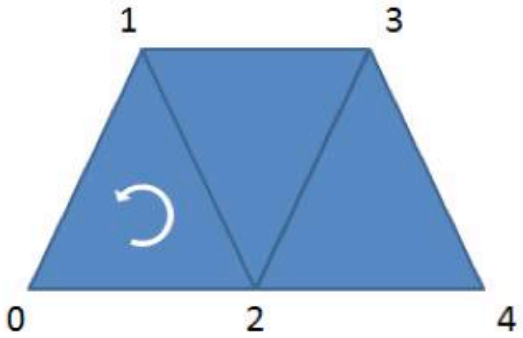
Comunicación con los shaders
Atributos. Usando glDrawArrays

```
// Asignar indices 0 y 1 a los atributos VertexPosition y VertexColor; linkar el shader
glBindAttribLocation(programHandle, 0, "VertexPosition");
glBindAttribLocation(programHandle, 1, "VertexColor");
glLinkProgram (programHandle);
          ...
// Crear dos buffers de atributos. Para cada uno: asociarles los datos (suponemos que
// están en dos arrays positionData y colorData), indicar el tamaño y posición,
// enlazarlos con la variable in del shader y habilitar los buffers.
GLuint vboHandles[2];
glGenBuffers(2, &vboHandles);

glBindBuffer(GL_ARRAY_BUFFER, vboHandles[0]);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), positionData, GL_STATIC_DRAW);
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL );
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, vboHandles[1]);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), colorData, GL_STATIC_DRAW);
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL );
glEnableVertexAttribArray(1);

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

¿normalizar? salto entre componentes salto inicial

Comunicación con los shaders
Atributos. Usando glDrawElements

Vertex and Indices



Index	Vertex Coordinates
0	(0,0)
1	(10,10)
2	(20,0)
3	(30,10)
4	(40,0)

coordinates

Vertex array = [0,0,10,10,20,0,30,10,40,0] → Vertex Buffer

Index array = [0,2,1,1,2,3,2,4,3] → Index Buffer

triangles

Triangles in the index array are *usually* but not necessarily defined counter-clockwise.

Comunicación con los shaders
Atributos. Usando glDrawElements

- En el shader:
  - Definir las variables como *in*
- En el programa:
  - Supongamos los datos creados y los buffers rellenos de datos
  - Antes de enlazar (*link*) el *shader*, asignar una localización para las variables *in*.
  - Asignar un *handle* a cada buffer y asociarle los datos
  - Asociar cada buffer a una variable *in*
  - Dibujar los elements

```
in vec3 VertexPosition;
in vec3 VertexColor;
```

```
float positionData[] = {
     -0.8f, -0.8f, 0.0f,
      0.8f, -0.8f, 0.0f,
      0.0f,  0.8f, 0.0f,
      1.6f,  0.8f, 0.0f
};

float colorData[] = {
      1.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f,
      0.0f, 0.0f, 1.0f,
      1.0f, 0.0f, 0.0f
};

uint indexData[] = {
      0, 1, 2,
      1, 3, 2
};
```

Comunicación con los shaders
Atributos. Usando glDrawElements

```
// Asignar indices 0 y 1 a los atributos VertexPosition y VertexColor; linkar el shader
glBindAttribLocation(programHandle, 0, "VertexPosition");
glBindAttribLocation(programHandle, 1, "VertexColor");
glLinkProgram (programHandle);
          ...
// Crear dos buffers de atributos. Para cada uno: asociarles los datos (suponemos que
// están en dos arrays positionData y colorData), indicar el tamaño y posición,
// enlazarlos con la variable in del shader y habilitar los buffers.
GLuint vboHandles[3];
glGenBuffers(3, &vboHandles);

glBindBuffer(GL_ARRAY_BUFFER, vboHandles[0]);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), positionData, GL_STATIC_DRAW);
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL );
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, vboHandles[1]);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), colorData, GL_STATIC_DRAW);
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL );
glEnableVertexAttribArray(1);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboHandles[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(uint), indexData, GL_STATIC_DRAW);

// Dibujar
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Comunicación con los shaders
Atributos

- Uso del calificador *layout*:
  - Sirve para asignar índices a las variables *in* en el propio *shader*
  - De esta forma se eliminan las llamadas a `glBindAttribLocation` en el programa OpenGL

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;
```

- Una tercera manera de asignar índices es dejar que el linker lo haga automáticamente (consultar libro):
  - Se utilizan las funciones `glGetActiveAttrib` y `glGetAttribLocation`

Comunicación con los shaders
Uniforms

- Las variables uniform son de sólo lectura: no pueden cambiar de valor dentro del shader.
- En el shader:
  - Definir las variables como *uniform*

```
uniform mat4 RotationMatrix;
```

- En el programa:
  - Supongamos los datos creados y con datos
  - Obtener la localización de la variable con `glGetUniformLocation`
  - Asociar la variable uniform con los datos del programa con `GlUniformXXXX`
  - Dibujar los arrays o elements

Comunicación con los shaders
Uniforms

```
// Supongamos la matriz ya creada
mat4 rotationMatrix = glm::rotate(mat4(1.0f), angle, vec3(0.0f,0.0f,1.0f));

// Localizar la varialbe y obtener su índice
GLuint location =glGetUniformLocation(programHandle, "RotationMatrix");

// Si la localización es válida, asociar la variable con la matriz de datos
if (location >= 0)
          glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);

// Conectar con los atributos, como se ha explicado
...

// Dibujar los arrays
glBindVertexArray(vaoHandle);
glDrawArrays(GL_TRIANGLES, 0, 3 );
```

número de matrices a asignar
(puede ser un array)

¿transponer?