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

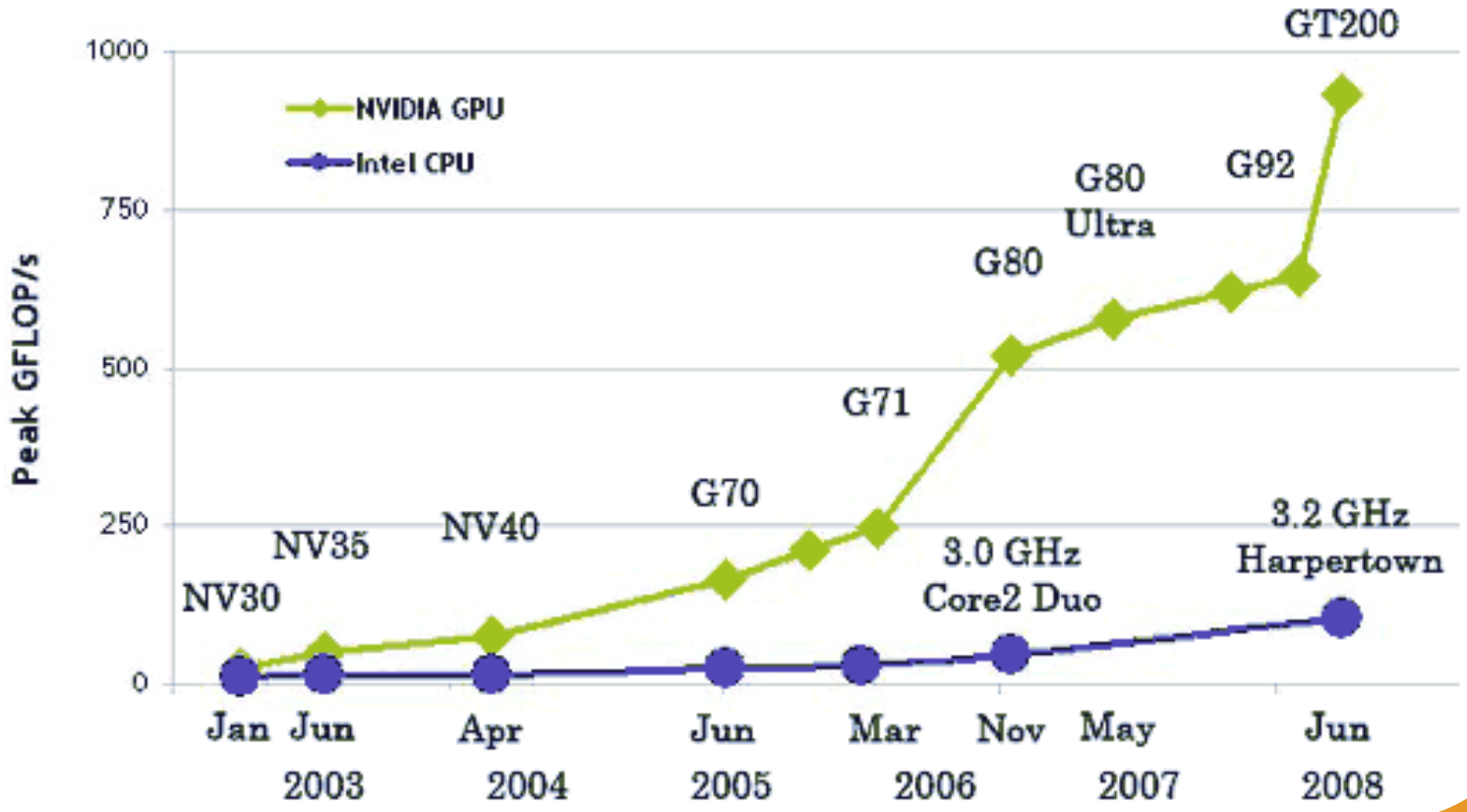
GPU vs. CPU

- 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

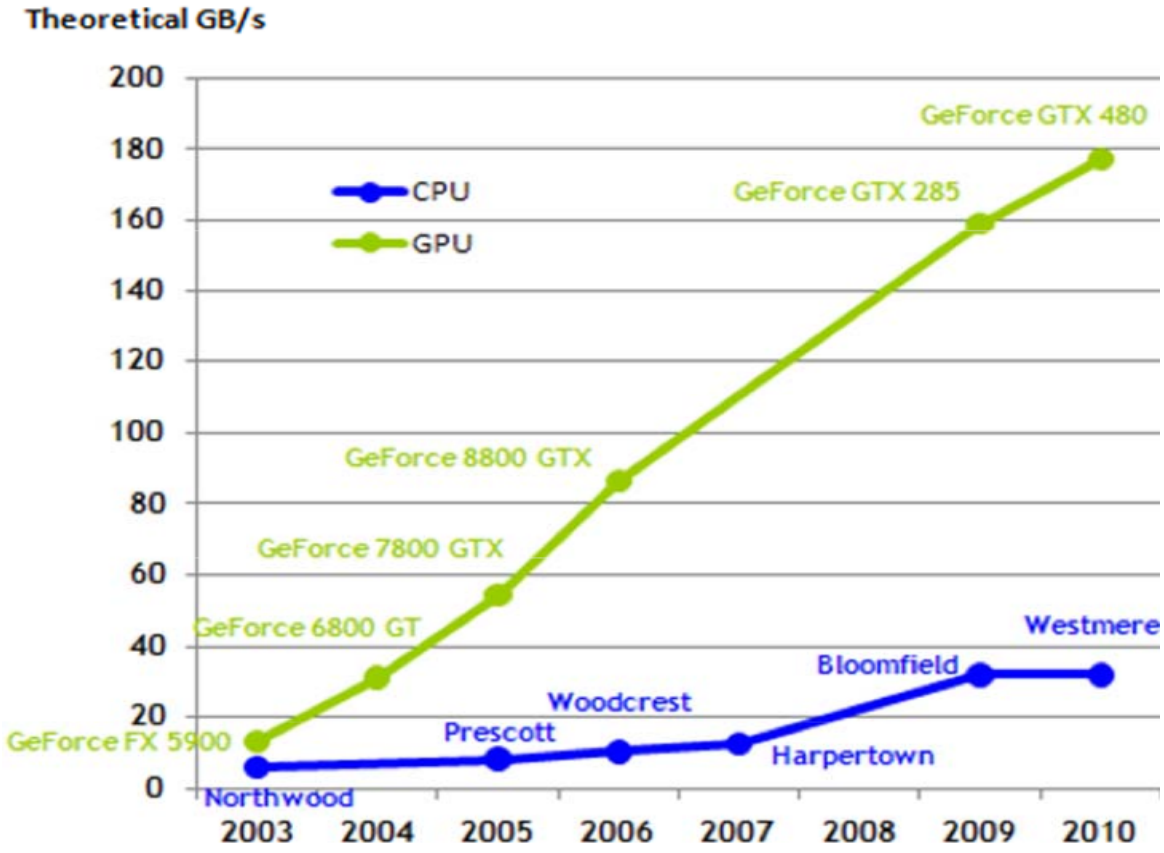
GPU vs. CPU

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

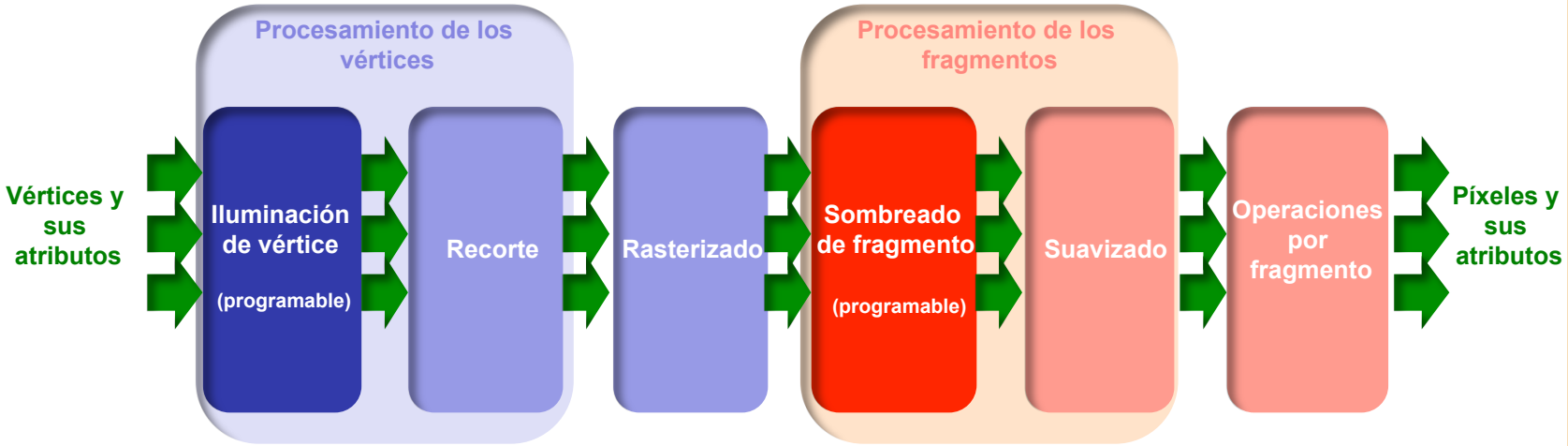
Aplicaciones de la GPU

- 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

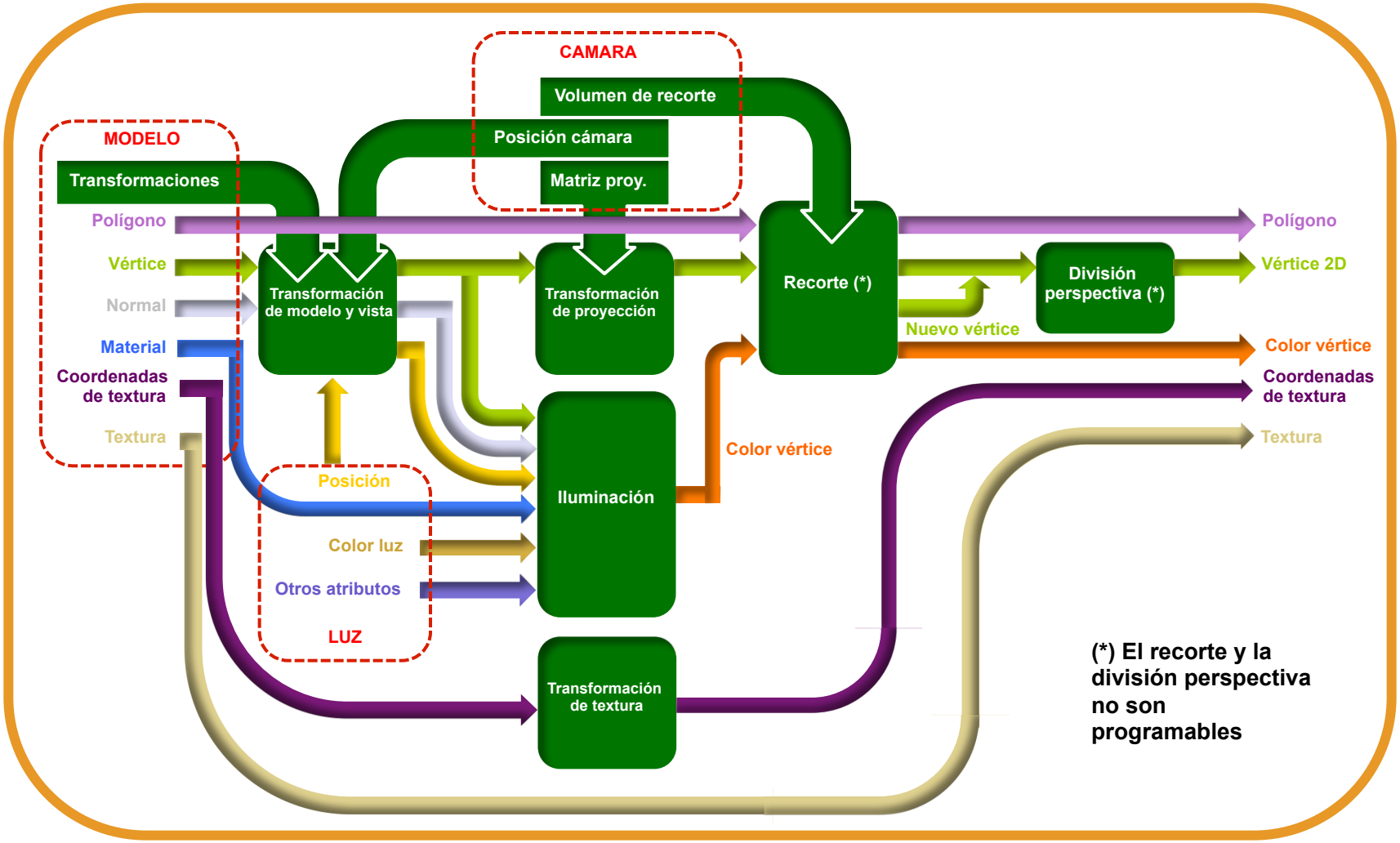
Programación de la GPU con GLSL (1.2)

- OpenGL *Shading Language*
- Veremos GLSL 1.2 (versiones posteriores incluyen nuevas funciones y otras se consideran obsoletas)
- Basado en ANSI C
- Permite programar algunas etapas del pipeline
- Pipeline 3D dividido en 4 etapas principales
  - Procesamiento de los vértices (programable)
  - Rasterizado
  - Procesamiento de los fragmentos (programable)
  - Operaciones sobre los fragmentos

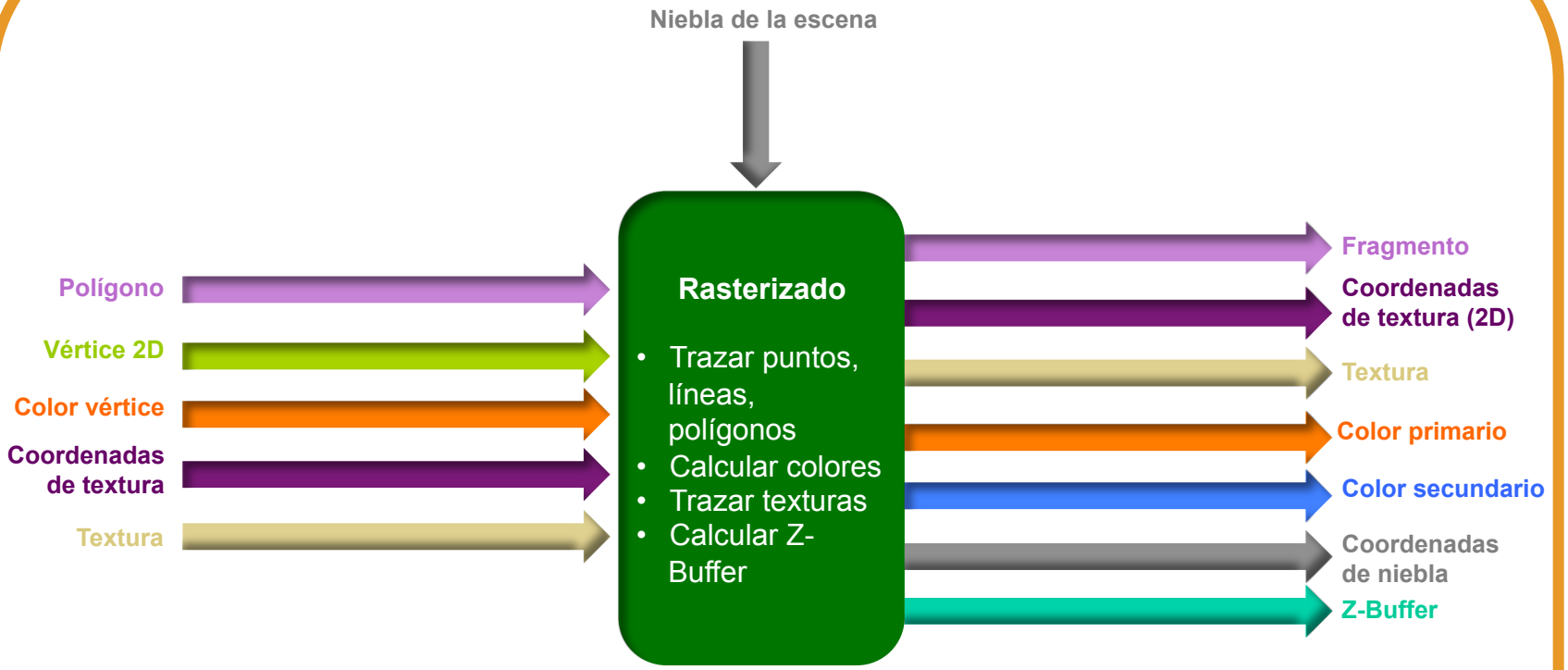
Pipeline OpenGL simplificado



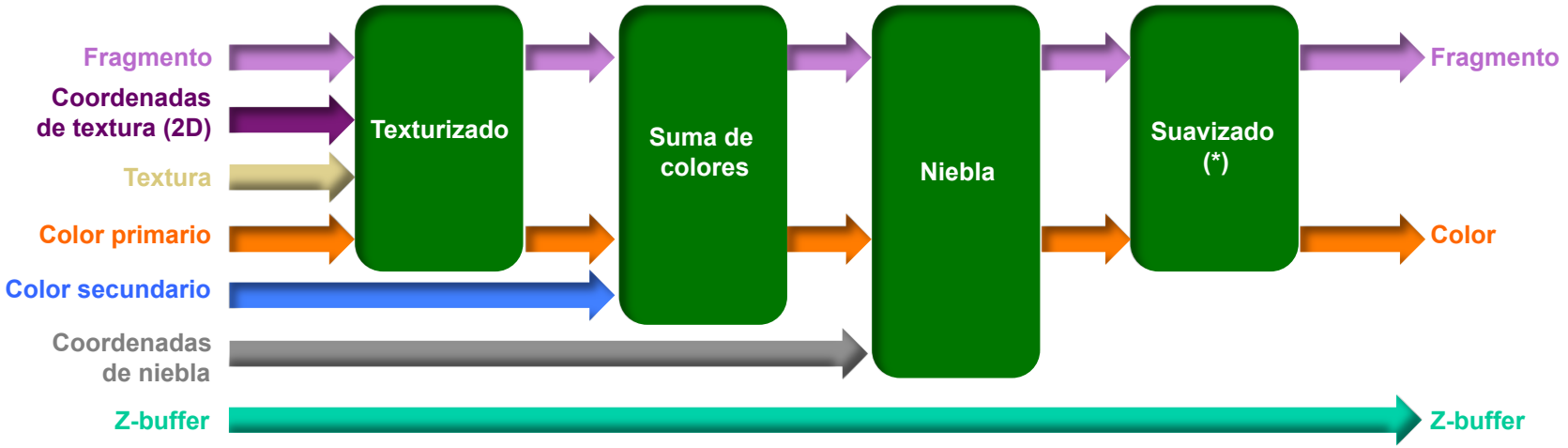
Pipeline vértices
Programable (*)



Pipeline raster
No programable

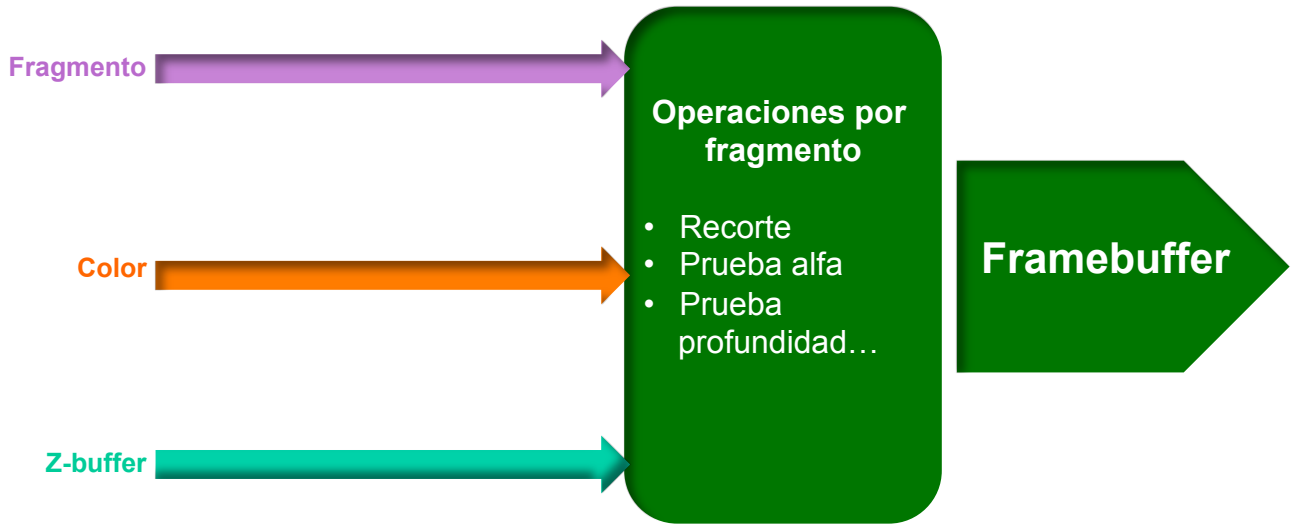


Pipeline fragmentos
Programable (*)



(*) El suavizado no es programable

Pipeline fragmentos
No programable



Shaders

- *Vertex shader*: programa de usuario que sustituye a procesamiento de vértices estándar de OpenGL:
  - Entrada: vértices y sus atributos
  - Salida: vértices transformados y otros atributos
- *Fragment shader*: programa de usuario que sustituye a procesamiento de fragmentos estándar de OpenGL:
  - 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	

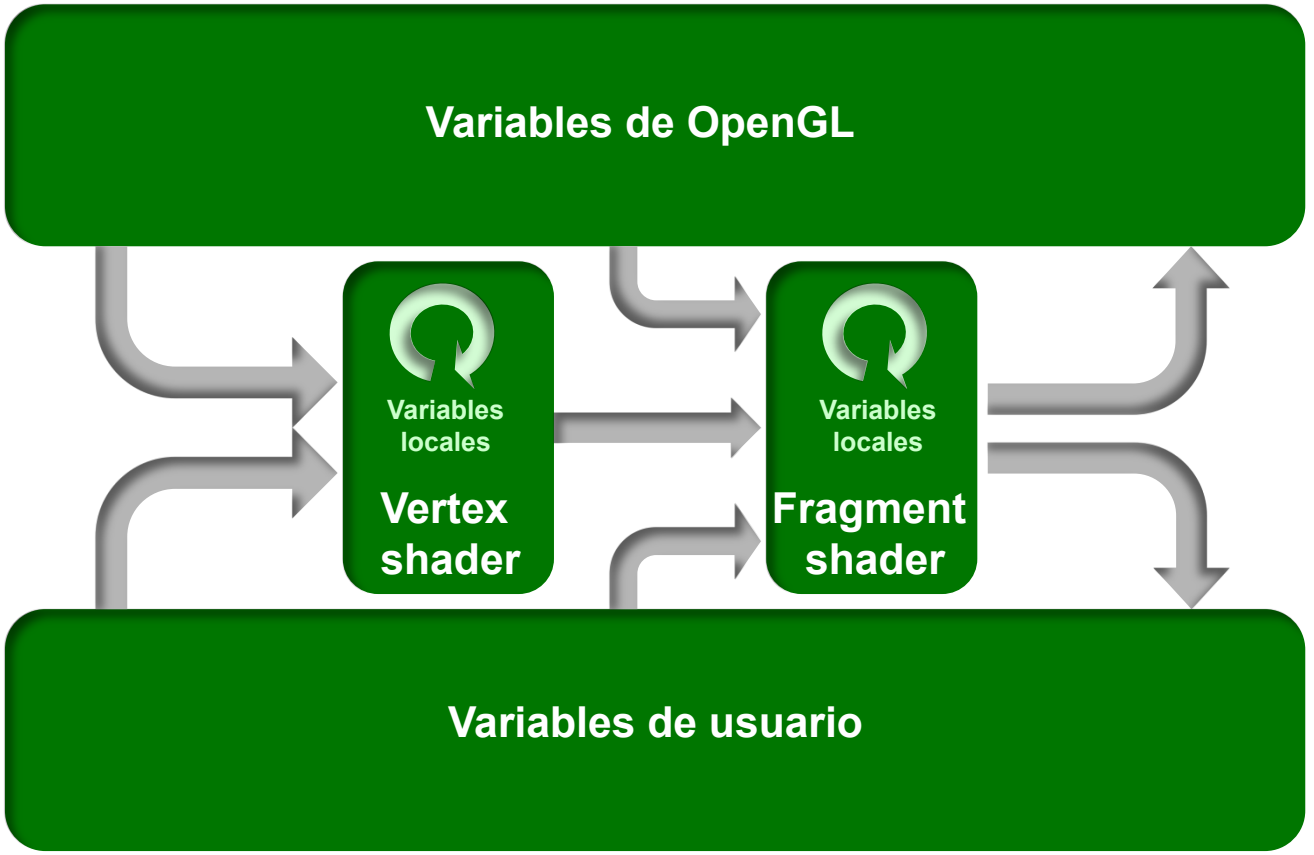
Tipos estructurados, funciones, estructuras de control

- 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

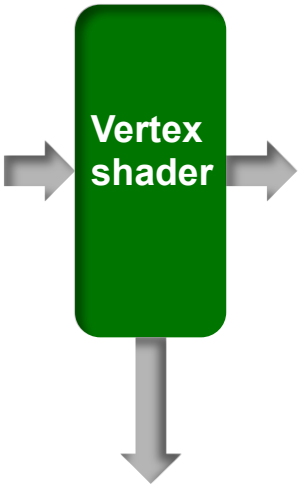


Variables incorporadas

- OpenGL se comunica con los *shaders* a través de variables incorporadas (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 (sólo lectura)
  - De salida
  - De salida hacia la siguiente etapa / entrada desde la anterior etapa

Variables incorporadas. Vertex shaders

ENTRADA		
Nombre	**Tipo**	**Descripción**
gl_Color	vec4	Color primario
gl_SecondaryColor	vec4	Color secundario
gl_Normal	vec3	Normal
gl_Vertex	vec4	Coord. del vértice
gl_MultiTexCoordn	vec4	Coord. de textura
gl_FogCoord	float	Coord. de niebla



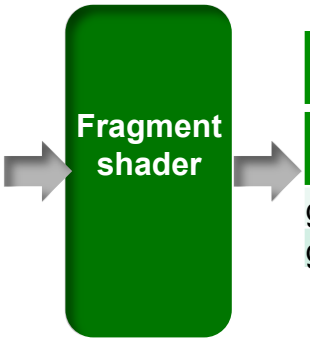
SALIDA		
Nombre	**Tipo**	**Descripción**
gl_Position	vec4	Coord. del vértice transformado
gl_PointSize	float	Tamaño del punto en píxeles
gl_ClipVertex	vec4	Coord. para recortar

SALIDA HACIA EL FRAGMENT SHADER		
Nombre	**Tipo**	**Descripción**
gl_FrontColor	vec4	Color primario frontal
gl_BackColor	vec4	Color primario posterior
gl_FrontSecondaryColor	vec4	Color secundario frontal
gl_BackSecondaryColor	vec4	Color secundario posterior
gl_TexCoord[]	vec4	Coord. de textura
gl_FogFragCoord	float	Coord. de niebla

Variables incorporadas. Fragment shaders

ENTRADA DESDE EL VERTEX SHADER		
Nombre	**Tipo**	**Descripción**
gl_Color	vec4	Color primario
gl_SecondaryColor	vec4	Color secundario
gl_TexCoord[]	vec4	Coord. de textura
gl_FogFragCoord	float	Coord. de niebla

ENTRADA		
Nombre	**Tipo**	**Descripción**
gl_FragCoord	vec4	Coord. de fragmento
gl_FrontFacing	bool	Es frontal



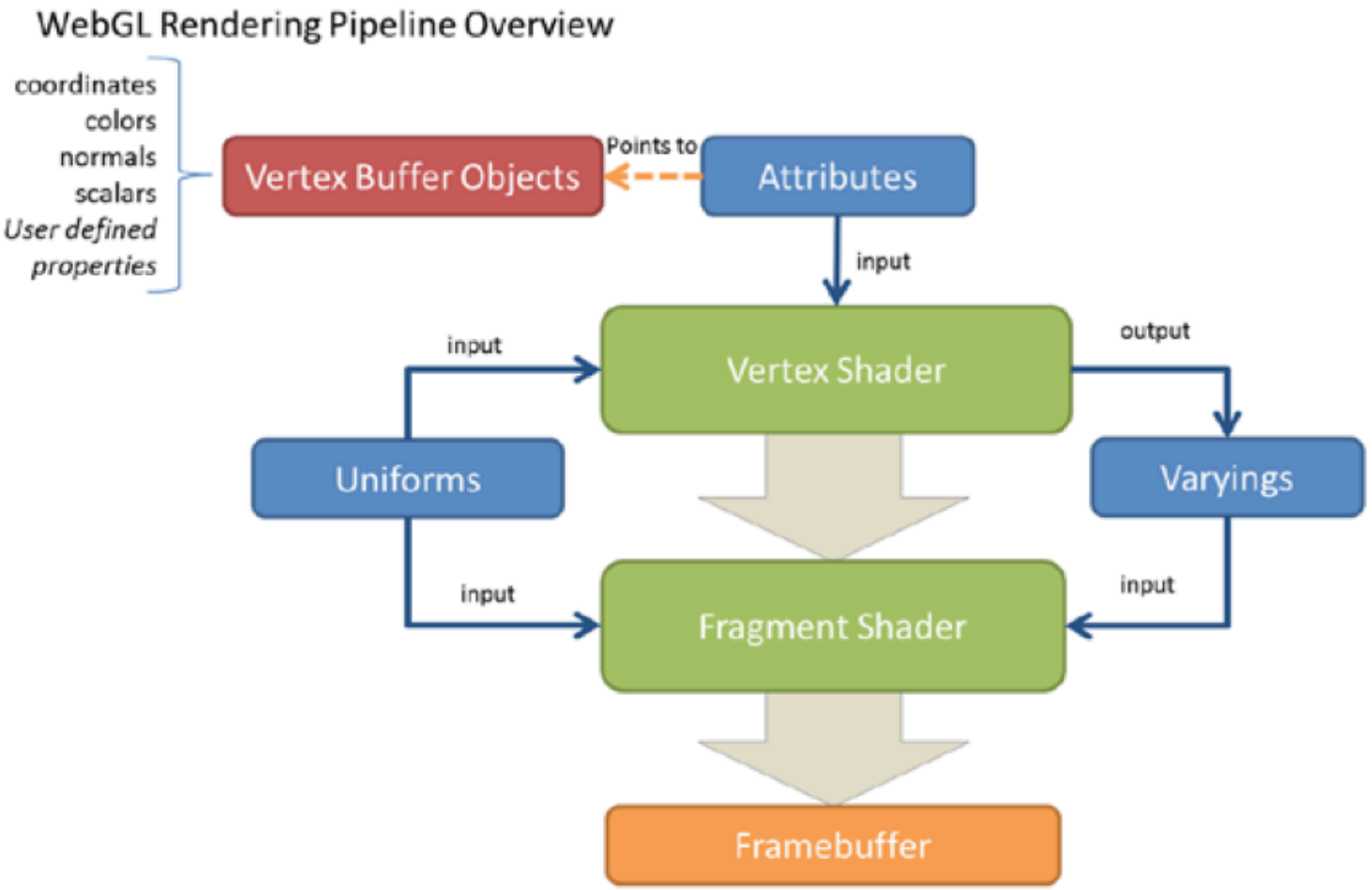
SALIDA		
Nombre	**Tipo**	**Descripción**
gl_FragColor	vec4	Color del fragmento
gl_FragDepth	float	Profundidad del fragmento

Variables de usuario

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

Calificador	Descripción
<ninguno>	Variables locales y parámetros de funciones
const	Valor constante. El valor se fija durante la compilación. Al estilo de C++.
attribute	Variable de entrada, de solo lectura, **sólo** para *vertex shaders*. Representan los atributos del vértice procesado (coordenadas, normal, color…), por lo tanto su valor es distinto en cada llamada del *vertex shader*.
uniform	Variable de entrada, para *vertex shaders* y *fragment shaders*. Representan atributos comunes para toda la escena (posición luces, niebla…), por lo tanto su valor es constante dentro de cada ciclo de dibujado.
varying	Variable de comunicación entre *shaders* (de salida para los *vertex shaders* y que sirven de entrada para los *fragment shaders*)

Variables de usuario



Otros elementos predefinidos

- Además de las variables incorporadas, hay otros elementos predefinidos:
  - Constantes: **`gl_MaxLights`**, **`gl_MaxClipPlanes`**, **`gl_MaxTextureUnits`**…
  - Variables de estado: **`gl_ModelViewMatrix`**, **`gl_ModelViewMatrixInverse`**, **`gl_ModelViewMatrixTranspose`**, **`gl_ProjectionMatrix`**, **`gl_TextureMatrix`**, **`gl_NormalMatrix`**, **`gl_MaterialParameters`**…
  - Funciones: **`sin`**, **`cos`**, **`pow`**, **`log`**, **`min`**, **`max`**, **`length`**, **`dot`**, **`cross`**, **`normalize`**…

Vertex shaders

- El *vertex shader* se llama para cada vértice
- Este *shader* manipula los datos por vértice: coordenadas de vértice, normales, colores y 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

- Al activar un vertex shader, se deshabilita parte de la funcionalidad fija
- Se deshabilita:
  - No se aplican las transformaciones modelview y projection
  - No se transforman las normales a coordenadas de vista, ni se reescalan ni se normalizan
  - No se aplica la iluminación por vértices
  - No se realizan los cálculos de material
  - No se generan automáticamente las coordenadas de textura
- Se mantiene
  - Normalización del color (clamp)
  - División perspectiva, mapeado en el viewport, escalado del rango de profundidad
  - Recorte
  - Ensamblaje de primitivas (primitive assembly) y rasterización
  - Determinación de la cara frontal
  - Sombreado plano (flat shading)

Vertex shaders

- No es obligatorio sustituir toda la funcionalidad fija, pero aquello que no se haga en el shader no lo hará OpenGL
- 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: posición, normal, color, textura…
  - Estado de OpenGL: matrices, luces

Vertex shaders

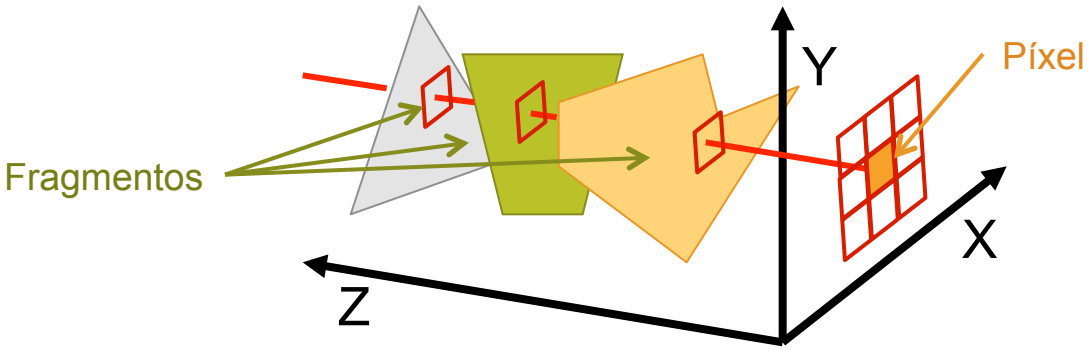
- Como mínimo, debemos calcular:
  - La posición de los vértices con **`gl_Position`**
  - Si queremos color, el color de los vértices con **`gl_FrontColor`**
  - Si queremos texturas, las coordenadas de textura **`con gl_TexCoord`**
- Otros cálculos:
  - Pasar datos al *fragment shader* utilizando variables **`varying`**
  - No es posible pasar datos entre *vertex shaders*

Vertex shaders

```
uniform vec3 lightPosition;   //valor fijo proveniente de nuestro programa
varying vec3 normal;          //valor que se envía al fragment shader
varying vec3 lightVector;     //valor que se envía al fragment shader

void main(){

          // Transformar el vértice a coordenadas de vista
          vec4 eyeVertex = gl_ModelViewMatrix * gl_Vertex;

          // Transformar y normalizar el vector normal
          normal = normalize(gl_NormalMatrix * gl_Normal);

          // Calcular el vector que va del vértice a la luz
          lightVector = normalize(lightPosition - eyeVertex.xyz);

          // Transformar la posición del vértice
          gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

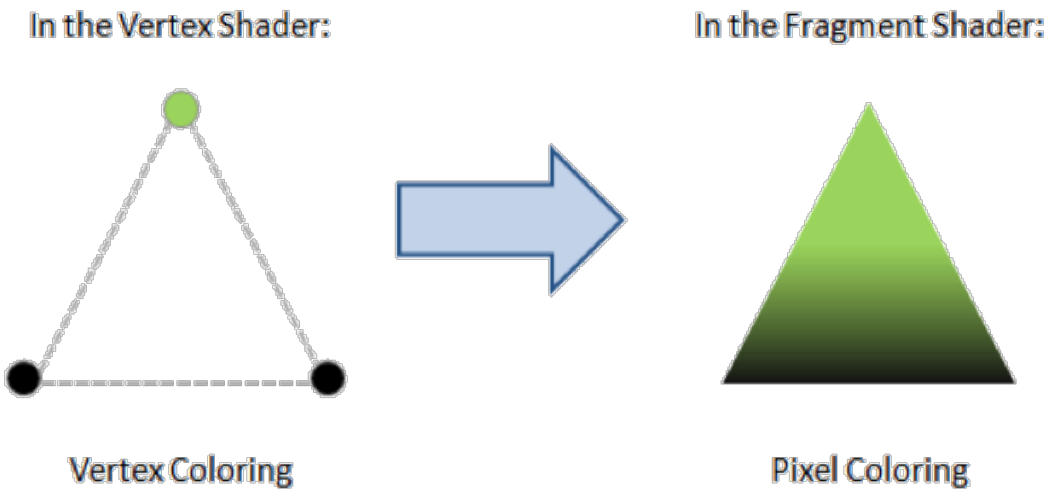
Fragment shaders

- 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

- Al activar un *fragment shader*, se deshabilita parte de la funcionalidad fija:
  - No se aplican las texturas
  - No se aplica la suma de colores
  - No se aplica la niebla
- No es obligatorio sustituir toda la funcionalidad
- 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)

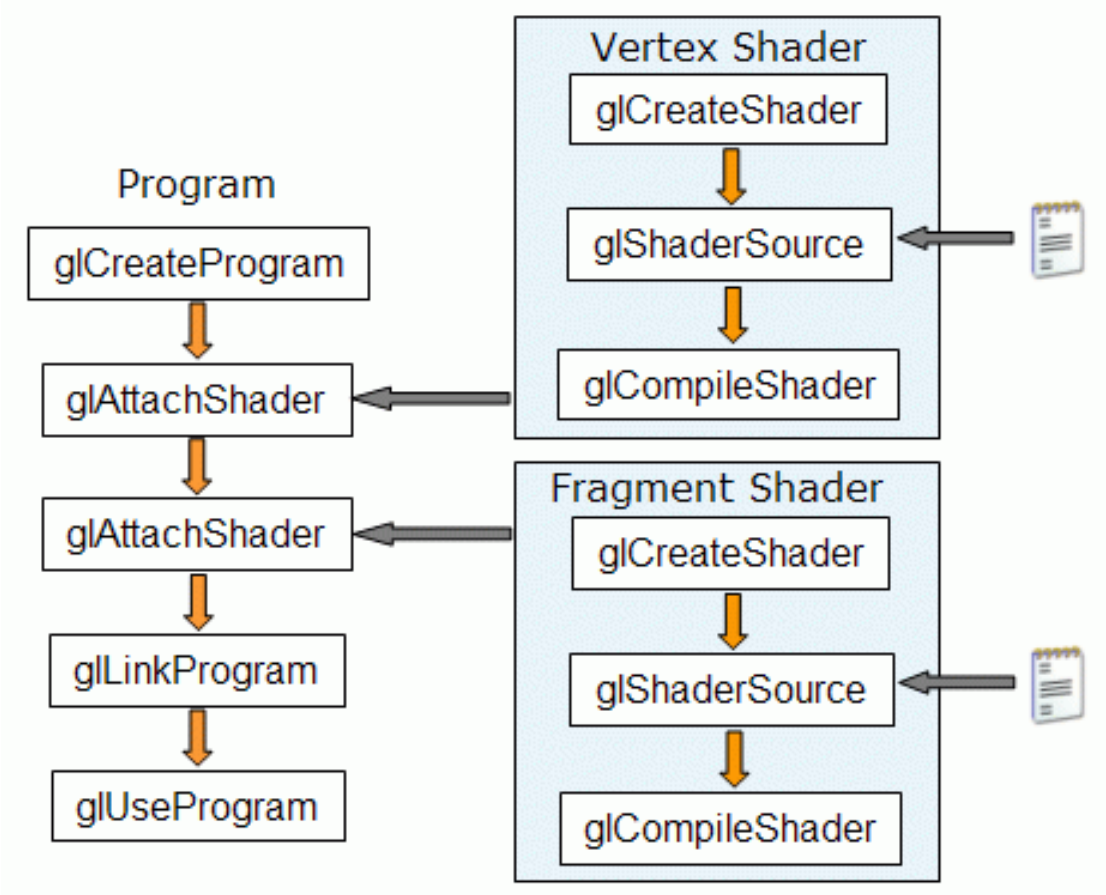
Fragment shaders

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

Fragment shaders

```
uniform vec3 color;           // valor fijo proveniente de nuestro programa
uniform vec3 ambient;         // valor fijo proveniente de nuestro programa
varying vec3 normal;          // valor proveniente del vertex shader
varying vec3 lightVector;     // valor proveniente del vertex shader

void main(){

          // Normalizar los vectores que provienen del vertex shader
          normal = normalize(normal);
          lightVector = normalize(lightVector);

          // Calcular el color difuso
          vec3 diffColor = color * max(0.0, dot(normal, lightVector));

          // Calcular el color de salida (color final del fragmento)
          gl_FragColor = vec4(ambient + diffColor, 1.0);
}
```

Utilizar shaders desde OpenGL



Ejemplo

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

          vsHandle = glCreateShader (GL_VERTEX_SHADER);
          fsHandel = glCreateShader (GL_FRAGMENT_SHADER);
          vsContent = MiFuncionParaLeerFicheros ("toon.vert");
          fsContent = MiFuncionParaLeerFicheros ("toon.frag");

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

          glCompileShader (vsHandle);
          glCompileShader( fsHandle);

          programHandle = glCreateProgram();
          glAttachShader (programHandle , fsHandle);
          glAttachShader (programHandle , vsHandle);
          glLinkProgram (programHandle );
          glUseProgram (programHandle );
}
```

Comunicar OpenGL con GLSL

- Estado de OpenGL
  - Se conoce directamente a través de las variables incorporadas
- Uniform
  - Obtener la localización de la variable en el shader

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

  - Pasar el dato

```
glUniform{2|3|4}{f|i}{v}(GLint loc, TYPE v);
```

- Varying
  - Obtener la localización de la variable en el shader

```
glBindAttributeLocation(GLint program, GLint
index,const GLchar* name);
```

  - Pasar el dato

```
glVertexAttrib{1|2|3|4}{s|f|d}{v}(GLint index,
TYPE v);
```

Ejemplo comunicación

```
// En el shader
uniform float specIntensity;
uniform vec4 specColor;
uniform vec4 colors[3];

// En el programa OpenGL
GLint loc1,loc2,loc3;
float specIntensity = 0.98;
float sc[4] = {0.8,0.8,0.8,1.0};
float colors[12] = {0.4,0.4,0.8,1.0,
                    0.2,0.2,0.4,1.0,
                    0.1,0.1,0.1,1.0};

loc1 = glGetUniformLocation(p,"specIntensity");
glUniform1f(loc1,specIntensity);

loc2 = glGetUniformLocation(p,"specColor");
glUniform4fv(loc2,1,sc);

loc3 = glGetUniformLocation(p,"colors");
glUniform4fv(loc3,3,colors);
```

Ejemplo 1: Shaders mínimos

- *Vertex shader* mínimo:
  - Transformación estándar para los vértices
  - Vértice transformado = projection*modelview*vértice original

```
// Tres versiones equivalentes (o casi)

void main() {
     gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}

void main() {
     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

void main() {
     gl_Position = ftransform();
}
```

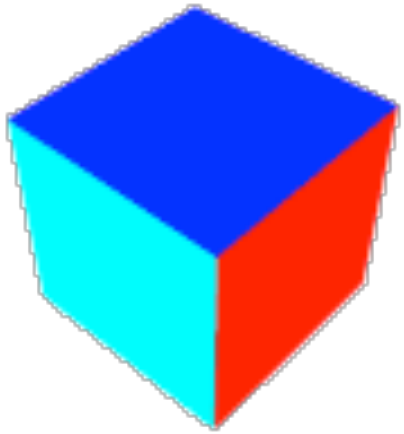
Ejemplo 1: Shaders mínimos

- *Fragment shader* mínimo:
  - El mismo color para todos los píxeles

```
void main()
{
          gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

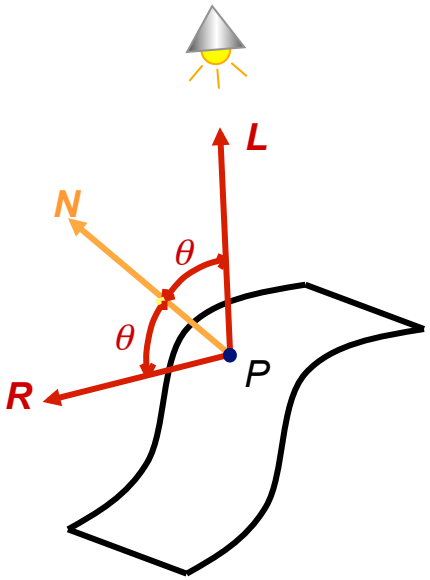
Ejemplo 2: Asignando color

```
//Vertex shader
void main()
{
          gl_FrontColor = gl_Color;
          gl_Position = ftransform();
}

//Fragment shader
void main()
{
          gl_FragColor = gl_Color;
}
```



Ejemplo 3: Iluminación difusa + *shading* Gouraud

- Recordemos el cálculo de la iluminación difusa



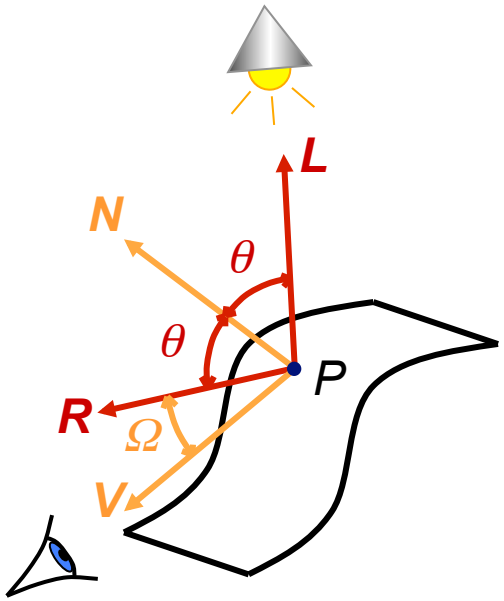
$$I_{difusa} = I_d\, k_d(\lambda)\, \cos\theta = I_d\, k_d\,(\lambda)\, (L \cdot N) \qquad 0 \le \theta \le 2\pi$$

Ejemplo 3: Iluminación difusa + *shading* Gouraud

```
// Vertex shader
uniform vec3 uPosicionLuz;    //posición de la luz
uniform vec4 uDifusaLuz;      //componente difusa de la luz
uniform vec4 uDifusaMaterial; //componente difusa del material
varying vec4 vColorFinal;     //valor de salida para el fragment shader

void main(void) {
          vec4 PosCoordVista = gl_ModelViewMatrix * gl_Vertex;
          vec3 N = normalize(gl_NormalMatrix * gl_Normal);
          vec3 L = normalize(PosCoordVista.xyz - uPosicionLuz);
          float CosDifusa= dot(N,L);
          vColorFinal = uDifusaMaterial * uDifusaLuz * CosDifusa;
          vColorFinal.a = 1.0;
          gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

// Fragment shader
varying vec4 vColorFinal;

void main(void) {
          gl_FragColor = vColorFinal;  //en realidad, el color se interpola
}
```

Ejemplo 4: Reflexión de Phong + *shading* Gouraud

- Recordemos el modelo de reflexión de Phong



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

Ejemplo 4: Reflexión de Phong + *shading* Gouraud

```
// Vertex shader
uniform vec3 uPosicionLuz;          //posición de la luz
uniform vec4 uAmbientalLuz;         //componente ambiental de la luz
uniform vec4 uDifusaLuz;            //componente difusa de la luz
uniform vec4 uEspecularLuz;         //componente especular de la luz
uniform vec4 uAmbientalMaterial;    //componente ambiental del material
uniform vec4 uDifusaMaterial;       //componente difusa del material
uniform vec4 uEspecularMaterial;    //componente especular del material
uniform float uBrillo               //brillo de la componente especular (n)
varying vec4 vColorFinal;           //valor de salida para el fragment shader

void main(void) {
          vec4 PosCoordVista = gl_ModelViewMatrix * gl_Vertex;
          vec3 N = normalize(gl_NormalMatrix * gl_Normal);
          vec3 L = normalize(PosCoordVista.xyz - uPosicionLuz);
          vec3 V = normalize(-vec3(PosCoordVista.xyz));
          vec3 R = reflect(L,N);
          float CosDifusa = max(dot(N,L),0.0);
          float CosEspec = max(dot(R,V),0.0);
          vColorFinal = uAmbientalLuz * uAmbientalMaterial +
                        uDifusaMaterial * uDifusaLuz * CosDifusa;
                        uEspecularMaterial * uEspecularMaterial * pow(CosEspec,uBrillo);
          vColorFinal.a = 1.0;
          gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```