Programando la GPU (I)

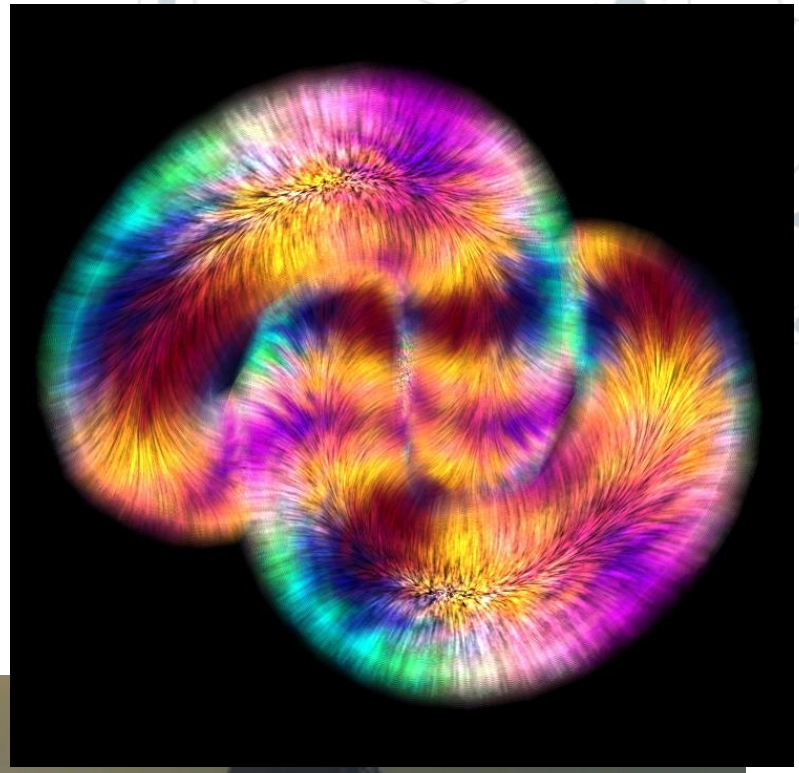
OpenGL Shading Language

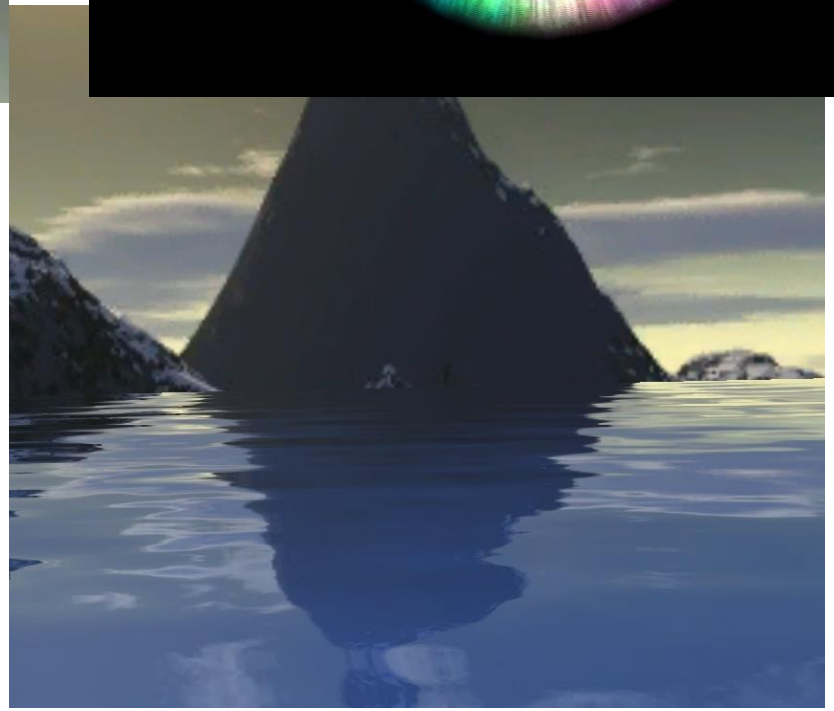
flickr.com/photos/guypaterson

Bibliografía:

- Superbiblia 7ª ed, pp. 205-225, 156-164
- Especificación GLSL 4.6

Índice





Introducción

- Un *shader* es un programa escrito para ejecutarse directamente en la GPU
- Lenguajes de programación de shaders para gráficos
  - OpenGL Shading Language (GLSL)
  - Microsoft High-Level Shader Language (HLSL)
  - Nvidia CG
  - Metal Shading Language
  - Pixar RenderMan, Sony Pictures Imageworks Open Shading Language (OSL)
  - …
- Lenguajes de propósito general para GPU
  - Nvidia CUDA
  - OpenCL
  - DirectCompute
  - Compute Shaders en OpenGL
  - SYCL
  - …

https://en.wikipedia.org/wiki/Shading_language

Introducción

- Historia de OpenGL/GLSL

OpenGL	GLSL	Fecha
1.0	-	1992
1.1	-	1993
1.2	-	1998
1.3	-	2001
1.4	-	2002
1.5	-	2003
2.0	1.10	Sept. 2004
2.1	1.20	Agosto 2006
3.0	1.30	Agosto 2008
3.1	1.40	Marzo 2009
3.2	1.50	Agosto 2009

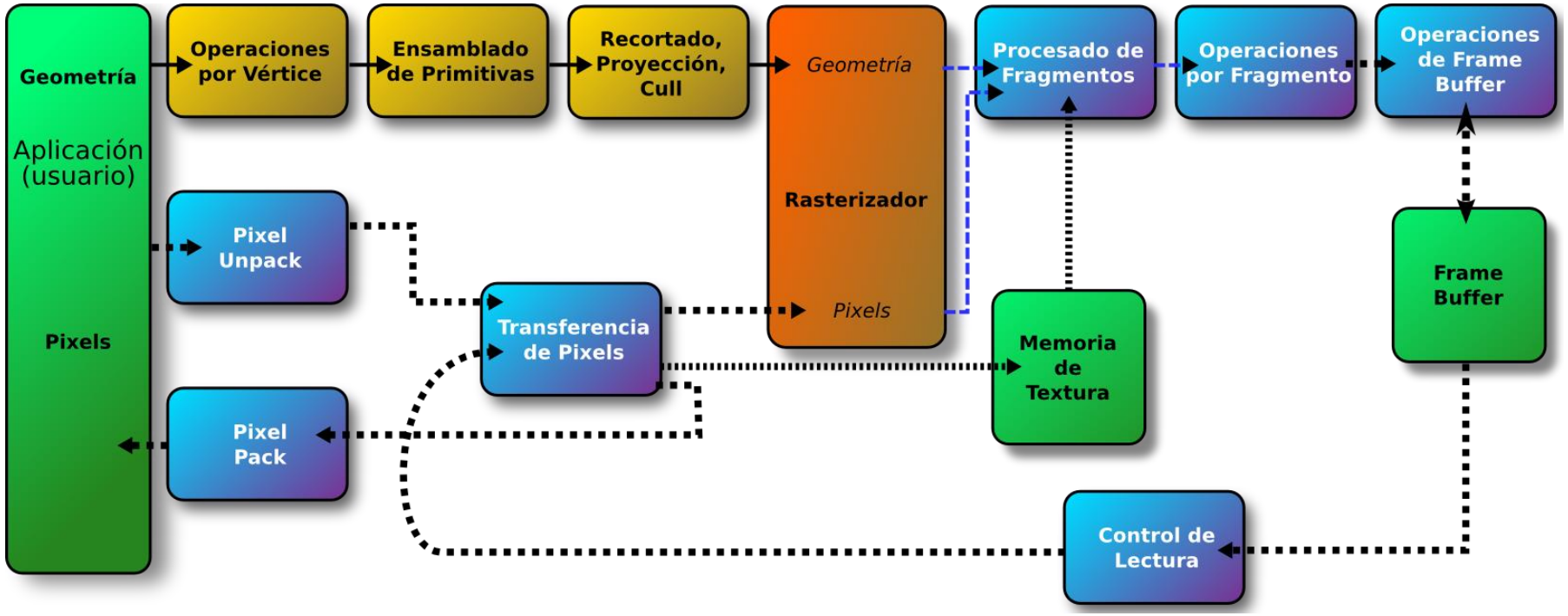
OpenGL	GLSL	Fecha
3.3	3.30	Marzo 2010
4.0	4.00	Marzo 2010
4.1	4.10	Julio 2010
4.2	4.20	Agosto 2011
4.3	4.30	Agosto 2012
4.4	4.40	Julio 2013
4.5	4.50	Agosto 2014
4.6	4.60	Julio 2017

OpenGL
Pre 2.0: Tubería de función fija

- Antes de la versión 2.0, OpenGL no daba mucha flexibilidad para definir las operaciones que se ejecutan dentro de la tarjeta
  - Como mucho, se podía cambiar ciertos parámetros de la máquina de estados, como el color actual, matrices de transformación, forma de aplicar las texturas, etc.
- La tubería estaba (y sigue estando) dividida en dos grandes etapas:
  - Procesado de vértices
    - Transformar, calcular iluminación…
  - Procesado de píxeles
    - Aplicar texturas, niebla, blending, color final del fragmento…

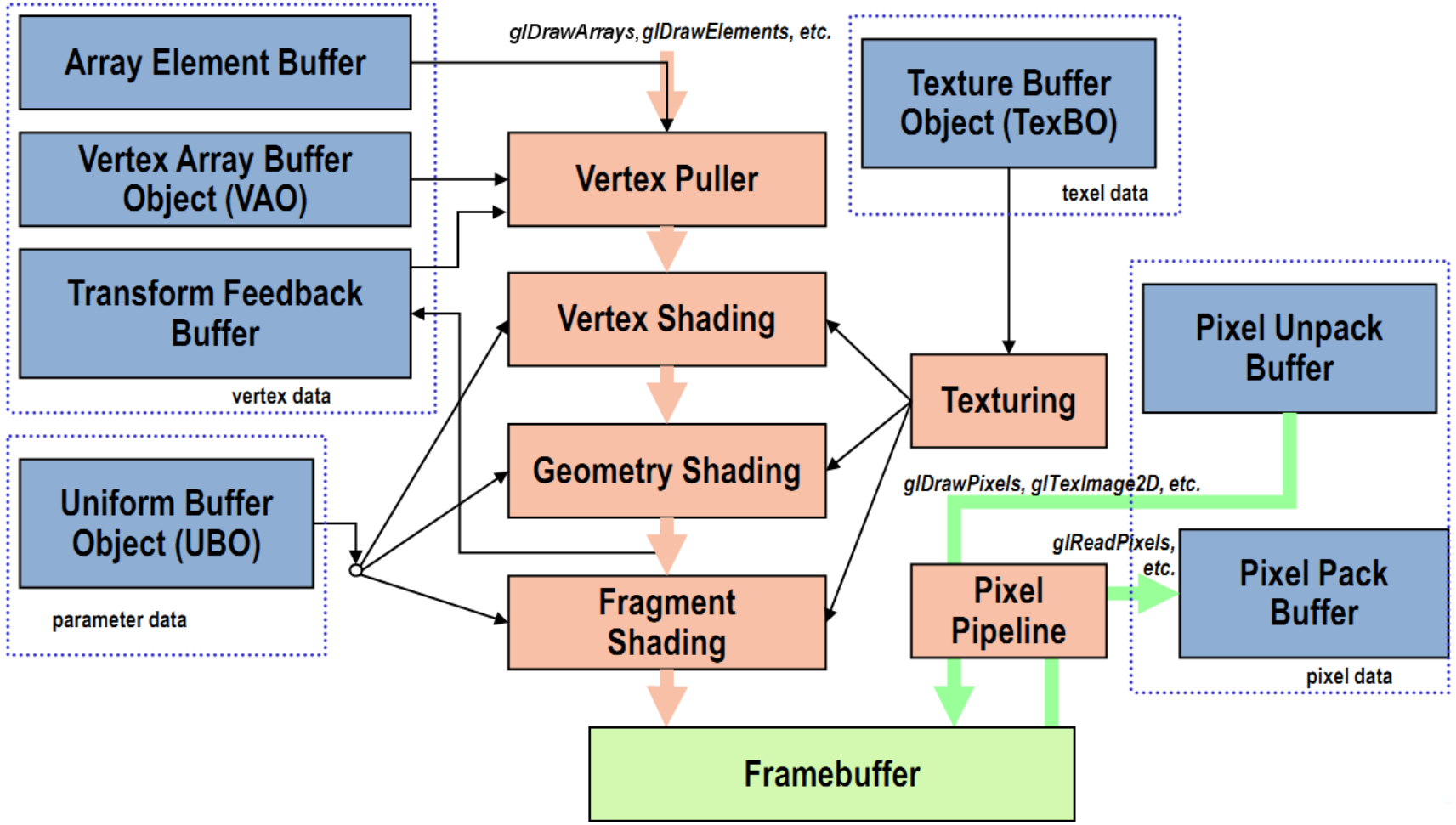
OpenGL

Pre 2.0: Tubería de función fija



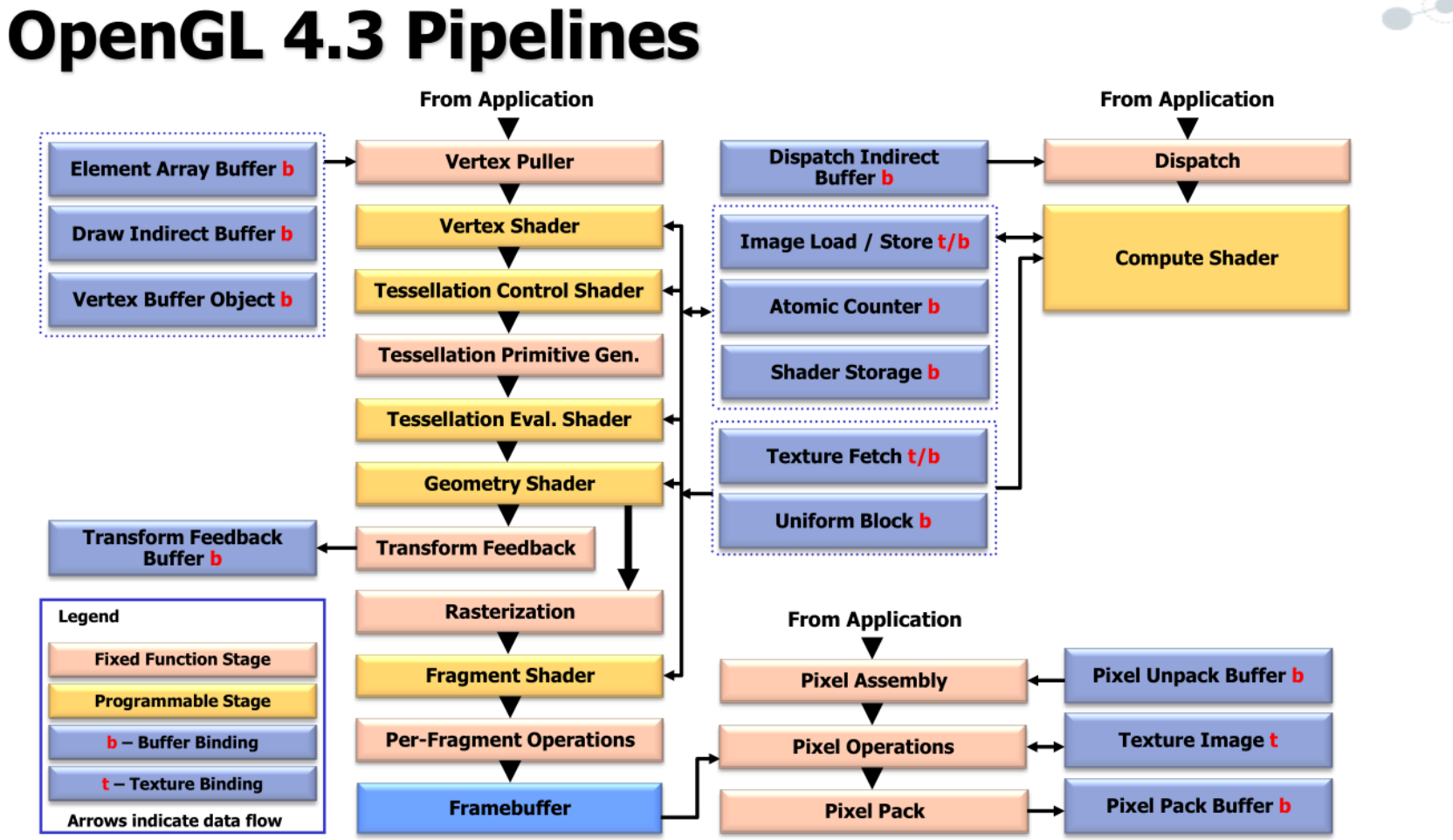
Jose Luis Hidalgo

OpenGL
Tubería de OpenGL 3.3



OpenGL3_Introduction_GDC_2009.pdf

OpenGL
Tubería de OpenGL 3.3



http://www.khronos.org

Shaders en OpenGL 4.3

- Seis tipos de *shaders*:
  - De vértice (desde GL 2.0)
  - De fragmento (desde GL 2.0)
  - De geometría (desde GL 3.2)
  - De control y de evaluación de teselación (desde GL 4.0)
  - De computación (desde GL 4.3)
  - El único obligatorio es el shader de vértice (aunque sin uno de fragmento no veremos nada)
- El *driver* de OpenGL se encarga de compilar los *shaders* para generar un programa
- Un programa está compuesto por uno o varios shaders

Shaders en OpenGL 4.3

Shaders de vértice

- Procesan los datos de entrada asociados a cada vértice para producir unos datos de salida
- Procesan cada vértice de forma independiente
- Operaciones típicas:
  - Transformación de vértices (obligatoriamente, hay que escribir en la variable predefinida **`gl_Position`**)
  - Transformación y normalización de normales
  - Generación de coordenadas de textura
  - Transformación de coordenadas de textura
  - Iluminación
  - Aplicación del color del material

Shaders en OpenGL 4.3

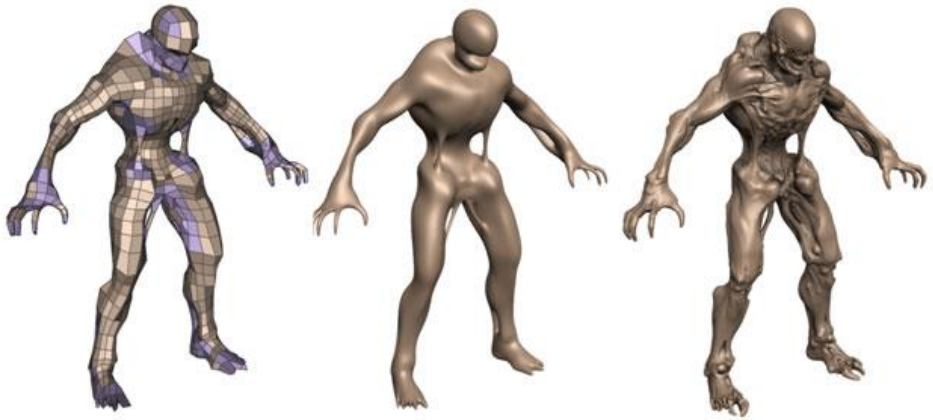
Shaders de fragmento

- Trabajan sobre cada fragmento generado por el rasterizador
- Operaciones:
  - Operaciones sobre valores interpolados
  - Aplicación de la textura
  - Efectos por fragmento (Phong, *bump mapping*, niebla…)
- Un *fragment shader* no puede cambiar la posición del fragmento que está procesando, ni acceder a los vecinos

Shaders en OpenGL 4.3

Shaders de teselación

- Reciben la salida del shader de vértice
- Trabajan a nivel de vértice, pero tienen acceso a todos los vértices del parche
- Pueden generar cualquier tipo de primitiva a partir del parche de entrada
- Generan geometría al vuelo, dentro de la GPU



http://www.nvidia.com/object/tessellation.html

Teselado + *displacement mapping*

Shaders en OpenGL 4.3

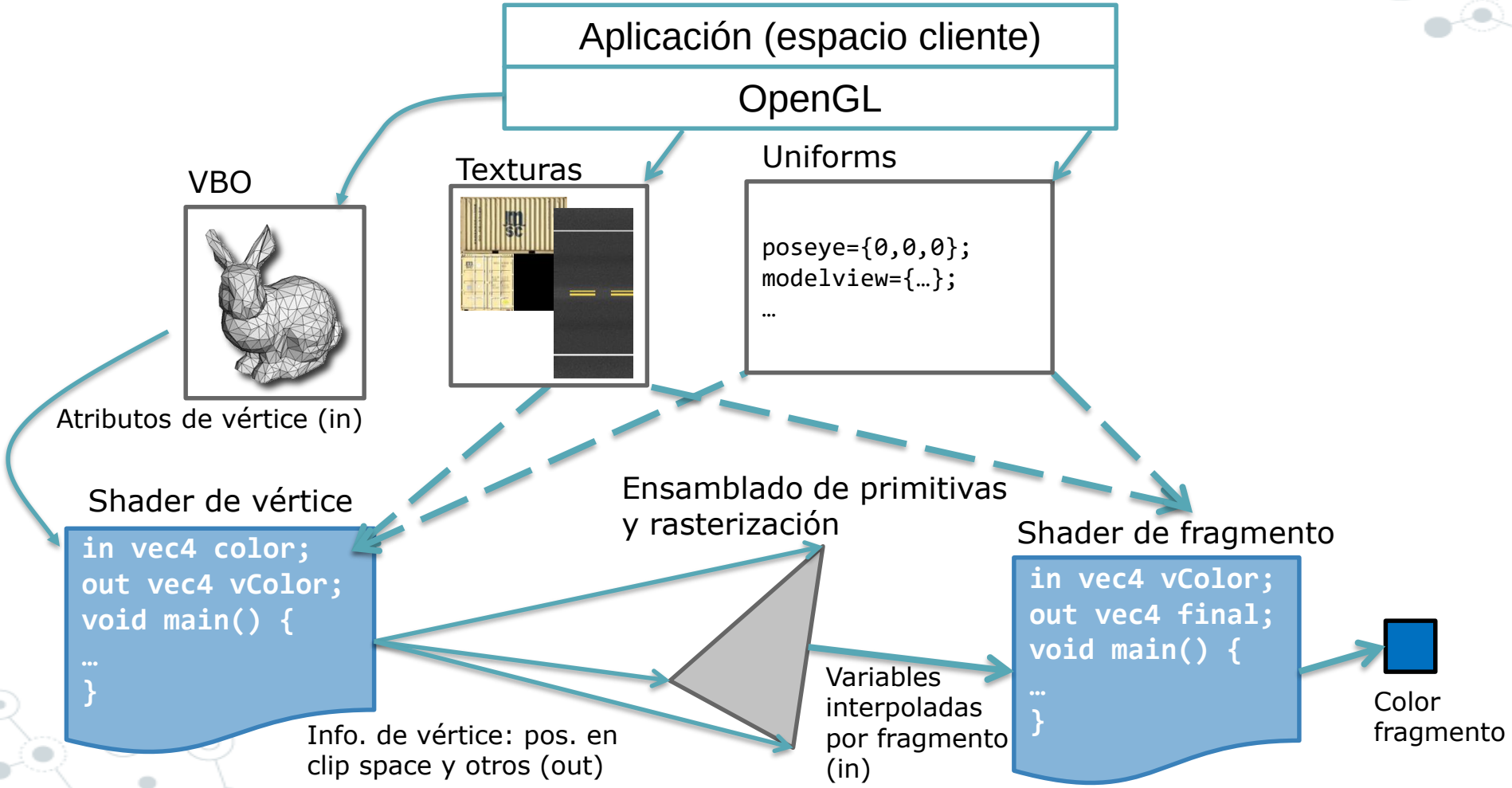
Shaders de geometría

- Se ejecutan después del de vértice (o teselación, si existe)
- Trabajan a nivel de primitiva (recibe todos los vértices de la primitiva y, quizá, vértices adyacentes)
- Descartan la primitiva de entrada, y pueden generar cero o más primitivas (todas del mismo tipo, aunque puede ser distinto del original)
- Aplicaciones:
  - extrudir un polígono para convertirlo en un volumen (p.e., un volumen de sombra), generar las 6 caras de un cubemap en una pasada, *transform feedback*, calcular dos vistas de la escena en una pasada, etc.

Shaders en OpenGL 4.X

- La aplicación envía información a los shaders mediante:
  - atributos: visibles únicamente en los shader de vértice. Un atributo siempre va asociado a un vértice
  - texturas: accesibles desde cualquier shader
  - *uniforms*: son variables globales a las que escribe la aplicación, accesibles desde cualquier shader
- El shader de vértice puede mandar información al de fragmento:
  - constante (mismo valor para todos los fragmentos de una primitiva)
  - interpolada

Flujo de información en OpenGL 3



GLSL
El lenguaje

- Para escribir *shaders* en las primeras GPU programables se usaba ensamblador
  - Problemas: difícil de depurar, específico de una arquitectura
- GLSL es un lenguaje compilado, basado en C/C++
- Comentarios: **`/* */`** y **`//`**
- Cada tipo de shader de un programa empieza a ejecutarse en la función **`main`**:

```
#version 330 core
void main() {

}
```

GLSL
El lenguaje

- Características
  - Usa el mismo lenguaje para procesar vértices, teselación, geometría y fragmentos
  - Soporte nativo de operaciones vectoriales y matriciales
  - Tipado más estricto que C/C++
  - Calificadores de tipo en los parámetros de las funciones para especificar entrada/salida (no hay punteros)

GLSL
Preprocesador

- **`#if, #ifdef #define, #undef, #endif…`**
- **`#version <número> [<perfil>]`**
  - Debe ser la primera instrucción del shader
  - Define la versión de GLSL que necesita el shader (p.e., 430 para GLSL 4.30, ver transparencia 5)
  - Los shaders que no declaren la versión, se asume que están diseñados para GLSL 1.10
  - **`<perfil>: core, compatibility`**. Por defecto, **`core`**
- **`#error <msg>`**

GLSL
Tipos de datos

- Tipos de datos en GLSL 4.6:
  - **`float, double, int, uint`** (32 bits), **`bool`**
  - Constantes: `123, 12U, 2.3, 2.3f, 2.30000001LF, true, false`
  - No hay: punteros, caracteres, cadenas
  - Vectores 2, 3 y 4D de los tipos básicos:
    - **`vec2, dvec2, ivec2, uvec2, bvec2, vec3, dvec3, ivec3, uvec3, bvec3, vec4, dvec4, ivec4, uvec4, bvec4`**
  - Matrices rectangulares de floats y doubles, de 2, 3 y 4 componentes:
    - **`mat2`** (o **`mat2x2`**), **`mat3`**, **`mat4`**, **`mat2x3`**, **`mat4x3`**, en general, **`mat<cols>x<filas>`**, o **`dmat`** para matrices de **`double`**
  - Otros tipos opacos (*handlers*) para referenciar texturas y otros objetos

GLSL
Tipos de datos

- Inicialización de variables:
  - `float a, b=3.0, c;`
  - `const int Size=4;`
  - Mediante constructores:
    - `vec4 v = vec4(1.0, 2.0, 3.0, 4.0);`
    - `v = vec4(0.0, 0.0, 0.0, 1.0);`
    - `mat2 m = mat2(1.0, 2.0, 0.0, 4.0);`
    - `mat2 m = mat2(vec2(1.0, 2.0), vec2(0.0, 4.0));`

$$m = \begin{bmatrix} 1.0 & 0.0 \\ 2.0 & 4.0 \end{bmatrix}$$

    - `vec3 v=vec3(0.5); // v=(0.5, 0.5, 0.5)`
    - `vec4 w = vec4(v, 1.0);`
  - No se pueden inicializar variables *`in`*, *`out`* o de tipos opacos

GLSL
Tipos de datos

- Inicialización de variables:
  - Mediante constructores:
    - `mat2 m=mat2(1.0);` $m = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$
    - `vec4 v = vec4(1.0, 2.0, 3.0, 4.0);` $m = \begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$
    - `vec2 u = vec2(v);  // u = (1.0, 2.0)`
    - `mat2 m = mat2(v);`
    - En general, la parte derecha debe tener el mismo número o más componentes que la izquierda

GLSL
Tipos de datos

- Conversión de tipos:
  - Implícitas: (**int**) -> **uint**, (**int,uint**)->**float**, (**int**, **uint**, **float**)->**double** (y sus vectores, matrices)
  - Explícitas (mediante constructores)
    - `float f = 2.3;`
    - `bool b = bool(f);  // b=true`
    - `int i = int(3.1);`
    - `vec4 v = vec4(2);`
- Operadores:
  - Los mismos que C, excepto operadores sobre punteros (**&, ***) y casting. Añade el operador lógico ^^ (or exclusivo)

GLSL
Tipos de datos

- Operaciones con vectores:
  - Acceso a componentes:
    - **`v[0]`** ó **`v.x`** ó **`v.r`** ó **`v.s`**
    - (**`x`**, **`y`**, **`z`**, **`w`**), (**`r`**, **`g`**, **`b`**, **`a`**), (**`s`**, **`t`**, **`p`**, **`q`**)
  - Acceso a varios componentes:
    - **`vec3 v3; vec2 v2;`**
    - **`v2=v3.xz;`**
  - Acceso a elementos de una matriz:
    - **`float f; vec4 v; mat4 m;`**
    - **`v=m[0];  // Primera columna`**
    - **`f=m[3][1]; // Segundo elem. de la última columna`**
    - **`m[3]=vec4(1.0); // Pone a 1.0 la última columna`**

GLSL
Tipos de datos

- *Swizzling* (mezclar un cocktail)
  - `vec4 v4;`
  - `vec3 v3=v4.rgb;`
  - `vec3 v3=v4.rrb;`
  - `vec3 v4=v4.wzyx;`
  - `vec4 pos;`
  - `pos.xw=vec2(4.0, 0.0);`
  - `pos.yx=vec2(0.0, 1.0);`
  - `color.bgra=color.rgba;`

GLSL
Tipos de datos

- Operaciones con vectores:
  - `vec2 v, u; float f;`
  - `v=v+u+f; // v.x=v.x+u.x+f; v.y=v.y+u.y+f;`
  - `v++;  // v.x++; v.y++;`
  - `vec4 v, u; mat4 m;`
  - **`v*u`** // producto componente a componente
  - **`v*m`** // vector (fila) * matriz
  - **`m*v`** // matriz * vector (columna)
  - **`m*m`** // matriz * matriz (producto algebraico)

GLSL
Tipos de datos

- Arrays

```
vec4 puntos[10];  // 0..9
vec3 sinm[]; // Se puede declarar sin tamaño…
vec3 sinm[10]; //…y dárselo después
```

  - ó

```
sinm[7]=vec3(1.0); // …o el compilador lo calculará
```

  - Inicialización con constructores
    - `float diff[5] = float[5](1.0, 2.0, 3.0, 4.0, 5.0);`
  - Se puede consultar su tamaño:
    - `diff.length() -> 5` (también para vec y mat)
  - A partir de OpenGL 4.3, arrays de arrays
    - `float diff2[4][10];`

GLSL
Tipos de datos

- Estructuras

```
struct light {          // Definición
   vec3 position;
   vec3 color;
};
light mi_luz; // Declaración. No hace falta struct
light otra(vec3(0, 1, 0), vec3(1,1,1)); // Constructor por defecto
```

GLSL

Calificadores de almacenamiento

- Una variable puede calificarse como:
  - **`const`**: variable de sólo lectura
  - **`in`**: variable recibida desde una etapa anterior (sólo lectura)
  - **`out`**: pasada a la siguiente etapa
  - **`uniform`**: variable de entrada con datos pasados desde la aplicación
- Ejemplos

```
const float PI = 3.141592;
uniform vec4 color;
in vec4 vertice;
out vec4 verticeTransf;
```

GLSL

Instrucciones de control de flujo

- Debe existir una única función **`main`** por cada shader de vértice, geometría, fragmento…
- Antes de entrar a **`main`**, se inicializan las variables globales
- Bucles:
  - **`for, while, do-while, break, continue, return`**
- Selección
  - **`if, if-else, ?:, switch`**
- Las condiciones deben ser booleanas, y se usa evaluación en cortocircuito
- **`discard`**: descartar el fragmento
- No hay **`goto`**

GLSL
Funciones

- Se permite sobrecarga dependiendo de los parámetros
- Se deben respetar los tipos de los parámetros
- Una función acaba con **`return`**
- No se permite recursión
- Se pueden devolver **`arrays`**
- Los parámetros se pasan por valor
- Existen parámetros de entrada y de salida
  - Calificados como **`in`**, **`const in`**, **`out`** o **`inout`**

```
void ComputeCoord(in vec3 normal, vec3 tangent, inout vec3 coord)
 bool [4] foo(float a, int b, out float c) {…}
```

- Se pueden usar funciones definidas en otros shaders del programa, incluyendo su cabecera

GLSL
Funciones predefinidas

Angle & Trigonometry Functions [8.1]
Component-wise operation. Parameters specified as *angle* are assumed to be in units of radians. T is float, vec2, vec3, vec4.

T **radians**(T *degrees*)	degrees to radians
T **degrees**(T *radians*)	radians to degrees
T **sin**(T *angle*)	sine
T **cos**(T *angle*)	cosine
T **tan**(T *angle*)	tangent
T **asin**(T *x*)	arc sine
T **acos**(T *x*)	arc cosine
T **atan**(T *y*, T *x*) T **atan**(T *y_over_x*)	arc tangent
T **sinh**(T *x*)	hyperbolic sine
T **cosh**(T *x*)	hyperbolic cosine
T **tanh**(T *x*)	hyperbolic tangent
T **asinh**(T *x*)	hyperbolic sine
T **acosh**(T *x*)	hyperbolic cosine
T **atanh**(T *x*)	hyperbolic tangent

Sección de la especificación

Exponential Functions [8.2]
Component-wise operation. T is float, vec2, vec3, vec4.

T **pow**(T *x*, T *y*)	x^y
T **exp**(T *x*)	e^x
T **log**(T *x*)	ln
T **exp2**(T *x*)	2^x
T **log2**(T *x*)	$\log_2$
T **sqrt**(T *x*)	square root
T **inversesqrt**(T *x*)	inverse square root

Tablas extraídas de la tarjeta de referencia rápida: https://www.khronos.org/developers/reference-cards

GLSL
Funciones predefinidas

Common Functions [8.3]
Component-wise operation. T is float, vec2, vec3, vec4. Ti is int, ivec2, ivec3, ivec4. Tu is uint, uvec2, uvec3, uvec4. bvec is bvec2, bvec3, bvec4, bool.

T **abs**(T *x*) Ti **abs**(Ti *x*)	absolute value
T **sign**(T *x*) Ti **sign**(Ti *x*)	returns -1.0, 0.0, or 1.0
T **floor**(T *x*)	nearest integer <= *x*
T **trunc**(T *x*)	nearest integer with absolute value <= absolute value of *x*
T **round**(T *x*)	nearest integer, implementation-dependent rounding mode
T **roundEven**(T *x*)	nearest integer, 0.5 rounds to nearest even integer
T **ceil**(T *x*)	nearest integer >= *x*
T **fract**(T *x*)	*x* - floor(*x*)
T **mod**(T *x*, float *y*) T **mod**(T *x*, T *y*)	modulus
T **modf**(T *x*, out T *i*)	separate integer and fractional parts
T **min**(T *x*, T *y*) T **min**(T *x*, float *y*) Ti **min**(Ti *x*, Ti *y*) Ti **min**(Ti *x*, int *y*) Tu **min**(Tu *x*, Tu *y*) Tu **min**(Tu *x*, uint *y*)	minimum value
T **max**(T *x*, T *y*) T **max**(T *x*, float *y*) Ti **max**(Ti *x*, Ti *y*) Ti **max**(Ti *x*, int *y*) Tu **max**(Tu *x*, Tu *y*) Tu **max**(Tu *x*, uint *y*)	maximum value

T **clamp**(T *x*, T *minVal*, T *maxVal*) T **clamp**(T *x*, float *minVal*, float *maxVal*) Ti **clamp**(Ti *x*, Ti *minVal*, Ti *maxVal*) Ti **clamp**(Ti *x*, int *minVal*, int *maxVal*) Tu **clamp**(Tu *x*, Tu *minVal*, Tu *maxVal*) Tu **clamp**(Tu *x*, uint *minVal*, uint *maxVal*)	min(max(*x*, *minVal*), *maxVal*)
T **mix**(T *x*, T *y*, T *a*) T **mix**(T *x*, T *y*, float *a*)	linear blend of *x* and *y*
T **mix**(T *x*, T *y*, bvec *a*)	true components in *a* select components from *y*, else from x
T **step**(T *edge*, T *x*) T **step**(float *edge*, T *x*)	0.0 if *x* < *edge*, else 1.0
T **smoothstep**(T *edge0*, T *edge1*, T *x*) T **smoothstep**(float *edge0*, float *edge1*, T *x*)	clip and smooth
bvec **isnan**(T *x*)	true if *x* is NaN
bvec **isinf**(T *x*)	true if *x* is positive or negative infinity

GLSL
Funciones predefinidas

Geometric Functions [8.5]
These functions operate on vectors as vectors, not component-wise. Tf=float, vec*n*. Td =double, dvec*n*. Tfd= float, vec*n*, double, dvec*n*.

float **length**(Tf *x*) double **length**(Td *x*)	length of vector
float **distance**(Tf *p0*, Tf *p1*) double **distance**(Td *p0*, Td *p1*)	distance between points
float **dot**(Tf *x*, Tf *y*) double **dot**(Td *x*, Td *y*)	dot product
vec3 **cross**(vec3 *x*, vec3 *y*) dvec3 **cross**(dvec3 *x*, dvec3 *y*)	cross product
Tfd **normalize**(Tfd *x*)	normalize vector to length 1
Tfd **faceforward**(Tfd *N*, Tfd *I*, Tfd *Nref*)	returns *N* if dot(*Nref*, *I*) < 0, else -*N*
Tfd **reflect**(Tfd *I*, Tfd *N*)	reflection direction *I* - 2 * dot(*N*,*I*) * *N*
Tfd **refract**(Tfd *I*, Tfd *N*, float *eta*)	refraction vector

Matrix Functions [8.6]
N and *M* are 1, 2, 3, 4.

mat **matrixCompMult**(mat *x*, mat *y*) dmat **matrixCompMult**(dmat *x*, dmat *y*)	component-wise multiply
mat*N* **outerProduct**(vec*N* *c*, vec*N* *r*) dmat*N* **outerProduct**(dvec*N* *c*, dvec*N* *r*)	outer product (where *N* != *M*)
mat*NxM* **outerProduct**(vec*M* *c*, vec*N* *r*) dmat*NxM* **outerProduct**(dvec*M* *c*, dvec*N* *r*)	outer product
mat*N* **transpose**(mat*N* *m*) dmat*N* **transpose**(dmat*N* *m*)	transpose
mat*NxM* **transpose**(mat*MxN* *m*) dmat*NxM* **transpose**(dmat*MxN* *m*)	transpose (where *N* != *M*)
float **determinant**(mat*N* *m*) double **determinant**(dmat*N* *m*)	determinant
mat*N* **inverse**(mat*N* *m*) dmat*N* **inverse**(dmat*N* *m*)	inverse

Vector Relational Functions [8.7]
Compare *x* and *y* component-wise. Sizes of the input and return vectors for any particular call must match. Tvec=vec*n*, uvec*n*, ivec*n*.

bvec*n* **lessThan**(Tvec *x*, Tvec *y*)	<
bvec*n* **lessThanEqual**(Tvec *x*, Tvec *y*)	<=
bvec*n* **greaterThan**(Tvec *x*, Tvec *y*)	>
bvec*n* **greaterThanEqual**(Tvec *x*, Tvec *y*)	>=
bvec*n* **equal**(Tvec *x*, Tvec *y*) bvec*n* **equal**(bvec*n* *x*, bvec*n* *y*)	==
bvec*n* **notEqual**(Tvec *x*, Tvec *y*) bvec*n* **notEqual**(bvec*n* *x*, bvec*n* *y*)	!=
bool **any**(bvec*n* *x*)	true if any component of *x* is true
bool **all**(bvec*n* *x*)	true if all comps. of *x* are true
bvec*n* **not**(bvec*n* *x*)	logical complement of *x*

Noise Functions [8.14]
Returns noise value. Available to fragment, geometry, and vertex shaders. *n* is 2, 3, or 4:

float **noise1**(Tf *x*) vec*n* **noisen**(Tf *x*)

Ejemplo de programa completo

- Shader de vértice mínimo

```
// The ShadedIdentity Shader
// Vertex Shader
// Richard S. Wright Jr.
// OpenGL SuperBible

#version 330

in vec4 vVertex;  // Vertex position attribute
in vec4 vColor;   // Vertex color attribute

out vec4 vVaryingColor; // Color value passed to fragment shader

void main(void) {
  vVaryingColor = vColor;  // Copy the color value
  gl_Position = vVertex;  // Pass along the vertex position
}
```

Ejemplo de programa completo

- Shader de fragmento mínimo

```
// The ShadedIdentity Shader
// Vertex Shader
// Richard S. Wright Jr.
// OpenGL SuperBible

#version 330

out vec4 vFragColor;  // Fragment color to rasterize
in vec4 vVaryingColor; // Incoming color from vertex stage

void main(void) {
  vFragColor = vVaryingColor;  // Interpolated color to fragment
}
```

Ejemplo de programa completo

Código cliente

Definición de los atributos de vértice

Framebuffer

Shader de vértice

```
in vec4 vVertex;
in vec4 vColor;

out vec4 vVaryingColor;

void main(void) {
  vVaryingColor = vColor;
  gl_Position = vVertex;
}
```

Shader de fragmento

```
out vec4 vFragColor
in vec4 vVaryingColor;

void main(void) {
  vFragColor = vVaryingColor;
}
```

Carga, compilación e instalación de los shaders

- Los pasos a seguir para instalar un shader en la GPU son:
  1. Crear el objeto shader, uno por cada tipo (**`glCreateShader`**)
  2. Proporcionar el código fuente de cada shader (**`glShaderSource`**)
  3. Compilar cada shader (**`glCompileShader`**)
  4. Comprobar errores de compilación (**`glGetShaderiv`**, **`glGetShaderInfoLog`**)
  5. Crear el programa (**`glCreateProgram`**)
  6. Adjuntar los shaders al programa (**`glAttachShader`**)
  7. Establecer los índices de los atributos (**`glBindAttribLocation`**)
  8. Enlazar el programa (**`glLinkProgram`**)
  9. Comprobar resultado del enlazado (**`glGetProgramiv`**)

  (ver con más detalle al final de la presentación)

GLSL

Comunicando la aplicación y el *shader*: *uniforms*

- Los atributos van asociados a cada vértice. Los uniform, por otro lado, son constantes durante una función de dibujo (p.e. `glDrawArrays`)
- En los shaders son de sólo lectura
  - Aunque un shader puede inicializar el uniform, con el valor que se usará si la aplicación no escribe en el mismo
- Se escribe en ellas desde el código cliente
  - Para ello, hay que buscar primero su localización, después de tener el programa enlazado (**`glGetUniformLocation`**)
  - Usar una función adecuada para escribir en la variable, una vez que el shader está en uso (**`glUniform*`**, **`glUniformMatrix*`**)
  - Dentro del shader, se usa como una variable global de sólo lectura

GLSL

Comunicando la aplicación y el *shader*: *uniforms*

- Para obtener la localización de una variable *uniform*, el programa debe estar enlazado:
  - **`GLint glGetUniformLocation(GLuint program, const GLchar *name)`**
  - devuelve -1 si no la encuentra
  - NO se puede pedir la posición de una estructura, pero sí la posición de un campo de la estructura (p.e. `miluz.Pos`), o la posición de un elemento del array (p.e. `colores[0]`, también funciona `colores`)
  - ¡Cuidado! Las variables declaradas que no se usan en el shader desaparecen en la compilación
- A partir de OpenGL 4.3, puedes decidir tú la localización de los uniform:

```
layout (location = 14) uniform vec4 lightPosition;
```

GLSL

Comunicando la aplicación y el *shader*: *uniforms*

- Escribir a una variable *uniform* desde la aplicación (el shader debe estar en uso):
  - **`void glUniform{1|2|3|4}{f|d|i|ui}(GLint location, TYPE v)`**
    - Para escribir en variables de 1, 2, 3 o 4 componentes
    - Para escribir en variables de tipo **`bool`**, se puede usar la función para escribir enteros
  - **`void glUniform{1|2|3|4}{f|d|i|ui}v(GLint location,`**
    `GLuint count, const TYPE *v)`
    - Para escribir en *uniform* de tipo array
    - **`count`** indica cuántos elementos del array se van a escribir

```
float misejes[2][3]={
         {1.0f, 0.0f, 0.0f}, {0.0f, 1.0f, 0.0f}};
[…]
GLint ejesLoc = glGetUniformLocation(program, "ejes");
glUniform3fv(ejesLoc, 2, misejes);
```

```
uniform vec3 ejes[2];
```

GLSL

Comunicando la aplicación y el *shader*: *uniforms*

- Escribir a una variable *uniform* de tipo matriz desde la aplicación:
  - **`void glUniformMatrix{2|3|4|cxr}{f|d}v(GLint location, GLuint count, GLboolean transpose, const GLfloat *v)`**
    - Para cargar matrices en *uniforms*
    - **`count`** es el número de matrices a cargar
    - Donde tanto **`c`** como **`r`** pueden ser 2, 3 o 4
    - **`transpose`** indica si se debe cargar por filas

GLSL

Comunicando la aplicación y el *shader*: *uniforms*

- En la aplicación:

```
…
glLinkProgram(pid);
GLint myUniformId= glGetUniformLocation(pid, "myUniform");
if (myUniformId < 0) exit(-1);
…
glUseProgram(pid);
…
glUniform4f(myUniformId, 0.3f, 0.1f, 0.1f, 1.0f);
```

- En el shader:

```
uniform vec4 myUniform = vec4(0.0);
in vec4 Color;
void main() {
 …
 FinalColor=Color+myUniform;
…
```

GLSL

Comunicando la aplicación y el *shader*: *uniforms*

```
// Flat Shader
// Vertex Shader
// Richard S. Wright Jr.
// OpenGL SuperBible

#version 330

// Transformation Matrix
uniform mat4 mvpMatrix;

// Incoming per vertex
in vec4 vVertex;

void main(void) {
  gl_Position = mvpMatrix  * vVertex;
}
```

```
// Flat Shader
// Fragment Shader
// Richard S. Wright Jr.
// OpenGL SuperBible

#version 330

// Make geometry solid
uniform vec4 vColorValue;

// Output fragment color
out vec4 vFragColor;

void main(void) {
  vFragColor = vColorValue;
}
```

GLSL

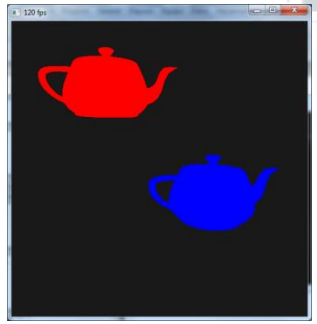
Comunicando la aplicación y el *shader*: *uniforms*

constantshadinguniform.*vert*

```
#version 330

$GLMatrices

in vec3 position;

void main() {
  gl_Position = modelviewprojMatrix * vec4(position,1.0);
}
```

constantshadinguniform.*frag*

```
#version 330

uniform vec4 primitive_color;
out vec4 final_color;

void main() {
  final_color = primitive_color;
}
```



ej4-1

$GLMatrices

- PGUPV sustituye la línea “$GLMatrices” por:

```
layout (std140) uniform GLMatrices {
   mat4 modelMatrix;
   mat4 viewMatrix;
   mat4 projMatrix;
   mat4 modelviewMatrix;
   mat4 modelviewprojMatrix;
   mat3 normalMatrix;
};
```

GLSL
Variables predefinidas

- Shaders de vértice:

```
in int gl_VertexID; // Índice del vértice que se está procesando
out gl_PerVertex {
  vec4 gl_Position; // Escribir aquí la posición en clip space
  float gl_PointSize; // Tamaño en píxels del punto
  float gl_ClipDistance[]; // Distancias a los planos de recorte
  float gl_CullDistance[]; // GL4.5
}
```

- Shaders de fragmento:

```
in vec4  gl_FragCoord; // Coordenadas en el S.C. de la ventana
in bool  gl_FrontFacing; // Cara frontal?
in float gl_ClipDistance[]; // Distancias a los planos de recorte
in float gl_CullDistance[];
…
out float gl_FragDepth; // Profundidad
…
```

GLSL

Texturas y shaders

- Es fácil trabajar con texturas usando *shaders*, ya que se puede leer desde varias texturas y combinar los datos libremente
- Usos alternativos de las texturas:
  - acceder a una textura dependiendo del valor leído en otra textura, almacenar resultados intermedios, tablas de valores precalculados para funciones complejas, almacenar normales, factores de perturbación de normales, valores de brillo…
- Las texturas son accesibles desde cualquier tipo de *shader*
- La aplicación debe establecer la textura y su modo de acceso
- Los shaders pueden modicar las coordenadas de textura definidas por el modelo, o incluso generarlas al vuelo

GLSL
Texturas y shaders

- Para usar una textura en un *shader* la aplicación debe (ver el Tema 3):
  1. Activar una unidad de textura
     - **`glActiveTexture`**
  2. Crear un objeto textura y vincularlo a la unidad
     - **`glGenTextures, glBindTexture, glTexImage…`**
  3. Establecer los parámetros (wrapping, filtrado, etc)
     - **`glTexParameter`**
  4. Comunicar al shader la unidad de textura donde se encuentra la textura

GLSL
Texturas y shaders

- Para acceder a una textura desde un *shader*, se debe definir una variable *uniform* de tipo **`sampler`**.
- La aplicación debe escribir en dicha variable la unidad en la que se encuentra la textura a usar
  - **`GLint texunitloc= glGetUniformLocation(pid,"texUnit");`**
  - **`glUniform1i(texunitloc, 0);`**
- El tipo de sampler usado debe corresponder con el tipo de textura cargada en la unidad
  - **`sampler1D, sampler2D, sampler3D, samplerCube…`**
- Para acceder a los datos de la textura, se usa la función:
  - **`vec4 texture(sampler2D sampler, vec2 tc)`**

GLSL
Texturas y shaders

- Ejemplo (**textureReplace.vert**):

p4

```
#version 420 core

$GLMatrices
in vec4 position;
in vec2 texCoord;
out vec2 texCoordFrag;
void main() {
  texCoordFrag = texCoord;
  gl_Position = modelviewprojMatrix * position;
}
```

GLSL
Texturas y shaders

- Ejemplo (**textureReplace.frag**):

p4

```
#version 420 core

uniform sampler2D texUnit;

in vec2 texCoordFrag;
out vec4 fragColor;

void main() {
  fragColor = texture(texUnit, texCoordFrag);
}
```

GLSL

Texturas y shaders

- Para muestrear texturas de tipo entero y entero sin signo:
  - **`isampler*`**, **`usampler*`**
- Para obtener el valor de un texel específico:
  - **`vec4 texelFetch(sampler1D s, int P, int lod)`**
  - **`vec4 texelFetch(sampler2D s, ivec2 P, int lod)`**
  - **`uvec4 texelFetch(usampler3D s, ivec3 P, int lod)`**
- Para obtener el tamaño de la textura muestreada:
  - **`int textureSize(sampler1D s, int lod)`**
  - **`ivec2 textureSize(sampler2D s, int lod)`**
  - **`ivec3 textureSize(sampler3D s, int lod)`**

Shaders en PGUPV

- Estudia las clases `PGUPV::Program` y `PGUPV::Shader` y los ejemplos suministrados

GLSL
Instalando shaders

- Los pasos para instalar un programa en la GPU son:
  1. Cargar y compilar los *shaders*
  2. Enlazar los *shaders* en un programa
  3. Instalar el programa
  4. Si existen *uniform* o atributos definidos por el usuario, vincularlos con variables de la aplicación

GLSL
Instalando shaders

1. Cargar y compilar los shaders
    1. **`GLuint glCreateShader(GLenum sType)`**
        - devuelve un identificador distinto de cero
        - **`sType={GL_VERTEX_SHADER | GL_FRAGMENT_SHADER | GL_GEOMETRY_SHADER}`**
    2. **`void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)`**
        - **`shader:`** identificador
        - **`count, string:`** cantidad y líneas de código del shader
        - **`length:`** vector de longitudes de cadena, o NULL si ASCIIZ

GLSL

Instalando shaders

1. Cargar y compilar los shaders
   3. **void glCompileShader(GLuint shader)**
      - **shader: identificador**
   4. **glGetShaderiv(id, GL_COMPILE_STATUS, &compiled);**

```
if(compiled == GL_FALSE) {
  GLint length;
  glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
  GLchar *log=new GLchar [length];
  glGetShaderInfoLog(id, length, &length, log);
  cerr << “Comp. error: " << log << endl;
  delete [] log;
  glDeleteShader(id);
  return 0;
}
```

GLSL
Instalando shaders

2. Enlazar los shaders en un programa
    1. **`GLuint glCreateProgram(void)`**
        - devuelve un identificador de programa distinto de cero
    2. **`void glAttachShader(GLuint program, GLuint shader)`**
        - **`program:`** identificador de programa
        - **`shader:`** identificador de shader
    3. **`void glLinkProgram(GLuint program)`**
        1. **`program:`** identificador de programa

GLSL

Instalando shaders

2. Enlazar los shaders en un programa

4.
```
glGetProgramiv(pid, GL_LINK_STATUS, &linked);
if (linked == GL_FALSE) {
      GLint length;
      glGetShaderiv(pid, GL_INFO_LOG_LENGTH, &length);
      GLchar *log=new GLchar [length];
      glGetProgramInfoLog(pid, length, &length, log);
      cerr << “Link error: " << log << endl;
      delete [] log;
      glDeleteProgram(pid);
      return 0;
   }
```

GLSL
Instalando shaders

3. Instalar el programa
   1. **`void glUseProgram(GLuint program)`**
      - **`program`**: identificador de programa. Si se pasa 0, entonces se desconectan los procesadores de vértice y fragmento

- Haciendo limpieza…
  - **`void glDeleteShader(GLuint shader)`**
  - **`void glDeleteProgram(GLuint program)`**

GLSL
Instalando shaders