

# Programando la GPU (II)

---

*Shaders de vértice e  
iluminación*



[flickr.com/photos/guypaterson](https://www.flickr.com/photos/guypaterson)

Bibliografía:

- Superbiblia 7ªed, 121-135, 567-577



# Índice

---

- Recordatorio
- Atributos
- *Uniform Buffer Objects*
- Responsabilidades del shader de vértice
- Iluminación en OpenGL
  - Implementando la tubería fija con *shaders*
- Sombreado en OpenGL
  - Sombreado por vértice
  - Sombreado por píxel
- Apéndice: detalles de implementación de UBO



# Recuerda

---

- Para enviar información desde la aplicación a los *shaders*, hay tres opciones:
  - **attribute**: variable que cambia a menudo (cada vértice), de la aplicación al shader de vértice
  - **uniform**: variable que cambia poco (cada lote de primitivas), de la aplicación a todos los shaders
  - **texturas**: las establece la aplicación y son accesibles desde cualquier shader
- Para comunicar un shader y el siguiente:
  - **out/in**: para valores interpolados para cada píxel, de *shader* de vértice a *shader* de fragmento



# Recuerda

## Atributos

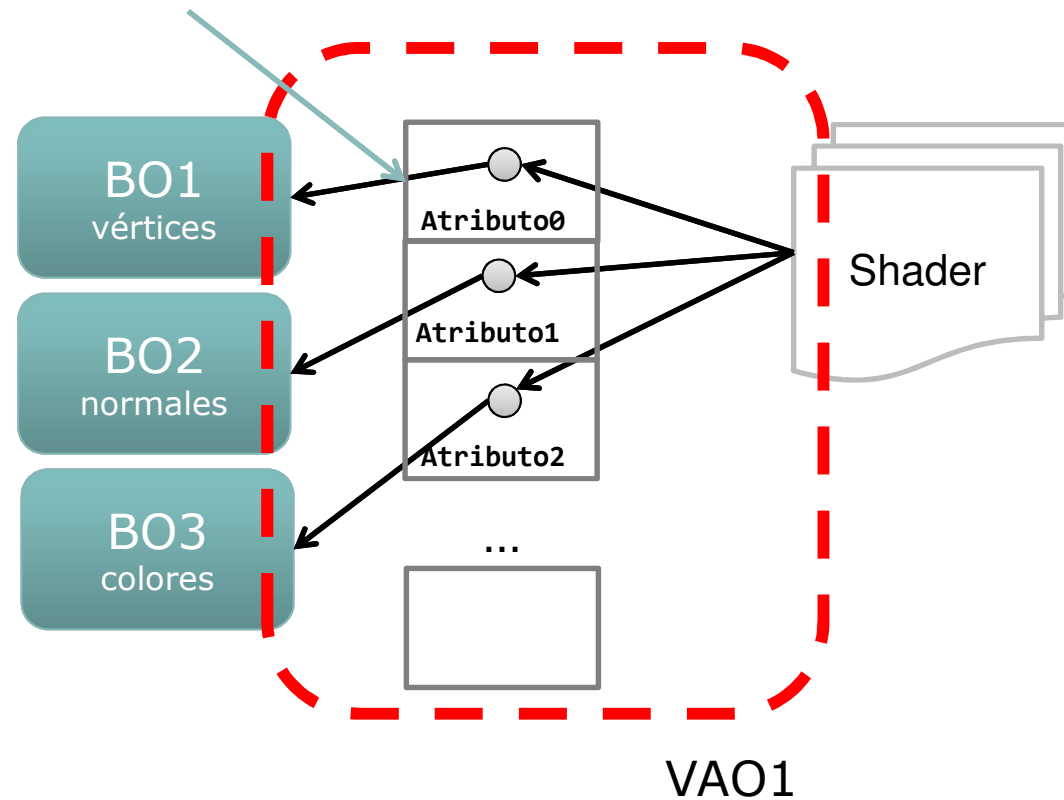
---

- OpenGL permite definir, como mínimo, 16 atributos por vértice (de 0 a 15). Máximo efectivo:  
`glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &m)`
- Cada atributo se almacena en un `vec4`. Si falta algún componente, se aumenta a (X, 0, 0, 1)
- Cada atributo está conectado con una variable `in` del shader de vértice, y sólo es visible en dicho shader
- Es útil almacenar siempre el mismo tipo de atributo en la misma posición (p.e., posición en 0, normal en 1, colores en 2...)
- En el tema 1 se estudió cómo almacenar los datos de un atributo en VBO y cómo usar los puntos de vinculación:

# Recuerda

## Atributos, BO y VAO

```
glVertexAttribPointer(0, ...);  
glEnableVertexAttribArray(0);
```





# Conectando atributos con variables en el shader

---

- El siguiente paso es conectar las variables del shader con un índice de atributo:

```
#version 330 core
$GLMatrices
```

textureReplace.vert

```
in vec4 position;
in vec2 texCoord; } atributos
```

```
out vec2 texCoordFrag;
```

```
void main() {
    texCoordFrag = texCoord;
    gl_Position = modelviewprojMatrix * position;
}
```



# Atributos

## Conexión antes de enlazar el shader

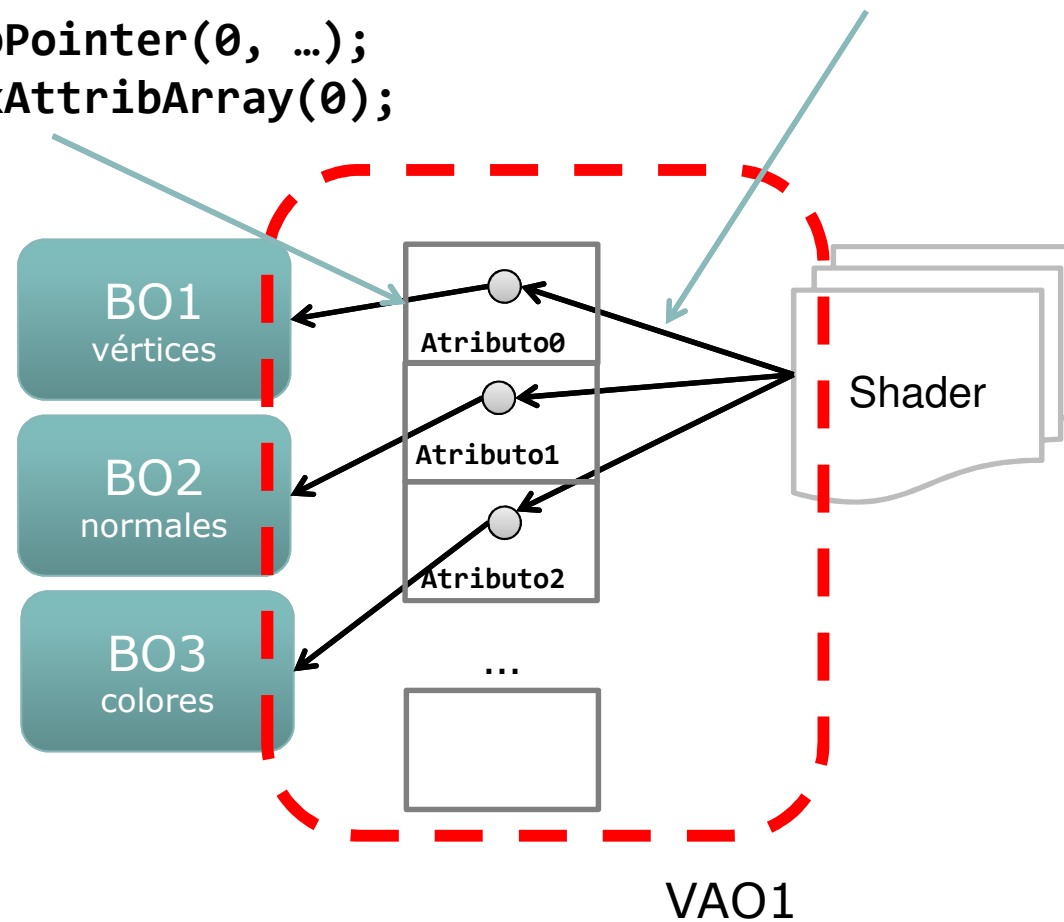
---

- El siguiente paso es conectar las variables del shader con un índice de atributo
- Antes de enlazar el programa (`glLinkProgram`):
- `void glBindAttribLocation(GLuint program, GLuint attribLocation, const GLchar *name)`
  - `program`: identificador del programa
  - `attribLocation`: índice del atributo
  - `name`: nombre de la variable del shader que recibirá el atributo
- Ejemplo:  
`glBindAttribLocation(pid, 0, "position");`

# Atributos

## Conexión antes de enlazar el shader

```
glVertexAttribPointer(0, ...);  
glEnableVertexAttribArray(0);  
glBindAttribLocation(pid, 0, "nombre")
```







# Atributos

## Conexión después de enlazar el shader

---

- Otra opción es dejar a OpenGL que asigne el atributo a un índice, y luego preguntarle por dicha asignación (el programa debe estar ya enlazado):
- **`GLint glGetAttribLocation(GLuint program, const GLchar *name)`**
  - **program**: identificador del programa
  - **name**: nombre de la variable del shader que recibirá el atributo
  - devuelve el índice del atributo
- Ejemplo:

```
GLuint attribLoc;  
attribLoc = glGetAttribLocation(pid, "position");
```



# Atributos

## Conexión desde el propio shader

---

- La última opción consiste en que el shader especifique el índice que utilizará un atributo, usando el calificador `layout`:

```
#version 330 core
$GLMatrices
layout(location = 0) in vec4 position;
layout(location = 2) in vec2 texCoord;
out vec2 texCoordFrag;
void main() {
    texCoordFrag= texCoord;
    gl_Position = modelviewprojMatrix * position;
}
```



# Variables out-in

---

- Son variables en las que escribe un shader (*out*) y recibe (interpoladas o no) el siguiente (*in*)
- No son accesibles desde el programa cliente (CPU)
- Deben tener el mismo nombre y los mismos calificadores (excepto *in/out*) en ambos shaders

textureReplace.vert

```
[...]
in vec4 position;
in vec2 texCoord;
out vec2 texCoordFrag;
void main()
{
    texCoordFrag = texCoord;
    [...]
```

textureReplace.frag

```
[...]

in vec2 texCoordFrag;
out vec4 fragColor;

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



# Variables out-in

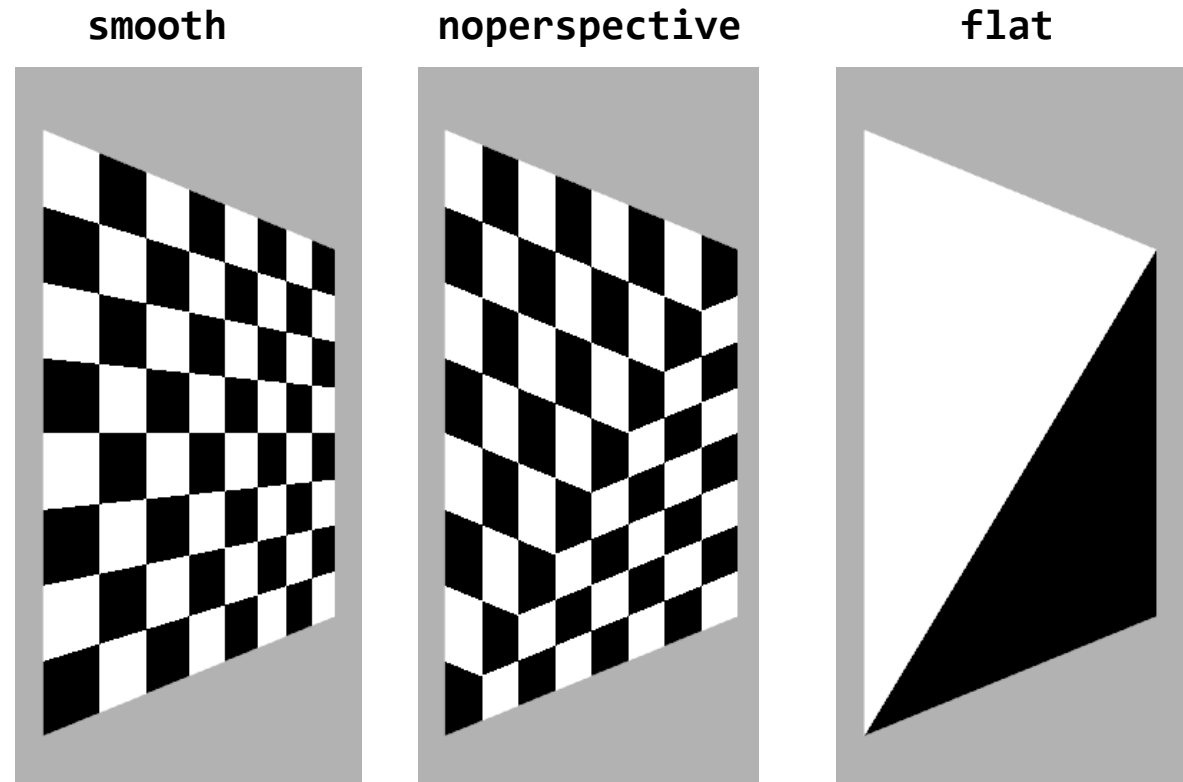
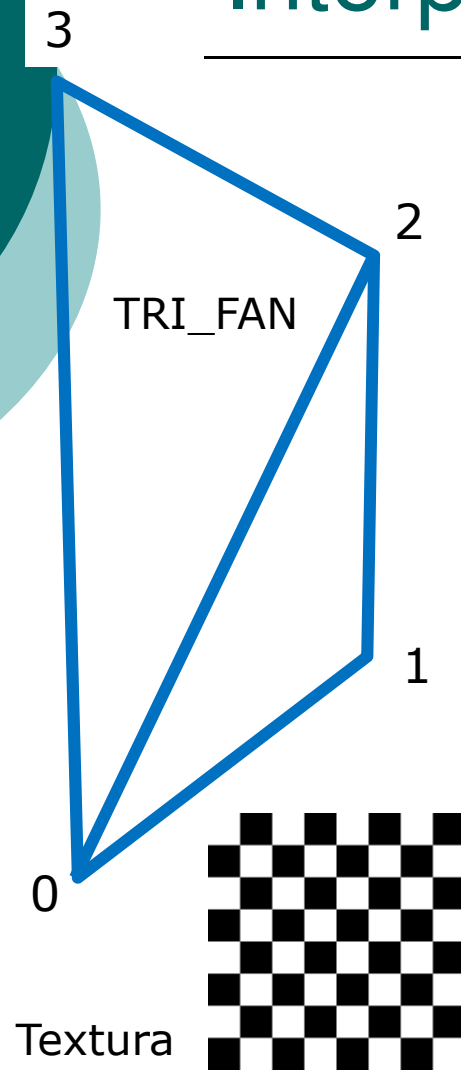
---

- Calificadores para especificar el tipo de interpolación durante la rasterización:
  - smooth, flat, noperspective
- Por ejemplo:

```
smooth out vec2 texCoordSmooth;  
flat out vec2 texCoordFlat;  
noperspective out vec2 texCoordNoPers;
```

# Interpolación de variables in-out

ej5-1



# Interpolación de variables in-out

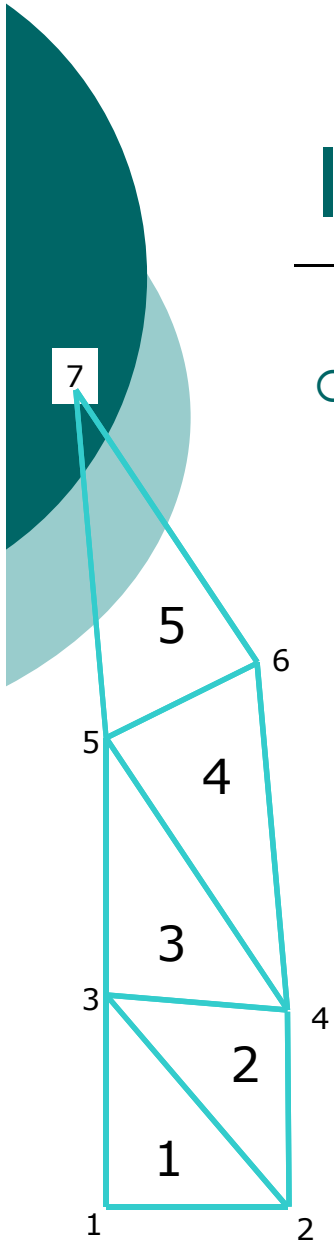
## ○ Provoking vertex

- Define de qué vertice se tomará el atributo al usar la interpolación constante (*flat*)
- `glProvokingVertex(GL_{LAST|FIRST}_VERTEX_CONVENTION)`

| Primitive type of polygon $i$ | First vertex convention | Last vertex convention                |
|-------------------------------|-------------------------|---------------------------------------|
| point                         | $i$                     | $i$                                   |
| independent line              | $2i - 1$                | $2i$                                  |
| line loop                     | $i$                     | $i + 1$ , if $i < n$<br>1, if $i = n$ |
| line strip                    | $i$                     | $i + 1$                               |
| independent triangle          | $3i - 2$                | $3i$                                  |
| triangle strip                | $i$                     | $i + 2$                               |
| triangle fan                  | $i + 1$                 | $i + 2$                               |

Polígono  $i$ -ésimo  
Vértices 1..n

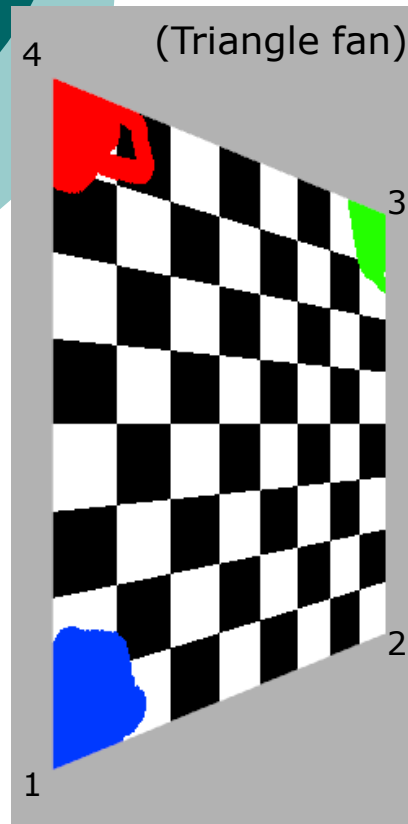
Tabla 13.2 espec. GL 4.6



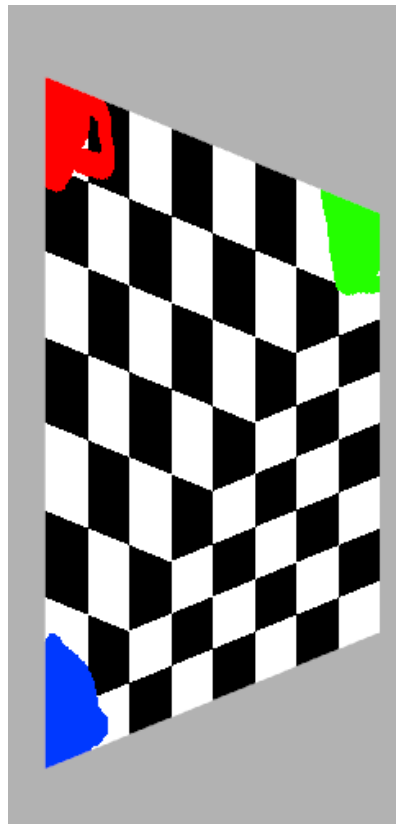
# Interpolación de variables in-out

ej5-1

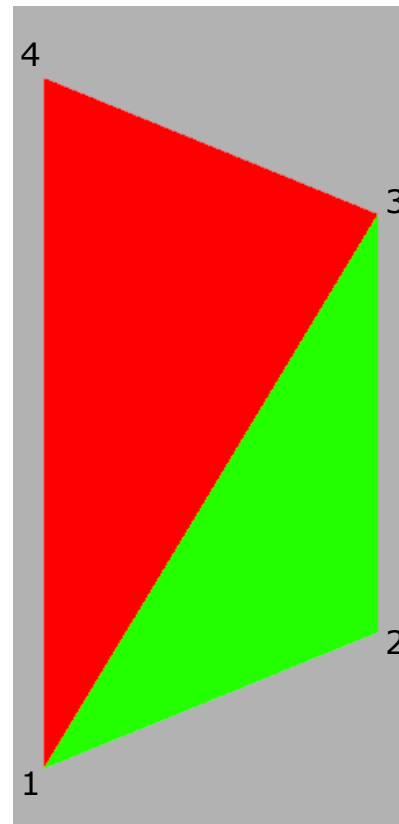
**smooth**



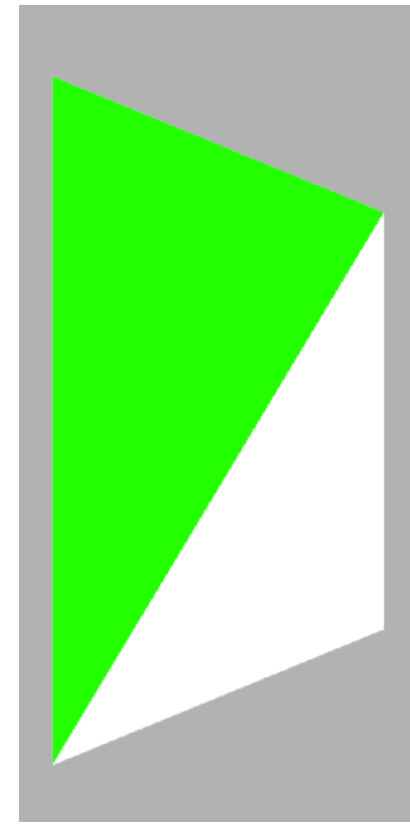
**noperspective**



**flat, last vertex**



**flat, first vertex**





# Bloques de interfaz

---

- Es posible agrupar variables “in” y “out” en bloques

```
// Shader de vértice
out Bump {
    vec3 normal;
    vec3 tangent;
} vs_bump;
```

```
// Shader de fragmento
in Bump {
    vec3 normal;
    vec3 tangent;
} fs_bump;
```





# *Uniform Buffer Objects*

---

- OpenGL 3.3 eliminó la mayoría de variables de estado que mantenía automáticamente la librería:
  - matrices de transformación, definición de materiales, definición de luces, cámara...
- Esto nos obliga a implementar y mantener este estado en la aplicación...
- lo que implica mantener una cantidad grande de uniforms



# *Uniform Buffer Objects*

---

- Los UBO permiten gestionar bloques de uniform fácilmente, y compartirlos entre distintos shaders
  - Los uniforms “normales” (definidos en el bloque por defecto) se almacenan en el objeto shader, mientras que los UBO no (están en un BO)
- Es habitual mantener las matrices de transformación, la definición del material de la primitiva actual o las luces activas en UBO
- Esto permite, por ejemplo, cambiar el valor de varias variables uniform en bloque con una sola llamada
- Los miembros de un UBO sólo pueden ser de tipos transparentes (vecN, float, double...) (no sampler2D, etc)



# *Uniform Buffer Objects*

---

- En el shader, se declaran como si fueran una estructura:

```
uniform GLMatrices ← Nombre del bloque
{
    mat4 modelMatrix;
    mat4 viewMatrix;
    mat4 projMatrix;
    mat4 modelviewMatrix;
    mat4 modelviewprojMatrix;
    mat3 normalMatrix;
} glmats; ← Instancia (opcional)
```



# *Uniform Buffer Objects*

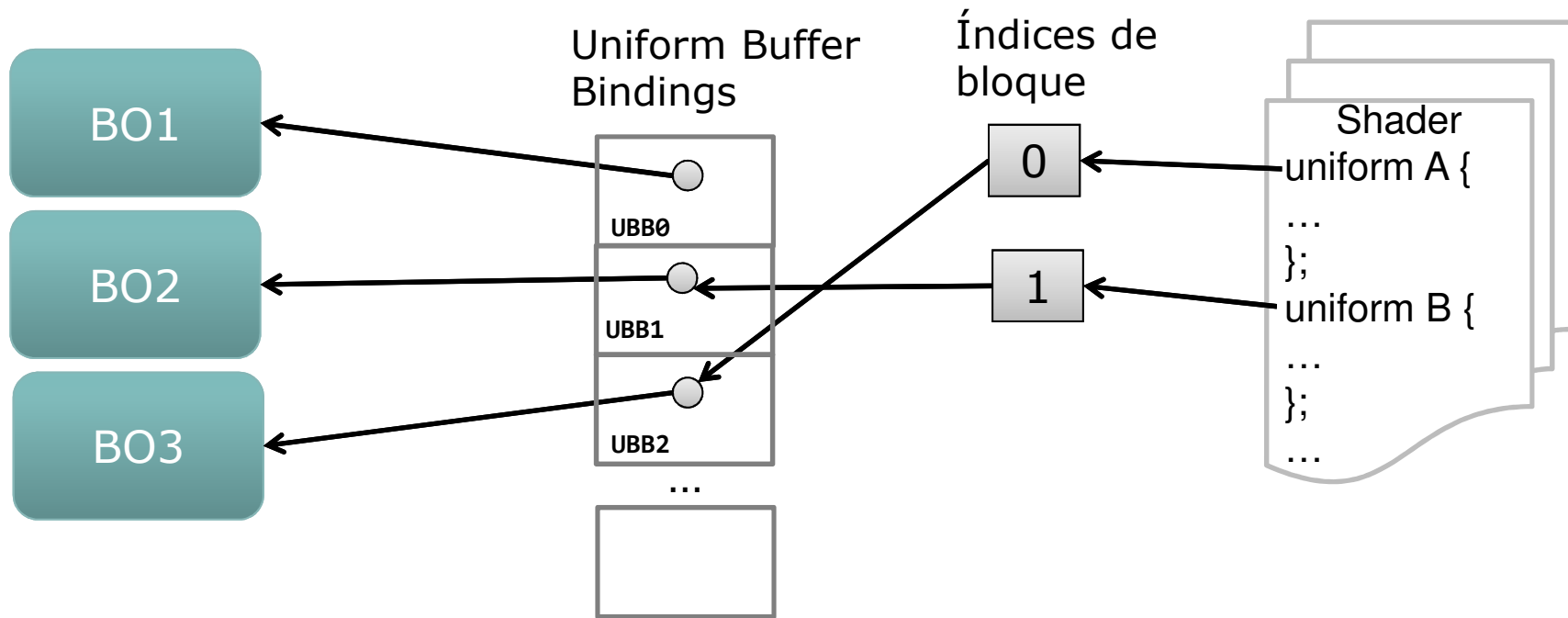
---

- Para acceder a los miembros de un UBO en el shader, se usa el nombre del campo, o el nombre de la instancia.campo

```
#version 330
uniform GLMatrices
{
    mat4 modelMatrix;
    mat4 viewMatrix;
    mat4 projMatrix;
    mat4 modelviewMatrix;
    mat4 modelviewprojMatrix;
    mat3 normalMatrix;
};
in vec4 vVertex;
void main(void) {
    gl_Position = modelviewprojMatrix * vVertex;
}
```

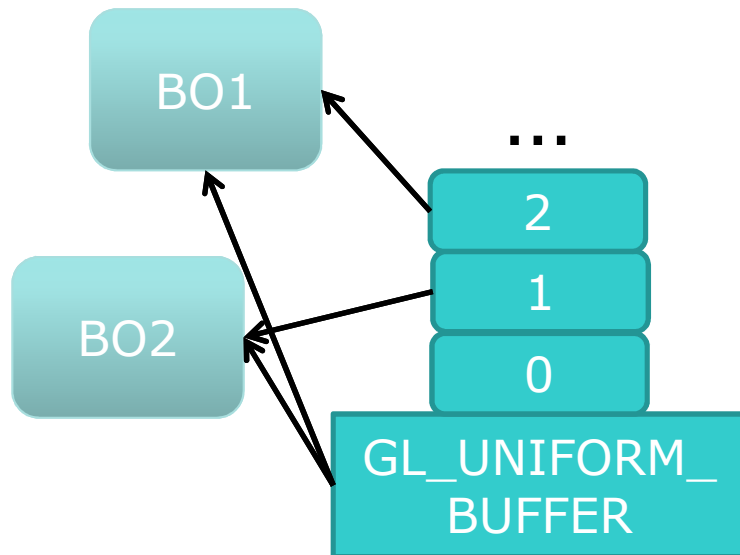
# Uniform Buffer Objects

- Para vincular un bloque de uniforms con un BO y hacerlo accesible al shader hay que dar varios pasos:



# Uniform Buffer Objects

- El punto de vinculación **GL\_UNIFORM\_BUFFER** es un punto de vinculación indexado:
  - Aparte del punto de vinculación “tradicional”, puede haber más de un BO vinculado en distintos índices



```
glBindBufferBase(GL_UNIFORM_BUFFER, 2, BO1);  
glBindBufferBase(GL_UNIFORM_BUFFER, 1, BO2);
```

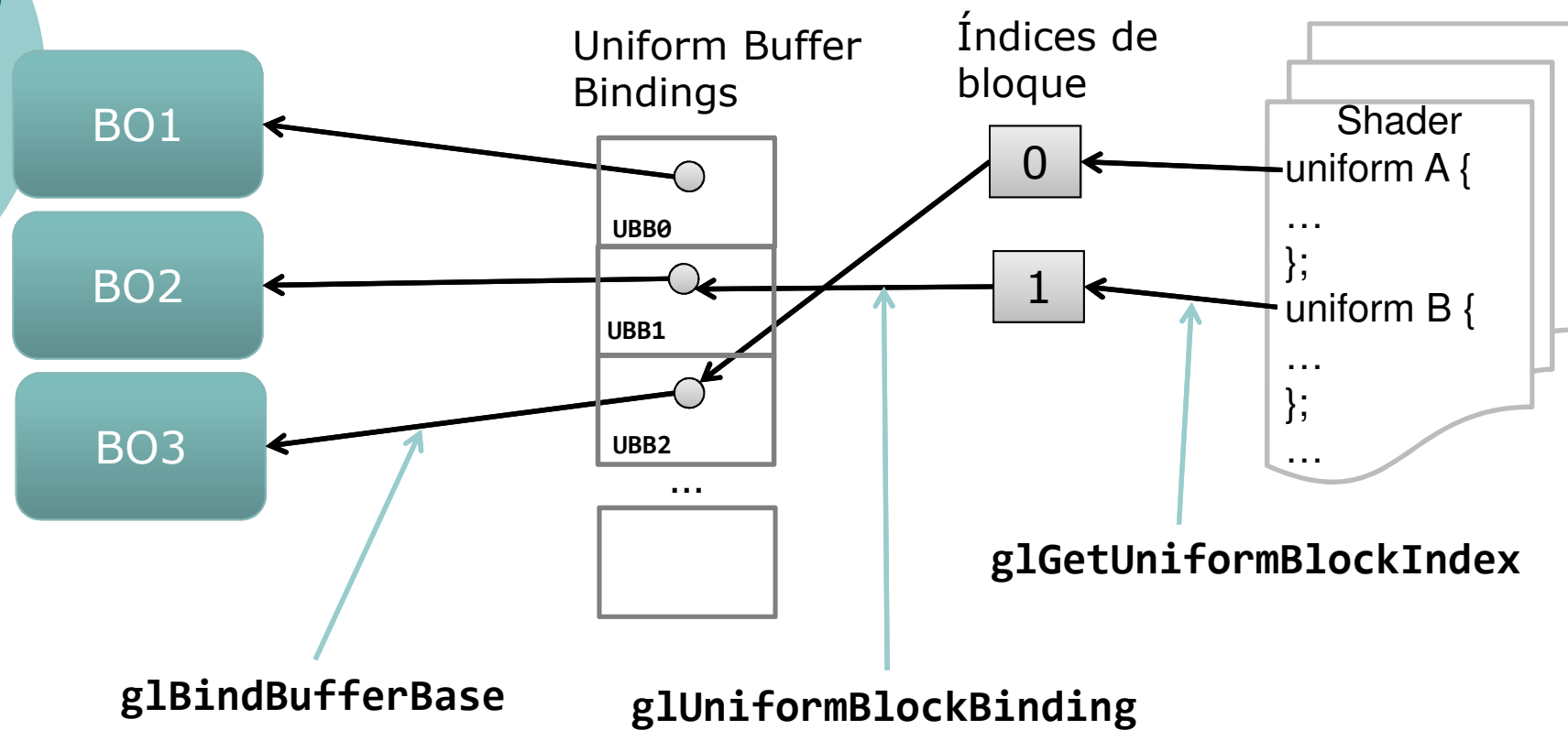


## *Uniform Buffer Objects*

---

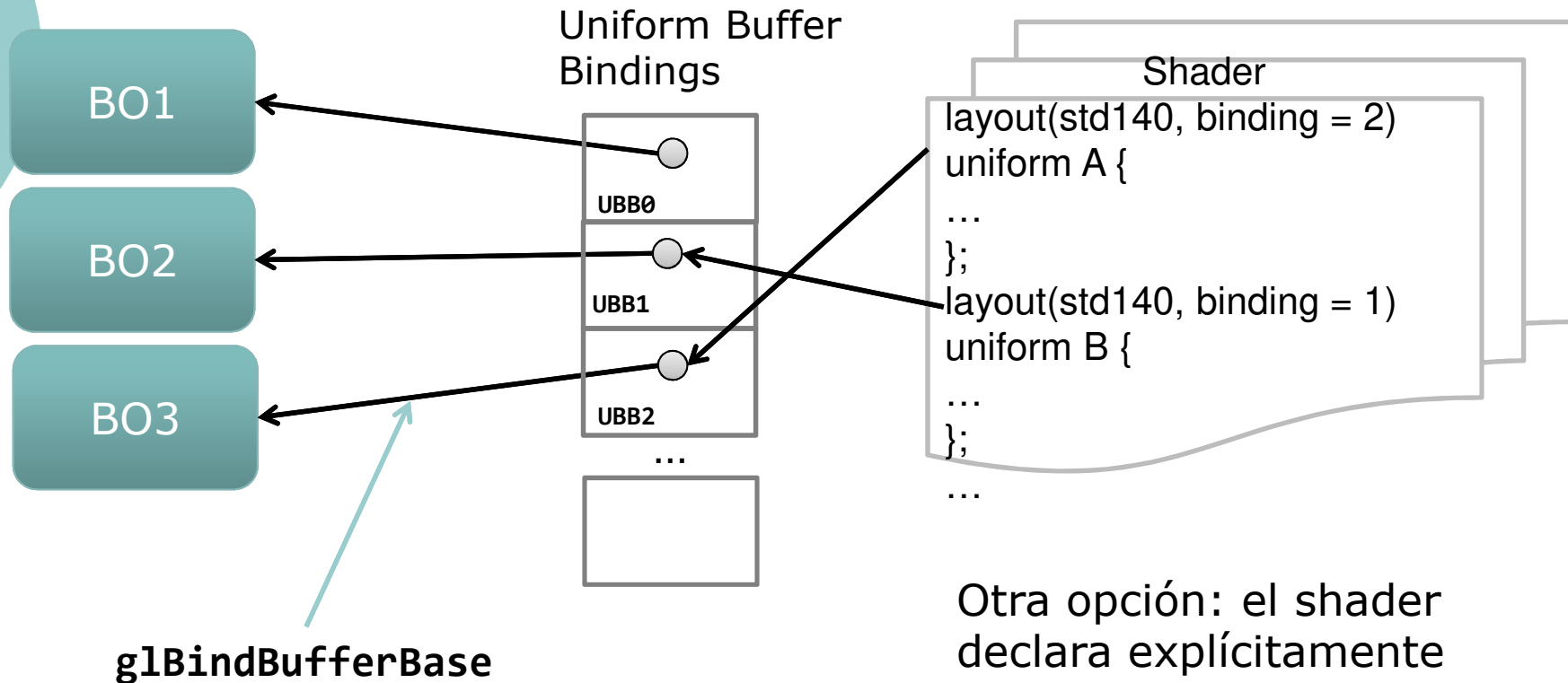
- Para calcular el número de puntos de vinculación de bloques de uniform:
  - `glGetIntegerv(GL_MAX_UNIFORM_BUFFER_BINDINGS, &i)`
  - En NVIDIA RTX 3070: 84
- Hay un número máximo de bloques de uniform que se pueden usar por:
  - programa: `GL_MAX_COMBINED_UNIFORM_BLOCKS`
  - etapa: `GL_MAX_{VERTEX|GEOMETRY|FRAGMENT|TESS_CONTROL|TESS_EVALUATION}_UNIFORM_BLOCKS`
  - En NVIDIA RTX 3070: 84, 14, 14, 14, 14, 14.

# Uniform Buffer Objects





# Uniform Buffer Objects



Otra opción: el shader declara explícitamente el índice de vinculación (GL 4.2)



# *Uniform Buffer Objects*

## Organización de la memoria

---

- Aún hay que definir la disposición de los campos en la memoria del BO.
- Hay dos opciones:
  - dejar a OpenGL que distribuya las variables según su criterio, y luego preguntarle (esta es la opción por defecto)
  - especificar una disposición estándar, donde se usan una serie de reglas predefinidas y comunes para todas las implementaciones de OpenGL
    - Cuidado con la alineación de los miembros de una estructura que hace el compilador de C++
- Ver detalles en el apéndice o en el código



## *Uniform Buffer Objects*

---

- Para modificar una variable de un bloque de uniform tan sólo hay que escribir en su Buffer Object :
  - `glBufferData`, `glBufferSubData`, `glMapBuffer`, etc.



## Responsabilidades del shader de vértice

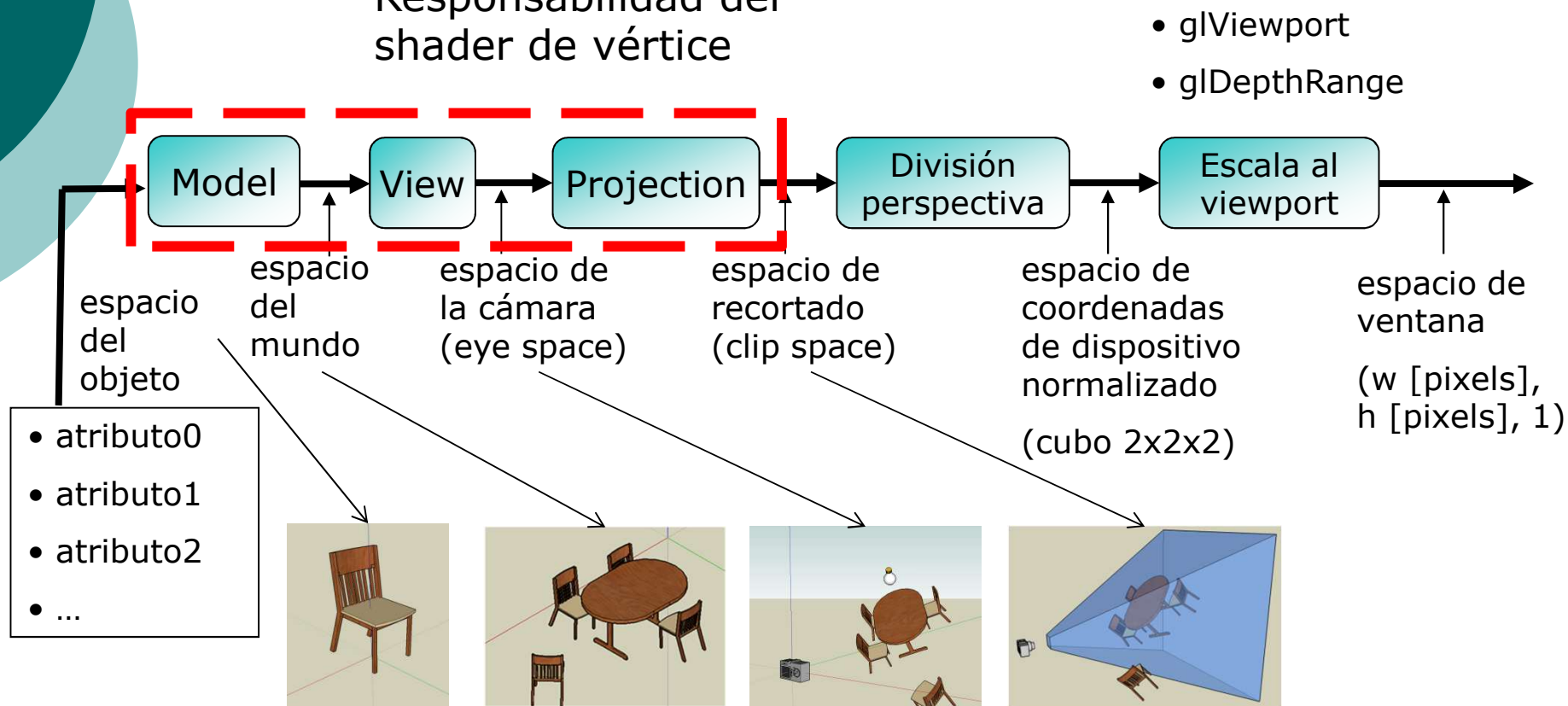
---

- El shader de vértice tiene varias responsabilidades:
  - transformar la posición del vértice al espacio de clipping (esto también se puede hacer en cualquier punto de la tubería de vértices)
  - pasar la información de los atributos que necesiten otros shaders
  - transformar las normales, calcular la componente de iluminación por vértice, quizá transformar coordenadas de textura...

# Responsabilidades del shader de vértice

## Transformación de la posición

### Responsabilidad del shader de vértice





# Responsabilidades del shader de vértice

## Transformación de la posición

---

- Multiplicar la posición por el producto de las matrices  $\text{PROJECTION} * \text{VIEW} * \text{MODEL}$ :

```
in vec4 position;  
gl_Position = modelviewprojMatrix * position;
```

- Para los cálculos de iluminación, se suele usar la posición del vértice en el espacio de la cámara:

```
vec4 ecPosition; vec3 ecPosition3;  
if (NeedEyePosition) {  
    ecPosition = modelviewMatrix * position;  
    ecPosition3 = (vec3(ecPosition)) / ecPosition.w;  
}
```



# Responsabilidades del shader de vértice

## Transformación de la normal

---

- Es común tener que llevar las normales al espacio de la cámara, pero no se puede usar la matrix MODELVIEW. Se debe usar:

```
in vec3 normal;  
out vec3 normalOut;  
normalOut = normalMatrix * normal;  
Donde normalMatrix se ha calculado como:  
mat3 normalMatrix =  
    transpose(inverse(mat3(modelViewMatrix)));
```

- Si hace falta normalizarla:

```
normalOut = normalize(normalOut);
```



# Iluminación en OpenGL

## Implementando la tubería fija con *shaders*

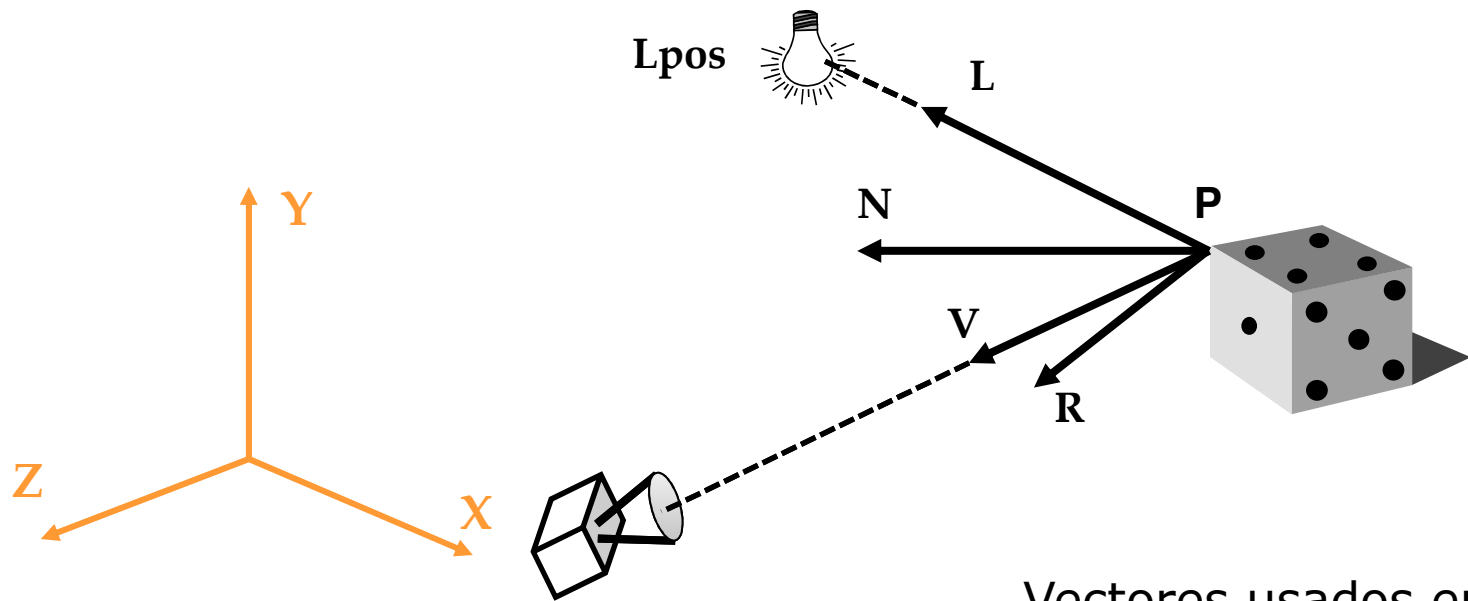
---

- Históricamente, para ahorrar transformaciones en los shaders, OpenGL llevaba la posición de las luces al espacio de la cámara
  - Por tanto, cuando se consulta la posición de una fuente de luz dentro de un *shader*, dicha posición venía dada en coordenadas de la cámara
- Para hacer el cálculo de iluminación por vértice, habrá que transformar tanto las posiciones como las normales al espacio de la cámara
  - Aplicar MODELVIEW a las posiciones de vértices
  - Aplicar la inversa traspuesta de MODELVIEW a las normales (disponible en la variable `normalMatrix` de PGUPV)



# Iluminación en OpenGL

Implementando la tubería fija con *shaders*



Vectores usados en el cálculo de la iluminación



# Iluminación en OpenGL

Implementando la tubería fija con *shaders*

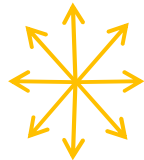
---

- El color calculado por OpenGL 1.x para un vértice es:
  - *color = componente emisorio del material + las contribuciones de cada fuente;*
  - Contribución de cada fuente:  
*contribución = factor de atenuación \* efecto foco \* (términos ambiental + difuso + especular);*

# Iluminación en OpenGL

Implementando la tubería fija con *shaders*

- Hay tres tipos de fuentes básicas:



Puntual

```
directional == 0  
spotCutoff == 180
```

Emite la misma intensidad en todas direcciones.



Direccional

```
directional != 0
```

Los rayos son paralelos y se asume que la fuente está en el infinito, en la dirección `posicion.xyz` (por tanto, los rayos van hacia `-posicion.xyz`). Se ignora la atenuación y el efecto foco. Por ello,  
`L=normalize(posicion.xyz)`



Focal

```
directional == 0  
spotCutoff < 180
```

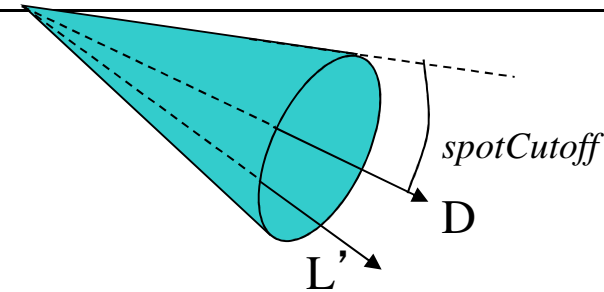
Define una posición y una dirección. Aquellos vértices que estén fuera del cono de luz no recibirán luz de esta fuente.

# Iluminación en OpenGL

## Implementando la tubería fija con *shaders*

- Contribución de las fuentes:

$$\text{factor de atenuación} = \frac{1}{\max(1, k_c + k_l d + k_q d^2)}$$



$$\text{efecto foco} = \begin{cases} 1 & \text{si la fuente no es focal (spotCutoff = 180.0)} \\ 0 & \text{si la fuente es focal, pero el vértice está fuera del cono iluminado} \\ \max(L' \cdot D, 0)^{\text{spotExponent}} & \text{en otro caso} \end{cases}$$

donde:

- d: distancia de la fuente al vértice
- L' : el vector unitario que apunta desde la fuente al vértice
- D: la dirección del foco (*spotDirection*)
- el vértice está dentro del cono si:  $\max(L' \cdot D, 0) \geq \cos(\text{spotCutoff}) = \text{spotCosCut off}$

# Iluminación en OpenGL

Implementando la tubería fija con *shaders*



- Contribución de las fuentes:

- Término ambiental:

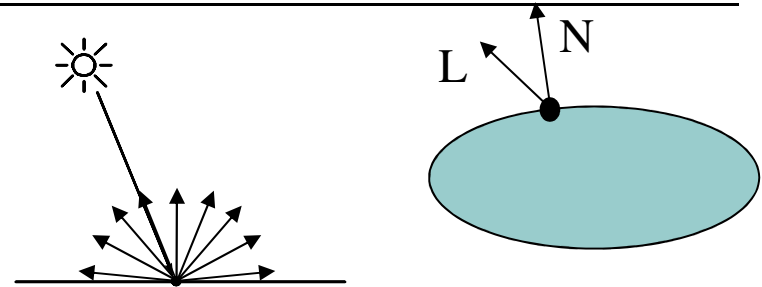
comp. ambiente de la luz  $\times$  refl. ambiente del material

- Término difuso:

$\max(L \cdot N, 0) \times$  comp. difusa de la luz  $\times$  refl. difusa del material

- donde:

- L: el vector unitario que apunta desde el vértice a la fuente
    - N: normal unitaria al vértice

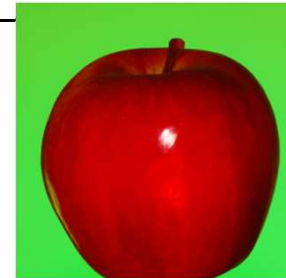


# Iluminación en OpenGL

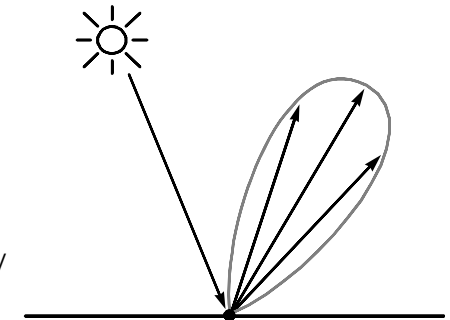
## Implementando la tubería fija con *shaders*

### ○ Contribución de las fuentes:

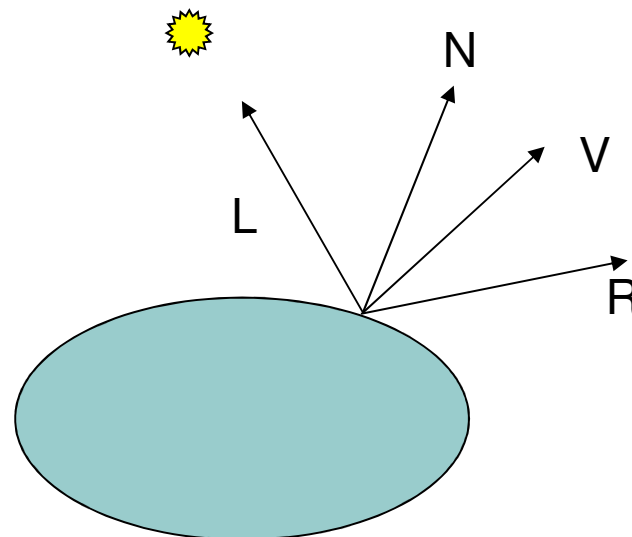
- Término especular:
- Sólo hay término especular si  $L \cdot N > 0$



flickr.com/photos/revilla/



$\max(R \cdot V, 0)^{shininess} \times \text{comp. especular de la luz} \times \text{refl. especular del material}$



R es el reflejo de L con respecto a la normal. Se puede calcular con la función GLSL:

$R = \text{reflect}(-L, N)$

(N debe ser unitario)

```
#version 330
```

```
$GLMatrices  
$Lights  
$Material
```

```
in vec3 position;  
in vec3 normal;
```

```
out vec4 color;
```

```
void main() {
```

```
    // Normal en el espacio de la cámara
```

```
    vec3 eN = normalize(normalMatrix * normal);
```

```
    // Vértice en el espacio de la cámara
```

```
    vec3 eposition = vec3(modelviewMatrix * position);
```

```
    // Vector vista (desde vértice a la cámara)
```

```
    vec3 V = normalize(-eposition.xyz);
```

```
    // Cálculo de la iluminación
```


```
    color = iluminacion(eposition, eN, V);
```

```
    gl_Position = modelviewprojMatrix * position;
```

```
}
```

Durante la compilación, PGUPV sustituye estas directivas por la definición de los bloques de uniform correspondientes. Con una sólo llamada a `connectUniformBlock` antes de compilar, PGUPV se encarga de todo

gouraud.vert  
en ej5-2



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

**\$GLMatrices**


---

```
struct LightSource {
    vec4 ambient, diffuse, specular;
    vec4 positionWorld, positionEye; // Posición de la luz (mundo y cámara)
    vec3 spotDirectionWorld; // Dirección del foco en el S.C. del mundo
    int directional; // Es direccional?
    vec3 spotDirectionEye; // Dirección del foco en el S.C. de la cámara
    int enabled; // Está encendida?
    float spotExponent, spotCutoff, spotCosCutoff; // Datos del foco
    vec3 attenuation; //  $k_c$ ,  $k_l$ ,  $k_q$ 
};
layout (std140) uniform Lights {
    LightSource lights[4];
};
```

**\$Lights**

gouraud.vert





```
layout (std140) uniform Material {  
    vec4 diffuse;  
    vec4 ambient;  
    vec4 specular;  
    vec4 emissive;  
    float shininess;  
    int textureCount;  
};
```

**\$Material**

```
uint numDiffTextures() { return bitfieldExtract(textureCount, 0, 3); }  
uint numSpecTextures() { return bitfieldExtract(textureCount, 3, 3); }  
uint numNormTextures() { return bitfieldExtract(textureCount, 6, 3); }  
uint numHeightTextures() { return bitfieldExtract(textureCount, 9, 3); }  
uint numOpacTextures() { return bitfieldExtract(textureCount, 12, 3); }  
uint numAmbTextures() { return bitfieldExtract(textureCount, 15, 3); }
```

**gouraud.vert**

```
// Suponiendo fuentes puntuales
vec4 iluminacion(vec3 pos, vec3 N, vec3 V) {
    int i;
    // Componente emisiva del material
    vec4 color = emissive;
    for (i = 0; i < lights.length(); i++) {
        if (lights[i].enabled == 0) continue;
        // Vector iluminación (desde vértice a la fuente)
        vec3 L = normalize(vec3(lights[i].positionEye) - pos);
        // Multiplicador de la componente difusa
        float diffuseMult = max(dot(N, L), 0.0);
        float specularMult = 0.0;
        if (diffuseMult > 0.0) {
            // Multiplicador de la componente especular
            vec3 R = reflect(-L, N);
            specularMult = max(0.0, dot(R, V));
            specularMult = pow(specularMult, shininess);
        }
        color += lights[i].ambient * ambient +
                lights[i].diffuse * diffuse * diffuseMult +
                lights[i].specular * specular * specularMult;
    }
    return color;
}
```

gouraud.vert



# Sombreado en OpenGL

## Sombreado por vértice

---

```
#version 330
```

```
in vec4 color;  
out vec4 fragColor;
```

```
void main() {  
    fragColor = color;  
}
```

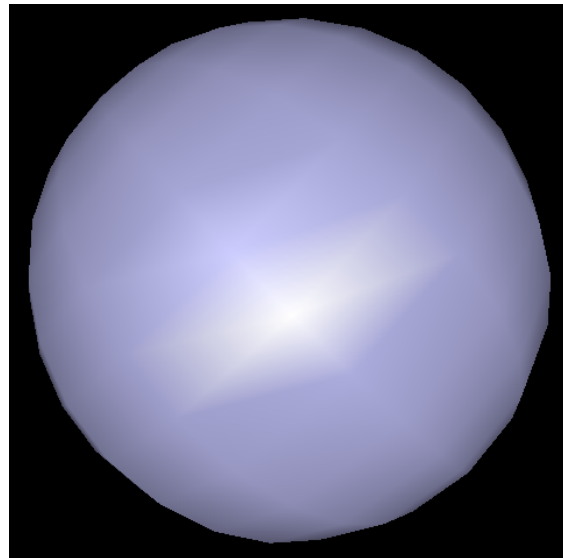
gouraud.frag

# Sombreado en OpenGL

## Sombreado por vértice

---

- Problemas de la iluminación por vértice

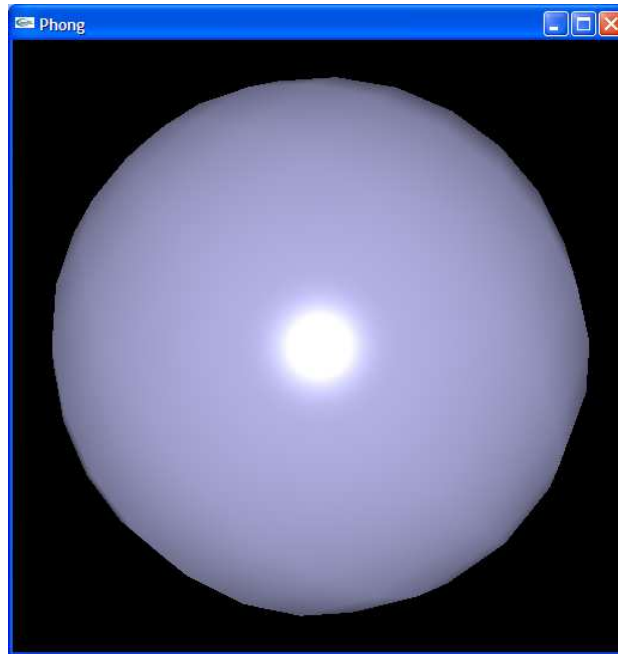


# Sombreado en OpenGL

## Sombreado por píxel

---

- El sombreado de Phong (por píxel) obtiene mejores resultados con la misma geometría:

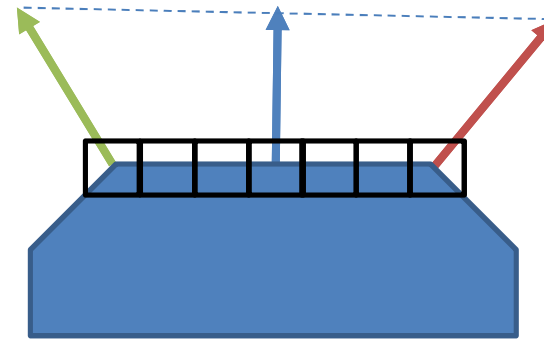


# Sombreado en OpenGL

## Sombreado por píxel

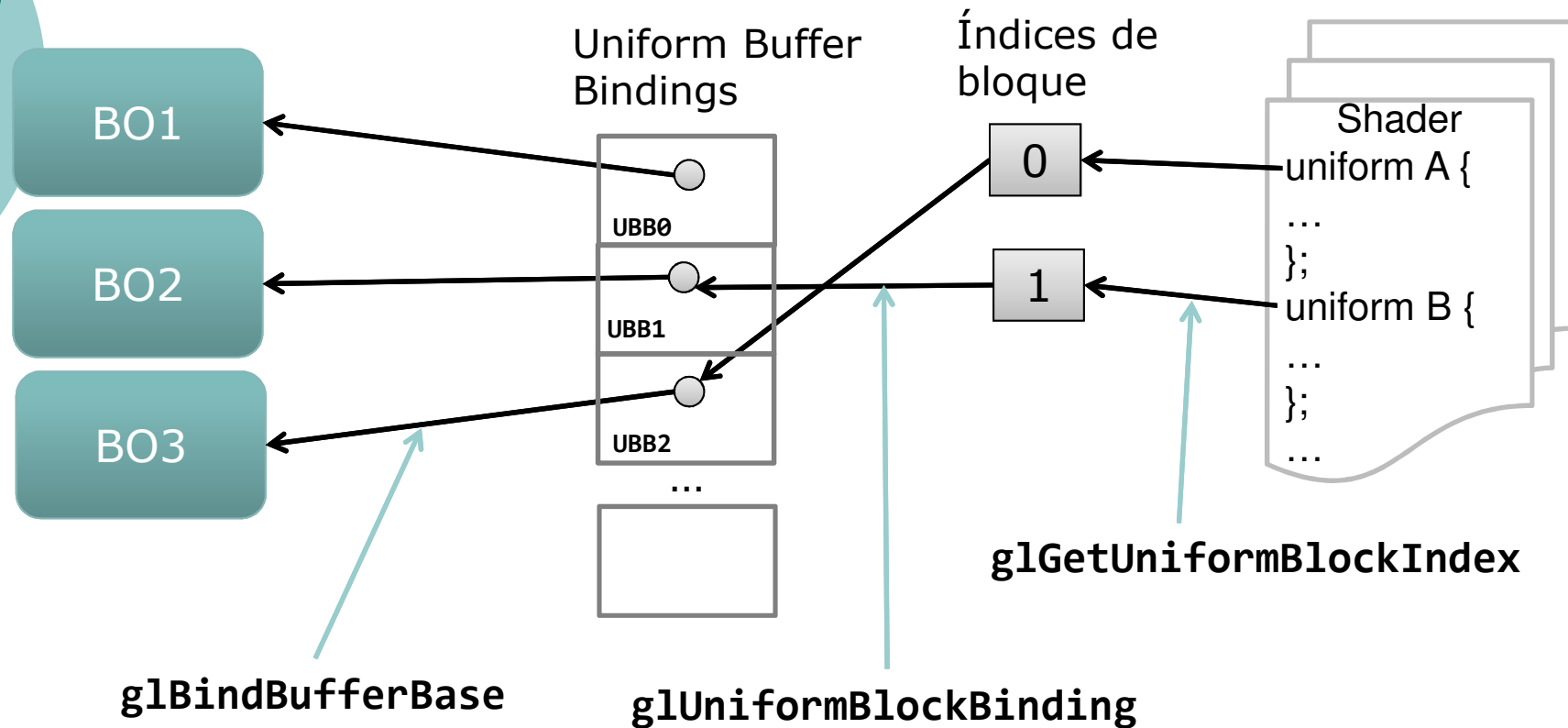
---

- Vertex shader
  - Definir  $N$  y la posición como *out* para pasarlos al shader de fragmento.
  - Llevarlos al espacio de la cámara
- Fragment shader
  - Renormalizar  $N$
  - Calcular  $L$  y  $V$  en el espacio de la cámara
  - Aplicar la ecuación de iluminación



# Uniform Buffer Objects

## Detalles de uso





# *Uniform Buffer Objects*

## Detalles de uso

---

- El primer paso es obtener el índice asignado por el compilador a cada bloque del shader:
  - `GLuint glGetUniformBlockIndex(GLuint program, const GLchar *uniformBlockName)`
    - **program**: identificador del programa
    - **uniformBlockName**: nombre del bloque de uniforms
    - devuelve el índice del bloque o `GL_INVALID_INDEX` si no lo encuentra





# *Uniform Buffer Objects*

## Detalles de uso

---

- Luego hay que vincular el índice de un bloque del shader el índice de un punto de vinculación:
  - `void glUniformBlockBinding(GLuint program, GLuint ubIndex, GLuint uniformBlockBinding)`
    - `program`: identificador del programa
    - `ubIndex`: índice del bloque obtenido en el paso anterior
    - `uniformBlockBinding`: índice del punto de vinculación del bloque. Un valor entre 0 y (número de puntos de vinculación-1)
- Esta vinculación se mantiene aunque el shader deje de estar activo



# *Uniform Buffer Objects*

## Detalles de uso

---

- Por último, hay que vincular un BO al mismo punto de vinculación:
  - **void glBindBufferBase(GLenum target, GLuint index, GLuint buffer)**
    - **target:** GL\_UNIFORM\_BUFFER
    - **index:** índice del punto de vinculación donde conectar el BO
    - **buffer:** identificador del buffer object (devuelto por **glGenBuffers**)



# *Uniform Buffer Objects*

## Organización de la memoria

---

- Primera opción: consultar a OpenGL las posiciones de cada variable dentro del UBO:
  - `glGetUniformIndices`
  - `glGetActiveUniformsiv`
  - Ver pp. 65 del libro rojo 8ª ed.
- Luego usar esas posiciones para escribir en zonas determinadas del BO



# *Uniform Buffer Objects*

## Organización de la memoria

---

- La segunda opción consiste en usar una organización de memoria estándar, y construir un tipo de datos equivalente en memoria cliente:

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



# *Uniform Buffer Objects*

Reglas de la distribución *std140*

---

1. If the member is a scalar consuming  $N$  basic machine units, the base alignment is  $N$ .
  2. If the member is a two- or four-component vector with components consuming  $N$  basic machine units, the base alignment is  $2N$  or  $4N$ , respectively.
  3. If the member is a three-component vector with components consuming  $N$  basic machine units, the base alignment is  $4N$ .
- ... y así hasta 10.
  - Ver pp. 146 de la especificación de OpenGL 4.6



# Uniform Buffer Objects

## Reglas de la distribución *std140*

---

- Otra opción es definir el bloque de uniform en el shader y consultar los desplazamientos, para construir la estructura de datos a posteriori.
- Por ejemplo:

```
layout (std140) uniform TransformBlock {  
    float scale;  
    vec3 translation;  
    float rotation[3];  
    mat4 proj_matrix;  
};
```

ej5-3



# Uniform Buffer Objects

## Reglas de la distribución *std140*

---

- Pulsar Ctrl+i en el ejemplo ej5-3:

[...]

GL\_MAX\_COMBINED\_UNIFORM\_BLOCKS: 84

GL\_MAX\_VERTEX\_UNIFORM\_BLOCKS: 14

GL\_MAX\_UNIFORM\_BLOCK\_SIZE: 65536

[...]

GL\_ACTIVE\_UNIFORMS: 4

GL\_ACTIVE\_UNIFORM\_BLOCKS: 1

[0]: TransformBlock

GL\_UNIFORM\_BLOCK\_BINDING: 0

GL\_UNIFORM\_BLOCK\_DATA\_SIZE: 144

Members:

proj\_matrix, offset=80, size=1, type=GL\_FLOAT\_MAT4

rotation[0], offset=32, size=3, type=GL\_FLOAT

scale, offset=0, size=1, type=GL\_FLOAT

translation, offset=16, size=1, type=GL\_FLOAT\_VEC3



# *Uniform Buffer Objects*

## Reglas de la distribución *std140*

---

- La estructura correspondiente en memoria de CPU podría ser:

```
struct paddedfloat {  
    GLfloat f;  
    char _pad[sizeof(vec4)-sizeof(float)];  
};  
struct TransformBlock {  
    GLfloat scale;  
    char _pad1[sizeof(vec4)-sizeof(GLfloat)];  
    glm::vec3 translation;  
    char _pad2[sizeof(vec4)-sizeof(vec3)];  
    paddedfloat rotation[3];  
    glm::mat4 proj_matrix;  
};
```