

## **Shaders**

### **Vértices y Fragmentos**

Material original: Ana Gil Luezas  
Ejemplos: Antonio Gavilanes  
Adaptación al curso 24/25: Alberto Núñez  
Departamento de Sistemas Informáticos y Computación  
Universidad Complutense de Madrid

- ☐ Accedemos a la información de OpenGL a través de variables
- ☐ Se aplican a la geometría – generalmente – en el espacio de coordenadas del modelo y produce una geometría en el espacio recortado “clip” 3D.
  - ☐ Usa las variables de entrada, y las modifica para que éstas sean accesibles – como salida – para otros shaders.
- ☐ En esencia, sustituye las operaciones de
  - ☐ Transformación del vértice
  - ☐ Transformación de la normales
  - ☐ Normalización de normales
  - ☐ Manejo de la luz por vértice
  - ☐ Manejo de las coordenadas de textura
- ☐ Las funciones que NO realiza (las hace las funciones fijas del pipeline)
  - ☐ Recortado de volumen
  - ☐ División homogénea
  - ☐ Mepeado del Puerto de vista
  - ☐ Culling de la cara posterior
  - ☐ Modo polígono
  - ☐ Modo offset

# OpenGL: Vertex Shaders

Fixed-function: posición, color, coordenadas de textura, . . .

**in:** valores de un vértice ( $v$ ) de la malla  
(position and attributes)

Vertex  
Coordinates  
Attribute  
Variables  
Uniform  
Variables  
Textures

Vertex  
Shader

Output  
Variables  
 $gl\_Position$

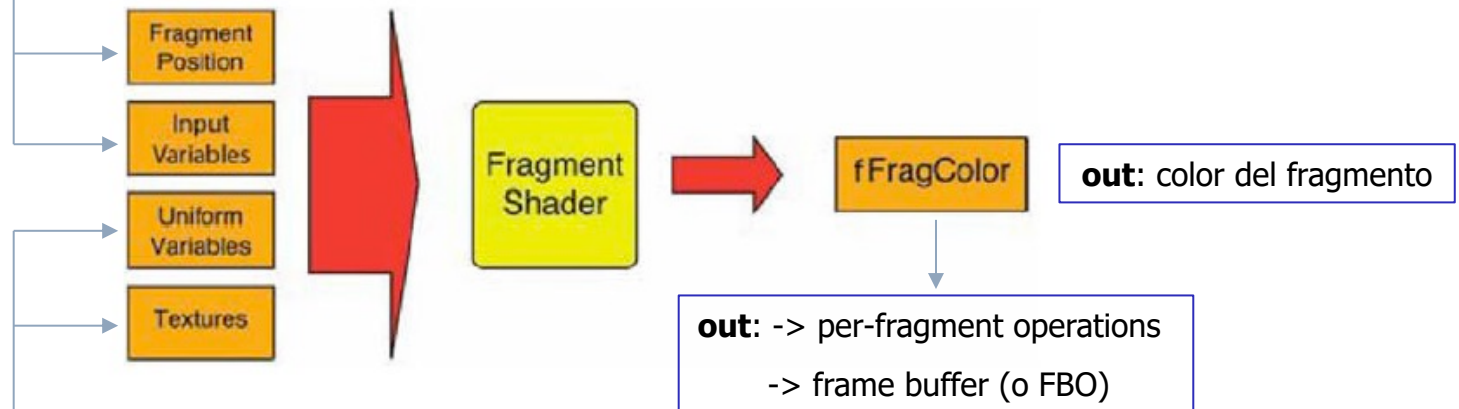
**out:** valores del vértice  
que serán interpolados

**uniform:** datos globales del programa  
(constantes en cada ejecución del programa).  
Accesibles también en el *fragment shader*

Clipping coordinates:  $\mathbf{cv} = \text{Projection} * \text{View} * \text{Model} * \mathbf{v}$   
View-space:  $\mathbf{vv} = \text{View} * \text{Model} * \mathbf{v}$   
World-space:  $\mathbf{wv} = \text{Model} * \mathbf{v}$

- ☐ Se aplican sobre los fragmentos para determinar el color del pixel
- ☐ El proceso de rasterización interpola
  - ☐ Color
  - ☐ Profundidad
  - ☐ Coordenadas de textura
- ☐ El shader utiliza esa interpolación – e información adicional - para generar el color final del pixel
- ☐ El cómputo de los fragmentos se realiza en paralelo
- ☐ Este tipo de shader reemplaza o añade las siguientes operaciones
  - ☐ Cómputo del color
  - ☐ Texturización
  - ☐ Luz por pixel
  - ☐ Niebla
  - ☐ Descarte de pixels en los fragmentos
- ☐ No reemplaza las siguientes operaciones
  - ☐ Blending
  - ☐ Stencil, depth y scissor tests
  - ☐ Operaciones de punteado
  - ☐ Operaciones de raster al escribir el pixel en el framebuffer

**in** del fragment shader (valores de un fragmento)  $\leftrightarrow$  **out** del vertex shader interpolados, y predefinidas: `gl_FragCoord` (Screen coordinates), `gl_FrontFacing`, ...



**uniform:** datos globales del programa (constantes en cada ejecución del programa). Accesibles también en el *vertex shader*

# Vertex Shader variables (Recordatorio)

- ❑ Variables definidas en la aplicación que dan acceso a la información del estado de cada vértice

- ❑ Atributos de entrada (per vertex -> mesh): in (no se pueden modificar)

```
in vec4 vertex;    // Coordenadas de posición
in vec3 normal;    // Vector normal
in vec2 uv0;       // Coordenadas de textura 0
```

- ❑ Atributos de salida (in transformados): out (hay que darles valor)

```
out vec4 gl_Position; // predefinida obligatoria
out vec2 vUv0;        // coordenadas de textura 0
```

- ❑ Transformaciones: uniform (constantes del programa)

```
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;
uniform mat3 normalMatrix;
uniform mat2 texCoordMatrix;
```

- ❑ Texturas: uniform sampler2D nombreTex

# Fragment Shader variables (Recordatorio)

- ❑ Valores de entrada (per fragment -> interpolados): **in** (no se pueden modificar)

```
in vec4 gl_FragCoord;    // Predefinida asociada a gl_Position (out del VS)
in bool gl_FrontFacing;  // Predefinida
in vec2 vUv0;            // En el vertex Shader: out vec2 vUv0;
```

- ❑ Valores de salida (para cada píxel): **out**

```
out vec4 fFragColor;      // En versiones anteriores, predefinida gl_FragColor
```

- ❑ **uniform** (constantes del programa)

```
uniform float intLuzAmb;
```

- ❑ Texturas: uniform

```
uniform sampler2D materialTex;
```

# Fragment Shader: Variables predefinidas

## ❑ `bool gl_FrontFacing`

- ❑ Si true → El fragmento corresponde a la cara frontal (*front*)

```
if (gl_FrontFacing)
    color = frontColor;
else
    color = backColor;
```

- ❑ Puede ser necesario ajustar la variable preguntando si ha invertido el orden de los vértices

- ❑ Añade el parámetro `uniform float Flipping;`

- ❑ -1 → está invertido; 1 → no está invertido

- ❑ Puede ser útil para definir la variable

```
bool frontFacing = (Flipping > -1)? gl_FrontFacing : !gl_FrontFacing;
```

- ❑ Y utilizar `frontFacing` en lugar de `gl_FrontFacing` en los condicionales.

```
if (frontFacing)
    color = frontColor;
else
    color = backColor;
```

- ❑ Se puede pasar este parámetro desde la aplicación:

- ❑ `param_named_auto Flipping render_target_flipping // -1 o 1`



# Fragment Shader: Variables predefinidas

- ❑ Para descartar un fragmento

- ❑ `discard`

- ❑ El fragmento queda descartado y no seguirá el proceso de renderizado
    - ❑ Efecto return

- ❑ `vec4 gl_FragCoord`

- ❑ Coordenadas del fragmento en *Screen space*
  - ❑ Origen abajo a la izquierda (*lower-left*)

```
if (gl_FragCoord.y < 12 || gl_FragCoord.x < 12)
    discard;
```

- ❑ Para renderizar las caras frontales y traseras de un fragmento, utilizar

- ❑ `cull_hardware none` y `cull_software none`
  - ❑ Útil si queremos tener control sobre el renderizado de las dos caras del fragmento
  - ❑ Por ejemplo, para ver la parte interior de objetos huecos

- ☐ Las coordenadas de textura definen cómo se asigna una imagen a una geometría.
- ☐ Una coordenada de textura se asocia a cada vértice de la geometría e indica qué punto de la imagen de textura debe asignarse a ese vértice.
- ☐ Las coordenadas de textura no se almacenan con “apariencia”, sino en cada geometría individualmente.
  - ☐ Esto permite que geometrías separadas compartan una apariencia con una textura de imagen y, sin embargo, muestren porciones distintas de esa imagen en cada geometría.
- ☐ Cada coordenada de textura es, como mínimo, un par  $(u,v)$ 
  - ☐ Ubicación horizontal y vertical en el espacio de textura.
  - ☐ Los valores suelen estar en el intervalo  $[0,1]$ .
  - ☐ El origen  $(0,0)$  está en la parte inferior izquierda de la textura.

- ❑ Las coordenadas de la textura también pueden tener valores opcionales  $w$  y  $q$ 
  - ❑ Estas coordenadas son opcionales
  - ❑ Posibles representaciones:  $(u,v,w)$ ,  $(u,v,q)$  o  $(u,v,w,q)$
- ❑  $w$  se utiliza para
  - ❑ mapeados de textura más complejos en el espacio 3D.
  - ❑ al renderizar, junto con los valores de transformación de la textura, como rotación, escalado y desplazamiento.
  - ❑  $w$  es un valor extra contra el que multiplicar los valores de transformación de la textura
  - ❑ Similar a cuando se transforma una posición del espacio-objeto al espacio-pantalla 3D.
  - ❑ Al multiplicar el  $uvw$  con los valores de la transformación de proyección, se obtienen dos coordenadas (a menudo llamadas  $s$  y  $t$ ) que se mapean en una textura 2D.
- ❑  $q$  se utiliza para escalar las coordenadas de la textura cuando se emplean técnicas como la interpolación proyectiva.

# Ejemplo de shader: vértices y fragmentos

## ❑ Vertex Shader

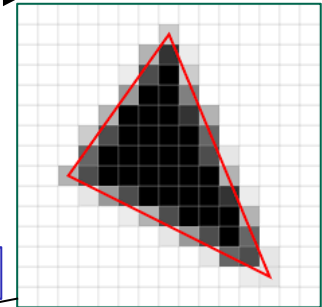
```
#version 330 core
in vec4 vertex;
uniform mat4 modelViewProjectionMatrix;

void main(void) {
    gl_Position = modelViewProjectionMatrix * vertex;
}
```

Model C.

```
vec4 vertices[] = {
    {-0.5, -0.5, 0.0, 1},
    { 0.5, -0.5, 0.0, 1},
    { 0.0,  0.5, 0.0, 1},,};
```

Clipping C.



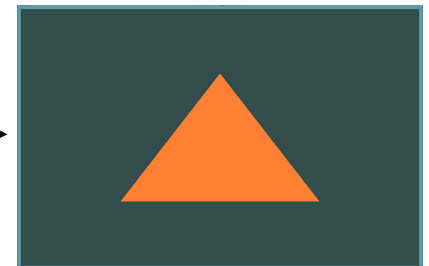
Screen C.

## ❑ Fragment Shader

```
#version 330 core
out vec4 fFragColor;

void main(void) {
    fFragColor = vec4(1.0, 0.5, 0.2, 1.0);
}
```

RGBA Z

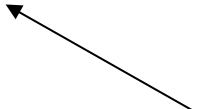


# Ejemplo de shader: vértices

- ❑ GLSL Vertex Shader: al menos pasa, en `gl_Position`, las coordenadas de los vértices en Clip-space, al proceso de recorte.
- ❑ El rasterizador las interpolará, junto con todos los valores `out`, y los fragmentos así generados, pasarán al fragment shader.
- ❑ Cada ejecución procesa 1 vértice y genera 1 vértice

```
// Archivo .glsl
#version 330 core
in vec4 vertex;           // Atributos de los vértices a procesar
in vec2 uv0;              // Coordenadas de textura 0
uniform mat4 modelViewProjMat; // Constante - uniform - de programa
out vec2 vUv0;            // out del vertex shader

void main() {
    vUv0 = uv0;            // Se copian las coordenadas de textura
    gl_Position = modelViewProjMat * vertex; // Obligatorio
}                          // (Clipping coordinates)
```



predefinido: out vec4

# Ejemplo de shader: fragmentos

- ❑ GLSL Fragment Shader: Mezcla de dos texturas
- ❑ Cada ejecución procesa 1 fragmento y genera 1 color

// Archivo.glsl

```
#version 330 core

uniform sampler2D texturaL;           // Tipo sampler2D para texturas 2D
uniform sampler2D texturaM;           // -> unidades de textura (int)
uniform float BF;                     // Blending factor
uniform float intLuzAmb;              // Luz ambiente blanca
in vec2 vUv0;                         // Out (del vertex shader)
out vec4 fFragColor;                 // Out (del fragment shader)

void main() {
    vec3 colorL = vec3(texture(texturaL, vUv0)); // Acceso a téxel
    vec3 colorM = vec3(texture(texturaM, vUv0)); // Configuración!
    vec3 color = mix(colorL, colorM, BF) * intLuzAmb; // Mix -> (1-BF).colorL + BF.colorM
    fFragColor = vec4(color, 1.0); // Out
}
```

## ☐ **mix(x, y, p)**

- ☐ Linearly interpolate between two values
- ☐ **x**: Specify the start of the range in which to interpolate
- ☐ **y**: Specify the end of the range in which to interpolate
- ☐ **p**: Specify the value to use to interpolate between *x* and *y*.

## ☐ **normalize(v)**

- ☐ Calculates the unit vector in the same direction as the original vector
- ☐ **v**: Specifies the vector to normalize

## ☐ **transpose(m)**

- ☐ calculate the transpose of a matrix
- ☐ **m**: Specifies the matrix of which to take the transpose

## ☐ **texture(sampler, p)**

- ☐ retrieves texels from a texture
- ☐ **sampler**: Specifies the sampler to which the texture from which texels will be retrieved is bound.
- ☐ **p**: Specifies the texture coordinates at which texture will be sampled.

## ☐ **sin(angle) y cos(angle)**

- ☐ return the sine and cosene of the parameter angle
- ☐ Specify the quantity, in radians

## ☐ **dot(x, y)**

- ☐ calculate the dot product of two vectors
- ☐ **x**: Specifies the first of two vectors
- ☐ **y**: Specifies the second of two vectors

## ☐ **max(x, y)**

- ☐ return the greater of two values, *x* and *y*.

```
vertex_program exampleVS glsl{
    source exampleFS.glsl
    default_params{
        param_named_auto modelViewProjMat worldviewproj_matrix
    }
}
```

```
fragment_program exampleFS glsl{
    source exampleFS.glsl
    default_params{

        param_named texturaL    int    0
        param_named texturaM    int    1
        param_named      BF      float 0.5
        param_named intLuzAmb    float 1.0
    }
}
```

```
material materialName{
    technique{
        pass{
            vertex_program_ref exampleVS{
                fragment_program_ref exampleFS{

                    texture_unit{
                        texture texturaA.jpg 2d
                        tex_address_mode clamp
                        filtering bilinear
                    }
                    texture_unit {
                        texture texturaB.jpg 2d
                        tex_address_mode wrap
                    }
                }
            }
        }
    }
}
```

**exampleVS.glsl**

```
#version 330 core
in vec4 vertex;
in vec2 uv0;
uniform mat4 modelViewProjMat;
out vec2 vUv0;

void main() {
    vUv0 = uv0;
    gl_Position = modelViewProjMat * vertex;
}
```

```
#version 330 core
in vec2 vUv0;
uniform sampler2D texturaL;
uniform sampler2D texturaM;
uniform float BF;
uniform float intLuzAmb;
out vec4 fFragColor;

void main() {
    vec3 colorL = vec3(texture(texturaL, vUv0));
    vec3 colorM = vec3(texture(texturaM, vUv0));
    vec3 color = mix(colorL, colorM, BF) * intLuzAmb;
    fFragColor = vec4(color, 1.0);
}
```

**exampleFS.glsl**



- ❑ Supongamos que las coordenadas de textura en el shader de vértices son
  - ❑ `in vec2 uv0`
- ❑ Entonces nos podemos referir a las dos coordenadas mediante
  - ❑ `uv0.s` y `uv0.t`
- ❑ Para escalar una textura por un factor ZF (Zoom factor), a ambas coordenadas se les realizan las siguientes operaciones, por este orden:
  - ❑ Centrarlas en el cuadrado  $[-0.5, 0.5] \times [-0.5, 0.5]$  (Traslación)
  - ❑ Aplicarles el factor dado ZF (Escala)
  - ❑ Centrarlas de nuevo en el cuadrado  $[0, 1] \times [0, 1]$  (Traslación)
- ❑ Por ejemplo, pasando de un ZF de 0.5 a 1 sería:

```
vUv1.s = (uv0.s - 0.5) * (ZF) + 0.5;
```

```
vUv1.t = (uv0.t - 0.5) * (ZF) + 0.5;
```

- ❑ Para animar el escalado de la textura basta con que el factor de escalado varíe con el tiempo

- ❑ Podemos utilizar una variable del tipo

```
param_named_auto sintime sintime_0_2pi ...
```

- ❑ Si lo hacemos así, el factor de escalado varía dentro del siguiente rango

$\text{sintime} \in [-1,1]$

- ❑ Animación con escalado entre un rango de tamaños

- ❑ Supongamos que la animación debe pasar desde la textura a su tamaño normal a una textura que sea  $x$  veces su tamaño normal
- ❑ Supongamos que el factor de escalado se expresa como combinación lineal de dos escalares  $a$ ,  $b$ , es decir:
  - ❑  $ZF = \text{sintime} * a + b, \text{sintime} \in [-1,1]$

- ❑ Para obtener  $ZF$  plantea y resuelve un sistema de dos ecuaciones con dos incógnitas  $a$ ,  $b$

- ❑ Obtén las ecuaciones aplicando los siguientes valores:

- ❑  $\text{sintime} = -1 \rightarrow ZF = x$
- ❑  $\text{sintime} = 1 \rightarrow ZF = 1$

## ❑ Material (coeficientes de reflexión)

vec3 **Diffuse** (Ambient)

vec3 **Specular**

float **Shininess**

## ❑ Luces

vec3 **Position** / **Direction** // En World o View space

vec3 **Ambient**

vec3 **Diffuse**

vec3 **Specular**

## ❑ Punto a iluminar → En Word o View space

vec3 **vertex**; → En Word o View space

vec3 **normal**; → En Word o View space

## ❑ Cámara

vec3 **eye**; → En Word o View space

**Importante:** todas  
en el mismo sistema  
de coordenadas

# Componente difusa: Ley de Lambert

## ❑ Reflexión difusa (color)

- ❑ Se aplica un factor de reducción (diff) en función del coseno del ángulo  $\theta$  que forman los vectores  $\mathbf{n}$  (vector normal) y  $\mathbf{s}$ .

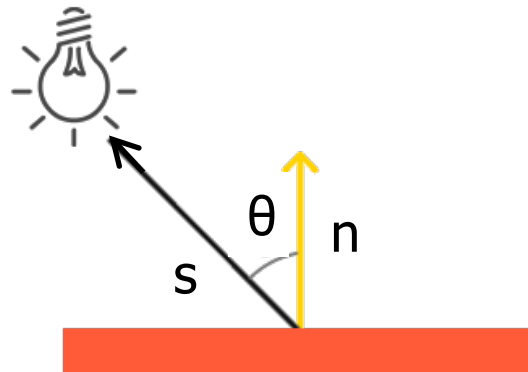
$\cos(\theta) = \text{dot}(\mathbf{n}, \mathbf{s})$  para vectores de magnitud 1

```
float diff = max(0, dot(n, s));
```

```
vec3 diffuse = diff * LightDiffuse * MaterialDiffuse;
```

- ❑ El vector normal de la cara *Back* es el opuesto al de la cara *Front* ( $\mathbf{n}$ ):

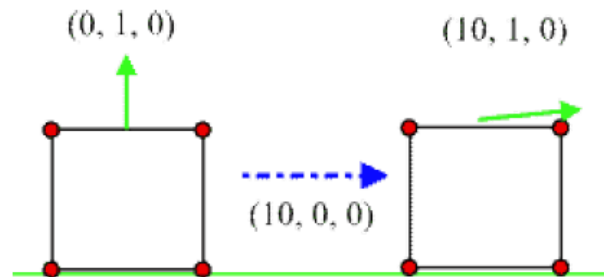
$\text{BackFaceNormal} = - \text{FrontFaceNormal} = - \mathbf{n}$



# Transformaciones de los vectores normales

## ❑ Traslaciones

- ❑ Los vectores normales **NO** se deben trasladar.
- ❑ Al trasladar un objeto sus vectores normales no cambian.



Si trasladamos el objeto y la normal

## ❑ Rotaciones

- ❑ Los vectores normales se deben rotar de la misma forma que el objeto
- ❑ Utilizando la misma rotación.

```
vec3 normal;  
mat4 atMat;    // Matriz afín    atMat =  $\begin{pmatrix} M & T \\ 0 & 1 \end{pmatrix}$   
  
vec3 nt = vec3(atMat * vec4(normal, 0.0));  
vec3 nt = mat3(atMat) * normal;
```

M: rotación y escala

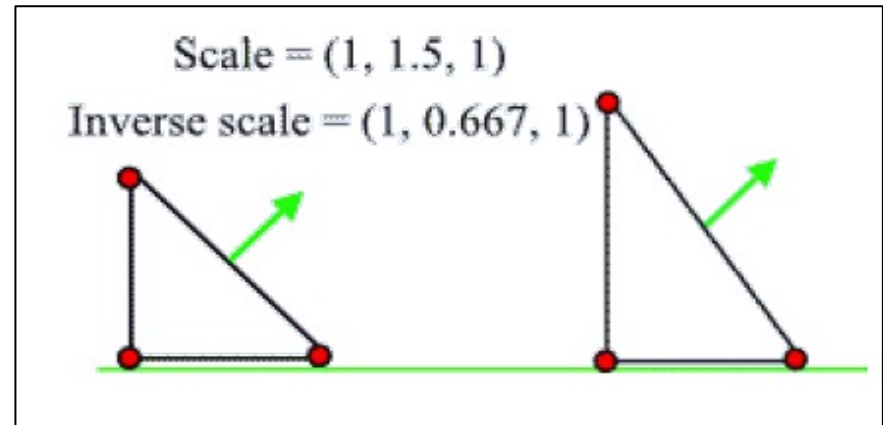
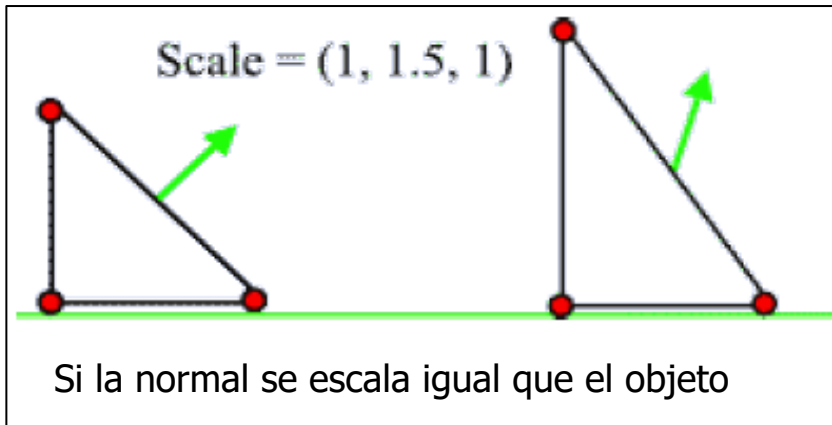
T: traslación

nt no se ve afectado por  
la traslación (T) de la matriz

# Transformaciones de los vectores normales

## ❑ Escalas

- ❑ La magnitud del vector se ve afectada → Hay que normalizarlo después de aplicarle la transformación.
- ❑ La **escala no uniforme** no preserva las normales
  - ❑ Los vectores dejan de ser perpendiculares, y los cálculos de iluminación quedarían distorsionados.



- ❑ Los vectores normales se deben escalar por la escala inversa.
- ❑ Hay que normalizar el vector con `normalize(vector)`

# Transformaciones de los vectores normales

## ❑ Normal matrix

- ❑ La transpuesta de la inversa de la submatriz de rotación y escala (*M: left-top 3x3 submatrix*).
- ❑ No se realiza la traslación, se realiza la rotación y la escala inversa.

$$atMat = \left( \begin{array}{c|c} M & T \\ \hline 0 & 1 \end{array} \right)$$

**Importante:** la matriz de proyección no tiene esta forma (no es afín)

```
vec3 normal;  
mat4 atMat;    // Matriz afín -> view, model o modelview matrix  
vec3 nt = mat3(transpose(inverse(atMat))) * normal;  
  
uniform mat4 normalMat;  
vec3 nt = normalize(vec3(normalMat * vec4(normal, 0.0)));
```

# Vertex Shader : Luz difusa

- ❑ Iluminación RGB difusa en view space (front & back faces)

```
in vec4 vertex;  
in vec3 normal;  
in vec2 uv0;
```

Vertex Shader

**uniform:** información sobre la fuente de luz y  
coeficientes de reflexión del material (Front / Back)

```
out vec2 vUv0;           // Coordenadas de textura  
out vec3 vFrontColor;    // Color RGB de la iluminación de la cara Front (normal)  
out vec3 vBackColor;     // Color RGB de la iluminación de la cara Back (-normal)
```

Fragment Shader

**uniform:** textura(s)

```
out vec4 fFragColor;     // color del fragmento ¿Front or Back face? gl_FrontFace
```



# Vertex Shader : Luz difusa

## ❑ Iluminación RGB difusa en view space (front & back faces)

```
uniform mat4 modelViewMat;           // View*Model matrix
uniform mat4 modelViewProjMat;       // Projection*View*Model matrix
uniform mat4 normalMat;              // transpose(inverse(modelView))

uniform vec3 lightAmbient;           // Intensidad de la luz ambiente
uniform vec3 lightDiffuse;           // Intensidad de la luz difusa
uniform vec4 lightPosition;          // Datos de la fuente de luz en view space
                                     // lightPosition.w == 0 -> directional light
                                     // lightPosition.w == 1 -> positional light

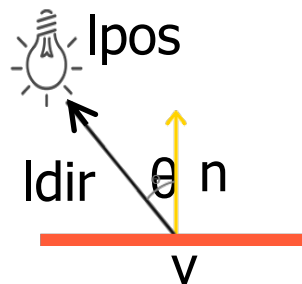
uniform vec3 materialDiffuse;        // Datos del material ¡Front=Back!
```

# Vertex Shader : Luz difusa

## ❑ Iluminación RGB difusa en view space (front & back faces)

GLSL permite funciones: por defecto los parámetros son `in` (por copia)

```
float diff(vec3 cVertex, vec3 cNormal){  
    vec3 lightDir = lightPosition.xyz;           // Directional light?  
    if (lightPosition.w == 1)                     // Positional light ?  
        lightDir = lightPosition.xyz - cVertex;  
  
    return max(dot(cNormal, normalize(lightDir)), 0.0); // dot: coseno ángulo  
}
```



## ❑ Iluminación RGB difusa en view space (front & back faces)

```
void main() {  
    vec3 ambient = lightAmbient * materialDiffuse;           // Ambient  
  
    // Diffuse en view space  
    vec3 viewVertex = vec3(modelViewMat * vertex);  
    vec3 viewNormal = normalize(vec3(normalMat * vec4(normal,0)));  
  
    vec3 diffuse = diff(viewVertex, viewNormal) * lightDiffuse * materialDiffuse;  
    vFrontColor = ambient + diffuse; // + Specular  
  
    diffuse = diff(viewVertex, -viewNormal) * lightDiffuse * materialDiffuse;  
    vBackColor = ambient + diffuse; // + specular  
  
    vUv0 = uv0;  
    gl_Position = modelViewProjMat * vertex; // En Clip-space  
}
```

# Fragment Shader : Luz difusa

## ❑ Iluminación y textura (front & back faces)

```
#version 330 core
in vec3 vFrontColor;           // Color de la iluminación interpolado
in vec3 vBackColor;           // Color de la iluminación interpolado
in vec2 vUv0;                 // Ccoordenadas de textura interpoladas
out vec4 fFragColor;
uniform sampler2D materialTex; // Front = Back

void main() {
    vec3 color = texture(materialTex, vUv0).rgb;
    if (gl_FrontFacing)
        color = vFrontColor * color;
    else
        color = vBackColor * color;

    fFragColor = vec4(color, 1.0);
}
```

En Ogre puede ser necesario ajustar ...

# Iluminación (Fragment shader)

- ❑ Para mejorar la precisión (mallas con poca resolución) se realizan los cálculos de la iluminación en el fragment shader.
- ❑ Vertex shader
  - ❑ No realiza los cálculos → No necesita los datos del material ni de las luces.
  - ❑ Además de las coordenadas de los vértices en Clip-space, tiene que pasar al fragment shader las coordenadas de los vértices y de los vectores normales transformadas al espacio mundial o de vista (ambos en el mismo espacio).

```
in vec4 vertex;  
in vec3 normal;  
in vec2 uv0;  
out vec2 vUv0;           // Coordenadas de textura  
out vec3 vXxxNormal;     // Coordenadas de la normal en Xxx space  
out vec4 vXxxVertex;     // Coordenadas del vértice en Xxx space
```

- ❑ Fragment shader
  - ❑ Realiza los cálculos → Necesita los datos del material y de las luces.

```
out vec4 fFragColor;     // color del fragmento ¿Front or Back face?
```

- ❑ Para la iluminación specular hace falta la posición de la cámara y las correspondientes componentes de la luz y el material.
- ❑ Para varias luces se puede utilizar un array de luces
- ❑ Para añadir más realismo se utilizan texturas para definir los coeficientes de reflexión de los materiales y los vectores normales (por fragmento)