Informática Gráfica II

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

Vertex Shaders

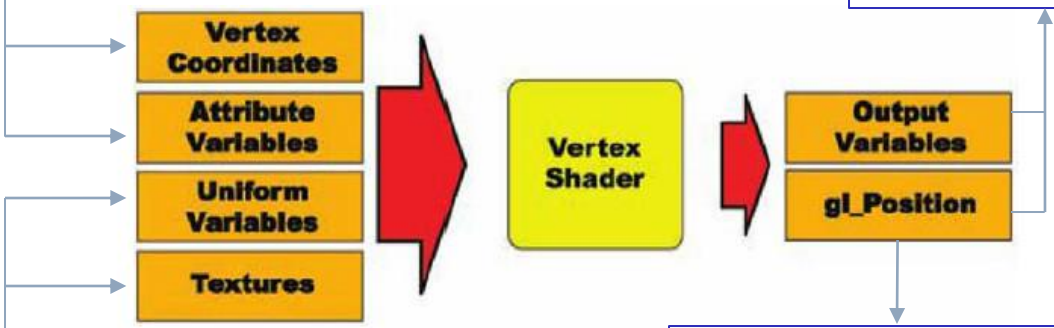
- Accedemos a la información de openGL a través de variables
- Se aplican a la geometría – gemeralmente – en el espacio de coordenadas del modelo y produce una geometría en el espacio recortado "clip" 3D.
  - Usa las variables de entrada, y las modifica para que éstas sean accesibles – como salida – 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 homegénea
  - Mepeado del Puerto de vista
  - Culling de la cara posterior
  - Modo polígono
  - Modo offset

openGL: Vertex Shaders

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

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

out: valores del vértice que serán interpolados



Clipping coordinates: **cv** = `Projection*View*Model*v`

View-space: **vv** = `View*Model*v`

World-space: **wv** = `Model*v`

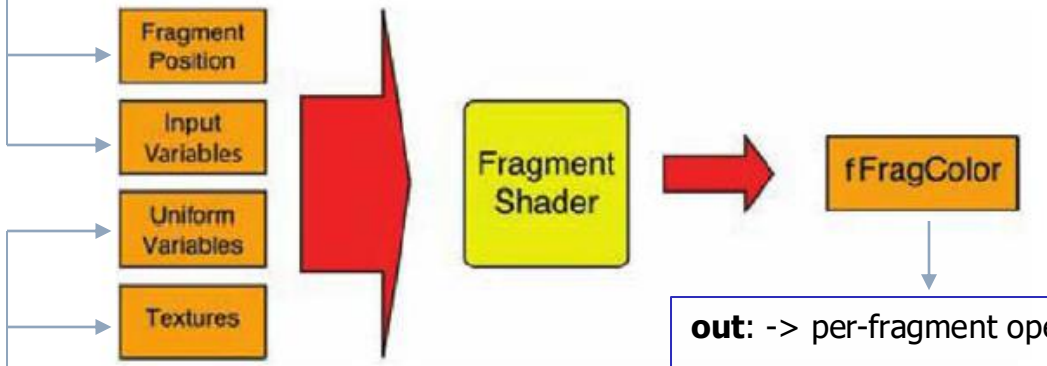
uniform: datos globales del programa (constantes en cada ejecución del programa). Accesibles también en el *fragment shader*

Fragment shaders

- 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

openGL: Fragment Shader

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



out: color del fragmento

out: -> per-fragment operations
-> frame buffer (o FBO)

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;      // Coordenada de textura 0
```

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

```
out vec4 gl_Position;  // Predefinida -> Posición del vértice
out vec2 vUv0;         // Coordenadas de textura 0
```

- Transformaciones: `uniform` (constantes del programa)
  - El el script del material

```
param_named_auto modelViewProjMat worldviewproj_matrix
param_named_auto normalMatrix normal_matrix
```

  - En el shader

```
uniform mat4 modelViewProjMat;
uniform mat3 normalMatrix;
```

- Texturas: `uniform` sampler2D nombreTex
  - Aunque en el shader de vértices es costoso.
  - Conviene limitar su uso a texturas pequeñas.

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`
  - En el script del material (shader de fragmentos)

```
param_named texture1 int 0
param_named texture2 int 1
```

  - En el shader:

```
uniform sampler2D texture1;
uniform sampler2D texture1;
```

En el material:

```
// Textura 0
texture_unit{
  texture lightMap.jpg
  tex_address_mode clamp
}
// Textura 1
texture_unit{
  texture spaceSky.jpg
  tex_address_mode clamp
}
```

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

Coordenadas de Textura

- 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.

Coordenadas de Textura

- Las coordenadas de la textura también pueden tener valores opcionales `w` y `q`
  - Estas coordenadas son opcionales
  - Posibles representaciones: (s, t), (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

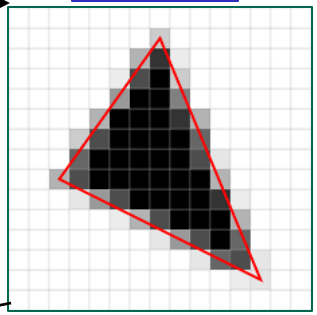
```
#version 330 core
in vec4 vertex;
uniform mat4 modelViewProjectionMatrix;

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.



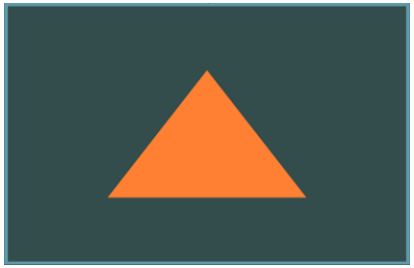
- Fragment Shader

```
#version 330 core
out vec4 fFragColor;

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

Screen C.

RGBAZ



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
  }
```

Funciones GLSL

- **`mix(x, y, p)`**
  - Interpola linealmente entre dos valores.
  - **`x`**: Especifica el inicio del intervalo en el que interpolar
  - **`y`**: Especifica el final del intervalo en el que interpolar
  - **`p`**: Especifica el valor a utilizar para interpolar entre *x* e *y*.
- **`normalize(v)`**
  - Calcula el vector unitario en la misma dirección que el vector original
  - **`v`**: Vector a normalizar
- **`traspose(m)`**
  - Calcula la traspuesta de la matriz **`m`**
- **`texture(sampler, p)`**
  - Obtiene el texel de la textura
  - **`sampler`**: Especifica el sampler al que está vinculada la textura de la que se recuperarán los texels.
  - **`p`**: Especifica las coordenadas de textura en las que se muestreará la textura.
- **`sin(angle) y cos(angle)`**
  - Calcula el seno y coseno del ángulo **`angle`**, respectivamente, en radianes
- **`dot(x, y)`**
  - Calcula el producto escalar de los vectores **`x`** e **`y`**.
- **`max(x, y)`**
  - Devuelve el valor mayor entre x e y.
- **`abs (x)`**
  - Calcula el valor absoluto de x
- **`distance (p0, p1)`**
  - Calcula la distancia entre los puntos **`p0`** y **`p1`**.

Ejemplo

Fichero .material

```
vertex_program exampleVS glsl{
  source exampleVS.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

Efecto zoom

- 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]x[-0.5, 0.5] (Traslación)
  - Aplicarles el factor dado `ZF` (Escala)
  - Centrarlas de nuevo en el cuadrado [0, 1]x[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;
```

Efecto zoom

- 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 ...
```

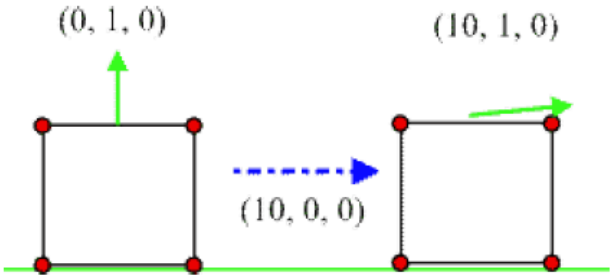
- Si lo hacemos así, el factor de escalado varía dentro del siguiente rango

  sintime ϵ [-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 = sintime * a + b, sintime ϵ [-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:
  - sintime=-1 → ZF=x
  - sintime=1 → ZF=1

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
```

$$atMat = \left(\frac{M}{0}\middle|\frac{T}{1}\right)$$

M: rotación y escala

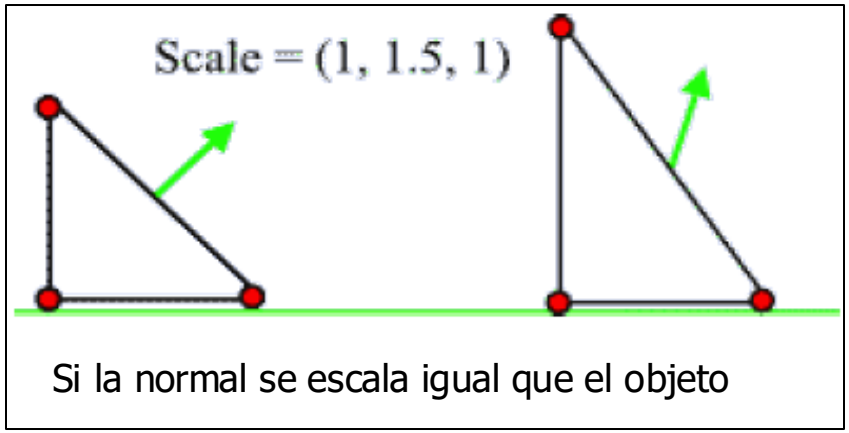
T: traslación

```
vec3 nt = vec3(atMat * vec4(normal, 0.0));
vec3 nt = mat3(atMat) * normal;
```

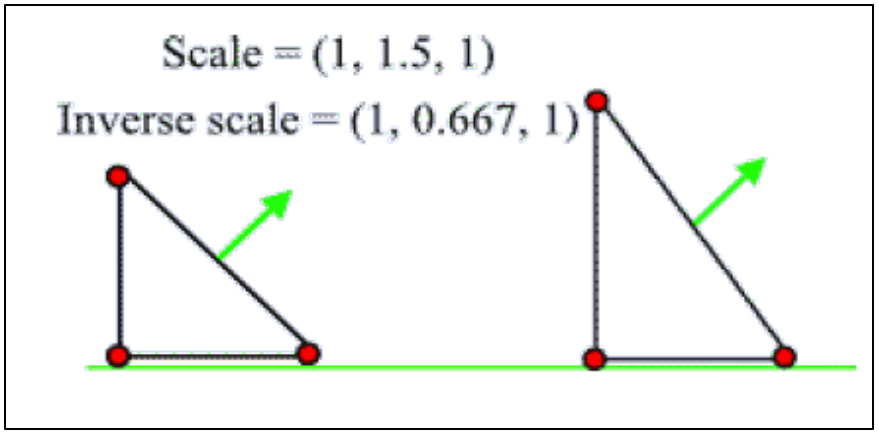
nt no se ve afectado por la traslación (T) de la matriz

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.



Si la normal se escala igual que el objeto



- 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.

$$atMat = \left(\frac{M}{0}\middle|\frac{T}{1}\right)$$

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)));
```

Iluminación

- Material (coeficientes de reflexión)

```
vec3  Diffuse (Ambient)
vec3  Specular
float Shininess
```

- Luces
  - **Posición**: light_position_(world | view | object)_space lightIndex `vec3 Ambient`
  - **Dirección**: light_direction_(world | view | object)_space lightIndex
  - **Color**: light_(diffuse | specular)_colour lightIndex
  - **Color de la luz en la escena global**: ambient_light_colour

Detalles en Tema 9 (slide 28)

- Punto a iluminar → `En World o View space`

```
vec3 vertex;      → En World o View space
vec3 normal;      → En World o View space
```

- Cámara

```
vec3 eye;         → En World o View space
```

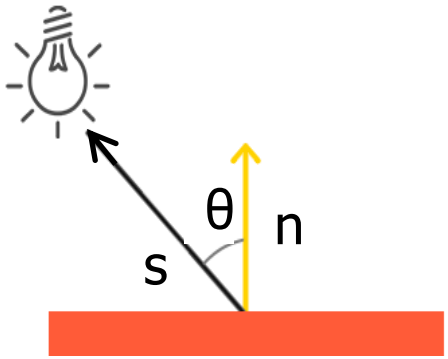
Importante: todas en el mismo sistema de coordenadas

Componente difusa: Ley de Lambert

- La ley de Lambert dice que la *componente difusa* que refleja una superficie ideal es proporcional al **coseno** del ángulo entre la **normal** de la superficie y la **dirección** hacia la luz.

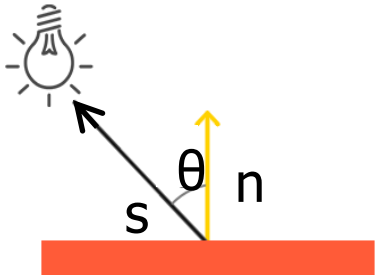
$$I = colorBase \cdot colorLuz \cdot \max(N \cdot L, 0);$$

- colorBase: Representa qué fracción de la luz refleja el material por el canal (R,G,B)
- colorLuz: Propiedades de la luz (color/brillo)
- N: normal de la superficie. Debe ser un vector unitario.
- L: Dirección hacia la luz desde el punto, unitaria y en el mismo espacio que N.



Componente difusa: Ley de Lambert

- Se aplica un factor de reducción (**diff**) en función del coseno del ángulo **θ** que forman los vectores **n** (vector normal) y **s** (dirección opuesta de la luz).
- Los vectores deben ser unitarios
  - Un vector $v=(v_x, v_y, v_z)$ es unitario si $|v| = 1$
  - Es decir, sqrt $((v_x)^2 + (v_y)^2 + (v_z)^2) = 1$
  - Si no lo es, se puede normalizar
- La dirección de la luz en Ogre apunta "en la dirección en la que brilla la luz"
  - Para aplicar Lambert, hay que negar el **vector de dirección**



- El vector normal de la cara *Back* es el opuesto al de la cara *Front* (n):

```
BackFaceNormal= -FrontFaceNormal = -n
```

```
cos(θ) = dot(n, s); // Vectores unitarios

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

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

Iluminación utilizando Lambert con shaders (VS)

- Parámetros del shader de vértices:

```
in vec4 vertex;                         // Posición del vértice
in vec3 normal;                         // Normal
in vec2 uv0;                            // Coordenadas de textura 0

uniform mat4 modelViewMat;              // View*Model matrix (espacio de la vista)
uniform mat4 modelViewProjMat;          // Projection*View*Model matrix  (espacio clip)
uniform mat4 normalMat;                 // upper-left 3x3 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!

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)
```

Iluminación utilizando Lambert con shaders (VS)

- Parámetros del shader de fragmentos:

```
in vec3 vFrontColor;                    // Color de la iluminación
in vec3 vBackColor;                     // Color de la iluminación

in vec2 vUv0;                           // Coordenadas de textura

uniform sampler2D materialTex;          // Textura


out vec4 fFragColor;                    // color del fragmento
```

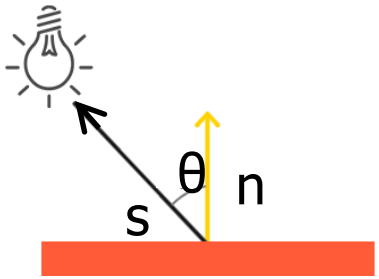
Vertex Shader : Luz difusa

- 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){
   // Posición de la luz (si es direccional, dirección)
   vec3 lightDir = lightPosition.xyz;
   // Si no es direccional, calculamos la dirección con el vértice (hacia la luz)
   if (lightPosition.w == 1)
        lightDir = lightPosition.xyz - cVertex;

   return max(dot(cNormal, normalize(lightDir)), 0.0); // dot: coseno ángulo
}
```



Vertex Shader : Luz difusa

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

```
void main() {
   vec3 ambient = lightAmbient * materialDiffuse;

   // Diffuse en view space (front)
   vec3 viewVertex = vec3(modelViewMat * vertex);
   vec3 viewNormal = normalize(normalMat * normal);
   vec3 diffuse = diff(viewVertex, viewNormal) * lightDiffuse * materialDiffuse;
   vFrontColor = ambient + diffuse;

   // Diffuse en view space (back)
   diffuse = diff(viewVertex, -viewNormal) * lightDiffuse * materialDiffuse;
   vBackColor = ambient + diffuse;

   vUv0 = uv0;
   gl_Position = modelViewProjMat * vertex;
}
```

Fragment Shader : Luz difusa

- Iluminación y textura (front & back faces)

```
#version 330 core

in vec3 vFrontColor;                    // Color de la iluminación (cara front)
in vec3 vBackColor;                     // Color de la iluminación (cara back)
in vec2 vUv0;                           // Coordenadas de textura

uniform sampler2D materialTex;          // Textura

out vec4 fFragColor;


void main() {

 vec3 color = texture(materialTex, vUv0).rgb;

     if (color.r > 0.6)
        discard;

    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 del mundo 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?
```

Iluminación

- 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)