Graphics Shaders

GLSL (OpenGL Shading Language)

https://www.khronos.org/opengl/

Ana Gil Luezas
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Graphics Hardware APIs

- Direct3D: HLSL (High Level Shading Language)
- OpenGL: GLSL (OpenGL Shading Language).

  Multiplataforma (CPU y GPU)

  OpenGL (CPU): especificación de una API para gestionar, desde la aplicación, la pipeline gráfica (GPU: máquina de estados de OpenGL)

  GLSL: lenguaje de programación de alto nivel (a la C) para programar ciertas etapas de la tubería de renderizado (máquina de estados de OpenGL)

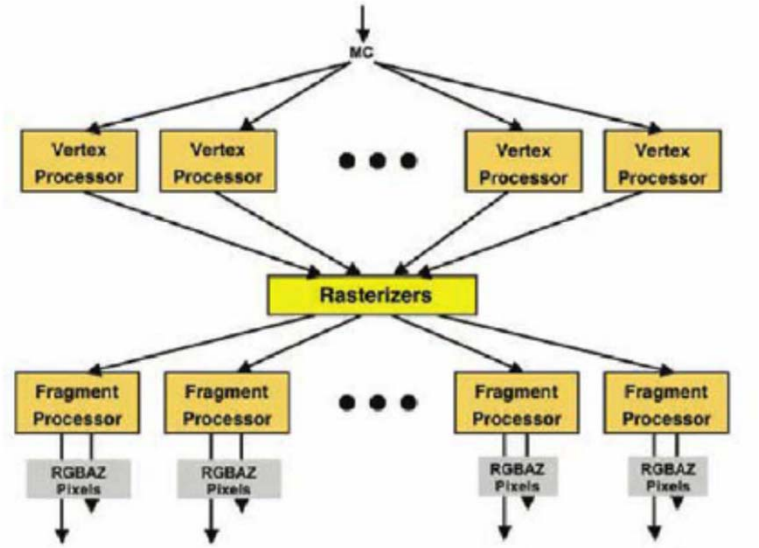
OpenGL v.	GLSL v.	Date
1.0	---	1992
…	…	…
1.5	---	2003
2.0	1.10	2004
2.1	1.20	2006
3.0	1.30	2008
3.1	1.40	2009
3.2	1.50	2009
3.3	3.30	2010
4.0	4.00	2010
…	…	…
4.6	4.60	2017

khronos.org/opengl/wiki/History_of_OpenGL

OpenGL Graphics Pipeline

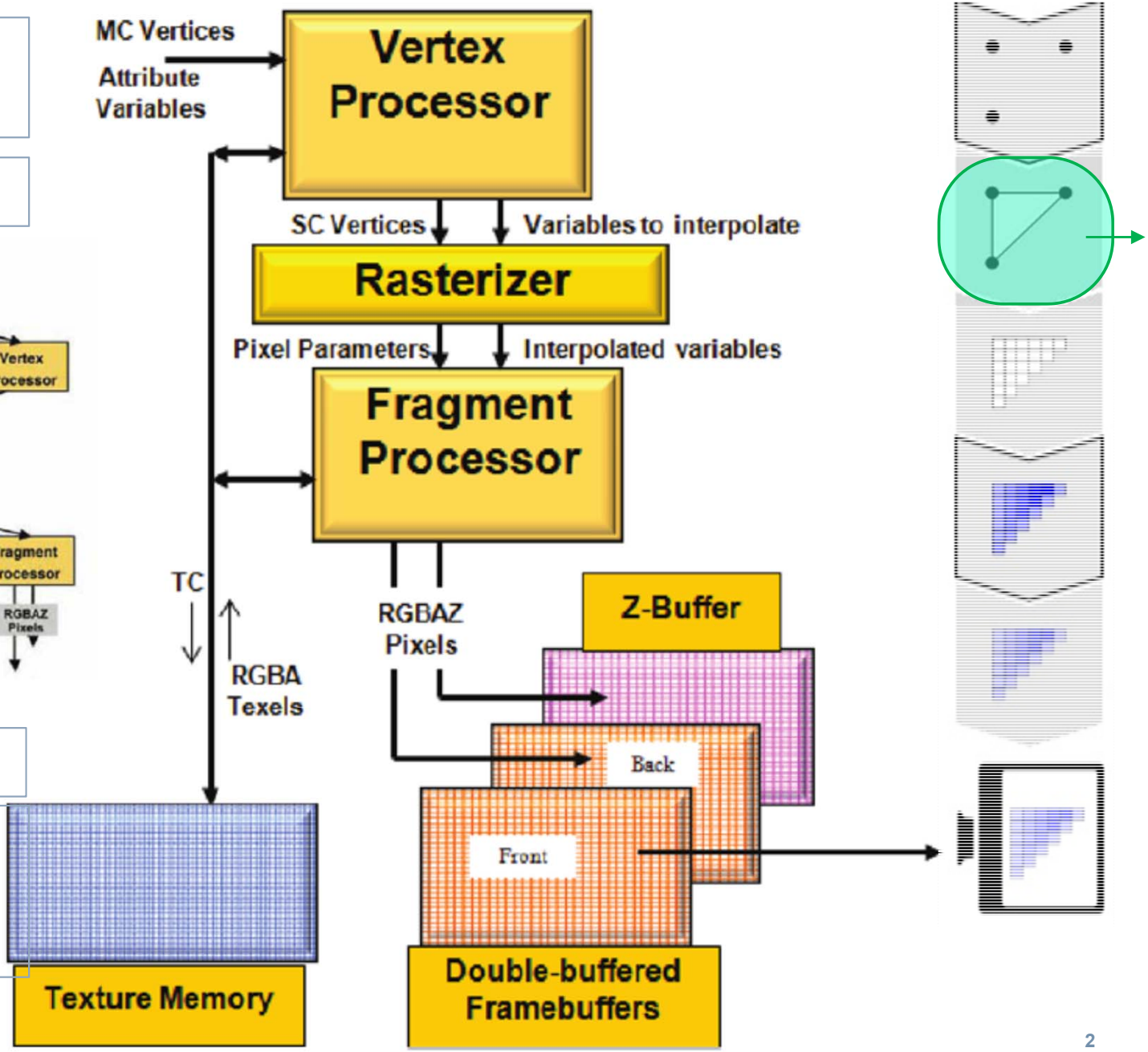
MC: Model Coordinates (mesh)

SC: Screen Coordinates

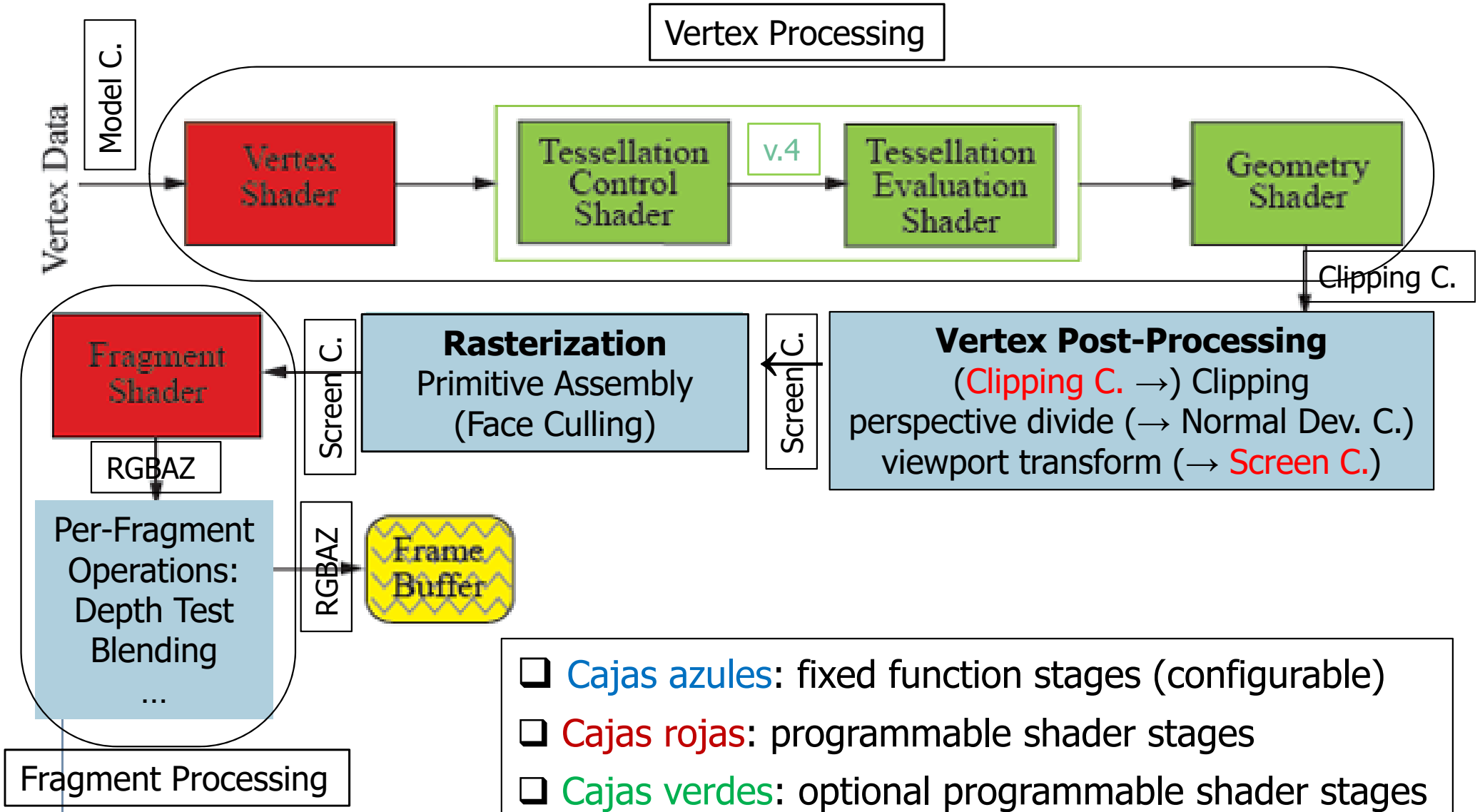


TC: Texture Coordinates

Texture units: filtering and tex_address_mode (OGRE: sampling state)



OpenGL Programmable Rendering Pipeline



- ❑ Cajas azules: fixed function stages (configurable)
- ❑ Cajas rojas: programmable shader stages
- ❑ Cajas verdes: optional programmable shader stages

Vertex and Fragment Shader example

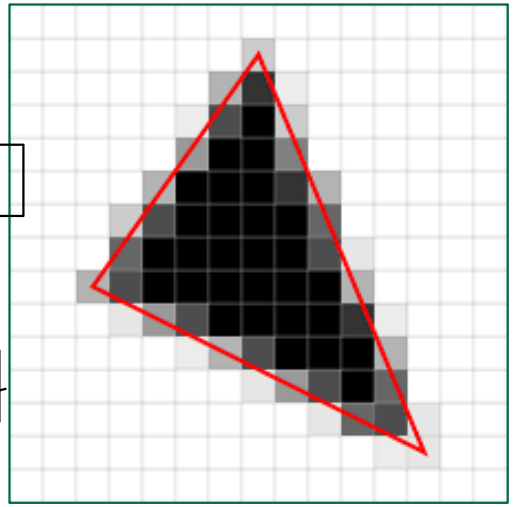
- Vertex Shader Source (archivo Ejemplo1VS.glsl)

```
#version 330 core

in vec4 vertex;

uniform mat4 modelViewProjectionMatrix;


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

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

Model C.

Clipping C.

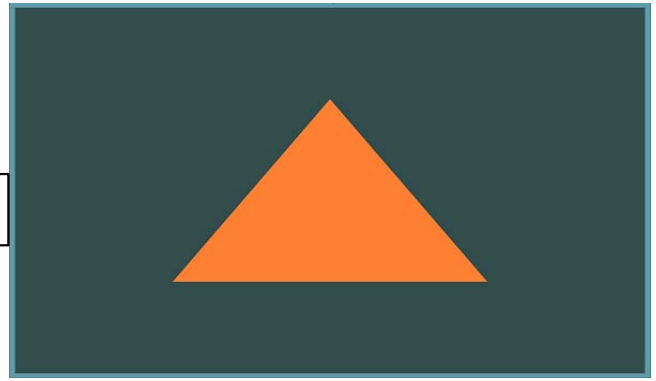


- Fragment Shader Source (archivo Ejemplo1FS.glsl)

```
#version 330 core

out vec4 fFragColor;


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

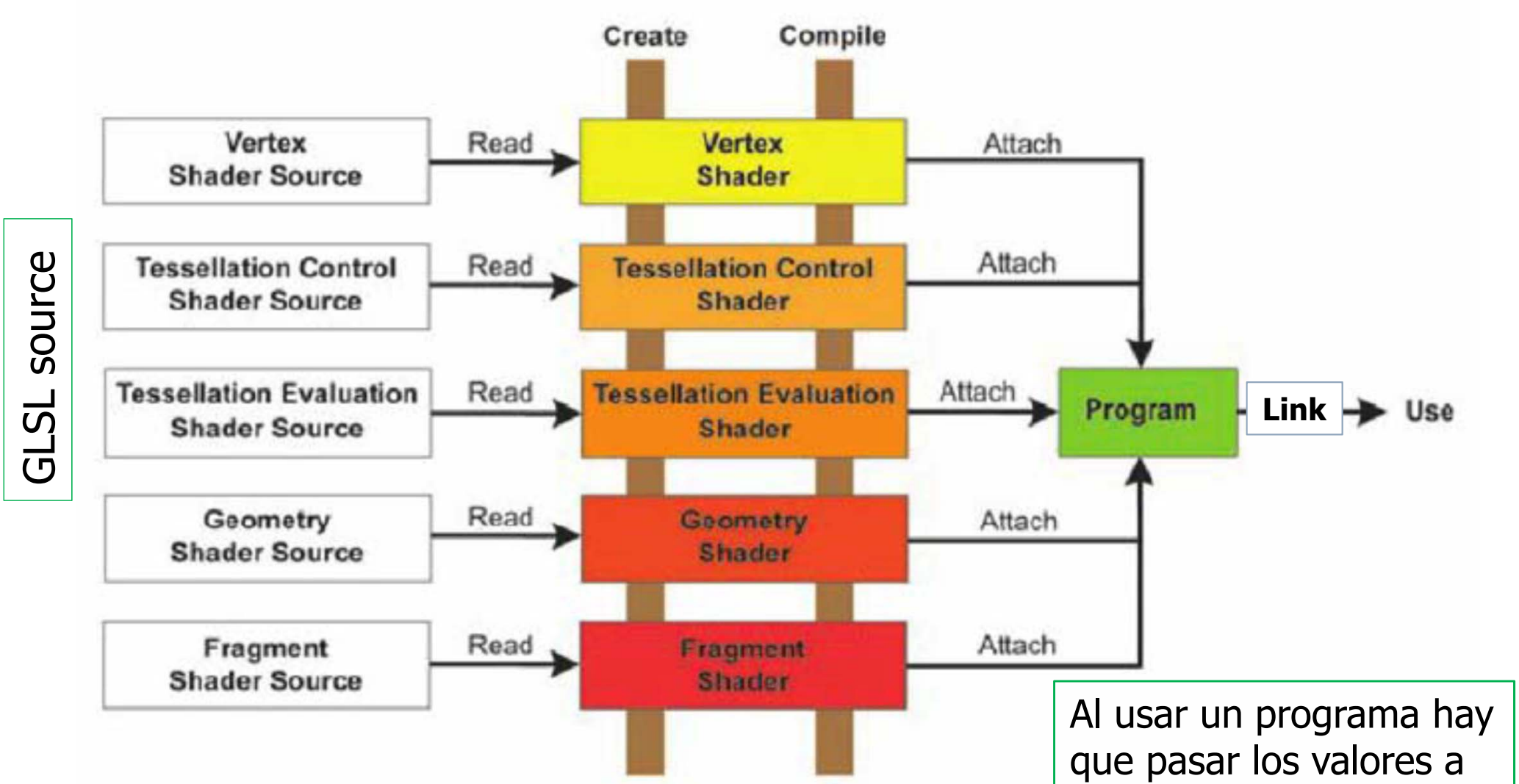
Screen C.

RGBAZ



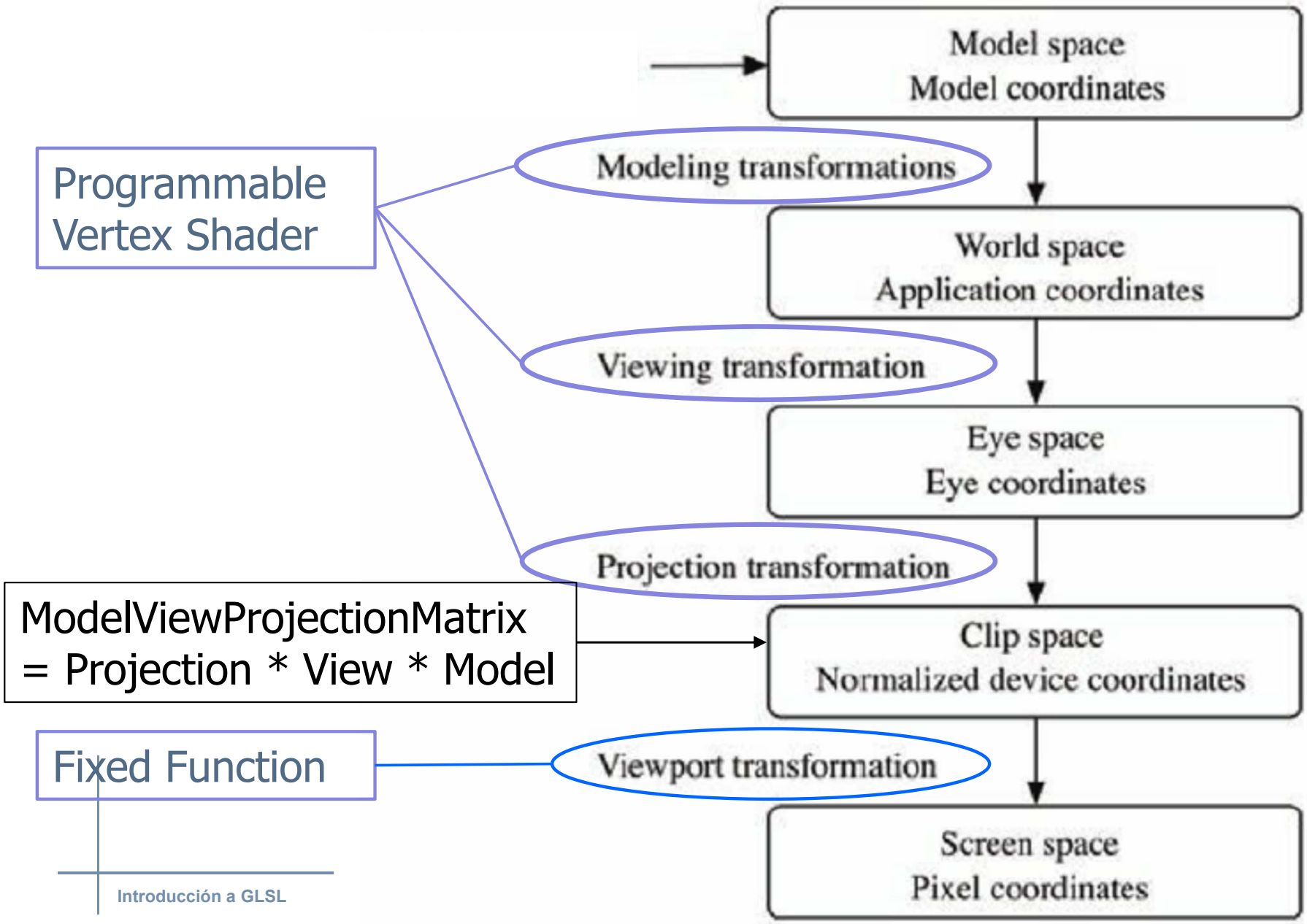
OpenGL Programmable Graphics Pipeline

CPU: comandos OpenGL para instalar el programa en GPU



Al usar un programa hay que pasar los valores a las uniform, y la malla

OpenGL Vertex Transformations

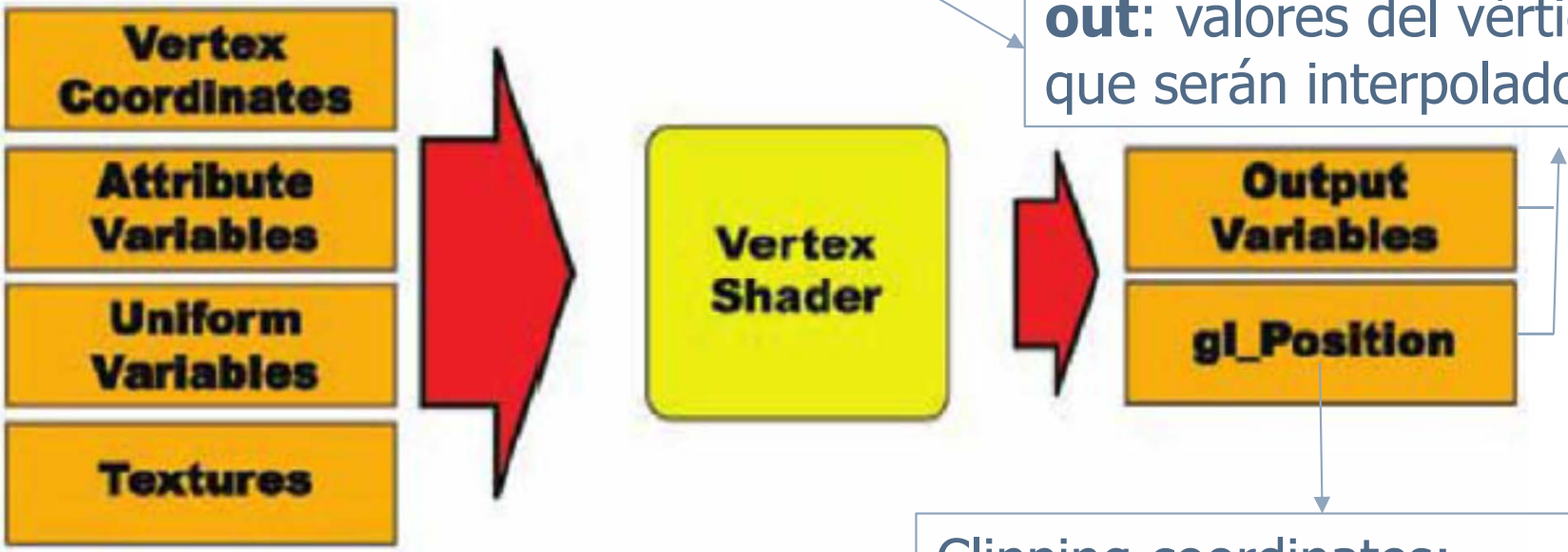


OpenGL Vertex Shader

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

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

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:

cv=Projection*View*Model * v

View-space: vv = View*Model * v

World-space: wv = Model * v

Vertex Shader variables

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

```
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;
```

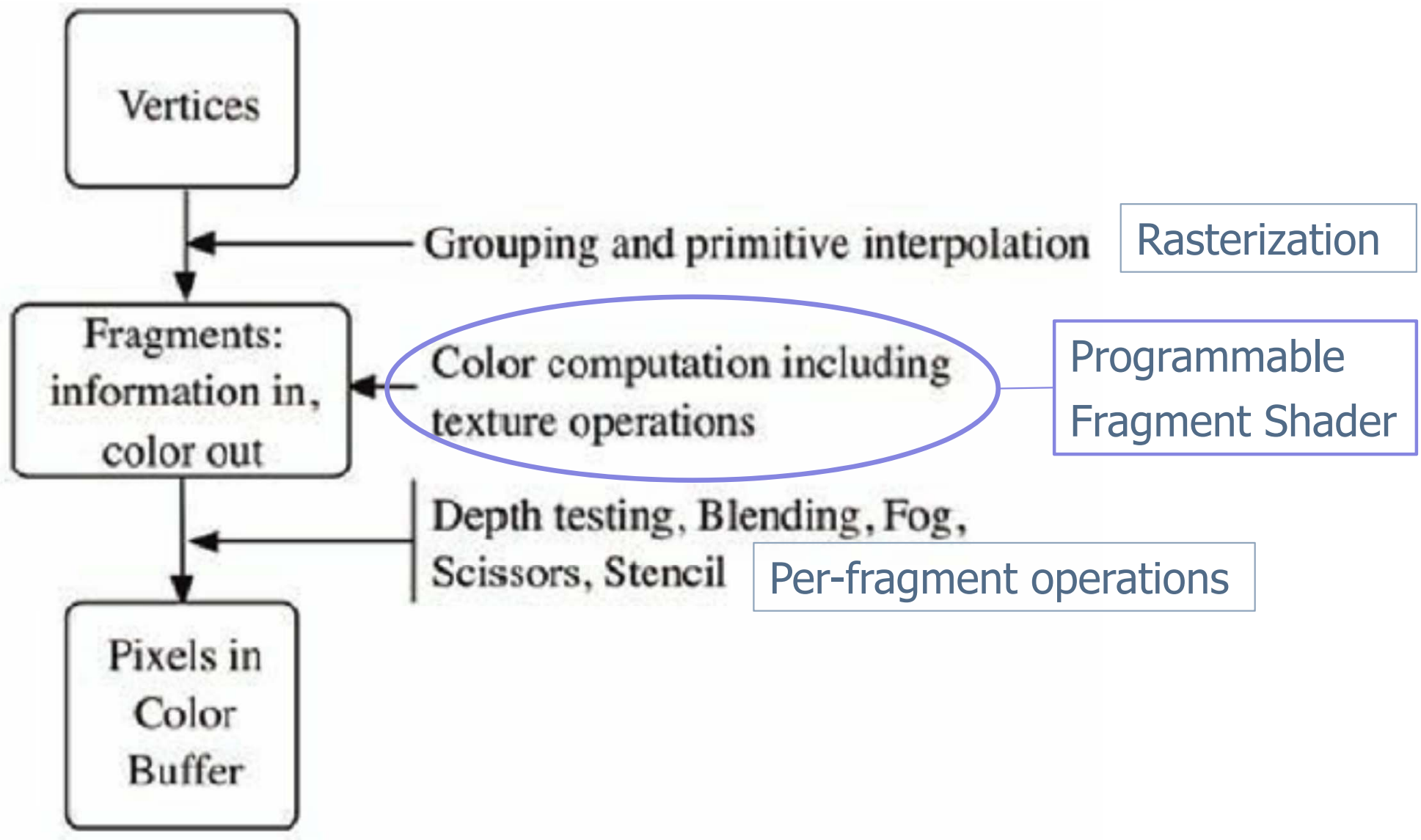
- Texturas: uniform sampler2D nombTex

Vertex Shader example

- 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

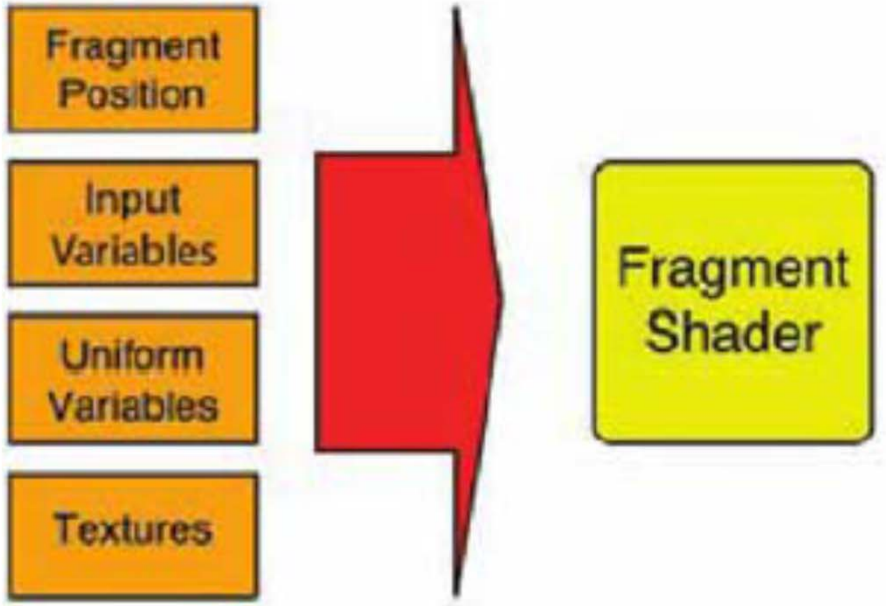
```
// archivo Ejemplo2VS.glsl
#version 330 core
in vec4 vertex;  // atributos de los vértices a procesar
in vec2 uv0;     // coordenadas de textura 0
uniform mat4 modelViewProjMat; // constante de programa
out vec2 vUv0;      // out del vertex shader

void main() {
  vUv0 = uv0; // se pasan las coordenadas de textura
  gl_Position = modelViewProjMat * vertex; //obligatorio
predefined: out vec4                       // (Clipping coordinates)
}
```

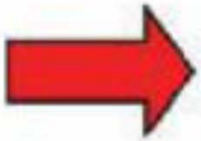
OpenGL Fragment Shader



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

Fragment Shader variables

- Valores de entrada (per fragment -> interpolados): in

```
// 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 example

- GLSL Fragment Shader: mezcla de dos texturas
  Cada ejecución procesa 1 fragmento

```
      // archivo Ejemplo2FS.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
}
```

Texture units

Los parámetros uniform para las texturas son del tipo samplerXD:

```
uniform sampler2D texName;
```

Los valores que pueden tomar son `int`, y representan la unidad de textura que se va a utilizar con la función GLSL predefinida

```
texture(texName, texCoord);
```

Esta función se configura especificando la forma de obtener el téxel:

- En Ogre (en el script del material):
  - `tex_address_mode`: wrap (repeat), clamp, mirror
  - `filtering`: nearest, linear, bilinear, none
- En OpenGL:

```
glTexParameteri(filter / wrap...)
```

Los valores para las constantes uniform se transfieren a la GPU cada vez que se va a usar un programa:

- En Ogre podemos hacerlo en el script del material

```
param_named nombreUniform tipo valor
param_named_auto nombreUniform nombreOgre
```

Por ejemplo:

```
param_named texturaL int 0 // unidad de textura 0
param_named_auto modelViewProjMat worldviewproj_matrix
```

- En OpenGL con comandos específicos:

```
glUseProgram(…) // para indicar el programa
glGetUniformLocation(…) // para localizar la variable
glUniform(…) // para transferir el valor a la variable
```

Para configurar opciones de la parte no programable:

- En Ogre en el script del material. Por ejemplo:

```
cull_hardware none
depth_check off
depth_write off
tex_address_mode clamp
filtering none
```

- En OpenGL con el comando glEnable(...) y funciones específicas. Por ejemplo:

```
glEnable(GL_CULL_FACE); glCullFace(GL_FRONT);
glEnable(GL_DEPTH_TEST); glDepthFunc(GL_ALWAYS);
glDepthMask(GL_FALSE); glTexParameteri(filter / wrap...)
```

OGRE: GPU Programs

- OGRE realiza todas las tareas referentes a la compilación, enlace y carga de los shaders que vayamos a utilizar para renderizar objetos de la escena (establecidos en el material del objeto).
- El gestor de recursos se encarga de analizar los archivos con el código fuente de los shaders.
- También nos permite establecer valores por defecto para las constantes uniform.
- En el script del material se especifican el shader de vértices, el shader de fragmentos, y los valores iniciales para las uniform.

  Cada vez que se ejecute un GPU-program se pasarán a la memoria de la GPU los valores especificados actualizados.
- Ejemplo: archivo PracticaGLSL.material ->

OGRE: GPU Programs

- Material `// archivo PracticaGLSL.material`

```
vertex_program Ejemplo2VS glsl //nombre para el shader
{
  source Ejemplo2VS.glsl // nombre del archivo del código
  default_params  // valores para las variable uniform
  {
   param_named_auto modelViewProjMat worldviewproj_matrix
  }
}
```

nombre dado en el shader

valor automático (Ogre) para la variable

```
                                                    // ->
```

https://ogrecave.github.io/ogre/api/latest/_high-level-_programs.html#Program-Parameter-Specification

OGRE: GPU Programs

```
// ->  // archivo PracticaGLSL.material

fragment_program Ejemplo2FS glsl //nombre para el shader
{
   source Ejemplo2FS.glsl  // nombre del archivo del código
   default_params  // valores para las variable uniform
   {
     param_named texturaL int 0 // 1º unidad de textura -> *
     param_named texturaM int 1 // 2º unidad de textura -> *
     param_named          BF float 0.5
     param_named intLuzAmb float 1.0
   }
}
```

nombre dado en el shader

tipo y valor para la variable

// ->

OGRE: GPU Programs

```
material IG2/ejemplo2GLSL  {
   technique  {
      pass  {
         vertex_program_ref Ejemplo2VS
         {  // params -> default_params
         }
         fragment_program_ref Ejemplo2FS
         {  // params -> default_params
         }
         texture_unit { // * -> int 0
            texture ejemploA.jpg 2d  // archivo
            tex_address_mode clamp   // sólo configuración
            filtering bilinear       // de acceso al texel
         }
         texture_unit {  // * -> int 1
            texture ejemploB.jpg 2d // archivo
            tex_address_mode wrap   // sólo configuración
         }                          // de acceso al texel
      }  }  }
```

OGRE: GPU Programs

- Podemos utilizar `time` para los valores `uniform`

```
param_named_auto tiempo time  // current elapsed time
param_named_auto tiempo1 time_0_1 1 // valores float en el intervalo [0..1] que se repiten cada 1 segundo
param_named_auto senotiempo sintime_0_2pi 60 // valores en el intervalo [sin(0)..sin(2pi)] que se repiten cada 60 segundos
```

- En el ejemplo anterior podemos modificar

```
param_named BF float 0.5
```

por

```
param_named_auto BF time_0_1 10
```

Fragment Shader: predefined variables

- `bool gl_FrontFacing` // true -> el fragmento corresponde a la cara front

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

  En Ogre puede ser necesario ajustar la variable preguntando si ha invertido el orden de los vértices (https://forums.ogre3d.org/viewtopic.php?t=47266):

  Añade el parámetro uniform float Flipping; // -1 -> está invertido, 1 -> no

  Utilízalo para definir la variable

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

  Y utiliza frontFacing en lugar de gl_FrontFacing en los condicionales.

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

  Para pasar el valor al nuevo parámetro:

  param_named_auto Flipping render_target_flipping // -1 o 1

- `vec4 gl_FragCoord` coordenadas del fragmento en Screen space (origin at lower-left origin)

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

- `discard` -> El fragmento queda descartado y no seguirá el proceso de renderizado (efecto return)

Iluminación

- Material (coeficientes de reflexión)

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

- Light(s)

```
vec3 Position / Direction   -> en Word 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

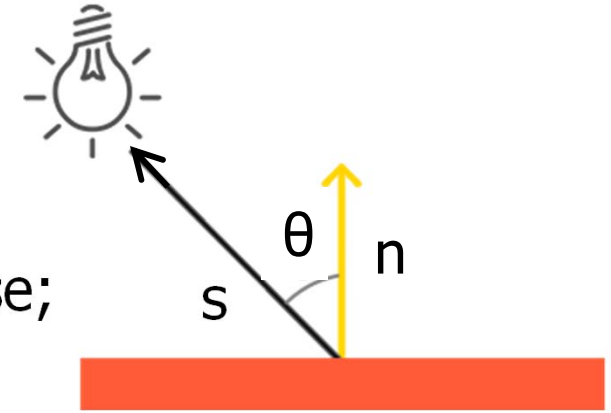
- Reflexión difusa (color)

  Se aplica un factor de reducción (diff) en función del coseno del ángulo **θ** que forman los vectores **n** (vector normal) y **s**.

  cos(θ) = dot(n, s) para vectores de magnitud 1

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

  vec3 diffuse = diff * LightDiffuse * MaterialDiffuse;

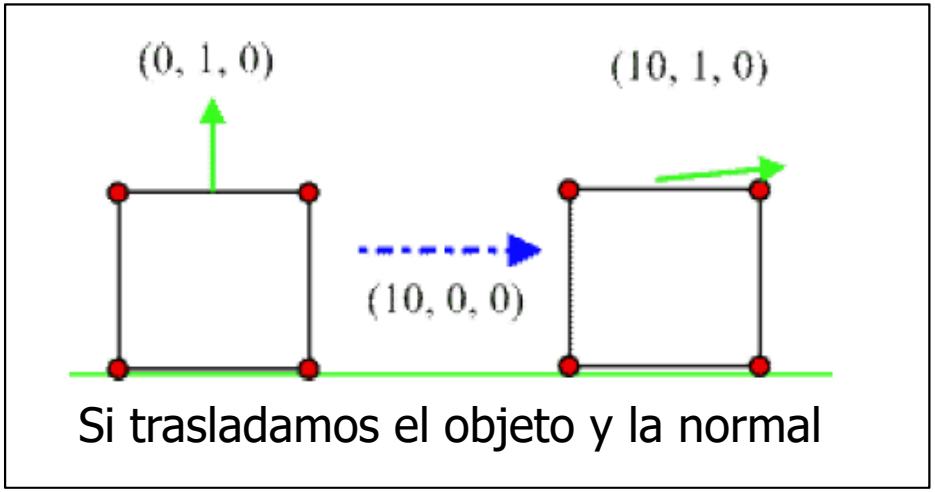


- El vector normal de la cara Back es el opuesto al de la cara Front (n):

  BackFaceNormal= - FrontFaceNormal = - 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, es decir, utilizando la misma rotación.

```
vec3 normal;
mat4 atMat;   // matriz afín   ->
. vec3 nt =  vec3(atMat * vec4(normal, 0.0));  // nt  no se ve afectado por
. vec3 nt = mat3(atMat) * normal ;              // la traslación (T) de la matriz
```

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

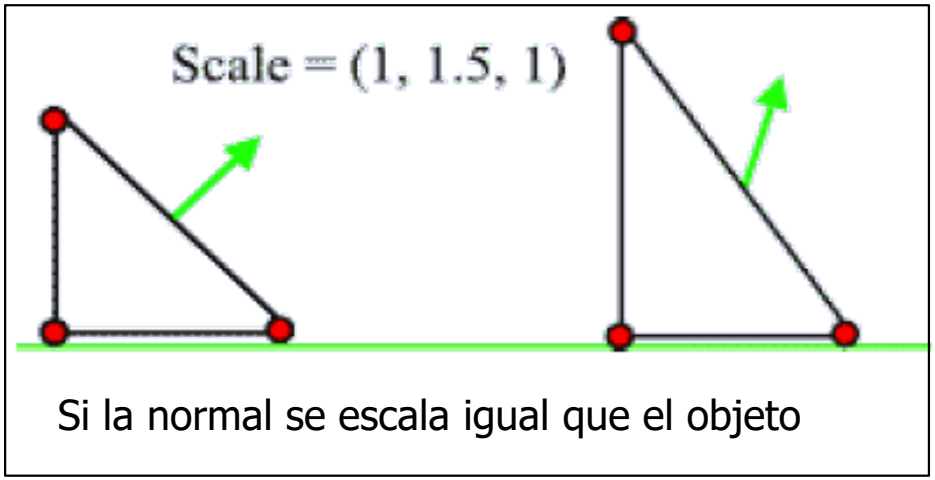
M: rotación y escala

T: traslación

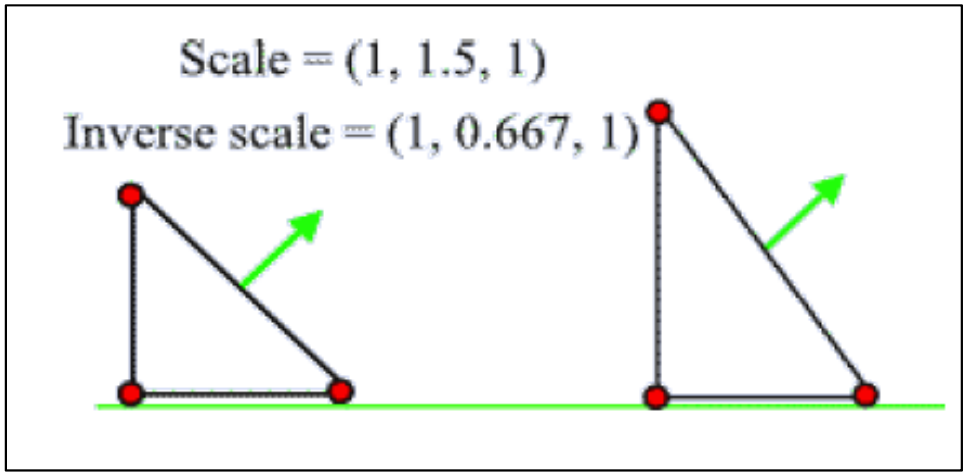
Transformaciones de los vectores normales

- 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 (`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 or modelview matrix
vec3 nt =  mat3(transpose(inverse(atMat)) * normal;
```

Mejor pasar la NormalMatrix:

```
uniform mat4 normalMat;
vec3 nt =  normalize(vec3(normalMat* vec4(normal, 0.0)));

// Mejor:
// uniform mat3 normalMat;
// vec3 nt =  normalize(normalMat * normal);
// pero en la versión de Ogre que usamos no existe
```

Vertex Shader example: diffuse lighting

- GLSL Vertex Shader: iluminación rgb difusa en view space (front & back faces)

```
in vec4 vertex; // datos de la malla
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 example: diffuse lighting

- GLSL Vertex Shader: iluminación rgb difusa en view space (front & back faces)

```
#version 330 core
                // datos de la malla: indicar cómo está estructurada
in vec4 vertex;   // layout (location = 0) in vec4 vertex;
in vec3 normal;   // layout (location = 1) in vec3 normal;
in vec2 uv0;      // layout (location = 2) in vec2 uv0;

out vec2 vUv0;          // coordenadas de textura
out vec3 vFrontColor;   // color rgb de la iluminación de la cara front
out vec3 vBackColor;    // color rgb de la iluminación  de la cara back
```

layout ...
lo pone Ogre

Vertex Shader example: diffuse lighting

- GLSL Vertex Shader: 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))

// struct Light { vec3 position; // datos de la fuente de luz
//                vec3 ambient, diffuse, specular; };
// uniform Light light;
```

GLSL permite structs, pero en Ogre da problemas: light.ambient -> lightAmbient

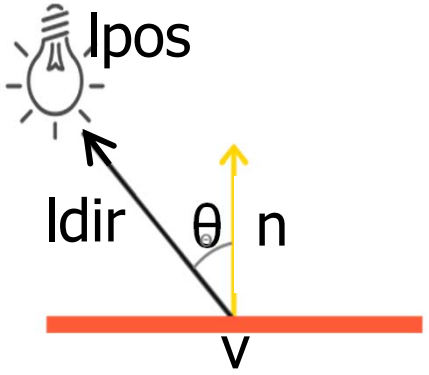
```
uniform vec3 lightAmbient;   // intensidades de la luz
uniform vec3 lightDiffuse;
uniform vec4 lightPosition;       // datos de la fuente de luz en view space
                // lightPosition.w == 0 -> directional light
                // lightPosition.w == 1 -> positional  light
uniform vec3 materialDiffuse;   // datos del material ¡Front=Back!
```

Vertex Shader example: diffuse lighting

- GLSL Vertex Shader: 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
}
```



Vertex Shader example: diffuse lighting

- GLSL Vertex Shader: iluminación rgb difusa en view space (front&back faces)

```
void main() {
   // ambient
   vec3 ambient = lightAmbient * materialDiffuse;
   // 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 example: diffuse lighting

- GLSL Fragment Shader: 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;        // coordenadas 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 …

Ogre: paso de parámetros uniform

- Valores Ogre para los parámetros con param_named_auto:

  Cuando el valor del parámetro va cambiando durante la ejecución (por ejemplo: la posición de la cámara, la matriz de proyección, …), Ogre se encarga de pasar en cada momento el valor actualizado.

  https://ogrecave.github.io/ogre/api/latest/class_ogre_1_1_gpu_program_parameters.html#a155c886f15e0c10d2c33c224f0d43ce3

  Ejemplos: param_named_auto `nombreUniform` nombreOgre

  param_named_auto … world_matrix `// matriz de modelado`

  param_named_auto … worldview_matrix `// matriz de modelado y vista`

  param_named_auto … inverse_transpose_world_matrix `// matriz de modelado para vectores`

  param_named_auto … inverse_transpose_worldview_matrix `// matriz de modelado y vista para vectores`

Ogre: paso de parámetros uniform

- Valores Ogre para los parámetros con param_named_auto:

  Ejemplos: param_named_auto `nombreUniform` nombreOgre

  param_named_auto … light_position 0 `// dirección/posición en coordenadas mundiales (word space) de la luz 0`

  param_named_auto … light_position_view_space 0 `// dirección/posición en coordenadas de la cámara (view space) de la luz 0`

  param_named_auto … light_diffuse_colour 0 `// intensidad de la componente difusa de la luz 0`

  param_named_auto … camera_position `// en world space`

- También podemos dar valor a los parámetros con param_named: por ejemplo, para uniform vec3 `materialDiffuse` (vec3 -> float3):

  param_named `materialDiffuse` float3 0.5 0.5 0.5

  En este caso, también se puede indicar el valor diffuse del material mediante:

  param_named_auto `materialDiffuse` surface_diffuse_color // del script

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; // datos de la malla
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)

- GLSL C-like language
  https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language
  - Restringido en algunos aspectos (char, punteros, recursión) y aumentado para el uso de gráficos (vectores, matrices)
  - Un shader empieza con una declaración de versión, seguida por la declaración de variables in/out y uniforms, y termina con la función main().
  - Se pueden definir funciones. Los parámetros se declaran in/out.
  - Vertex shader: las variables in se corresponden con los atributos de los vértices de la malla, y las variables out con los valores interpolables (varying). Las in no se pueden modificar.
  - Las variables se declaran especificando su tipo
  - Fragment Shader: las variables in se tienen que corresponder con variables out del vertex Shader. Hay que dar valor a las out.
  - En el fragment shader se puede desechar un píxel con discard

GLSL

- Tipos de datos
  - Básicos: float, int, bool, double, uint
  - Vectores(XvecN): vecN para float, ivecN para int, bvecN para bool, …

    Se accede a los campos con .x, .y, .z, .w, .xy, .xyz, .s, .t, .rbg, …
  - Matrices: matN para matrices de N (2,3,4) float
  - Samplers: samplerND, sampler2D (para texturas 2d)
  - Registros: struct, se accede a los campos con punto

    ```
    struct vertice { vec4 posicion;
                     vec3 color; };
    vertice v;   v.posicion = vec4(0,1.5,0,1);
                 v.color = vec3(1,1,1);      v.posicion.x = v.posicion.y+1;
    ```
  - Arrays: vertice av[N];

GLSL

- Visibilidad de las variables:
  - Función
  - Shader (de vértices o de fragmentos)
  - Programa: por todos los shader (de vértices y de fragmentos)
- Calificadores de tipo:
  - const
  - in: para atributos de vértice en VBO, o para recoger información entre los shaders.
  - out: para pasar información entre los shaders.
  - uniform: para pasar datos de CPU a GPU al programa.

    Son constantes

    uniform para texturas: sampler2D, sampler3D, sampler1D
  - buffer: para pasar datos de CPU a GPU y viceversa (OpenGL 4.3)

GLSL

- Definición de funciones (NO recursivas)
  - Parámetros: in, out, inout (todos por copia)
- Operadores aritméticos y funciones predefinidas
- Instrucciones de control:
  - if, if-else
  - for, while, do-while
- Variables predefinidas (por shader):
  - Input: gl_FrontFacing, gl_FragCoord (fragment shader)
  - Output: gl_Position, gl_PointSize (vertex shader)
- Consultas:
  - gl_MaxTextureImageUnits

https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language