Informática Gráfica II

Pipeline de renderizado

&

GLSL (OpenGL Shading Language)

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

Graphics Hardware APIs

- Direct3D: HLSL (**H**igh **L**evel **S**hading **L**anguage)
- 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)

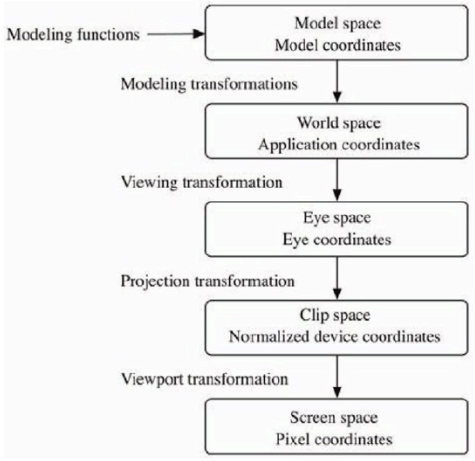
GLSL: Lenguaje de programación de alto nivel (similar a C) para programar ciertas etapas de la pipeline de renderizado (máquina de estados de OpenGL)

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

Fuente: khronos.org/opengl/wiki/History_of_OpenGL

Procesado de vértices en la pipeline de OpenGL

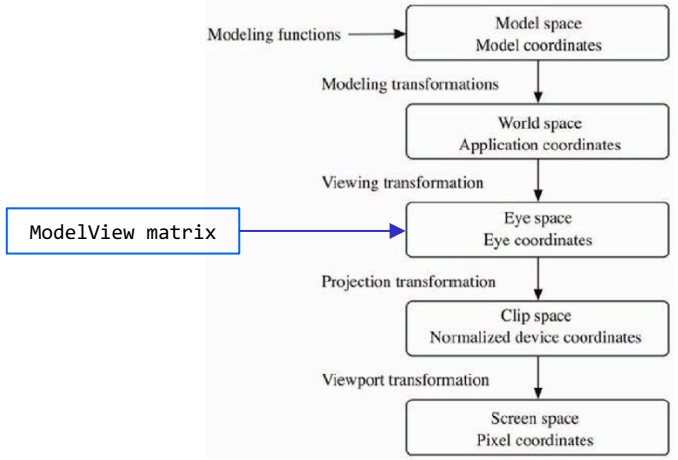
- La parte de geometría relativa al procesamiento de vértices sigue este esquema
- La primera etapa define la geometría fundamental de la escena
  - Cada pieza - de geometría - es creada es su propio espacio
    - Vectores normales
    - Coordenadas de textura
- En la segunda etapa se define el espacio global (world space) de la escena
  - Cada primitiva geométrica se sitúa en el espacio global
    - Transformaciones: Escala, rotación, traslación, . . .
  - La entrada de esta etapa es el conjunto de transformaciones
    - No afecta al color o los materiales
    - Modifica vértices, normales y geometría de la luz
  - La salida es un conjunto de vértices y normales modificado
    - Representa la geometría original en un espacio diferente



Fuente: Graphic Shaders: Theory and Practice

Procesado de vértices en la pipeline de OpenGL

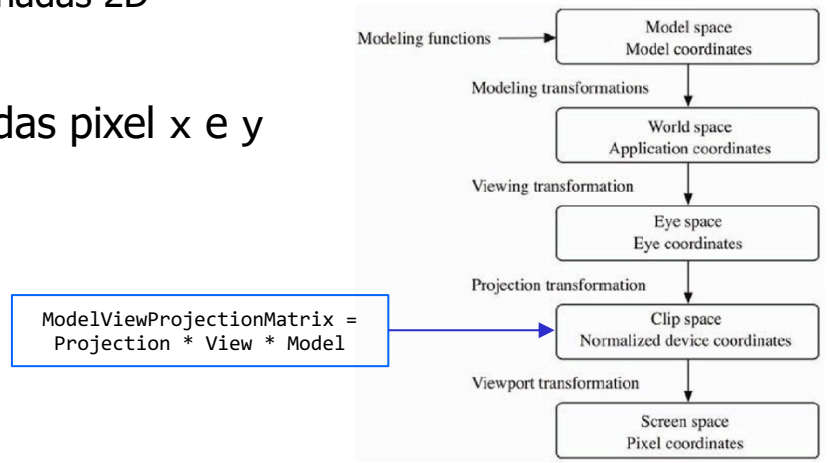
- La tercera etapa representa el espacio del ojo (*eye space*)
  - Se crea cuando se especifica la información de visualización de la escena
  - La entrada es la definición del entorno de visualización
  - Modifica la escena teniendo en cuenta el origen de coordenadas del ojo
  - Modifica vértices, normales y geometría de la luz
  - La matriz `ModelView` se crea en este punto
    - Se utiliza para transformar
      - Vértices
      - Normales
      - Posiciones de la luz
      - Direcciones de la luz
  - Se incluye información de cada vértice
    - Color



Fuente: Graphic Shaders: Theory and Practice

Procesado de vértices en la pipeline de OpenGL

- En la cuarta etapa se define el "clip space"
  - Se especifica la proyección de la escena en un plano de visualización
  - La entrada es la definición de la proyección
  - Se aplica una transformación de la proyección
  - Se crea un espacio 3D rectangular que sirve de entrada para la siguiente etapa
- La quita y última etapa
  - Utiliza la información del puerto de vista definido
  - Crea una representación de *pixels* para cada vértice
  - Dos operaciones
    - Recorte (clipping): se interpolan los pixels de los vértices en los bordes
    - Convertir el espacio 3D en coordenadas 2D
- La salida final:
  - Conjunto de vértices en coordenadas pixel x e y
    - Agrupamiento
    - Normales
    - Profundidad
    - Coordenadas de textura
    - color

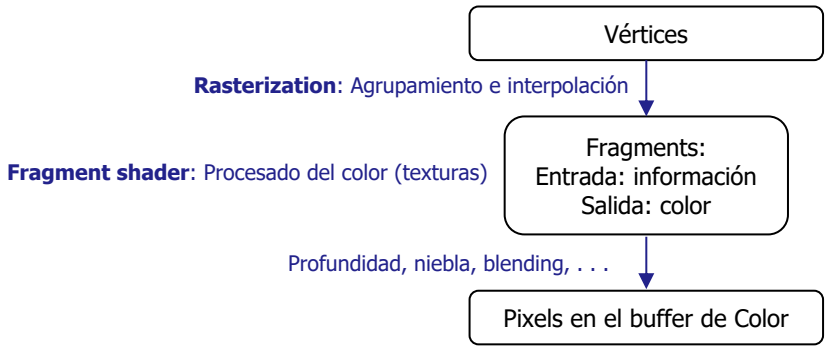



```
ModelViewProjectionMatrix =
Projection * View * Model
```

Fuente: Graphic Shaders: Theory and Practice

Pipeline de renderizado de OpenGL

- Pixel (en terminología GLSL)
  - Información sobre el aspecto (R,G,B,A,Z) que se va a escribir en el *framebuffer*
- Fragmento
  - *Futuro* pixel sobre el que aún no se ha procesado su información
- El esquema muestra una vista simplificada del "OpenGL rendering pipeline"
  - Aquí no se tienen en cuenta las normales
  - La luz se procesa por vértice
- En la segunda etapa, se procesa la pre-información y se genera el pixel
- La salida ya contine el color del pixel tal y como se almacena en el *framebuffer*

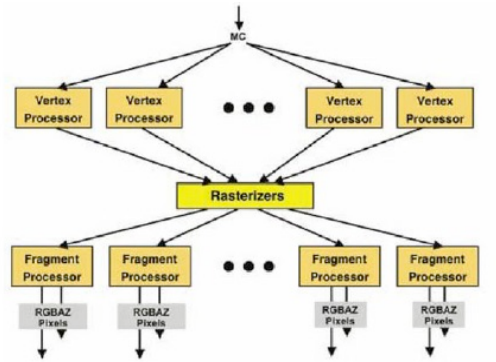


Fuente: Graphic Shaders: Theory and Practice

Pipeline de OpenGL en hardware gráfico

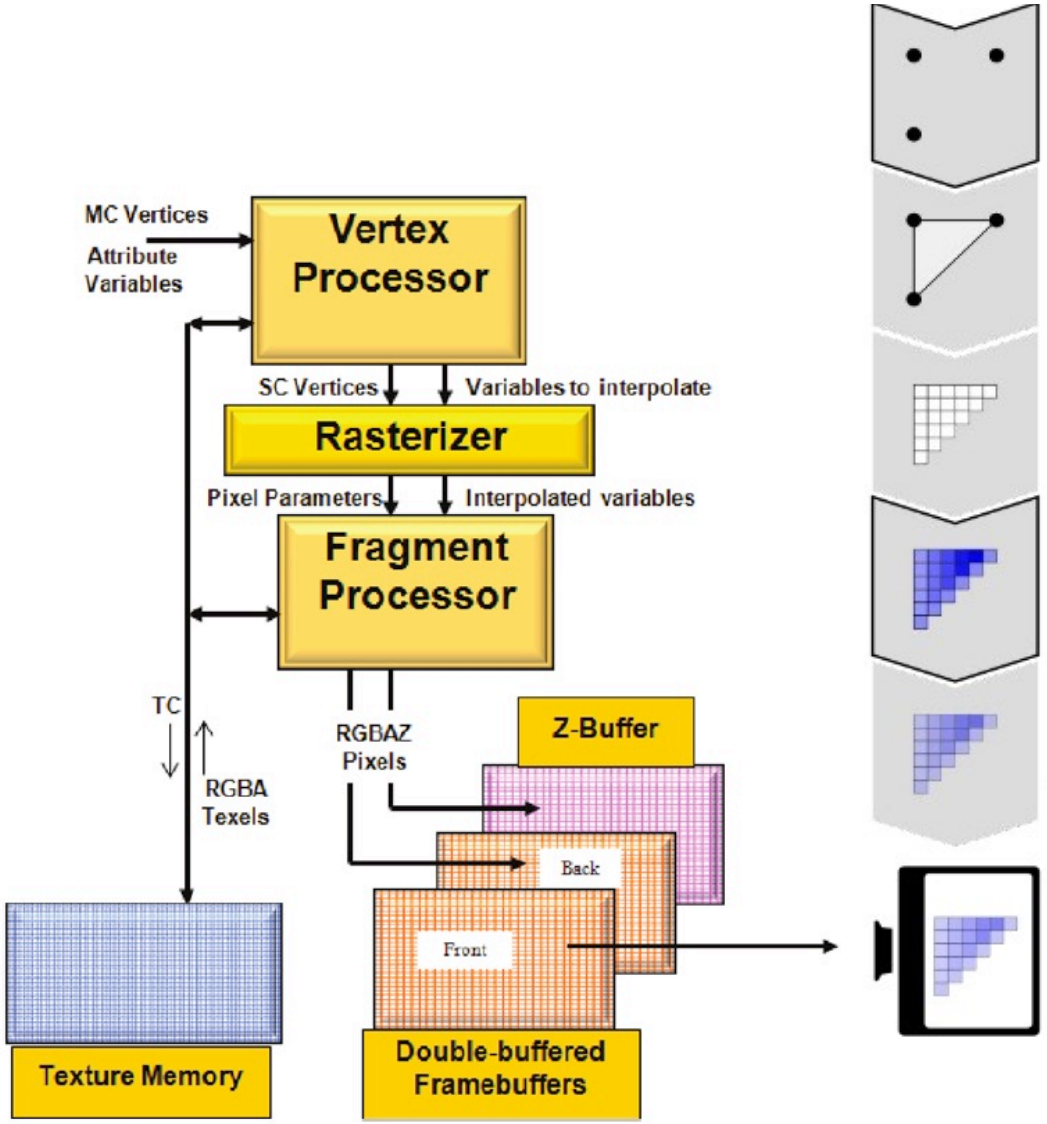
MC: Model Coordinates (mesh)

SC: Screen Coordinates



TC: Texture Coordinates

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



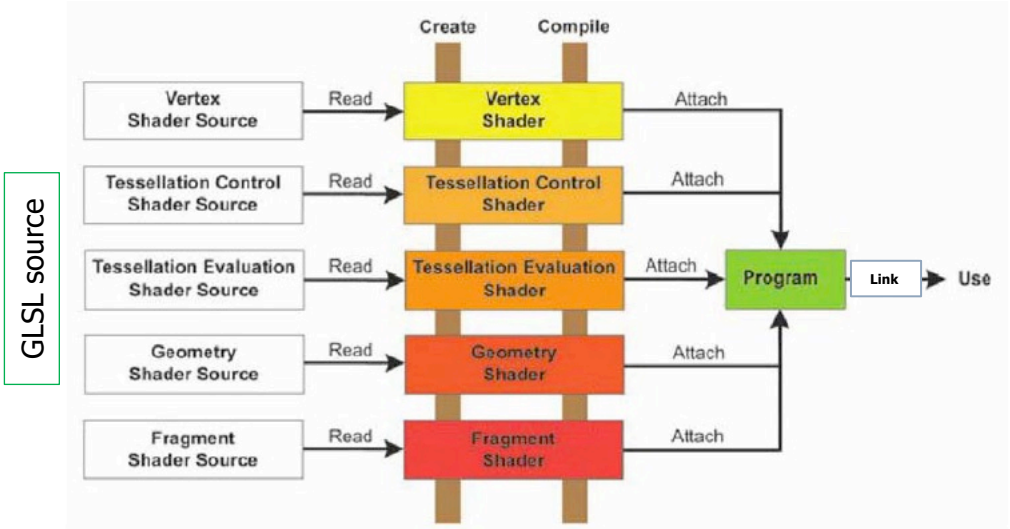
Fuente: Graphic Shaders: Theory and Practice

Pipeline de OpenGL en hardware gráfico

- Vertex Processor (primer esquema visto en Slide 2)
  - Entrada: vértices, normales, primitivas, color, luces, materiales y coordenadas de textura
  - Salida: Conjunto de vértices – como pixels – con color, profundidad, coordenadas de textura
- Rasterizer
  - Convierte los vertices – con información – en fragmentos
  - En esencia, interpola vértices para crear fragmentos
- El procesador de fragmentos
  - Crea el color a partir de la información
  - La salida son pixels RGBAZ
- Al final, se graban los pixels en el *frameBuffer*

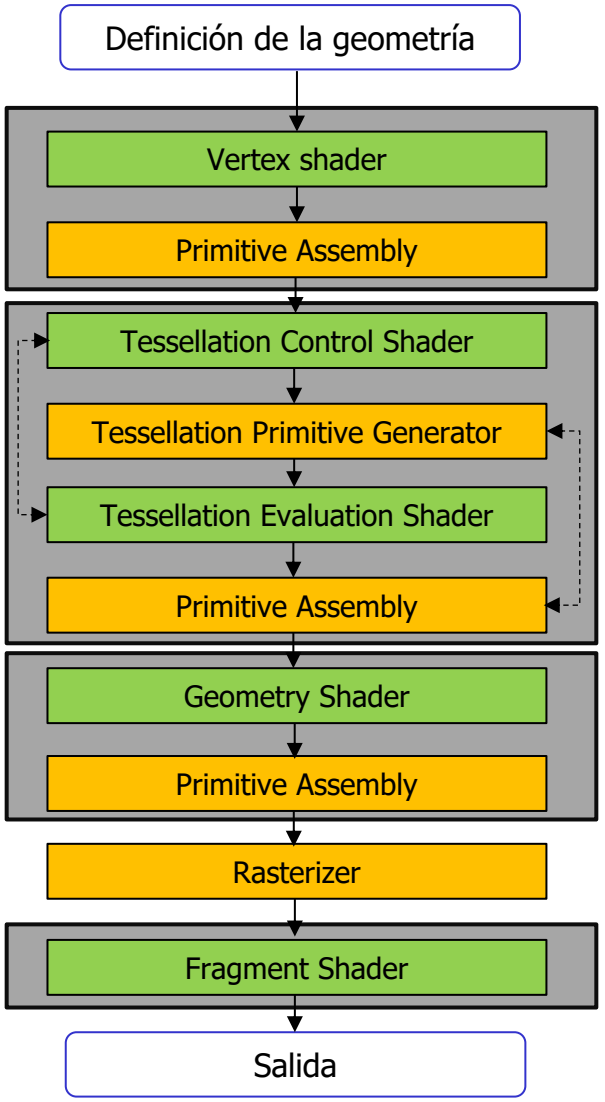
OpenGL: Pipeline gráfico programable

CPU: comandos OpenGL para instalar el programa en GPU



Tipos de shaders

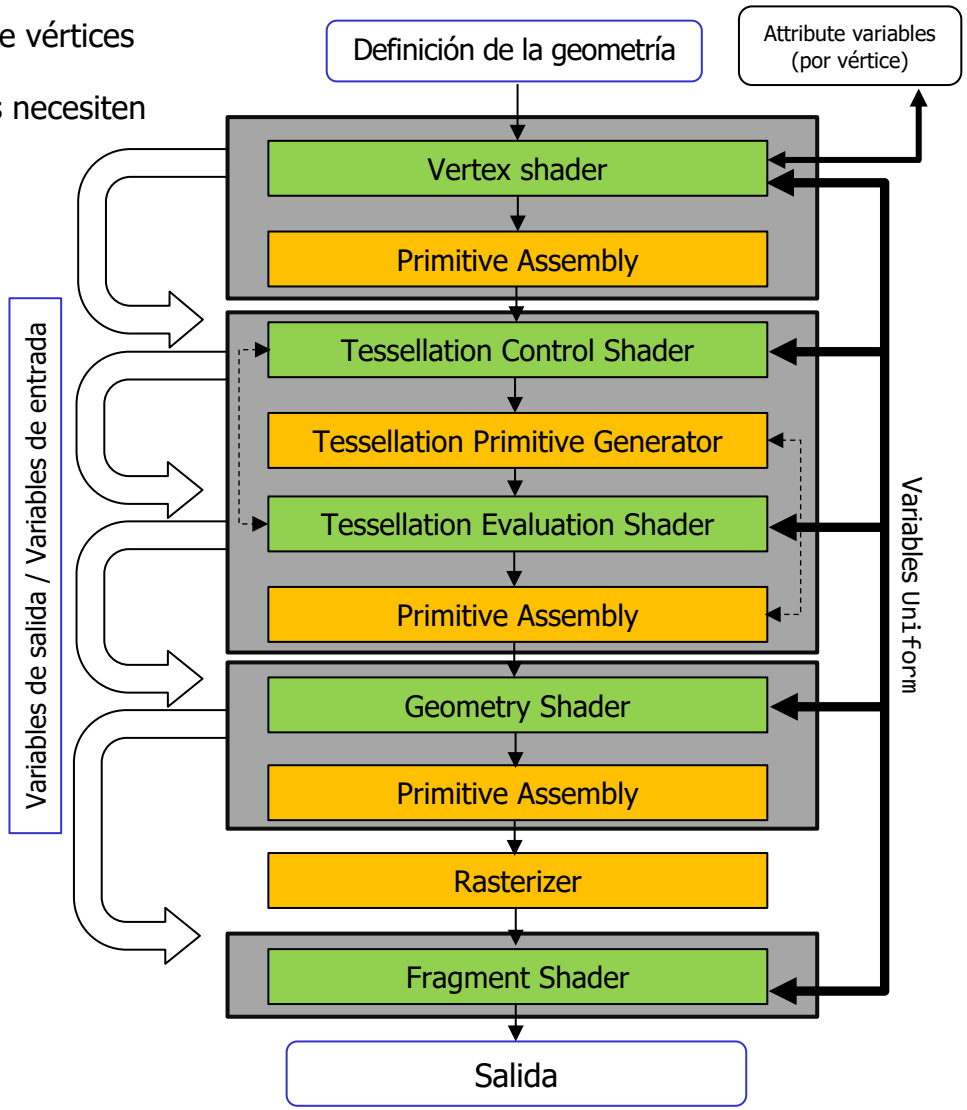
- Hay varios tipos de shaders:
  - De vértices (*vertex shaders*)
  - Control de teselación (*tessellation control shader*)
  - Evaluación de la teselación (*evaluation tessellation shader*)
  - De geometría (*Geometry shaders*)
  - De fragmentos (*Fragment shaders*)
- El siguiente esquema muestra más detalle de la pipeline
  - Cada shader es un camino adicional de la pipeline
  - Se pueden combinar y son opcionales
  - Aunque, en general. . .
    - si se usa cualquiera, también se incluye el de vértices



Fuente: Graphic Shaders: Theory and Practice

Pipeline gráfico con las etapas programables

- Todas las variables (*attributes*) son entrada al shader de vértices
- Las variables `Uniform` son entrada en todos los que las necesiten
  - Son definidas por la aplicación
  - No se pueden definir en el shader
- Los datos que se manden a otro shader
  - Variables tipo `out`
- El shader que lo necesite, lee la variable tipo `in`
- Un shader de vértices:
  - Solo opera sobre un vértice al mismo tiempo
  - Puede modificar su posición, normal o textura
- Un *Tessellation shader*, a partir de un conjunto de puntos, los interpola para crear una nueva geometría.
- Un *Geometry shader* parte de las primitivas gráficas del shader de vertices y crea una o más primitivas.
  - Procesan: geometría completa y color del vértice
- El Fragment shader opera sobre cada fragmento de forma individual, permitiendo cambiar el color del pixel



Fuente: Graphic Shaders: Theory and Practice

- 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` – no modificables - se corresponden con los atributos de los vértices de la malla
    - Las variables `out` con los valores interpolables (varying).
  - 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 variables de tipo `out`.
  - En el *Fragment shader* se puede desechar un píxel con `discard`

- Tipos de datos
  - Básicos: `float`, `int`, `bool`, `double`, `uint`
  - Vectores (XvecN) para N: 2, 3 y 4
    - `vecN` para float: vec2, vec3 y vec4
    - `ivecN` para int. ivec2, ivec3 e ivec4
    - `bvecN` para bool. bvec2, bvec3 y bvec4
    - Se accede a los campos con .x, .y, .z, .w, .xy, .xyz, .s, .t, .rbg, …
  - Vec4 (usos)
    - (`r`, `g`, `b`, `a`): Componentes de colores
    - (`x`, `y`, `z`, `w`): Componentes de geometría
    - (`s`, `t`, `p`, `q`): Componentes de coordenadas de textura
  - Matrices: `matN` para matrices NxN de `float` para N: 2, 3 y 4
    - `mat2`: matriz de 2x2
    - `mat3`: matriz de 3x3
    - `mat4`: matriz de 4x4

- Tipos de datos
  - 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];
  - Operadores
    - El de asignación es =
    - Los relacionales == y != pueden aplicarse a arrays y estructuras
      - Siempre que los arrays sean del mismo tamaño
      - Y las estructuras declaren los mismos tipos

- Matrices cuadradas NxN de `float` para N: 2,3 y 4
  - El tipo es `matN`
  - Funciones: `transpose`, `inverse`, `matrixCompMult`

```
mat4 matrix(1.0);                // 4x4 identity matrix
mat4 matrix = mat4 ( vec4,       // First column
                     vec4,       // Second column
                     vec4,       // Third column
                     vec4 );     // Fourth column

vec4 col = matrix[0];            // The first column
matrix[0][0]                     // The first entry of the first column
matrix[0].x                      // The first entry of the first column
matrix[1] = vec4(3.0, 3.0, 3.0, 0); // Sets the second column to all 3.0s
matrix[2][0] = 16.0;             // Sets the first entry of the third column to 16.0.
vec4 tra = matrix[3];            // Assigns the third column to the tra variable
```

- 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` : Constante que no puede usarse fuera del shader
  - `attribute`: Variable, utilizada sólo en el shader de vértices, establecida por la aplicación
  - `uniform`: Dato establecido fuera del shader y que no modifica su valor en el shader. Sirve para pasar datos *de CPU a GPU* al programa.
  - `uniform` para texturas: `sampler2D`, sampler3D, sampler1D
  - `In/out`: Variables para comunicar datos entre shaders.
    - `in`: Para atributos de vértice o para recoger información entre los shaders. Se lee la información del shader anterior en el pipeline. Son de **sólo lectura**.
    - `out`: Para pasar información entre los shaders. Son de **sólo escritura**. Pasa la información al shader siguiente en el pipeline.
  - `buffer:` para pasar datos de CPU a GPU y viceversa (OpenGL 4.3)

- Definición de funciones (NO recursivas)
  - Parámetros:
    - `const`: Se copia el valor al parámetro y no se modifica en la función
    - `in`: De entrada. No se modifica el valor
    - `out`: No tiene valor inicial, pero se asigna uno antes del `return`.
    - `inout`: Tiene valor inicial que– posiblemente – cambie antes del `return`
  - Operadores aritméticos y funciones predefinidas
    - Instrucciones de control:
      - `if`, `if-else`
      - `for`, `while`, `do-while`
  - Variables predefinidas (por shader):
    - output: `gl_Position`, `gl_PointSize` (vertex shader)
    - input: `gl_FrontFacing`, `gl_FragCoord` (fragment shader)
- Consultas: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

- Lo que no proporciona GLSL:
  - Casting de tipos
  - Promoción automática
  - Punteros
  - Strings
  - Enums

GLSL en OGRE: GPU Programs

- GLSL es el lenguaje de shaders nativo de la API OpenGL y no requiere *plugins*.
- 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.
  - En GLSL, no es necesario definir ningún punto de entrada, ya que siempre es `main()`
  - El código fuente de GLSL se compila en código nativo de la GPU y no en ensamblador intermedio.

GLSL en OGRE: GPU Programs

- Se hace referencia a los shaders en la sección de pase (`pass`) del script de materiales
  - Shaders de vértices, geometría, teselación y/o fragmentos.
- Los programas se definen por separado
  - Permite su reutilización entre muchos materiales distintos
  - Se define el programa una sola vez
- Un ejemplo, vinculamos un programa de vértices llamado `myVertexProgram`

```
vertex_program_ref myVertexProgram{
    param_indexed_auto 0 worldviewproj_matrix
    param_indexed      4 float4  10.0 0 0 0
}
```

  - 2 parámetros:
    - El primero es 'auto', lo que significa que no tenemos que suministrar un valor como tal, sólo un código reconocido (la matriz mundo/vista/proyección en este caso)
    - El segundo parámetro es un parámetro especificado manualmente, un float de 4 elementos.

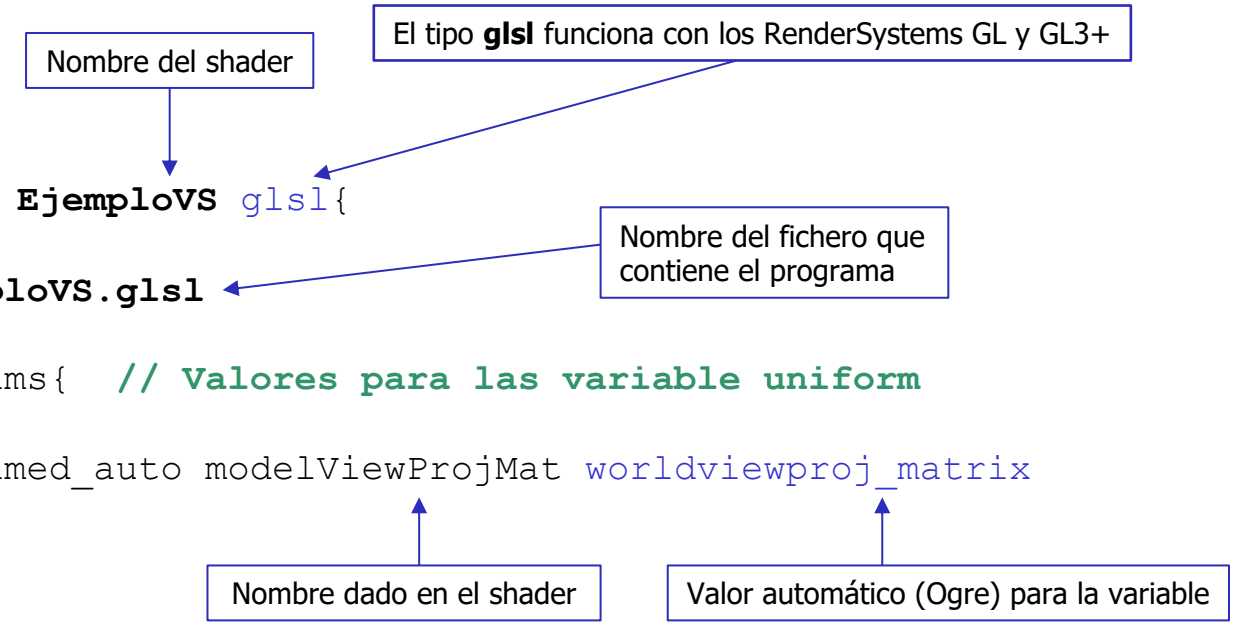
- La sintaxis para referenciar shaders de vértices y fragmentos es idéntica
  - `fragment_program_ref` para fragmentos y `vertex_program_ref` para vértices

GLSL en OGRE: GPU Programs

- Una única definición del shader puede ser utilizada por cualquier número de materiales
  - El programa debe ser definido antes de ser referenciado en la sección `pass` de un material.
- La definición de un programa puede incluirse en el propio script `.material`
  - También es posible definirlo en un script `.program` externo
    - Así podrá utilizarse en varios archivos `.material`
- Se garantiza que todos los scripts `.program` han sido analizados antes que **todos** los scripts `.material`
  - En nuestro caso, los ficheros de programas serán `.glsl`
- Al igual que los scripts `.material`, los scripts `.program` se leerán desde cualquier ubicación que esté en las rutas correspondientes (`resources.cfg`)
- Para definir un programa de vértices o fragmentos (en el fichero `.material`)
  - `vertex_program` y `fragment_program`
- Es posible especificar los parámetros por defecto que se utilizarán en el shader.
  - Esto se hace incluyendo una sección anidada **`default_params`**

GLSL en OGRE: GPU Programs

```
// Archivo example.material

vertex_program EjemploVS glsl{

   source EjemploVS.glsl

   default_params{   // Valores para las variable uniform

       param_named_auto modelViewProjMat worldviewproj_matrix

   }
}
```



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

GLSL en OGRE: GPU Programs

- Algunos atributos ya están incorporados **por defecto**, con su nombre en Ogre
  - Para los vértices (de entrada)

Nombre en Ogre	Índice	OpenGL
vertex	0	glVertex
Normal	2	glNormal
colour	3	glColor
Uv0-uv7	8-15	gl_MultiTexCoord0 - gl_MultiTexCoord7

  - Variables de salida – por defecto – para los fragmentos
    - `fFragColor` : Color del pixel
    - `gl_FragDepth`: Profundidad del pixel

GLSL en OGRE: GPU Programs

- Algunas matrices que que incorpora Ogre por defecto
  - `world_matrix` (matriz del mundo)
  - `view_matrix` (matriz de la vista actual)
  - `projection_matrix` (matriz de la proyección actual)
  - `worldview_matrix` (Matrices del mundo actual y la vista concatenadas)
  - `worldviewproj_matrix` (Matrices del mundo actual, vista y proyección concatenadas)
- Variables de la luz
  - `light_position` (Posición de la luz 0)
  - `light_position_view_space` (Posición de la luz 0)
  - `light_diffuse_colour` (Intensidad de la componente difusa de la luz 0)
- Variables de la cámara
  - `camera_position` (Posición de la cámara)

GLSL en OGRE: GPU Programs

- 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
```

  - Ejemplos:

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

- En OpenGL con comandos específicos:

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

GLSL en OGRE: GPU Programs

- Hay 4 formas de pasar parámetros al shader
  - **`param_named`**
    - Formato: `param_named <name> <type> <value>`
      - **name:** Nombre de la variable en el shader.
      - **type:** Puede ser `float4`, `matrix4x4`, `float<n>`, `int4`, `int<n>`
      - **value**: Lista de valores separados por espacios o tabulaciones
  - **`param_named_auto`**
    - Actualiza automáticamente un parámetro dado con un valor derivado
    - 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
      - No es necesario escribir código para actualizar los parámetros en cada fotograma.
    - Formato: `param_named_auto <name> <autoConstType>`

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

GLSL en OGRE: GPU Programs

- Hay 4 formas de pasar parámetros al shader
  - **`param_indexed`**
    - Similar a `param_named`, pero utiliza un índice en lugar de un nombre
    - Formato: `param_indexed <index> <type> <value>`
      - **index** es un entero relativo a la forma en que se almacenan las constantes en la tarjeta gráfica: en bloques de 4 elementos. Por ejemplo, si se definió un parámetro `float4` en el índice 0, el siguiente índice sería 1. Si se definió una matriz4x4 en el índice 0, el siguiente índice utilizable sería el 4, ya que una matriz 4x4 ocupa 4 índices.
      - **type** puede ser `float4`, `matrix4x4`, `float<n>`, `int4`, `int<n>`
      - **value**: Lista de valores separados por espacios o tabulaciones
  - **`param_indexed_auto`**
    - Equivalente a `param_named_auto` utilizando índices de parámetro en lugar de nombres
    - Formato: `param_indexed_auto <index> <autoConstType>`
      - **index** es un entero
      - **autoConstType** es uno de los tipos definidos por defecto.
        - Por ejemplo `world_matrix`

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

GLSL en OGRE: GPU Programs

- Ejemplos de valores Ogre para los parámetros con `param_named_auto`:

```
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                   // Posición de la cámara (en world space)

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
```

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

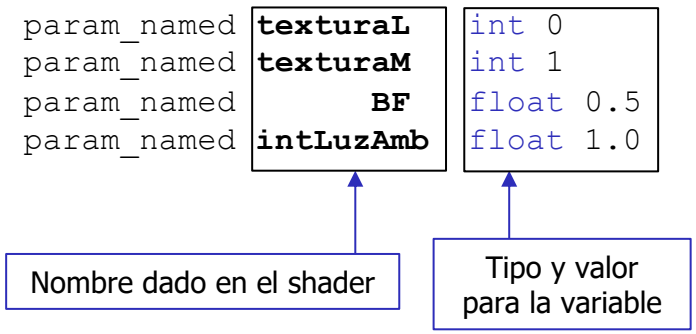
```
param_named materialDiffuse float3 0.5  0.5  0.5
```

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

```
param_named_auto materialDiffuse surface_diffuse_color
```

GLSL en OGRE: GPU Programs

```
// Archivo example.material


fragment_program EjemploFS glsl{

    source EjemploFS.glsl                    // Nombre del archivo del código
    default_params{                          // Valores para las variable uniform

        param_named texturaL    int 0        // 1° unidad de textura
        param_named texturaM    int 1        // 2° unidad de textura
        param_named      BF     float 0.5    // Valor float
        param_named intLuzAmb   float 1.0    // Valor float
    }
}
```



GLSL en OGRE: GPU Programs

```
// Archivo example.material


material example/ejemploGLSL{
    technique{
        pass{
           vertex_program_ref EjemploVS{
             // Parámetros para el shader
           }
           fragment_program_ref EjemploFS{
             // Parámetros para el shader
           }
           texture_unit {
              texture ejemploA.jpg 2d
              tex_address_mode clamp
              filtering bilinear
           }
           texture_unit {
              texture ejemploB.jpg 2d
              tex_address_mode wrap
           }
        }
    }
}
```

GLSL en OGRE: GPU Programs

- Los parámetros `uniform` para las texturas son del tipo `samplerXD`

```
uniform sampler2D texName;
```

- 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…)
```

GLSL en OGRE: GPU Programs

- 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…)

GLSL en OGRE: GPU Programs

- Podemos utilizar time para los valores uniform

```
// Tiempo transcurrido
param_named_auto tiempo time

// Valores float en el intervalo [0..1] que se repiten cada 1 segundo
param_named_auto tiempo1 time_0_1 1

// Valores en el intervalo [sin(0)..sin(2pi)] que se repiten cada 60 segundos
param_named_auto senotiempo sintime_0_2pi 60
```

- En el ejemplo anterior podemos modificar

```
param_named BF float 0.5
```

por

```
param_named_auto BF time_0_1 10
```