TEMA 5. INTRODUCCIÓN A LA PROGRAMACIÓN EN GPU

Objetivos:

- *Conocer los fundamentos de las tarjetas gráficas y su funcionamiento.*
- *Conocer los fundamentos de la programación con vertex y fragment shaders, según el estándar OpenGL 4.x*

5.1. ESTRUCTURA DE LA TARJETA GRÁFICA.

Las tarjetas gráficas han evolucionado de forma espectacular en la última década, en gran medida por las necesidades de la industria que hace uso de los sistemas gráficos (cine, videojuegos, etc.).

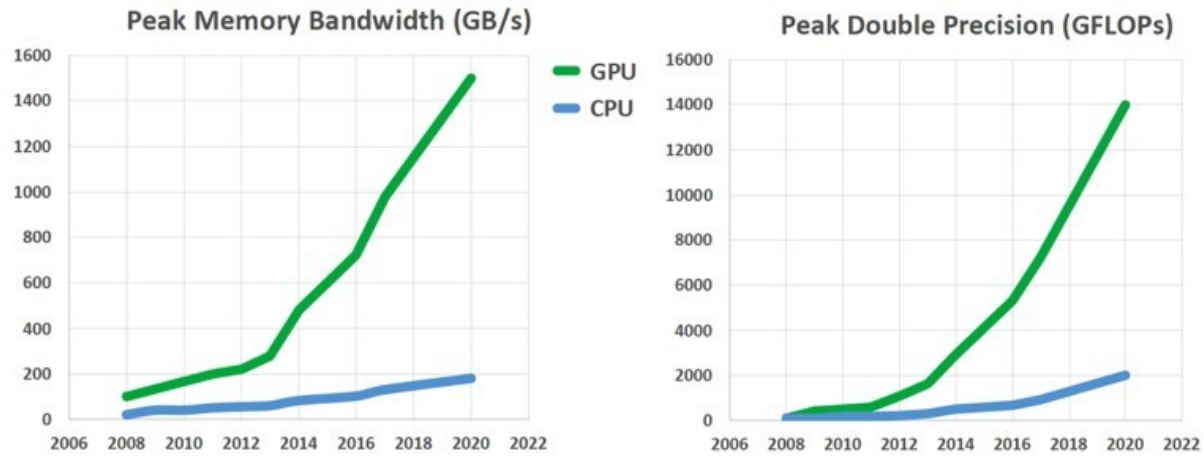

Ilustración 135 Operaciones en coma flotante por segundo.

Paralelamente a esta evolución, se han visto beneficiadas otras áreas que en principio no tenían que verse afectadas por dicho progreso, como es el caso del procesamiento paralelo de propósito general. En una tendencia iniciada por NVIDIA, las GPU (Graphics Processing Unit) han ido transformarse desde una arquitectura orientada únicamente a la renderización 3D a lo que ahora se llaman GPGPU (General Purpose GPU), de forma que las APIs gráficas se han utilizado para desarrollar algoritmos paralelos en áreas tan diversas como la síntesis de proteínas, gestión de precios en tiempo real, reconstrucción de resonancias magnéticas nucleares o consultas masivas a bases de datos. Además, este incremento en la necesidad de paralelismo ha llevado a los fabricantes de CPUs a evolucionar los procesadores, incluyendo más núcleos e instrucciones SIMD, pero nunca alcanzando la capacidad de procesamiento de las GPUs, como se aprecia en la Ilustración , donde también se pude ver como las GPU siguen la ley de Moore mucho mejor que las CPU.

Con anterioridad a las GPU existían las tarjetas VGA, que no era más que un controlador con una memoria dedicada encargado de generar imágenes en un dispositivo de video, como el monitor. En la década de los noventa, este controlador fue "complicándose" con componentes para realizar tareas de rasterización, texturización y sombreado básico.

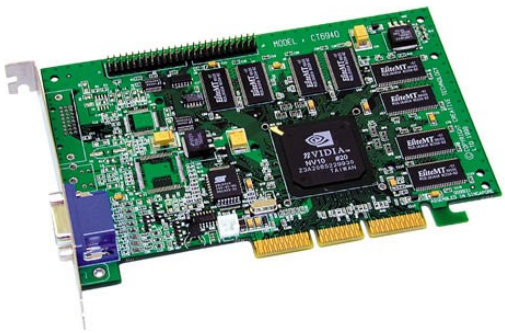

Ilustración 136 NVIDIA GeForce 256. Primera tarjeta gráfica en usar el término GPU

En 1999, Nvidia presentó la GeForce 256, que puede ser considerada como la primera GPU. En aquel momento existían otras compañías que aún perviven, como ATI.

Las primeras GPUs estaban optimizadas para tareas como coger un conjunto de coordenadas, colores, texturas y parámetros de luz y generar una imagen de forma eficiente. Con el tiempo, las GPU cada vez son más programables, y esos pequeños programas, denominados *shaders* se pueden apicar a cada triángulo, vértice y pixel, por lo que se pueden obtener efectos más complejos (reflexiones, refracciones, etc.). A ello se le añade que lo que inicialmente era aritmética entera ahora es aritmética de punto flotante de doble precisión.

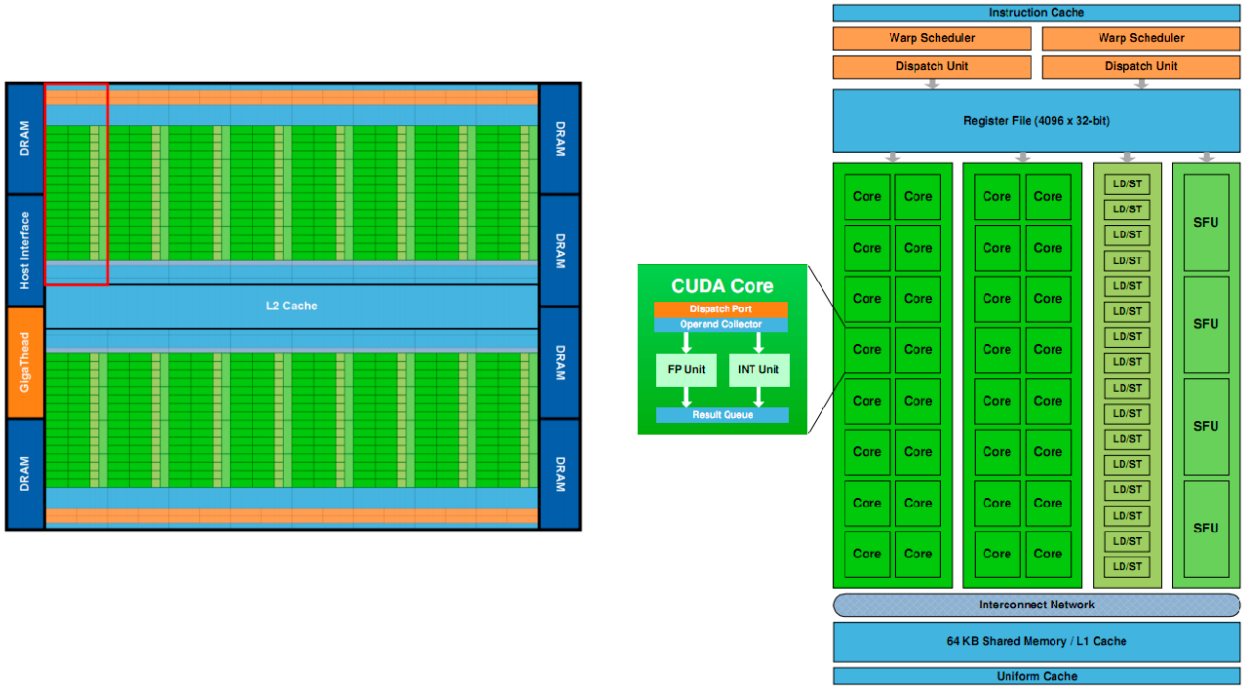

Ilustración 137 Estructura de una GPU tipo "Fermi" de NVIDIA

En las primeras generaciones de GPUs, los núcleos de los procesadores se dividían entre procesadores de vértices y de fragmentos, y de ahí viene la primera evolución de OpenGL desde el pipeline fijo al programable. A partir de 2008, todos los cores admiten el mismo conjunto de instrucciones y por tanto se igualan en cuanto a hardware. La capacidad de cómputo viene repartida por los 16 multiprocesadores (que se muestra en detalle en la Ilustración 137 dcha.). Cada uno de estos multiprocesadores dispone de procesadores escalares (con su ALU y FPU), denominados CUDA cores. Además, cada multiprocesador tiene su registro, unidades de carga y almacenamiento y 4 unidades de funciones especiales (SFU), que realizan funciones como el seno y el coseno.

El hardware de la GPU está diseñado para calcular operaciones sobre millones de píxeles, con miles de vértices y enormes texturas, pero al contrario que las CPUs se permite una mayor latencia, pues cumpliendo 60 ciclos por segundo, el usuario no notará retrasos. Esto es un alivio, porque apenas hay sitio para cachés. Una diferencia con la memoria RAM que usa la CPU es que la de la de la GPU está optimizada para el manejo de arrays 2D, es decir, texturas. Además, la caché de la GPU, que existe, suele ser de sólo lectura, ya que la escritura del dato final se realiza al final del shader.

Las instrucciones GPU son inherentemente SIMD (Single Instruction Multiple Data). En los sistemas gráficos, los shaders ejecutarán el mismo programa sobre miles de datos, como por ejemplo los cálculos de iluminación o la aplicación de texturas. Un ejemplo puede ser la aplicación de una función, p.ej. una traslación, sobre un array bidimensional. En CPU ello se haría con un doble bucle anidado.

Sin embargo, en GPU tan sólo hay que dar una instrucción para aplicar la función, y todos los datos de la matriz se calculan en paralelo.

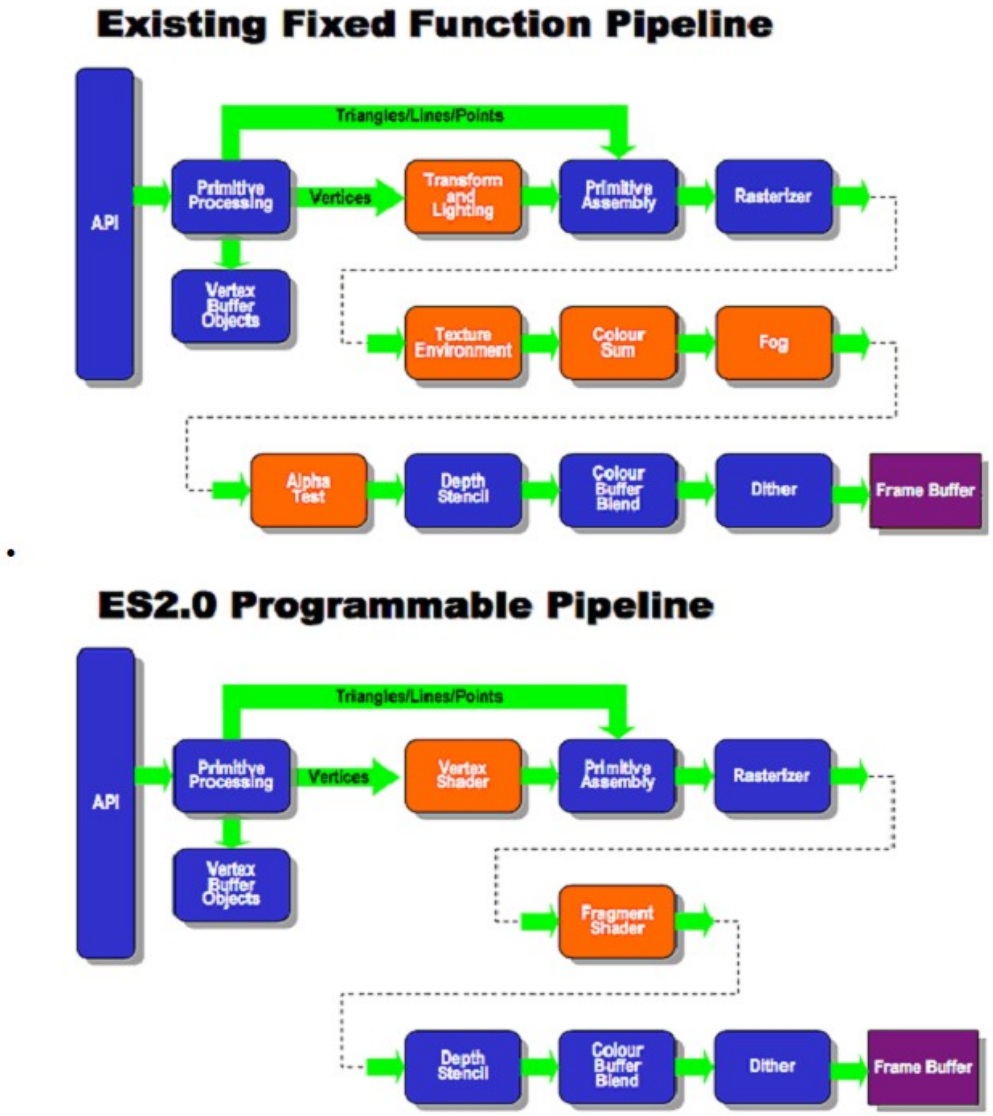

Ilustración 138 Pipeline Fijo - vs PipeLine Programable OpenGL ES 2.0

5.2. Optimización del rendering mediante la programación de shaders en GPU.

En la Ilustración 138 se muestra la transformación que ha sufrido la ejecución de programas gráficos desde los inicios de OpenGL al momento actual. De hecho, se muestra sólo el cauce gráfico de OpenGL ES 2.0, que es más sencillo que el de OpenGL 4.3, por ejemplo, ya que sólo permite programar el *vertex shader* y el *fragment shader.*

Los *shaders* de OpenGL están escritos en un lenguajes denominado GLSL (OpenGL Shading Language). Es un lenguaje inspirado en C, pero modificado para encajar mejor en los procesadores gráficos. No es necesario ningún compilador aparte, sino que el mismo OpenGL lo compila. La idea es:

- El código fuente del shader se trata como una cadena de texto que se guarda en una variable.
- Esta variable y todos los shaders que haya en sus respectivas variables se compilan
- Se construye un programa *objeto* que enlaza todos los *shaders* juntos.

Podemos decir que el *vertex shader* se ocupa de la geometría y el *fragment shader* de generar la imagen final.

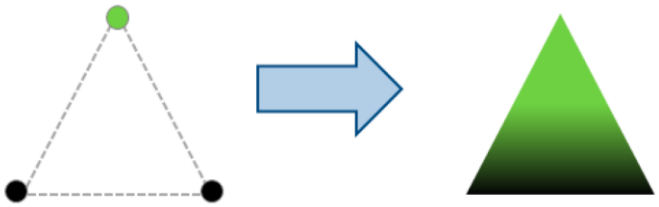

Ilustración 139 Información afectada por el vertex shader (Izda) y por el fragment shader (Dcha)

Vertex shader

En *vertex shader* toma la información de los vértices y su entorno que estén almacenados en el sistema OpenGL y los prepara en un conjunto de variables uniformes y de atributo, de forma que se puedan realizar cálculos sobre éstos.

El procesador de vértices trabaja sobre geometría, que normalmente está en coordenadas del modelo, y produce geometría que está referenciada en el sistema de coordenadas de vista 3D. La proyección y recortado se realizan en etapas posteriores del cauce gráfico.

¿Qué operaciones que antes se hacían en el cauce fijo se pueden programar ahora en un procesador de vértices?

- Transformaciones de vértices
- Transformaciones de normales (incluida su normalización)
- Gestión de la iluminación por vértice
- Gestión de las coordenadas de textura

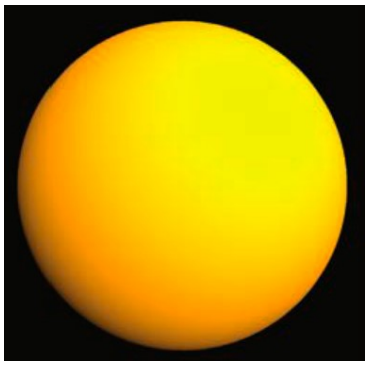

Ilustración 140 Esfera amarilla.

Sin embargo, hay más operaciones que no se pueden realizar en este shader y se dejan para etapas posteriores:

- Recorte del volumen de visualización
- División por coordenadas homogéneas
- Mapeado del viewport
- Eliminación de partes traseras
- Desplazamiento y relleno de polígonos, etc.

Podríamos decir que la parte primordial del vertex shader es tomar todas las variables de atributo y usarlas o copiarlas en variables de salida para posteriores *shaders.*

Un procesador de vértices tiene varias salidas, siendo las más importantes los vértices transformados y el color asociado a cada vértice. Por supuesto, también se pueden calcular o recalcular normales y coordenadas de textura, así como modificar coordenadas de vértice.

En el Código T5- 1 podemos ver un ejemplo de un *vertex shader* que realiza los cálculos necesarios para la esfera amarilla de la Ilustración 140. En él se calculan para cada vértice la intensidad de la luz según el valor de la normal, las coordenadas del observador y la posición de la luz.

Nótese como en el Código T5- 1 el color no se modifica. Su único interés es el cálculo de la variable vLightIntensity, que será la que en el procesador de fragmentos se utilice para calcular el color real.

Hay que pensar que para cada vértice de nuestro programa se ejecutará el código del main, por lo que hay que leer este código como si fuera "*para cada vértice, que se pasará en aVertex, aNormal y aColor*"…

```
uniform mat4 uModelViewMatrix; // Matriz 4x4 ModelView (global)
uniform mat4 uModelViewProjectionMatrix; // Matriz 4x4 ModelView*Projection
uniform mat3 uNormalMatrix;  // Matriz de normal: view*traspuesta(inversa(Model))

in vec4 aVertex; // entrada al shader: vértice
in vec4 aNormal; // entrada al shader: normal en el vértice
in vec4 aColor;  // entrada al shader: color en el vértice

out vec4  vColor;   // salida del shader: color del vértice
out vec3  vMCposition; // salida del shader: posición XYZ del vértice
out float vLightIntensity;  //salida del shader: intensidad de luz en vértice

const vec3 LIGHTPOS = vec3( 3., 5., 10. ); // posición de la luz

void main( ) {
 vec3 transNorm   = normalize( uNormalMatrix * aNormal ); // Normal en SCV

 vec3 ECposition  = vec3( uModelViewMatrix * aVertex ); // Vertice en SCV

 vLightIntensity  = dot(normalize(LIGHTPOS - ECposition), transNorm);

 vLightIntensity  = abs( vLightIntensity ); // Intensidad de la Luz

 vColor       = aColor;

 vMCposition   = aVertex.xyz;

 gl_Position   = uModelViewProjectionMatrix * aVertex; // Posicion 2D del vertice
}
```

Código T5- 1 Vertex Shader de ejemplo

El procesador de vértices utiliza varios tipos de variables:

- **Variables de atributo.** Son las que almacenan las propiedades del vértice que se está procesando, y son variables de entrada. Sólo pueden ser leídas en el *vertex shader*, ya que sólo almacenan propiedades de los vértices (posición, color, normal, etc).
- **Variables uniformes.** Son constantes a lo largo del procesamiento de una primitiva, y son de sólo lectura. Al igual que las variables de atributo, vienen de la aplicación principal de OpenGL. Pueden especificar valores como:
  - Las matrices primarias de OpenGL: ModelView, Projection o Texture
  - Las matrices derivadas de OpenGL: Nomral, ModelViewProjection y ModelViewInverse
  - Los planos de recorte delanteros y traseros
  - Las propiedades del material: ambiente, difuso, especular, brillo y emisión.
  - Las propiedades de la luz: color, posición, dirección, cono y atenuación.
  - Los parámetros del efecto niebla: color, densidad, inicio fin.
- **Variables de salida**. Son los resultados del procesamiento, y se usan como variables de sólo escritura. Hay unas variables que son obligatorias, como `glPosition` (que tiene las posiciones del vértice en 3D en coordenadas de recortado) y `gl_PointSize` (que almacena,

opcionalmente, el tamaño del vértice en pixels). También se deben generar como salida, aunque no se modifiquen, otras como el color.

A la hora de gestionar las coordenadas de los vértices, hay que tener en cuenta en qué Sistema de Coordenadas las estamos pasando. Como convención se suele usar MC como prefijo para *Model-space Coordinates* o coordenadas del modelo y EC para *Eye-space Coordinates,* o coordenadas de observador. Se pueden calcular como:

```
vec3 MCposition= aVertex.xyz;
vec3 ECposition = (uModelViewMatrix * aVertex).xyz;
```

En resumen, el trabajo del procesador de vértices se muestra en la Ilustración 141

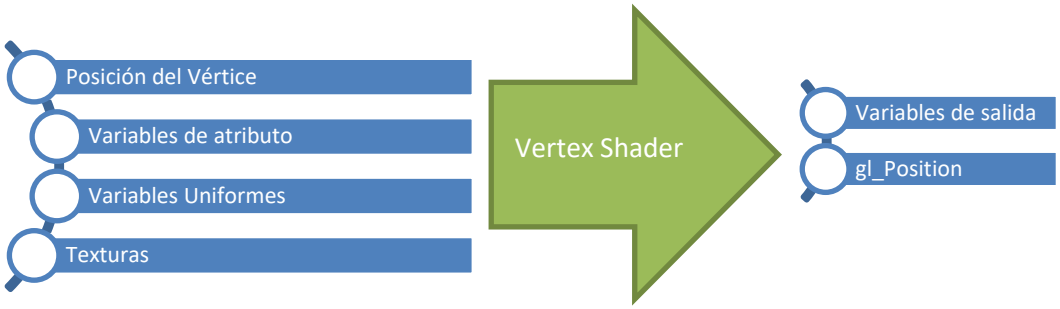

Ilustración 141 Entradas y salidas del procesador de vértices

Fragment shader

A veces se denomina procesador de fragmentos o de pixels, y operan sobre éstos para determinar su color. Sabemos, por la asignatura de Informática Gráfica, que la operación de rasterización interpola cantidades como los colores, profundidades y coordenadas de textura que sólo son conocidas para los vértices.

Igual que ocurría con el procesador de vértices, operaciones que antes eran automáticas en el pipeline fijo, ahora han de ser realizadas a mano:

- Cálculo de color,
- Texturización,
- Iluminación,
- Niebla, etc.

```
in vec4  vColor; // entrada al shader: color del vértice
in float vLightIntensity;  // entrada al shader: intensidad de la luz

out vec4 fFragColor; // salida del shader: color del pixel

void main( ) {
   fFragColor = vec4( vLightIntensity*vColor.rgb, 1. );
}
```

Código T5- 2 Fragment shader para generar la Ilustración 140

Hay una gran variedad de datos que se le pueden pasar a un procesador de fragmentos, bien desde el sistema OpenGL o del procesador de vértices. Estas variables pueden ser:

- **Uniformes.** Son variables proporcionadas por el sistema o por la aplicación. Podemos verlas como variables globales de sólo lectura.
- **De entrada.** Las variables de entrada al procesador de fragmentos son las de salida de la etapa anterior. En OpenGLES sería del procesador de vértices, pero hay otros procesadores que no veremos en la asignatura, como el de geometría o el de teselación, que se ejecutan entre el *vertex shader* y el *fragment shader* en un contexto OpenGL 4.x. Estas variables de entrada son los datos que se interpolan dentro de cada primitiva para obtener información suficiente al *fragment shader* y poder calcular el color de cada pixel. Las variables de entrada más importantes para el procesador de fragmentos son la posición, el color, las coordenadas de textura, la profundidad y cualesquiera otros datos que puedan servir para calcular el color del pixel:
    - Color. El color se calcula para cada pixel de formas muy distintas. La forma tradicional es interpolar linealmente los colores de cada vértice, o interpolar las coordenadas de textura y usar texturas para calcular el color. Una vez que se ha calculado, se guarda en la variable fFragColor.
    - Coordenadas de textura. Si se utilizan texturas, es posible obtener las coordenadas de texel para cada vértice. Estas coordenadas se pueden almacenar en una variable denominada vST
- **De Salida.** El procesador de fragmentos debe contener al menos una variable de salida para almacenar el color del pixel, definida como

```
out vec4 fFragColor;
```

IMPORTANTE: Los nombres y tipos de los parámetros de las variables de in del *fragment shader* deben coincidir uno a uno con los parámetros out del vertex shader.

Se pueden declarar otras variables del tipo vec4, y guardar en ella información de color, para guardarlo en otro buffer o sacarlo por otra pantalla. Esta información puede ser:

- Intensidad de la luz
- Coordenadas geométricas
- Coordenadas de textura
- Vectores de reflexión/refracción por cada vértice
- Vectores de los vértices a la fuente de luz
- Las coordenadas del fragmento en coordenadas de pantalla.

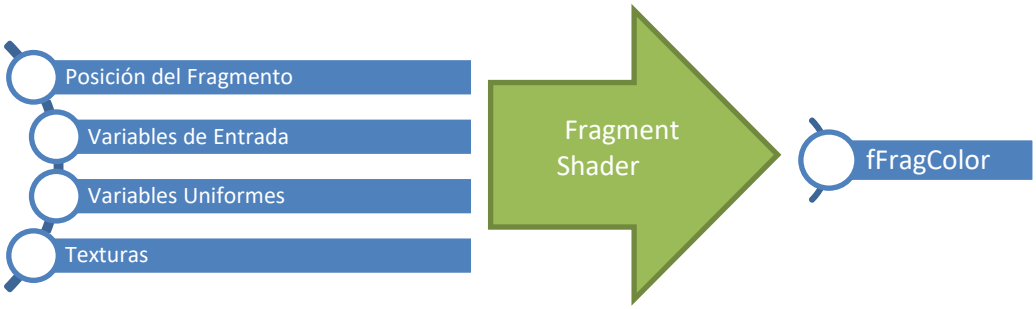

Ilustración 142 Entradas y salidas del procesador de fragmentos

El código de GLSL se parece mucho a C, con los operadores aritmético lógicos habituales e instrucciones de preprocesamiento como #define, #ifdef, etc.

GLSL incluye nuevos tipos que son de mucha utilidad en los gráficos por ordendor:

- Enteros escalares y vectoriales: `int, ivec2, ivec3, ivec4`
- Reales escalares y vectoriales: `float, vec2, vec3, vec4`
- Matrices cuadradas reales: `mat2, mat3, mat4`
- Matrices no cuadradas reales: `mat3x2`, etc.
- Booleanos escalares y vectoriales, `bool, bvec2,bvec3, bvec4`

El hecho de combinar elementos escalares y vectoriales hace que un operador como * esté altamente sobrecargado.

Otra propiedad interesante es que a los vectores se puede acceder con el tradicional [ ] , o bien usando atributos especiales como `.rgba` (si lo consideramos un color), `.xyzw` (geometría) o `.stpq` (coordenadas de textura). Por ejemplo, `aVertex.xyz` devuelve los tres primeros componentes de un vértice denominado `aVertex`. Si usáramos `aVertex.rgb` obtendríamos lo mismo, pero no se vería intuitivo en el código que estamos obteniendo coordenadas en lugar de colores.

Diferencias con C:

- No hay punteros
- No hay conversión de tipos
- No hay strings
- No hay enumeraciones

Ya hemos visto en los ejemplos anteriores cómo las variables, además del tipo, se puede especificar si es

- `const`, es decir que es constante en tiempo de compilación y no se puede referenciar fuera de ese shader concreto.
- `attribute`, si es una variable usada en el vertex shader que se envía desde la aplicación o desde el contexto OpenGL (<=1.3)
- `uniform`, si es una variable enviada desde fuera del shader y que puede variables como mucho una vez por primitiva
- `in`, si es una variable de entrada
- `out / varying` , si es una variable de salida.

Así mismo, los parámetros de las funciones en el shader pueden tipificarse como `in`, `out` o `inout`.

Algunas variables se van a utilizar mucho y se ven en muchos ejemplos, por lo que es bueno seguir un criterio homogéneo y casi estándar, a saber:

- `vec3 aVertex` : las coordenadas del vértice actual en coordenadas del modelo
- `vec3 aNormal` : las coordenadas de la normal del vértice actual en su sistema coordenado original
- `vec4 aColor` : el color del vértice actual
- `vec4 aTexCoordi` (i=0,1,2) – las coordenadas de textura de nivel i para el vértice actual.

Además, hay otras variables uniformes que se pueden acceder (y se suelen usar desde todos los shaders), como son:

- `mat4 uModelViewMatrix` : la matriz de modelview, que si recordáis es el producto de las matrices de transformación modelado y de transformación de vista
- `mat4 uProjectionMatrix`: la matriz de proyección
- `mat4 uModelViewProjectionMatrix`: el producto de las matrices MV y P
- `mat3 uNormalMatrix`: la matriz normal activa para un vértice concreto (es la inversa traspuesta de la matrix MV)

Normalmente se establece la siguiente convención a la hora de nombrar una variable:

Letra de inicio	Significa que la variable
a	Es un atributo por vértice dado por la aplicación
u	Es una variable uniforme dada por la aplicación
v	Es una salida del *vertex shader*
f	Es una salida del *fragment shader*

Esquema general de una aplicación OpenGL 4.x

Hasta ahora nos hemos fijado exclusivamente en los *shaders*, su funcionalidad y propósito, pero hemos obviado, intencionadamente, su contexto dentro de un programa general.

Cuando se usan *shaders,* no sólo hay que proporcionar el código de estos, sino también realizar algunos pasos, mostrados en la Ilustración 143, para integrarlos en nuestra aplicación.

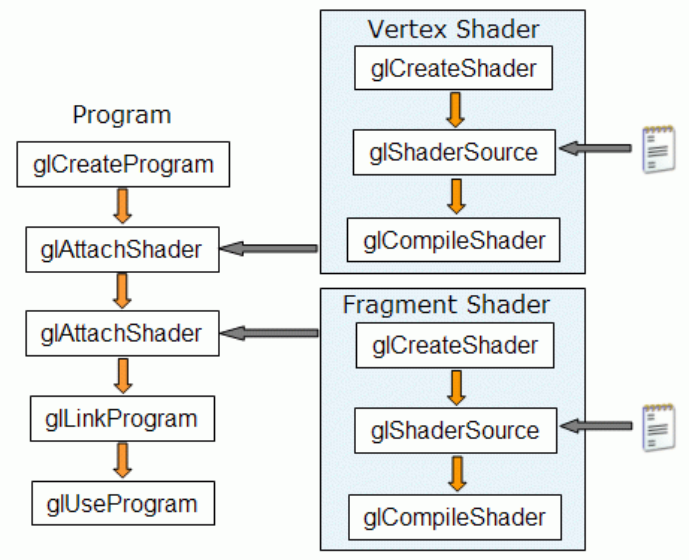

Ilustración 143 Pasos necesarios para crear los shaders

Estos pasos son:

1. Crear el código fuente de los *shaders* y guardarlo en sendos archivos.
2. Leer los archivos en una cadena de texto.
3. Crear un **objeto** *shader* para cada una de las cadenas que contienen al *shader*.
4. Asignar a cada objeto la cadena con el código fuente de su *shader*
5. Compilar cada objeto.
6. Crear un ***programa shader*** global
7. Conectar todos los *shaders* con ese programa global.
8. Enlazar el programa de shaders
9. Especificar que ese programa de shaders se usará en lugar de las funciones del pipeline fijo.

La Ilustración 143 se traduce a C en el Código T5- 3, y es usada en el Código T5- 4.

En el Código T5- 4 llama la atención la llamada a `glewInit()` y el `if` posterior. GLEW (OpenGL Extension Wrangler Library) (http://glew.sourceforge.net) proporciona en tiempo de ejecución mecanismos para determinar qué extensiones o funcionalidades de OpenGL están implementadas en el hardware subyacente. Por ejemplo, `glCreateProgram()` puede estar disponible como `glCreateProgramEXT()` o `glCreateProgramARB()`, por lo que tiene sentido usar el nombre genérico y dejar a GLEW que se encargue de averiguar cuál está implementada en el sistema.

```
void setShaders() {
        char *vs,*fs;

        v = glCreateShader(GL_VERTEX_SHADER);
        f = glCreateShader(GL_FRAGMENT_SHADER);

        vs = textFileRead("verticesshader.vert");
        fs = textFileRead("fragmentshader.frag");

        const char * vv = vs;
        const char * ff = fs;

        glShaderSource(v, 1, &vv,NULL);
        glShaderSource(f, 1, &ff,NULL);

        free(vs);free(fs);

        glCompileShader(v);
        glCompileShader(f);

        p = glCreateProgram();

        glAttachShader(p,v);
        glAttachShader(p,f);

        glLinkProgram(p);
        glUseProgram(p);
 }
```

Código T5- 3 Creación de Shaders en C

```
int main(int argc, char **argv) {
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
 glutInitWindowPosition(100,100);
 glutInitWindowSize(320,320);
 glutCreateWindow("MM 2004-05");

 glutDisplayFunc(renderScene);
 glutIdleFunc(renderScene);
 glutReshapeFunc(changeSize);
 glutKeyboardFunc(processNormalKeys);

 glEnable(GL_DEPTH_TEST);
 glClearColor(1.0,1.0,1.0,1.0);
//      glEnable(GL_CULL_FACE);

 glewInit();
 if (glewIsSupported("GL_VERSION_2_0"))
       printf("El sistema acepta OpenGL 2.0\n");
 else {
       printf("OpenGL 2.0 no soportado\n");
       exit(1);
 }
 setShaders();

 glutMainLoop();

 // just for compatibiliy purposes
 return 0;
}
```

Código T5- 4 Función main() que usa la función setShaders del Código T5-3

No es lo mismo un *shader* que un programa *shader*. El programa contiene compilados y linkados todos los shaders que se van a usar.

Pasar datos a los shaders

Cuando se escribe un programa con OpenGL, es necesario leer o crear en tiempo de ejecución la geometría de los objetos que están en la escena, así como las transformaciones que se aplican, bien a la matriz de modelado, a la de vista o a la de proyección.

Estos datos, como hemos visto antes, pueden estar en variables uniformes (algo así como "globales" para cada primitiva) o en variables de atributo (aquellos valores asociados a cada vértice, como el color, la posición, etc.).

Un asunto importante a reseñar: las variables se definen y usan en el *shader*, pero sus valores son dados en la aplicación principal.

Variables Uniformes

Se declaran de la forma habitual:

```
uniform type nombreVariable;
```

Por ejemplo

```
uniform vec4 myVar = {0.5, 0.2, 0.7, 1.0};
```

Esta sentencia asocia un tipo y un nombre a la variable, pero no le asigna un espacio de memoria salvo que se inicialice con unos valores, como en el ejemplo que hemos puesto. La asignación de un espacio de memoria sólo ocurre cuando se enlaza el programa. Mientras tanto, esta sentencia no es más que una declaración de intenciones.

¿Cómo sabe nuestro programa la dirección de la variable para poder asignarle datos? Para ello hay que saber el nombre que se le ha dado en el *shader,* y se obtiene su dirección como

```
GLint glGetUniformLocation(GLuint program, const char *name);
```

Veámoslo con un ejemplo. Si en el shader escribimos la declaración anterior de myVar (con o sin la incialización), podremos modificar sus valores de la siguientes dos maneras:

```
GLint myLoc = glGetUniformLocation(p, "myVar");
glProgramUniform4f(p, myLoc, 1.0f, 2.0f, 2.5f, 2.7f);
```

O bien

```
float myFloats[4] = {1.0f, 2.0f, 2.5f, 2.7f};
GLint myLoc = glGetUniformLocation(p, "myVar");
glProgramUniform4fv(p, myLoc, 1, myFloats);
```

Variables de atributo

Las variables de atributo son una manera de proporcionar datos por vértice al *vertex shader*, y sólo son usadas en este. Si por casualidad queremos que dichas propiedades se utilicen en etapas posteriores del cauce gráfico, hay que convertirlos en el *shader* en variables de salida.

Se declaran de la siguiente manera:

```
in tipo nombre;  // En OpenGL < 1. 3:  attribute  tipo  nombre;
```

Por ejemplo

```
in vec2 aUV;
```

Y, como ocurría con las variables uniformes, hay que obtener su localización en memoria después de enlazar los *shaders* y antes de poder asignarle cualquier valor con:

```
GLint glGetAttribLocation(GLuint program, const char *attribName);
```

GLSL es un lenguaje que va evolucionando con el tiempo. Hay una serie de variables predefinidas, que existen sí o sí, que se denominan de forma distinta:

OpenGL < 3.0	OpenGL >=3.0
gl_Vertex	aVertex
gl_Color	aColor
gl_Normal	aNormal
gl_ProjectionMatrix	uProjectionMatrix
gl_ModelViewMatrixInverse	uModelViewMatrixInverse
gl_ModelViewMatrix	uModelViewMatrix
gl_ModelViewProjectionMatrix	uModelViewProjectionMatrix
gl_NormalMatrix	uNormalMatrix

Además, desde OpenGL 3.2, es necesario usar atributos de vértice genéricos (no valen los predefinidos), e indicar su localización (un número entero que lo identifica en el contexto del shader) al declararlos:

```
layout(location = 2) in vec4 a_vec;
```

Y entonces, en el código de nuestro programa tras compilar y **antes de enlazar**, se ha de invocar a la siguiente función, una vez por cada atributo de vértice:

```
glBindAttribLocation( idProg, 2, "a_vec" );
```

Especificación de los atributos de vértice

Desde la aplicación se deben especificar las variables atributo de los vértices, normalmente como array, utilizando la siguiente instrucción:

```
void glVertexAttribPointer(GLuint index, GLint size,
                           GLenum type, GLboolean normalized,
                           GLsizei stride, const GLvoid *pointer);
```

Veámoslo en un ejemplo:

```
==== VERTEX SHADER =====
layout(0) vec2 coord2d;
layout(1) in vec3 v_color;
layout(2) out vec3 f_color;
void main(void) {
  gl_Position = vec4(coord2d, 0.0, 1.0);
  f_color = v_color;
}
==== FIN VERTEX SHADER =====

/*****/
// Variables globales o atributos de la clase
/*****/
GLuint vbo_triangle, vbo_triangle_colors;
GLint attribute_coord2d, attribute_v_color;

/*****/
// En la inicialización de los recursos
/*****/
GLfloat triangle_colors[] = {
    1.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    1.0, 0.0, 0.0,
};

glGenBuffers(1, &vbo_triangle_colors);
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle_colors);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(triangle_colors),
             triangle_colors,
             GL_STATIC_DRAW);

/*****/
En el método dibujar
/*****/
attribute_name = "v_color";
attribute_v_color = glGetAttribLocation(program, attribute_name);
if (attribute_v_color == -1) {
    fprintf(stderr, "Could not bind attribute %s\n", attribute_name);
    return 0;
 }


glEnableVertexAttribArray(attribute_v_color);
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle_colors);
glVertexAttribPointer(
    attribute_v_color, // atributo de vértice
    3,                // número de elementos por vértice (r,g,b)
    GL_FLOAT,          // tipo
    GL_FALSE,          // tomar los elementos tal cual
    0,                // sin datos extra entre uno y el siguiente
    0                 // desplazamiento del primer elemento
  );
```

BIBLIOGRAFÍA Y ENLACES DE INTERÉS

[Bailey2012] Mike Bailey, Steve Cunningham. **Graphics Shaders, Theory and Practice**, CRC Press, 2012

[Shreiner13] Shreiner, Dave et al. **OpenGL programming guide: the official guide to learning OpenGL, Versions 4.1.** Addison-Wesley 2013 (disponible online en la biblioteca)

https://developer.nvidia.com/content/gpu-gems

https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/

http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/

http://webglfundamentals.org/

http://www.html5rocks.com/en/tutorials/webgl/shaders/