Traducido del alemán al español - www.onlinedoctranslator.com

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

computación visual

Objetos gráficos y su programación.

Universidad de Darmstadt

Prof. Dr. Elke Hergenroether
Björn Frommer
Prof. Dr. Benjamín Meyer

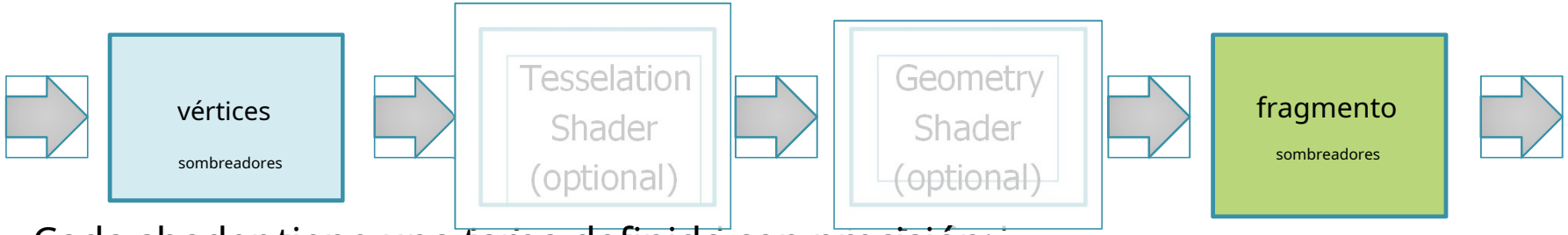
CAPÍTULO 6

programación de sombreadores

tubería de sombreado

- Los datos cargados todavía se pueden manipular antes de renderizar.
- Los cálculos deseados se realizan en los llamadossombreadoresimplementados, pequeños fragmentos de programa que se pueden ejecutar directamente en la GPU.

- Lenguaje informático de los programas shader: GLSL (OpenGL SteniendoLidioma)
  - Alternativas: HLSL para desarrollo puro de Windows/DirectX, Cg (descontinuado por NVidia, 2012)

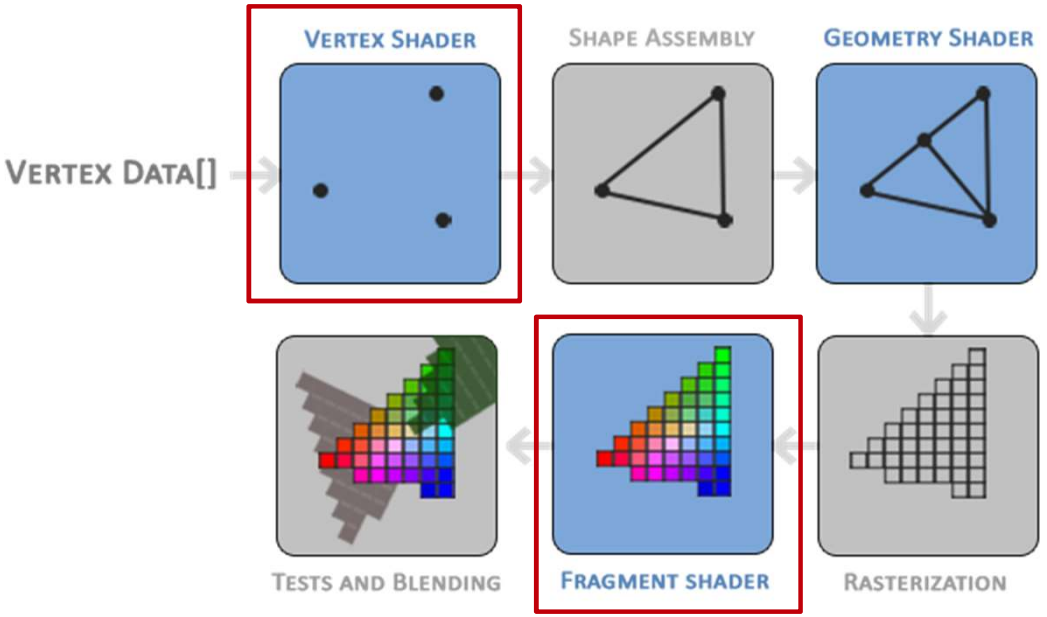


- Cada shader tiene una tarea definida con precisión:
  - Sombreadores de vértices:Calcula elposición finalcada vértice en la imagen de salida
  - Sombreadores de fragmentos:Calcula elcolor de cada píxelen la imagen de salida
    - (Shader de teselación: descomposición de los objetos en mallas triangulares más finas)
    - (Shader de geometría: cambiar los datos de geometría en tiempo de ejecución)

} Intercambio de espacio de almacenamiento versus rendimiento

tubería de sombreado

Sombreadores de vértices:Calcula elposición finalcada vértice en la imagen de salida



El sombreador de geometría es un sombreador opcional. Puede transformar primitivas individuales, como triángulos, y agregar vértices adicionales, por ejemplo.

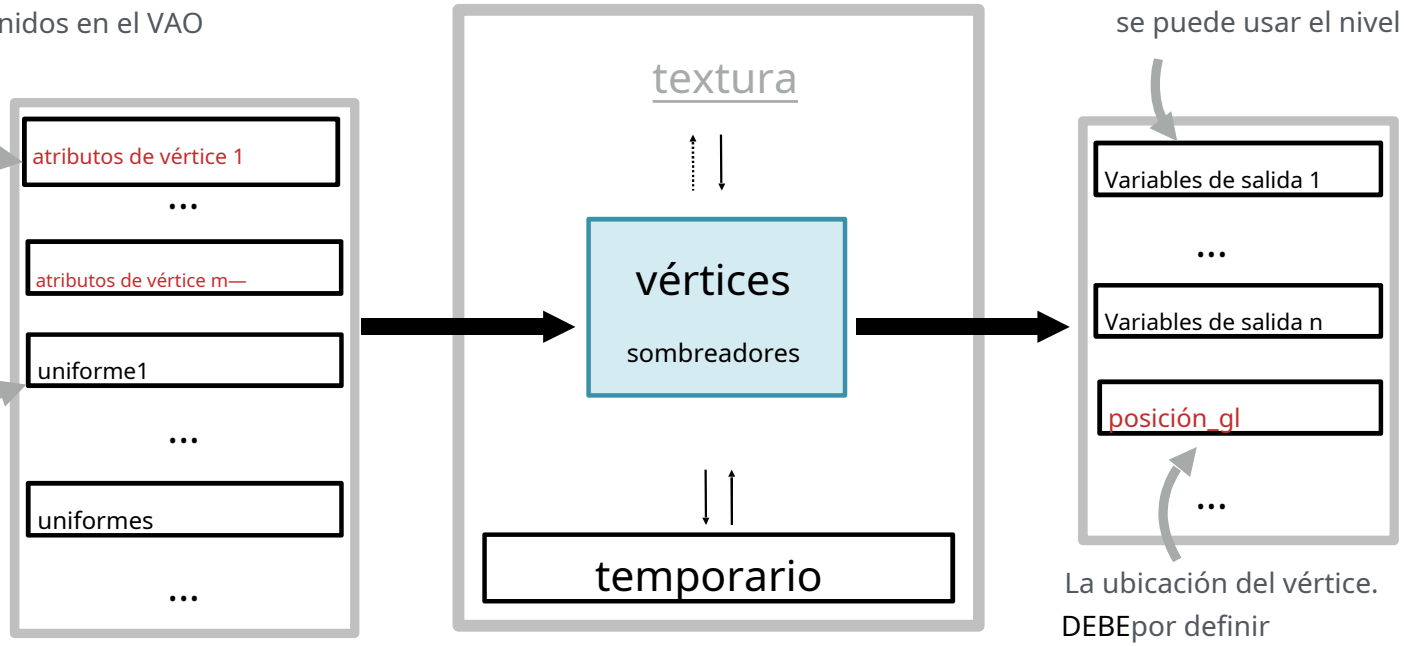
Sombreadores de fragmentos:
Calcula elcolor de cada píxel en la imagen de salida

De: https://learnopengl.com/Getting-started/Hello-Triangle

Etapa de sombreado de vértices

la entrada esun solo
vértices con todos sus
atributos definidos en el VAO

Se pueden definir múltiples salidas, una
como entrada para la siguiente
se puede usar el nivel de shader



La ubicación del vértice.
DEBEpor definir

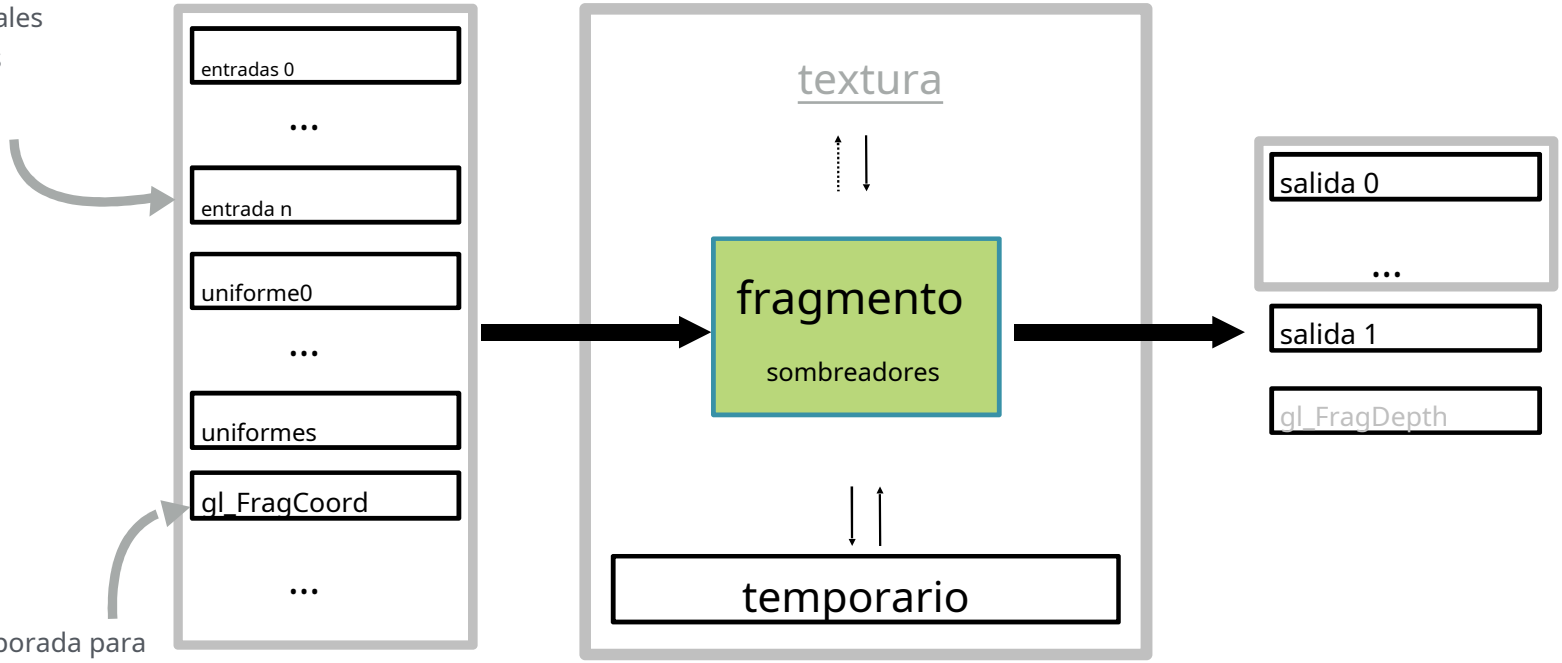
vértice adicional
variables independientes
puede cargar y
ser usado

programa Fpara un vértice cada uno

Simplificado: asigna el vértice de entrada a la pantalla

Etapa de sombreado de fragmentos

Recibe la salida como entrada
el nivel de shader anterior más
los adicionales opcionales
datos e integrados
Variables



Variable incorporada para
consultar la corriente
fragmentos (píxeles)

Programa para un fragmento a la vez!

Simplificado: define el color de salida para cada píxel

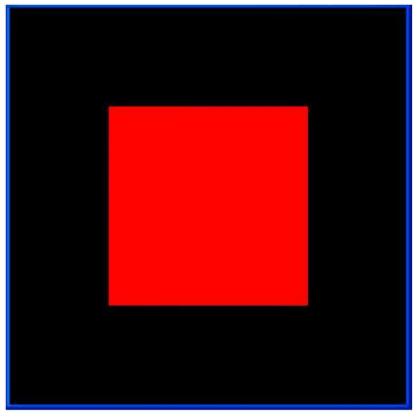
un ejemplo sencillo

sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0) en el vértice vec3;

modelo mat4 uniforme;

vacío principal()
{
        gl_Position = modelo * vec4(vértice, 1.0);
}
```

sombreadores de fragmentos

```
# versiones 330
fuera vec4 color;

vacío principal()
{
        color = vec4(1.0, 0.0, 0.0, 1.0);
}
```



producción

un ejemplo sencillo

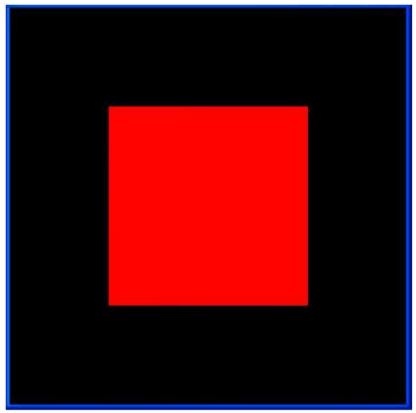
Definición del usado
Versión OpenGL/GLSL

sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0) en el vértice vec3;

modelo mat4 uniforme;

vacío principal()
{
        gl_Position = modelo * vec4(vértice, 1.0);
}
```

sombreadores de fragmentos

```
# versiones 330
fuera vec4 color;

vacío principal()
{
        color = vec4(1.0, 0.0, 0.0, 1.0);
}
```



producción

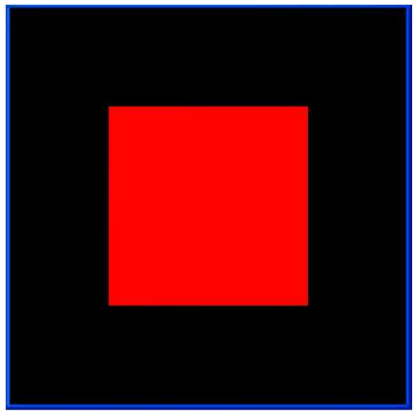
un ejemplo sencillo

sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0)en vec3 vértice;

modelo mat4 uniforme;

vacío principal()
{
        gl_Position = modelo * vec4(vértice, 1.0);
}
```

sombreadores de fragmentos

```
# versiones 330
fuera vec4 color;

vacío principal()
{
        color = vec4(1.0, 0.0, 0.0, 1.0);
}
```



producción

Forma general:

diseño (ubicación =índice) en nombre de tipo;

El índice se refiere a la respectiva VAO **atributo desactivado**

glVertexAttribPointer(GLuintíndice, ...)

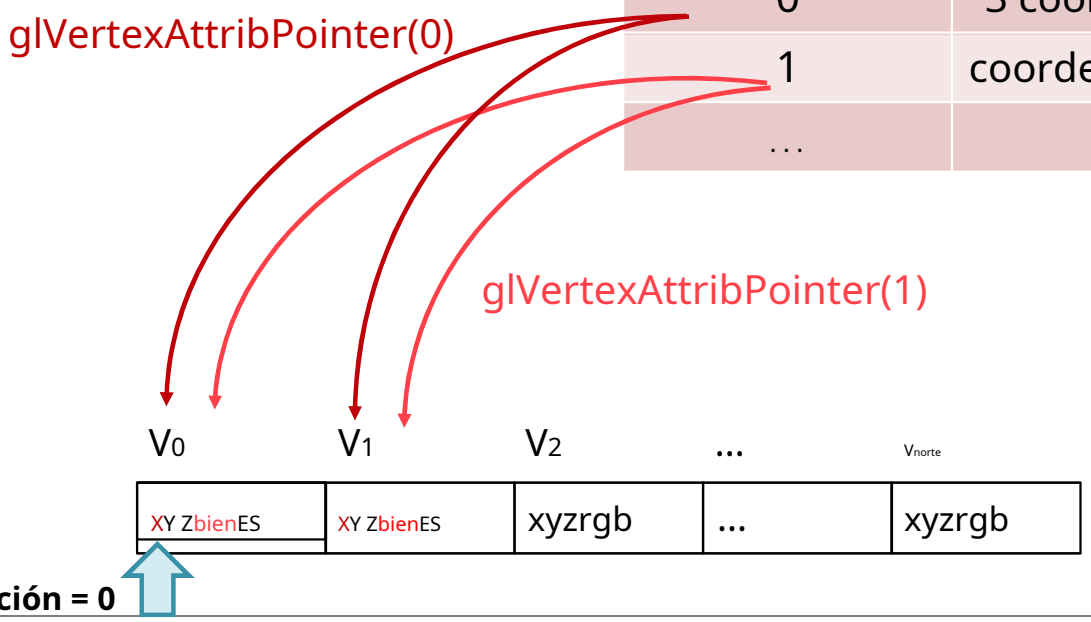
-El programador necesita saber qué contiene el atributo.

Objeto de búfer de vértices (VBO), Objeto de matriz de vértices (VAO) y VertexAttribPointer

Objeto de matriz de vértices (VAO)

Tarea:Conoce la estructura de los datos en el VBO. Los "VertexAttribPointers" son los objetos en esta matriz que almacenan/conocen la estructura de almacenamiento de los datos.

índice de atributos	Descripción
0	"3 coordenadas GL_FLOAT por vértice" "3
1	coordenadas GL_FLOAT por vértice" . . .
. . .	



Objeto de búfer de vértice (VBO)

Tarea:Recopila los datos y carga los datos de la CPU a la GPU, desde donde luego se visualiza, por ejemplo.

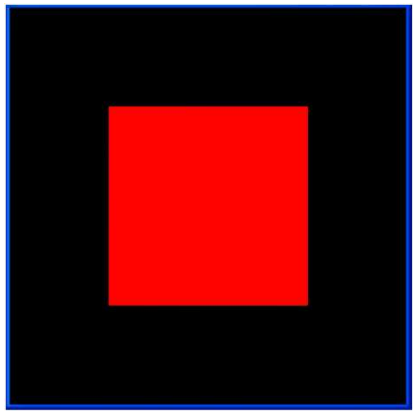
un ejemplo sencillo

sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0) envec3vértice;

uniformemat4modelo;

vacío principal()
{
        gl_Posición = modelo *vec4(vértice, 1.0);
}
```

sombreadores de fragmentos

```
# versiones 330
afueravec4color;

vacío principal()
{
        color =vec4(1,0, 0,0, 0,0, 1,0);
}
```



producción

tipos de datos

- punto flotante: flotar, vec2, vec3, vec4
- enteros: En t ivec2, ivec3, ivec4
- Booleano: booleano, bvec2, bvec3, bvec4
- Matriz: mat2 (2x2), mat3 (3x3), mat4 (4x4)

- Acceso a varios elementos del vector/matriz:
  . xyzw, .rgba, .stqp, [i], [i][j] (para matrices)

```
vec2 v2; vec3 v3; vec4 v4; v2.x

                                  //produce un flotador //Error:
v2.z                              indefinido para el tipo //da
v4.rgba                           un vec4 //da un vec2
v4.xy
v4.xgp                            //Error: componentes no coincidentes
```

Tipos de datos - Swizzling y Smearing

Valores R (lectura):

```
v4.wzyx      //   correcto, da vec4(w,z,y,x)
v4.xxx       //   correcto, da vec3(x,x,x)
v4.yyxx      //   correcto, duplicar xey da un vec4(y,y,x,x) Error:
v2.yyzz      //   demasiados componentes para el tipo
```

Valores L (escritura):

```
vec4 v4 = vec4( 1.0, 2.0, 3.0, 4.0); v4.xw =
vec2( 5.0, 6.0);                                  //5.0, 2.0, 3.0, 6.0) //(
v4.wx = vec2( 7.0, 8.0); v4.xx =                  8.0, 2.0, 3.0, 7.0)
vec2( 9.0,10.0); v4.yz = 11,0;                    //Error: x usado dos veces //
                                                  Error: tipo de discrepancia //(
v4.yz = vec2( 12.0 );                             8.0, 12.0, 12.0, 7.0)
```

Funciones predefinidas

Si es posible, use funciones integradas en su lugar ¡tu propio!

- Ángulos y trigonometría:
  - radianes, grados, sen, cos, tan, asen, acos, ...
- funciones exponenciales:
  - pow, exp, log, sqrt, ...
- Características generales:
  - abdominales, celular (redondea el valor de punto flotante transferido al siguiente número natural más alto),abrazadera (restringir un valor entre otros dos valores), mínimo máximo, ...
- Funciones geométricas:
  - cruzar, punto, longitud, normalizar, reflejar (Cálculo de la dirección de reflexión de un vector incidente (de luz), ...

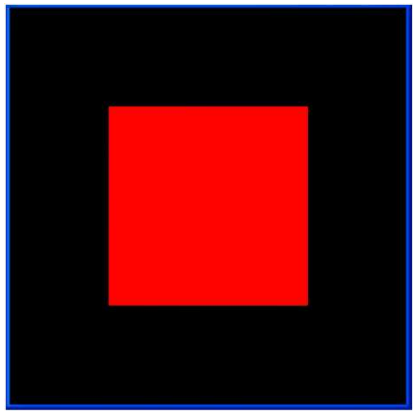
un ejemplo sencillo

sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0)envec3 vértice;

uniformemodelo mat4;

vacío principal()
{
        gl_Position = modelo * vec4(vértice, 1.0);
}
```

sombreadores de fragmentos

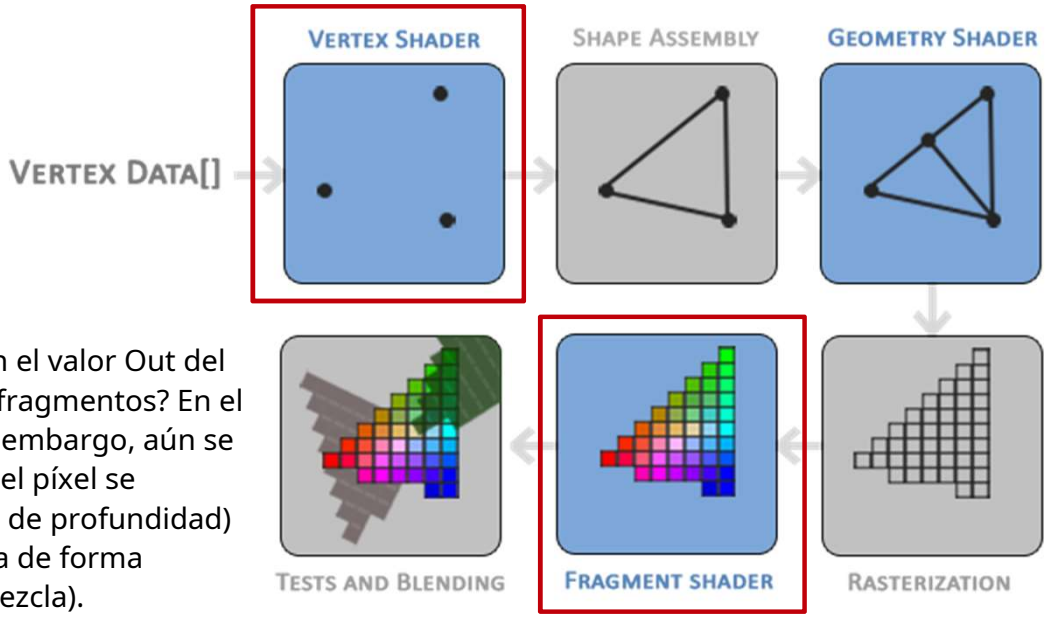
```
# versiones 330
afueravec4 color;

vacío principal()
{
        color = vec4(1.0, 0.0, 0.0, 1.0);
}
```



producción

tubería de sombreado

Sombreadores de vértices:Calcula elposición finalcada vértice en la imagen de salida



El sombreador de geometría es un sombreador opcional. Puede transformar primitivas individuales, como triángulos, y agregar vértices adicionales, por ejemplo.

¿Qué sucede con el valor Out del sombreador de fragmentos? En el último paso, sin embargo, aún se debe verificar si el píxel se procesa (prueba de profundidad) y/o si se muestra de forma transparente (mezcla).

Sombreadores de fragmentos:
Calcula elcolor de cada píxel en la imagen de salida

De: https://learnopengl.com/Getting-started/Hello-Triangle

tipos de almacenamiento

- Requerido para la comunicación entre shaders y la aplicación
- en
  - Enlace al shader de la etapa anterior
  - Entrada por vértice a OpenGL o sombreador de vértices de la aplicación (SOLO LECTURA) o: Entrada por fragmento a sombreador de fragmentos (SOLO LECTURA)
- afuera
  - Enlace desde el sombreador a la siguiente etapa
  - Pase el vértice (LEER/ESCRIBIR) al sombreador de fragmentos para la salida final, interpolada
- uniforme
  - Entrada a cualquier programa de sombreado desde OpenGL o cualquier aplicación (SOLO LECTURA)
  - constante durante el renderizado

uniformes

- Los uniformes son variables proporcionadas por el usuario desde la aplicación hasta los sombreadores.

- Subida de uniformes
  - Encuentra la ubicación de las variables en el programa shader
    - GLint glGetUniformLocation(ID del programa GLuint,const GLchar *nombre);
  - subir uniforme
    - void glUniform{1,2,3,4}{i,f}(ubicación de GLint, GLfloat v0[, v1, v2, v3]);
    - void glUniformMatrix{234}fv(ubicación de GLint, recuento de tamaño de GL, transposición de GLboolean, const Glfloat *valor);
    - contarespecifica el número de matrices a modificar. 1 si la variable de destino no es un campo de matriz y 1 o más si es una matriz de matrices.

direccionamiento desde
Variables de sombreado
por nombre de variable!

En nuestro marco: **(para matrices)**
ShaderProgram::setUniform(variable de cadena, valor de TIPO, [transposición bool]);

un ejemplo sencillo

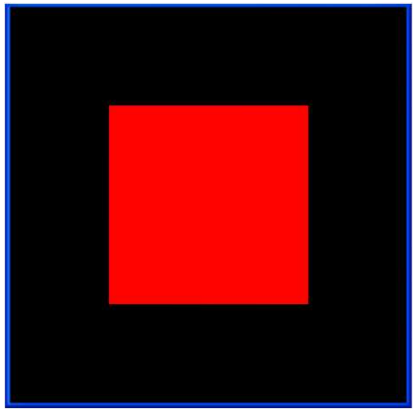
sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0) en el vértice vec3;

fuera vec4 pasar;

modelo mat4 uniforme;

vacío principal()
{
        gl_Position = modelo * vec4(vértice, 1.0); pasarEn =
        vec4(1.0, 0.0, 0.0, 1.0);
}
```

¡Mismo nombre y tipo de variable! →

sombreadores de fragmentos

```
# versiones 330
fuera vec4 color;

en vec4 paso;

vacío principal()
{
        color = pasar;
}
```



producción

un ejemplo sencillo

sombreadores de vértices

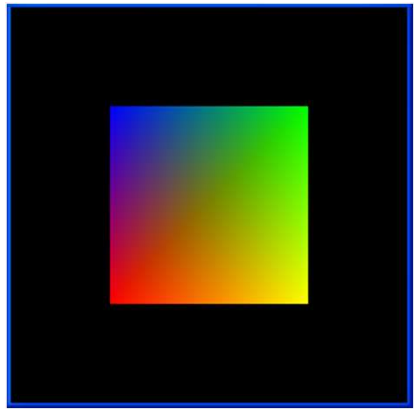
```
# versiones 330
diseño (ubicación = 0) en el vértice vec3; diseño
(ubicación = 1) en color vec3;

fuera vec4 pasar;

modelo mat4 uniforme;

vacío principal()
{
        gl_Position = modelo * vec4(vértice, 1.0); passOn =
        vec4(color, 1,0);
}
```

interpolación automática

entre los datos del vértice!

sombreadores de fragmentos

```
# versiones 330
fuera vec4 color;

en vec4 paso;

vacío principal()
{
        color = pasar;
}
```



producción

un ejemplo sencillo

sombreadores de vértices

```
# versiones 330
diseño (ubicación = 0) en el vértice vec3; diseño
(ubicación = 1) en color vec3;

fuera vec4 pasar;

modelo mat4 uniforme;

vacío principal()
{
        posición_gl=modelo *vec4(vértice, 1.0); passOn =
        vec4(color, 1.0);
}
```

Multiplicando la antigua posición del vértice por la matriz de transformación acumulada

Cualquier sombreador de vértices DEBE ¡llene la variable (incorporada) gl_Position con la posición del vértice homogéneo!

sombreadores de fragmentos

```
# versiones 330
fuera vec4 color;

en vec4 paso;

vacío principal()
{
        color=Transmitir;
}
```

Definición de las variables de salida

Cualquier sombreador de fragmentos DEBE definir al menos una variable de salida y esta con la final ¡Color de relleno del fragmento!

Uniformes - Diapositiva del Capítulo 6

- Los uniformes son variables proporcionadas por el usuario desde la aplicación hasta los sombreadores.

- Subida de uniformes
  - Encuentra la ubicación de las variables en el programa shader
    - GLint glGetUniformLocation(ID del programa GLuint,const GLchar *nombre);
  - subir uniforme
    - void glUniform{1,2,3,4}{i,f}(ubicación de GLint, GLfloat v0[, v1, v2, v3]);
    - void glUniformMatrix{234}fv(ubicación de GLint, recuento de tamaño de GL, transposición de GLboolean, const Glfloat *valor);
    - contarespecifica el número de matrices a modificar. 1 si la variable de destino no es un campo de matriz y 1 o más si es una matriz de matrices.

direccionamiento desde
Variables de sombreado
por nombre de variable!

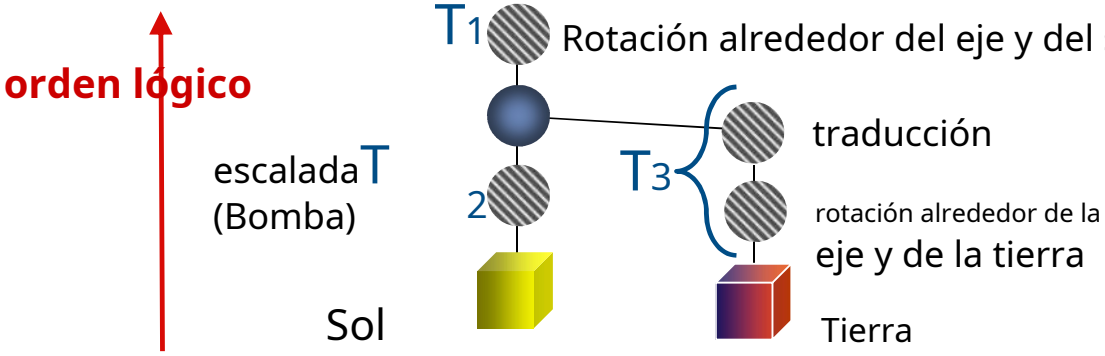
En nuestro marco: **(para matrices)**
ShaderProgram::setUniform(variable de cadena, valor de TIPO, [transposición bool]);

Scenegraph - implementación de muestra

pseudocódigo:

```
Matriz4f T1 = nueva Matriz4f(); T1.rotar( alfa,
0.0, 1.0, 0.0);

Matriz4f T2 = nueva Matriz4f(); escala
T2( s, s, s);

Matriz4f T3 = nueva Matriz4f();
T3.traducir(p1, p2, p3); T3.rotar( beta, 0.0,
1.0, 0.0);
```

```
// crea una nueva matriz T1
// rotación con ángulo alfa alrededor del vector (0,1,0)t

// crea una nueva matriz T2
// incluso escalar por factor s

// crea una nueva matriz T3
// traducción relativa al sol, posición (p1, pag2, pag3) // rotación con
ángulo beta alrededor del vector (0,1,0)t
```



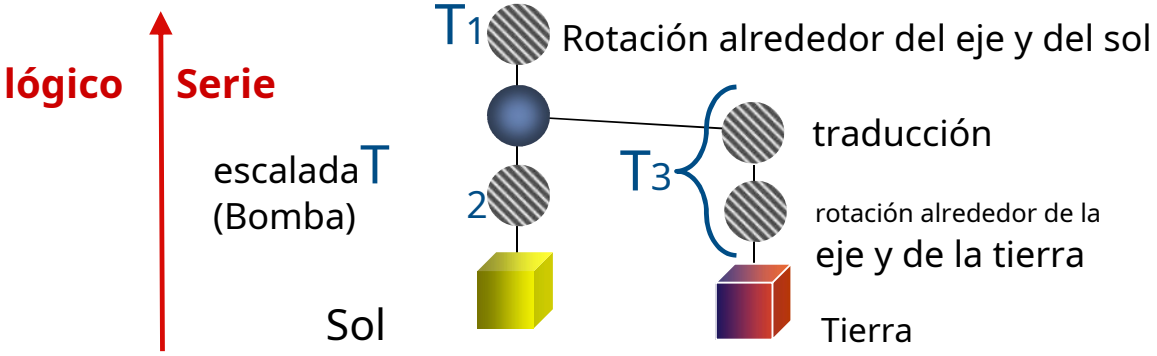
Matriz acumulada para..

la tierra: $METRO_{mi} = T1 * T3;$

el sol: $METRO_{s} = T1 * T2;$

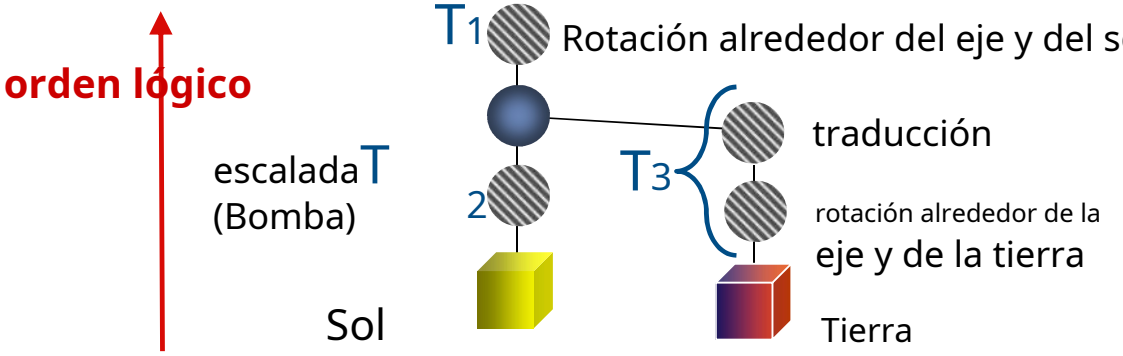
extracto deescena vacía::render(float dt)enEscena.cpp

```
// Primero, las matrices T1, T2 y T3 se llenan con los valores apropiados T1 = nueva Transformada; //
Crea la matriz identidad T1->rotate(glm::vec3(0, 0.2 * dt, 0));// Rotación En El Eje Y.

                                                    // El ángulo aumenta con by dt

T2 = nueva Transformada; // Genera la matriz identidad T2->scale(glm::vec3(scaling*dt, scaling*dt,
scaling*dt));// dt cambia la escala

T3 = nueva Transformada; // Genera la matriz identidad T3->translate(glm::vec3(0.8f, 0, 0));//
escalado único de la tierra T3->rotar(glm::vec3(0, 0.4f*dt, 0));
```



extracto de códigoEscena.cpp

```
//Cargar los shaders en: bool Escena::init ()
m_assets.addShaderProgram("shader",AssetManager::createShaderProgram("...vertex.glsl",
                                                                      ".../fragmento.glsl"));
m_shader = m_assets.getShaderProgram("shader"); m_shader-
>uso();
// En: escena vacía::render(float dt)
// Implementación de la rama izquierda del scenegraph - Sun: T1*T2 m_shader->setUniform("mm", T1-
>getTransformMatrix() * T2->getTransformMatrix(), false); // Implementación de la rama izquierda del scenegraph -
Tierra : T1*T3 m_shader->setUniform("mm", T1->getTransformMatrix() * T3->getTransformMatrix(), falso),
```



Las matrices se acumulan utilizando los parámetros uniformes; consulte el Capítulo 6

la tierra: $METRO_{mi} = T1 * T3;$

el sol: $METRO_{s} = T1 * T2;$