

GPU, OpenGL & Shaders

Nancy Hitschfeld K.

Daniel Calderón S.

9 Septiembre, 2020

GPU

Artículo

Discusión

Leer

Editar

Ver historial

Buscar en Wikipedia



Unidad de procesamiento gráfico

(Redirigido desde «[GPU](#)»)


Para otros usos de este término, véase [GPU \(desambiguación\)](#).

Unidad de procesamiento gráfico o **GPU** (*Graphics Processing Unit*) es un [coprocesador](#) dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del [procesador](#) central en aplicaciones como los videojuegos o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la *GPU*, la [unidad central de procesamiento](#) (*CPU*) puede dedicarse a otro tipo de cálculos (como la [inteligencia artificial](#) o los cálculos [mecánicos](#) en el caso de los [videojuegos](#)).

La *GPU* implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico. Una de las primitivas más comunes para el procesamiento gráfico en 3D es el [antialiasing](#), que suaviza los bordes de las figuras para darles un aspecto más realista. Adicionalmente existen primitivas para dibujar rectángulos, triángulos, círculos y arcos. Las GPU actualmente disponen de gran cantidad de primitivas, buscando mayor realismo en los efectos.

Las *GPU* están presentes en las [tarjetas gráficas](#).

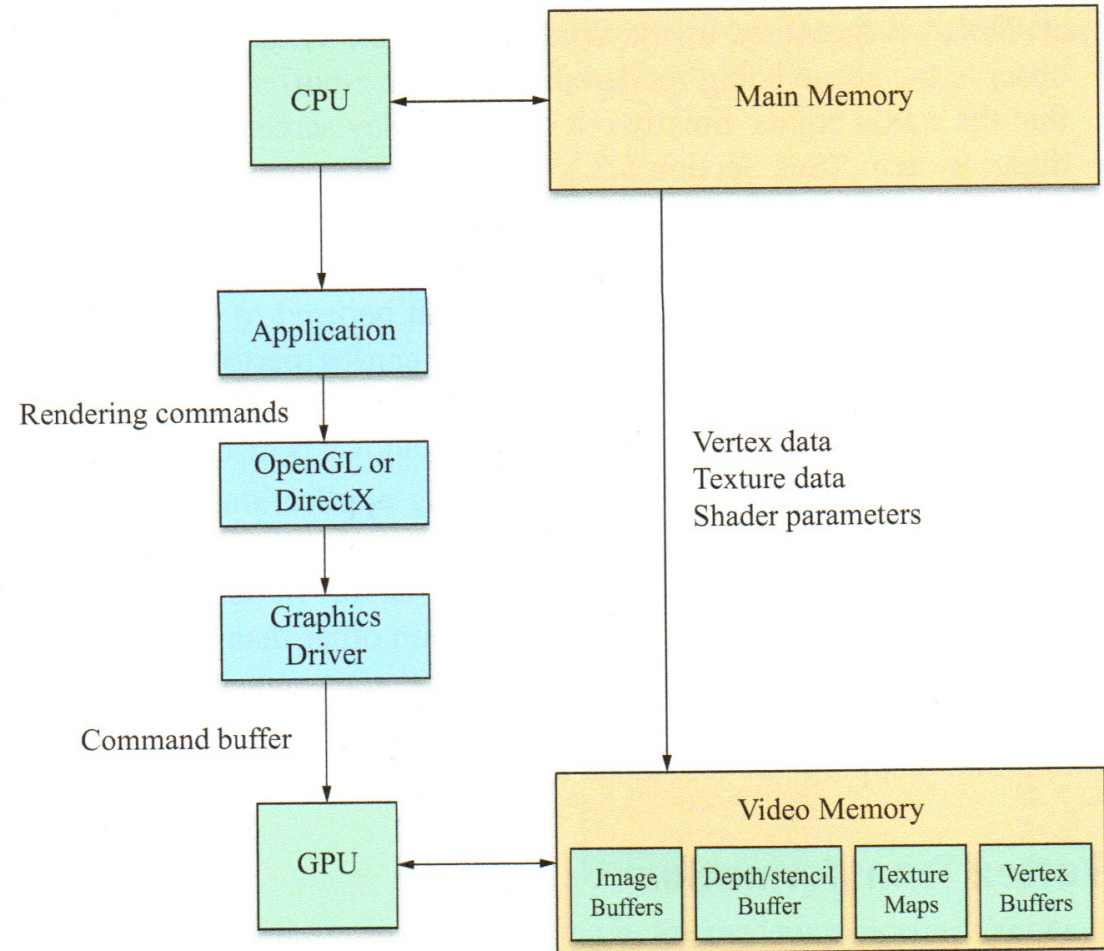


Unidad de procesamiento gráfico. 

[https://es.wikipedia.org/wiki/Unidad de procesamiento gr%C3%A1fico](https://es.wikipedia.org/wiki/Unidad_de_procesamiento_gr%C3%A1fico)

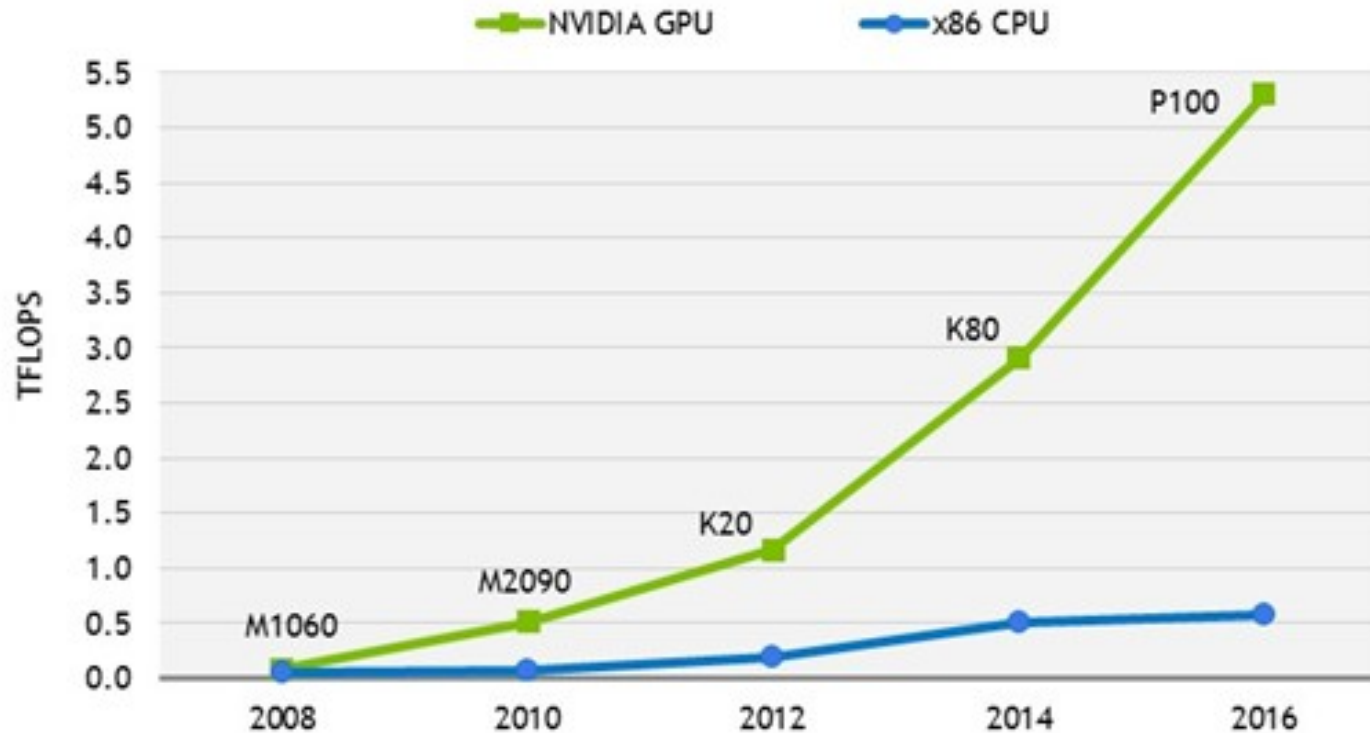
GPU: Graphics Processing Unit

- Hardware dedicado al procesamiento de gráficos
- Distintos fabricantes:
 - NVIDIA
 - AMD
 - Intel
 - Etc...
- Implementan especificaciones de OpenGL u otra API gráfica



Evolución de las GPU

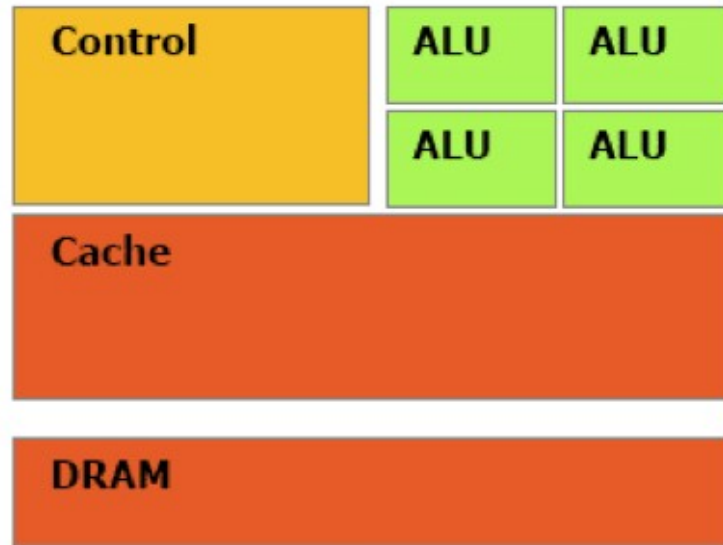
- La industria gráfica es dinámica
- Constantemente hay nuevas posibilidades y requerimientos



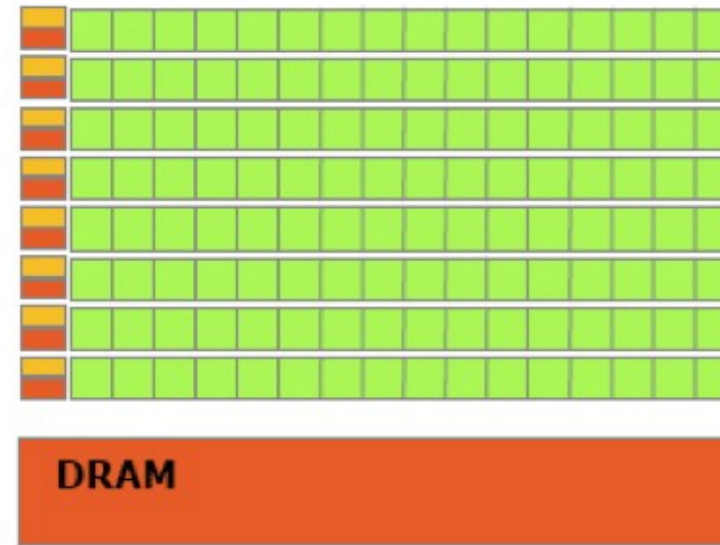
Teraflop:

Unidad de velocidad de cómputo igual a un millón de millones () de operaciones de punto flotante por segundo.

GPU vs CPU

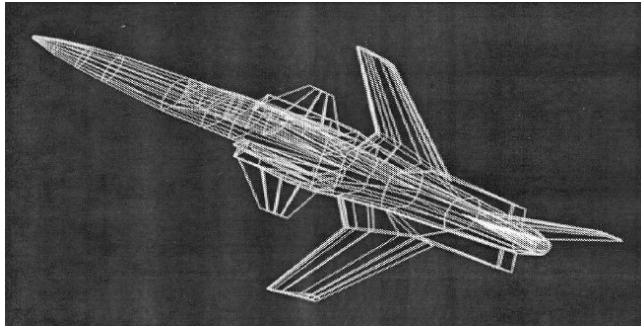


CPU



GPU

Generaciones de GPUs



Primera Generación:
Wireframe

- Transform, clip and project
- Solo líneas, no hay píxeles
- Previo a 1987



Segunda Generación:
Shaded solids

- Lightning!
- Filled polygons
- Depth buffer y color blending
- 1987 – 1992



Tercera Generación

- Todo más rápido
- Filtrado de texturas y antialiasing
- 1992-2001

GPUs: Fijas vs Programables

- Primeras GPUs presentaban funcionalidades fijas y muy limitadas
 - Definido por cada API
 - Ejemplo: Nintendo Wii (2006)
- GPUs modernas son altamente programables, permitiendo mayor flexibilidad

OpenGL



WIKIPEDIA
La enciclopedia libre

[Portada](#)

[Portal de la comunidad](#)

[Actualidad](#)

[Cambios recientes](#)

[Páginas nuevas](#)

[Página aleatoria](#)

[Ayuda](#)

[Donaciones](#)

[Notificar un error](#)

[Imprimir/exportar](#)

Artículo

Discusión

Leer

Editar

Ver

OpenGL

No debe confundirse con [OpenCL](#).

OpenGL (**Open Graphics Library**) es una especificación estándar que define una [API](#) multilenguaje y [multiplataforma](#) para escribir aplicaciones que produzcan gráficos [2D](#) y [3D](#). La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por [Silicon Graphics Inc. \(SGI\)](#) en 1992² y se usa ampliamente en [CAD](#), [realidad virtual](#), representación científica, visualización de información y [simulación de vuelo](#). También se usa en [desarrollo de videojuegos](#), donde compite con [Direct3D](#) en plataformas Microsoft Windows.

<https://es.wikipedia.org/wiki/OpenGL>

Qué es OpenGL?

- OpenGL es una de tantas especificaciones que nos permiten controlar la GPU
- Ejemplos de otras especificaciones:



OpenCL

Qué es OpenGL?

- OpenGL es una especificación
- No es “Código abierto” (*open source*)
 - Existen implementaciones que sí lo son
- Su fama se debe a que es multilenguaje y multiplataforma
 - Windows, Linux, MacOS.
- La especificación es implementada por los fabricantes de tarjetas de video: NVIDIA, AMD, etc...
- Entonces, se descarga?, se instala?
 - No!, es parte integral de los drivers de su tarjeta de video

OpenGL

- El objetivo de OpenGL es proveer una capa de abstracción del sub sistema gráfico
 - i.e. ejecutando una misma secuencia de instrucciones, se obtiene el “mismo” resultado, independiente del hardware subyacente
- Permite que el programa no sepa del hardware sobre el cual se ejecuta
- OpenGL intenta encontrar un balance entre alto y bajo nivel de abstracción
 - Debe ocultar procedimientos específicos al hardware
 - Debe permitir sacar un máximo provecho de él

Origen de OpenGL

- OpenGL nace de Silicon Graphics Inc. (SGI) y su [*IRIS GL*](#)
 - Integrated Raster Imaging System Graphics Library
- GL viene de *Graphics Library*
- Silicon Graphics Inc. Era una fábrica de estaciones de trabajo de alto rendimiento
 - Utilizar APIs propietarias encarecía aún más el trabajo
- A principios de los 90', en una valoración de la portabilidad, SGI modifica IRIS GL removiendo partes específicas de sistema y la libera como OpenGL (1992)

Origen de OpenGL (cont)

- OpenGL puede ser implementada sin pagar royalty por cualquiera ^{^^}
- SGI también establece el Architectural Review Board (ARB)
 - Miembros iniciales: Compaq, DEC, IBM, Intel y Microsoft
 - Diseña, gobierna y produce la especificación de OpenGL
 - Hoy en día ARB es parte del grupo Khronos (supervisa varios estándares libres)
- OpenGL tiene más de 25 años!

Versiones de OpenGL

Se introduce
modelo de
deprecación

Versión	Año
OpenGL 1.0	1992
OpenGL 1.1	1997
OpenGL 1.2	1998
OpenGL 1.2.1	1998
OpenGL 1.3	2001
OpenGL 1.4	2002
OpenGL 1.5	2003
OpenGL 2.0	2004
OpenGL 2.1	2006

Shaders! →

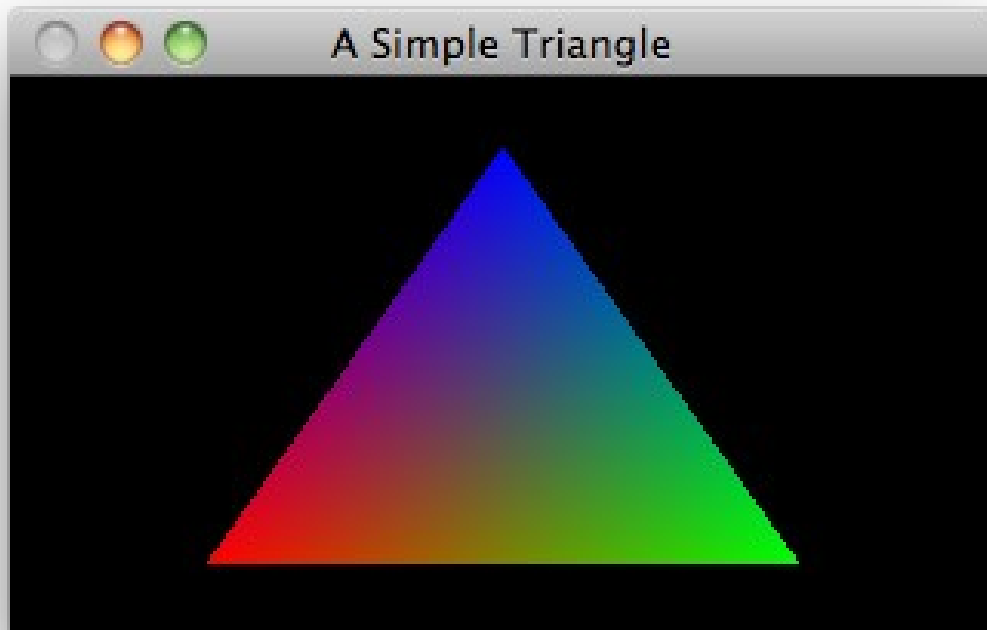
Versión	Año
OpenGL 3.0	2008
OpenGL 3.1	2009
OpenGL 3.2	2009
OpenGL 3.3	2010
OpenGL 4.0	2010
OpenGL 4.1	2010
OpenGL 4.2	2011
OpenGL 4.3	2012
OpenGL 4.4	2013
OpenGL 4.5	2014
OpenGL 4.6	2017



OpenGL

- Compatibility Profile
 - Pipeline con funciones fijas
- Core Profile
 - Pipeline con funcionalidades programables (shaders)
 - Mayor libertad para utilizar la GPU

Ejemplo: OpenGL Compatibility Profile



```
1 #include <GLUT/glut.h>
2 #include <GL/glut.h>
3
4 // Clears the current window and draws a triangle.
5 void display() {
6
7     // Set every pixel in the frame buffer to the current clear color.
8     glClear(GL_COLOR_BUFFER_BIT);
9
10    // Drawing is done by specifying a sequence of vertices. The way these
11    // vertices are connected (or not connected) depends on the argument to
12    // glBegin. GL_POLYGON constructs a filled polygon.
13    glBegin(GL_POLYGON);
14        glColor3f(1, 0, 0); glVertex3f(-0.6, -0.75, 0.5);
15        glColor3f(0, 1, 0); glVertex3f(0.6, -0.75, 0);
16        glColor3f(0, 0, 1); glVertex3f(0, 0.75, 0);
17    glEnd();
18
19    // Flush drawing command buffer to make drawing happen as soon as possible.
20    glFlush();
21 }
22
23 // Initializes GLUT, the display mode, and main window; registers callbacks;
24 // enters the main event loop.
25 int main(int argc, char** argv) {
26
27     // Use a single buffered window in RGB mode (as opposed to a double-buffered
28     // window or color-index mode).
29     glutInit(&argc, argv);
30     glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
31
32     // Position window at (80,80)-(480,380) and give it a title.
33     glutInitWindowPosition(80, 80);
34     glutInitWindowSize(400, 300);
35     glutCreateWindow("A Simple Triangle");
36
37     // Tell GLUT that whenever the main window needs to be repainted that it
38     // should call the function display().
39     glutDisplayFunc(display);
40
41     // Tell GLUT to start reading and processing events. This function
42     // never returns; the program only exits when the user closes the main
43     // window or kills the process.
44     glutMainLoop();
45 }
```

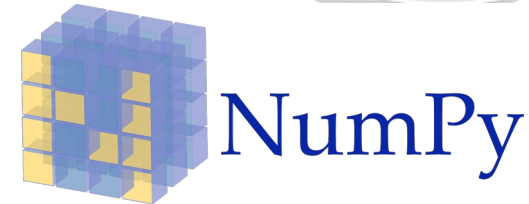
Ventanas y eventos

- OpenGL solo se encarga de la parte de gráfica controlando la GPU
- Una aplicación completa requiere además:
 - Ventana de visualización
 - Interacción con dispositivos periféricos (mouse, teclado, joystick, etc...)
- Para las tareas anteriores se utilizan otras librerías
 - GLUT (obsoleta)
 - QT
 - HTML/javascript
 - GLFW (actualmente utilizado en libros oficiales de OpenGL)

CC3501 Today

- En esta edición del curso utilizaremos:

- Python 3.5
- OpenGL Core Profile
 - ~ i.e. utilizaremos shaders
 - ~ source ~/python-cg/bin/activateObs: NO programaremos shaders, se darán
- GLFW para el manejo de ventanas y eventos
- Numpy para el manejo numérico



Nota:

Hasta el 2018 se utilizó en el curso:

- OpenGL Compatibility Profile
- PyGame para el manejo de ventanas y eventos

Pipeline Programable

- El mayor avance en los gráficos en tiempo real es el pipeline programable
- Introducido en la NVIDIA GeForce 3 (2001)
- Soportado en todas las tarjetas de video high-end modernas
 - NVIDIA, AMD, Intel

Shaders

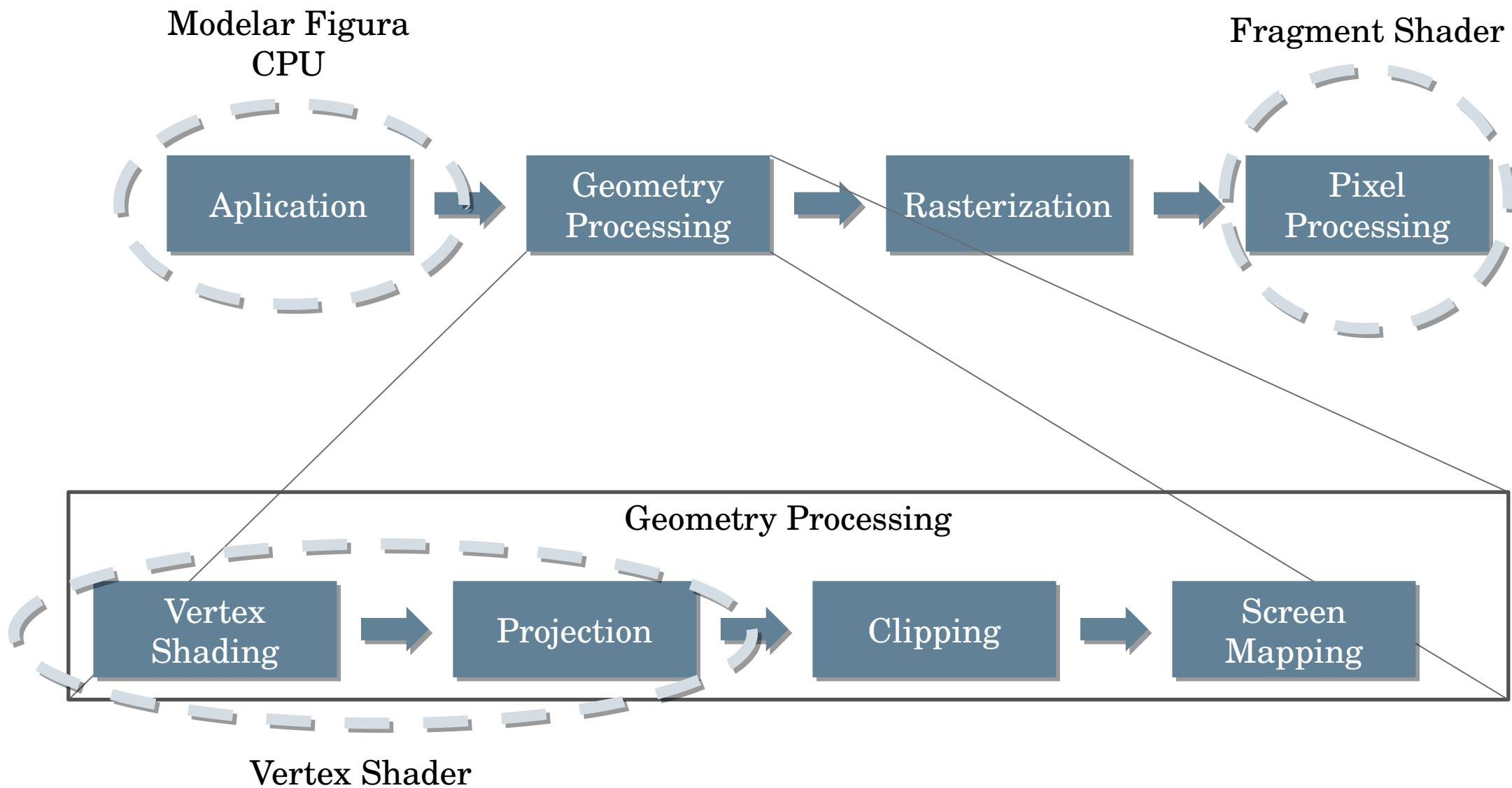
Shaders

- Shaders son programas personalizados que ejecutan partes del pipeline de rendering gráfico
- Muchos efectos no son posibles en el pipeline fijo de OpenGL
- Se ejecutan en la GPU
- Se programan en un lenguaje de shaders:
 - GLSL: OpenGL Shading Language (OpenGL)
 - HLSL: High Level shading Language (DirectX)
 - CG: C for Graphics (Nvidia, deprecado en 2012)



Shaders

- Hay varios tipos de shaders dependiendo la etapa del pipeline gráfico que se quiera modificar
- Utilizaremos solo 2 tipos de shaders:
 - Vertex Shader
 - Fragment Shader
- Pasos intermedios son ejecutados por los procesos por defecto de la GPU
- Ejemplos de programas en GLSL
 - <http://glslsandbox.com/>
 - *Sky is the limit!*

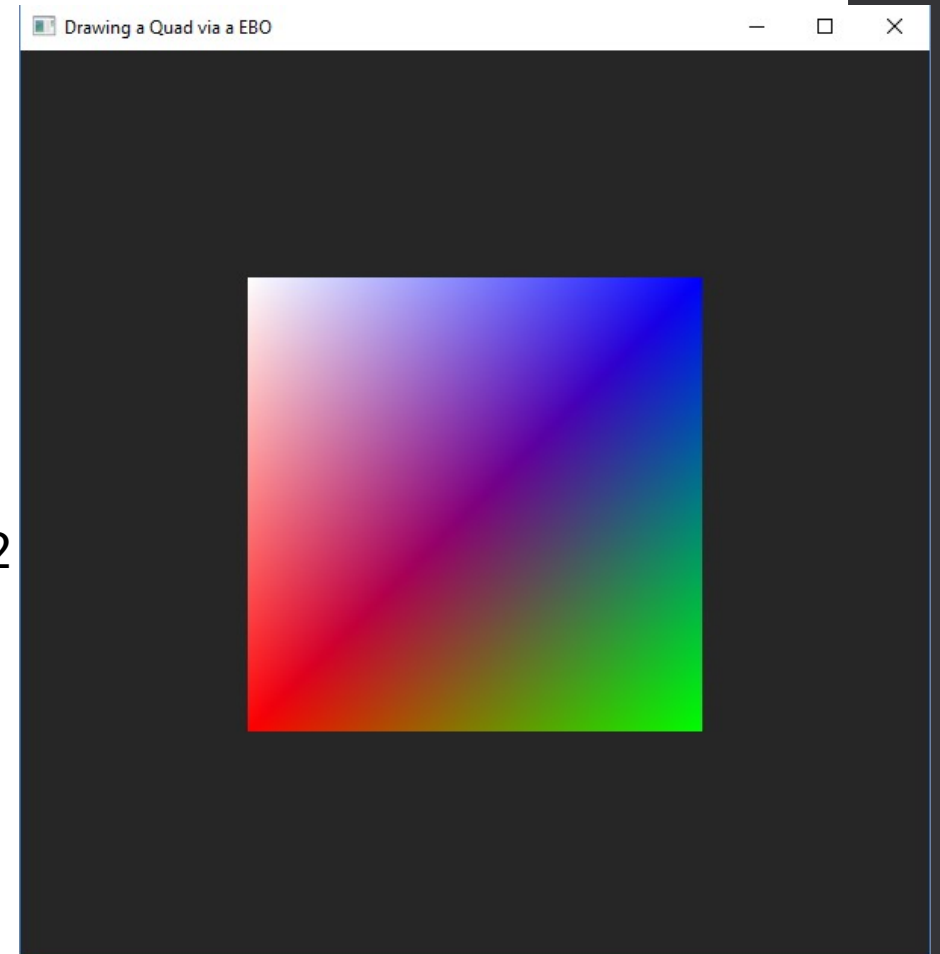


Describiendo una figura

```
# Defining data for each vertex
#
#           positions           colors
vertexData = [-0.5, -0.5, 0.0, 1.0, 0.0, 0.0,
              0.5, -0.5, 0.0, 0.0, 1.0, 0.0,
              0.5, 0.5, 0.0, 0.0, 0.0, 1.0,
              -0.5, 0.5, 0.0, 1.0, 1.0, 1.0]

# It is important to use 32 bits data
vertexData = np.array(vertexData, dtype = np.float32)

# Defining connections among vertices
# We have a triangle every 3 indices specified
indices = np.array(
    [0, 1, 2,
     2, 3, 0], dtype= np.uint32)
```

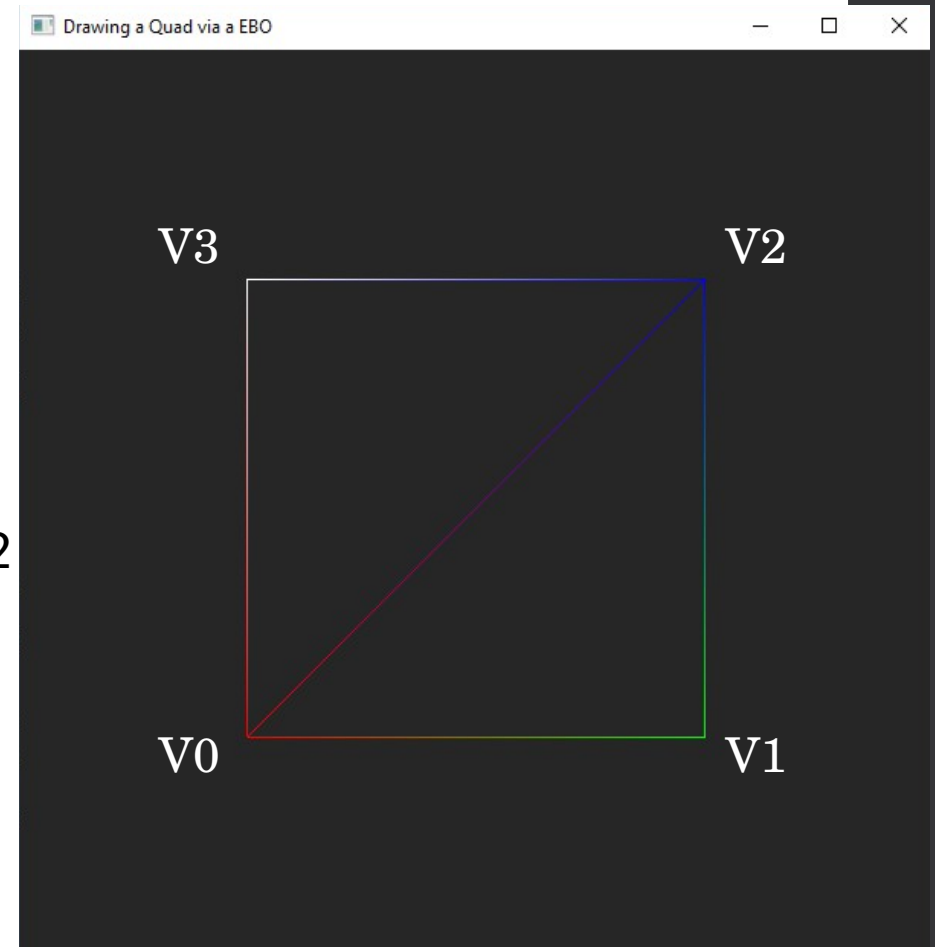


Describiendo una figura

```
# Defining data for each vertex
#
#           positions           colors
vertexData = [-0.5, -0.5, 0.0, 1.0, 0.0, 0.0,
              0.5, -0.5, 0.0, 0.0, 1.0, 0.0,
              0.5, 0.5, 0.0, 0.0, 0.0, 1.0,
              -0.5, 0.5, 0.0, 1.0, 1.0, 1.0]

# It is important to use 32 bits data
vertexData = np.array(vertexData, dtype = np.float32)

# Defining connections among vertices
# We have a triangle every 3 indices specified
indices = np.array(
    [0, 1, 2,
     2, 3, 0], dtype= np.uint32)
```



Estructurando la información a renderizar

- Vertex Array Object (VAO)
 - 1 para cada figura distinta
- Vertex Buffer Object (VBO)
 - 1 para cada objeto de la figura con características específicas
 - Un VAO puede tener varios VBO asociados
- Element Buffer Object (EBO)
 - Enlazado de la data del VBO
 - Permite reutilizar la data asociada a los vértices ya descritas, previniendo errores
 - Su uso no es estrictamente necesario

Estructurando la información a renderizar

```
# VAO, VBO and EBO and for the shape
```

```
vao = glGenVertexArrays(1)
```

```
vbo = glGenBuffers(1)
```

```
ebo = glGenBuffers(1)
```

```
# Vertex data must be attached to a Vertex Buffer Object (VBO)
```

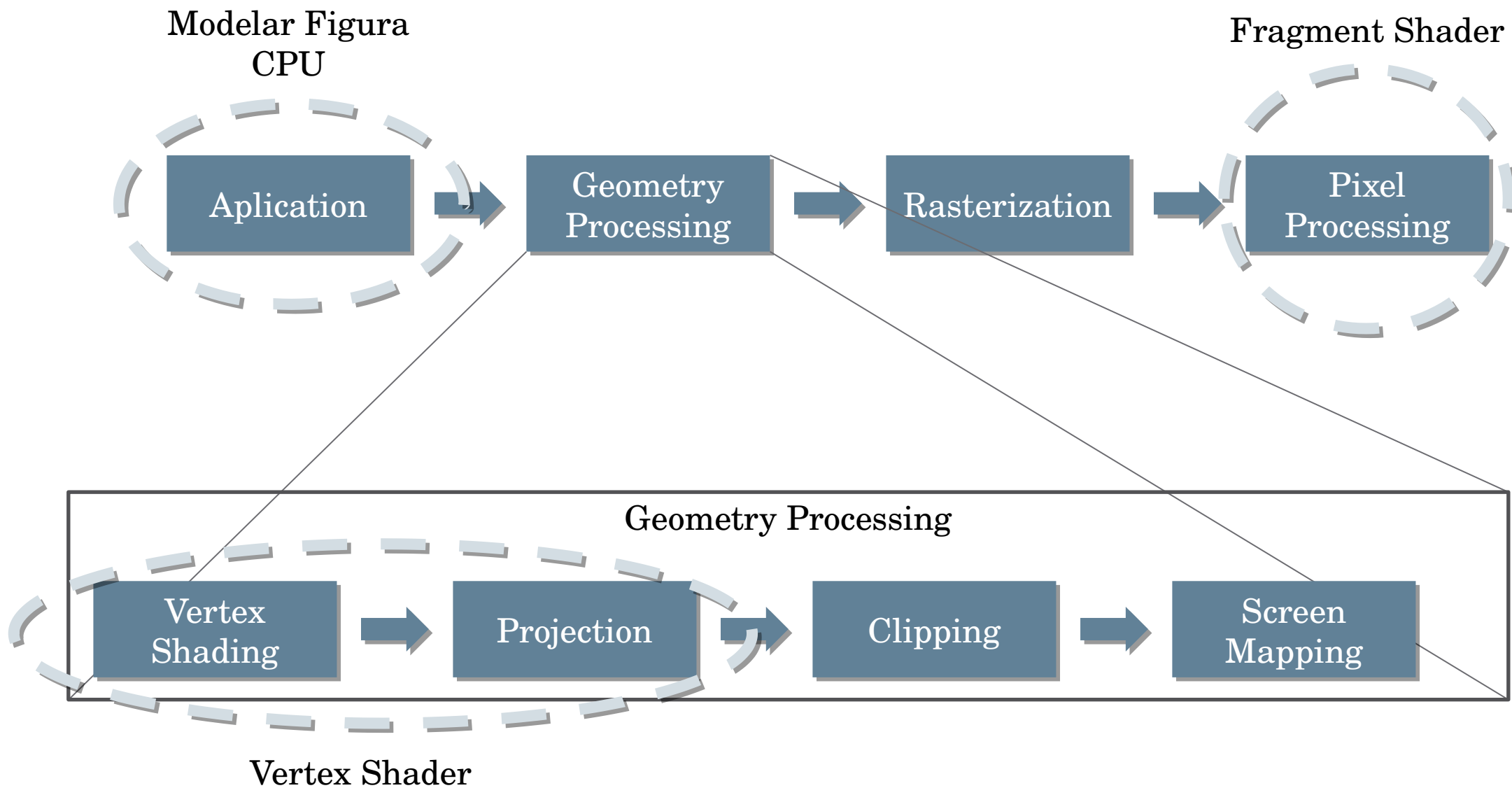
```
glBindBuffer(GL_ARRAY_BUFFER, vbo)
```

```
glBufferData(GL_ARRAY_BUFFER, len(vertexData) * INT_BYTES, vertexData,  
GL_STATIC_DRAW)
```

```
# Connections among vertices are stored in the Elements Buffer Object (EBO)
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)
```

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, len(indices) * INT_BYTES, indices,  
GL_STATIC_DRAW)
```



Vertex Shader

- Etapa inicial en el pipeline de rendering
- 2 tareas:
 - Calcula la posición de cada vértice
 - Evalúa la data en los vértices
 - ⋈ Normal
 - ⋈ Color / Material
 - ⋈ Coordenadas de textura
- Entradas:
 - Vértices en coordenadas de objetos
 - Atributos asociados a cada vértice
 - ⋈ Color
 - ⋈ Normal
 - ⋈ Coordenadas de textura
 - ⋈ Etc.
- Salidas:
 - ▢ Localización del vértice en coordenadas de clipping (-1 a 1 en los 3 ejes)
 - ▢ Color del vértice
 - ▢ Normal en el vértice
 - ▢ Etc.

Vertex Shader básico en GLSL

```
vertex_shader = ""  
    #version 130  
    in vec3 position;  
    in vec3 color;  
  
    out vec3 fragColor;  
  
    void main()  
    {  
        fragColor = color;  
        gl_Position = vec4(position, 1.0f);  
    }  
    ""
```

Fragment Shader

- Etapa que viene luego de la rasterización
- i.e. se tienen píxeles tentativos
- Cada uno contiene la información de los vértices interpolada
 - Color
 - Normal
 - Coordenadas de textura
 - Etc. (usos creativos)
- Entradas:
 - Salidas del vertex shader interpoladas por la GPU
- Salidas:
 - Color del píxel
 - Valor de profundidad
 - Posibilidad de descartar el pixel

Fragment Shader básico en GLSL

```
fragment_shader = ""  
#version 130  
  
in vec3 fragColor;  
out vec4 outColor;  
  
void main()  
{  
outColor = vec4(fragColor, 1.0f);  
}  
""
```

Pipeline Program

- Ensamblaje de shaders creado un pipeline de rendering
- Es posible tener muchos pipelines distintos
 - Distintos estilos de rendering
 - Ejemplos:
 - ⋈ Mañana, medio día, atardecer y noche
 - ⋈ Iluminación “realista” o colores simplificados
 - ⋈ Visión “infraroja”
 - ⋈ Reflexiones, agua, neblina, etc.
- En cualquier momento, solo un pipeline puede estar activo
- Una misma escena puede contener elementos renderizados con distintos pipelines

Creando un Shader Program

```
# Assembling the shader program (pipeline) with both shaders
shaderProgram = OpenGL.GL.shaders.compileProgram(
    OpenGL.GL.shaders.compileShader(vertex_shader, GL_VERTEX_SHADER),
    OpenGL.GL.shaders.compileShader(fragment_shader, GL_FRAGMENT_SHADER))

# Telling OpenGL to use our shader program
glUseProgram(shaderProgram)
```

Renderizando con un shader program

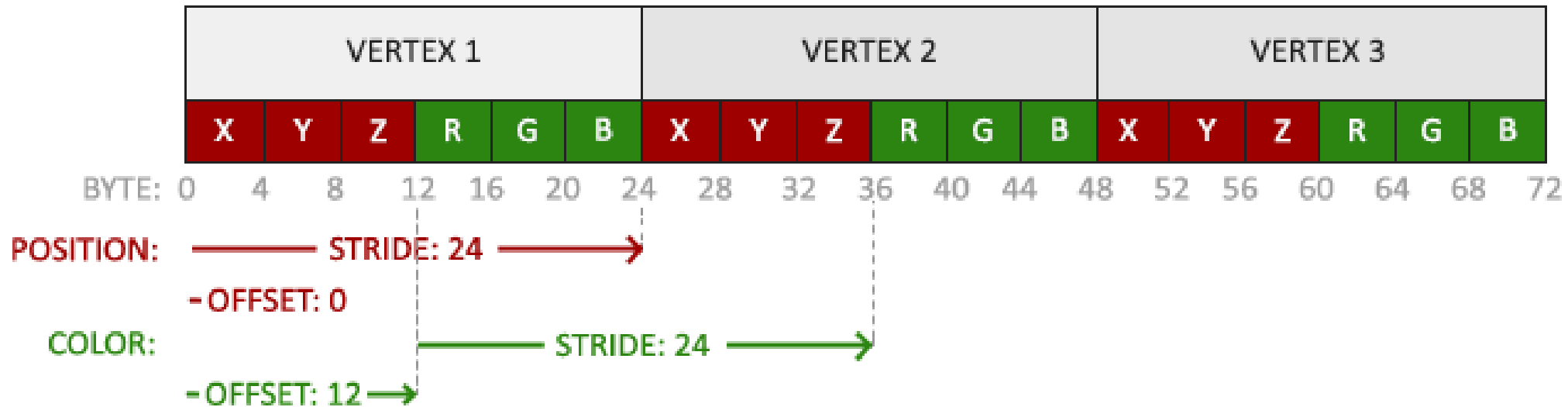
```
# Binding the proper buffers
glBindVertexArray(vao)
glBindBuffer(GL_ARRAY_BUFFER, vbo)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)

# Setting up the location of the attributes position and color from the VBO
# A vertex attribute has 3 integers for the position (each is 4 bytes),
# and 3 numbers to represent the color (each is 4 bytes),
# Henceforth, we have  $3*4 + 3*4 = 24$  bytes
position = glGetAttribLocation(shaderProgram, "position")
glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, 24, ctypes.c_void_p(0))
glEnableVertexAttribArray(position)

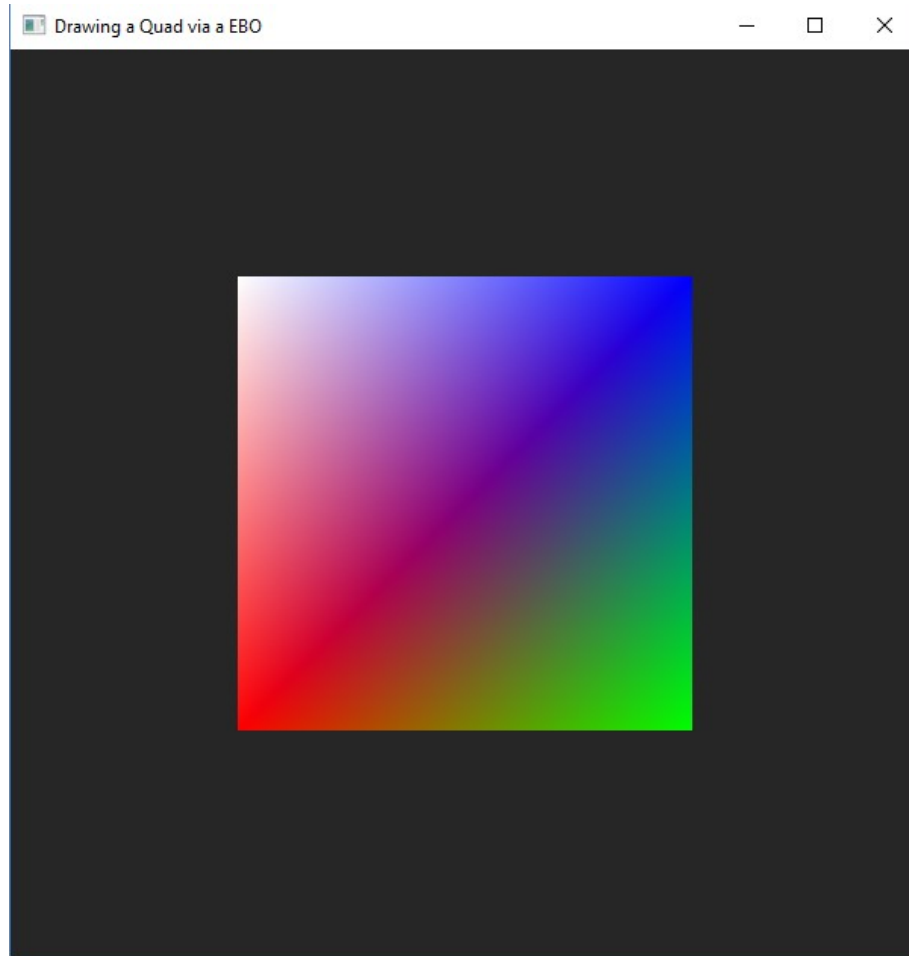
color = glGetAttribLocation(shaderProgram, "color")
glVertexAttribPointer(color, 3, GL_FLOAT, GL_FALSE, 24, ctypes.c_void_p(12))
glEnableVertexAttribArray(color)

# Render the active element buffer with the active shader program
glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
```

Descripción del VBO



Ejemplo OpenGL Core Profile



ex_quad.py

GLSL

GLSL

- Es el lenguaje de shading de OpenGL
- Hay varias versiones disponibles dependiendo el sistema donde se ejecutará
- A pesar de que no implementaremos shaders, es muy útil saber leerlos
- A continuación una breve guía

Tipos de datos

- Escalares:
 - float: 32 bit, casi IEEE-754.
 - int: Al menos 16 bits, representando un número entero
 - bool
- Vectoriales
 - vec[2 | 3 | 4]: vector de números floats
 - ivec[2 | 3 | 4]: vector de números enteros
 - bvec[2 | 3 | 4]: vector de booleanos
- Matrices
 - mat[2 | 3 | 4]: matrices cuadradas de distintos tamaños con valores de punto flotante
- Sampler
 - sampler[1 | 2 | 3]D: Permiten “muestrear” una imagen de textura

Operaciones

- Comportamiento análogo a C++
- Operaciones componente a componente para vectores y matrices
- Producto matricial entre matrices y vectores
- Ejemplos
 - `vec3 t = u * v;`
 - `float f = v[2];`
 - `v.x = u.x + f;`
- Swizzling
 - `vec4 v;`
 - `v.rgba;` // equivalente a `v`
 - `v.xy;` // es de tipo `vec2`
 - `v.zzz;` // es de tipo `vec3` con el valor de la componente `z` en cada componente
 - `v.b;` // equivalente a `v[2]`

Control del flujo

- Control de flujo
 - if-else
 - for, while, do
- Funciones
 - No hay soporte para recursión
 - main is el punto de entrada al shader
- Calificadores de parámetros
 - in: parámetro de entrada
 - out: salida
 - inout: entrada y salida

```
void ComputeTangent(  
    in vec3 N,  
    out vec3 T,  
    inout vec3 coord)  
{  
    if((dot(N, coord)>0)  
        T = vec3(1,0,0);  
    else  
        T = vec3(0,0,0);  
    coord = 2 * T;  
}
```

Built-in Functions

- Trigonometría:
 - `cos`, `sin`, `tan`, ...
- Exponenciales
 - `pow`, `log`, `sqrt`, ...
- Comunes
 - `abs`, `floor`, `min`, `clamp`, ...
- Geometría
 - `length`, `dot`, `normalize`, `reflect`, ...
- Relacional
 - `lessThan`, `equal`, ...
- Utilizar las funciones built-in siempre
 - Nunca implementarlas
- Disponibles dependiendo de la versión de GLSL
 - Ejemplo: `inverse(.)` no se encuentra disponible en las primeras versiones pero sí en 3.30
- A veces no implementadas por fabricantes ↯

Referencias

OpenGL SuperBible

Seventh Edition, Comprehensive Tutorial and Reference

Graham Sellers, Richard S. Wright Jr., Nicholas Haemel

Capítulos 1, 2 y 3

OpenGL Programming Guide

Ninth Edition

John Kessenich, Graham Sellers, Dave Shreiner

Capítulo 1

