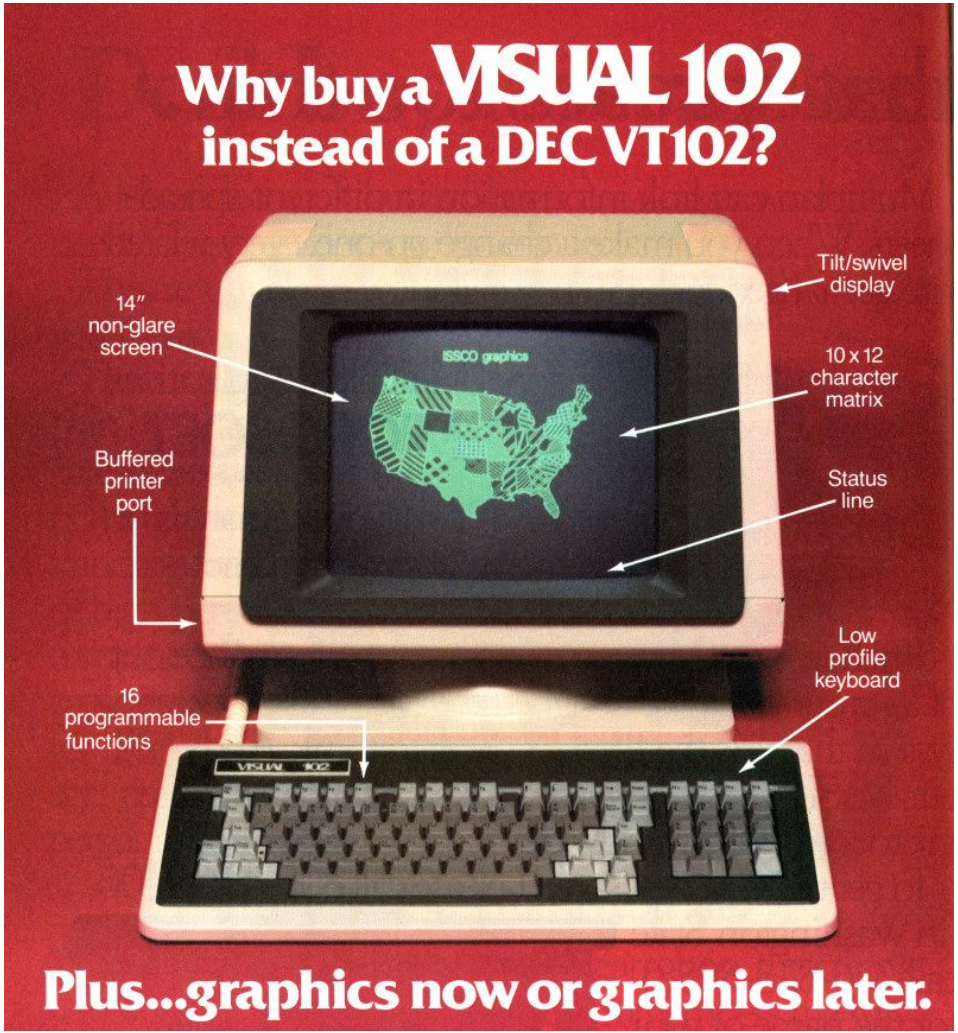


CC3501 GPU/OpenGL

Eduardo Graells-Garrido

Otoño 2023

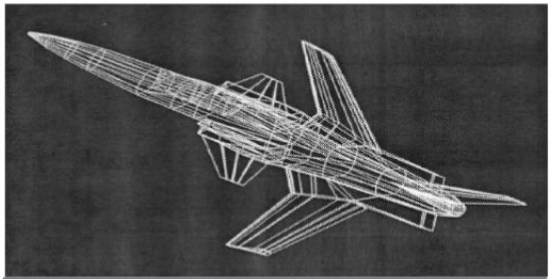
VintageComputing.com

Antes de las GPU

Hace algunas décadas no existían las arquitecturas de GPU.

Las tarjetas gráficas se encargaban de comunicarse con el monitor y de tener la imagen en memoria, o de hacer gráficos 2D, pero no más que eso.

Con el tiempo surgieron distintas tarjetas con chips especializados. Como PowerVR, 3dFx Voodoo, y otras. Cada una tenía su propia biblioteca de rendering y drivers, porque sus tecnologías eran únicas e independientes entre sí.



Primera Generación: Wireframe

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

Segunda Generación: Shaded solids

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

Tercera Generación

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

La “primera” GPU

En 1999, nvidia lanzó la tarjeta GeForce 256 y la llamó “la primera GPU”.

No es la primera en términos teóricos: lo que hacía ya era posible de antes. Pero en vez de ser una arquitectura ad-hoc, **sistematizó** la arquitectura, la puso al alcance de los consumidores (*gamers*), y puso sobre la mesa el término que ahora es ubicuo: **GPU**.

GPU: Unidad de Procesamiento Gráfico

“Un procesador de chip único que integra transformaciones, iluminación, configuración y recorte de triángulos, y un motor de graficación (*rendering*) capaz de procesar un mínimo de 10 millones de polígonos por segundo” (nvidia, 1999)

Una GPU es parte de una tarjeta gráfica.

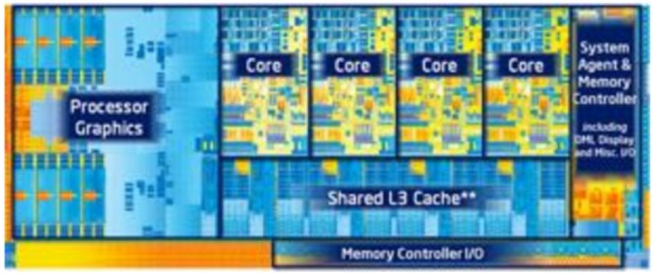
Hoy llamamos GPU a la tarjeta, pero en realidad es el procesador

Discreta: es una tarjeta dedicada al procesamiento gráfico. Contiene múltiples *cores* en múltiples *chips* y memoria especializada (VRAM).

Ejemplo: GeForce RTX 3080

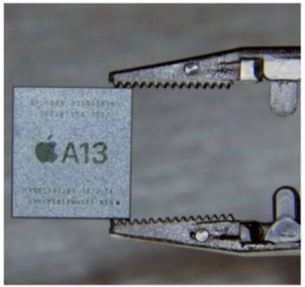
Integrada: es un chip de procesamiento gráfico con múltiples *cores*. Tiene acceso a la RAM del sistema.

discrete GPU card



integrated GPU: part of modern CPU die

smartphone GPU (integrated)



NVIDIA

matrox
Graphics for Professionals

AMD
RADEON
RX 6000 SERIES

Qualcomm

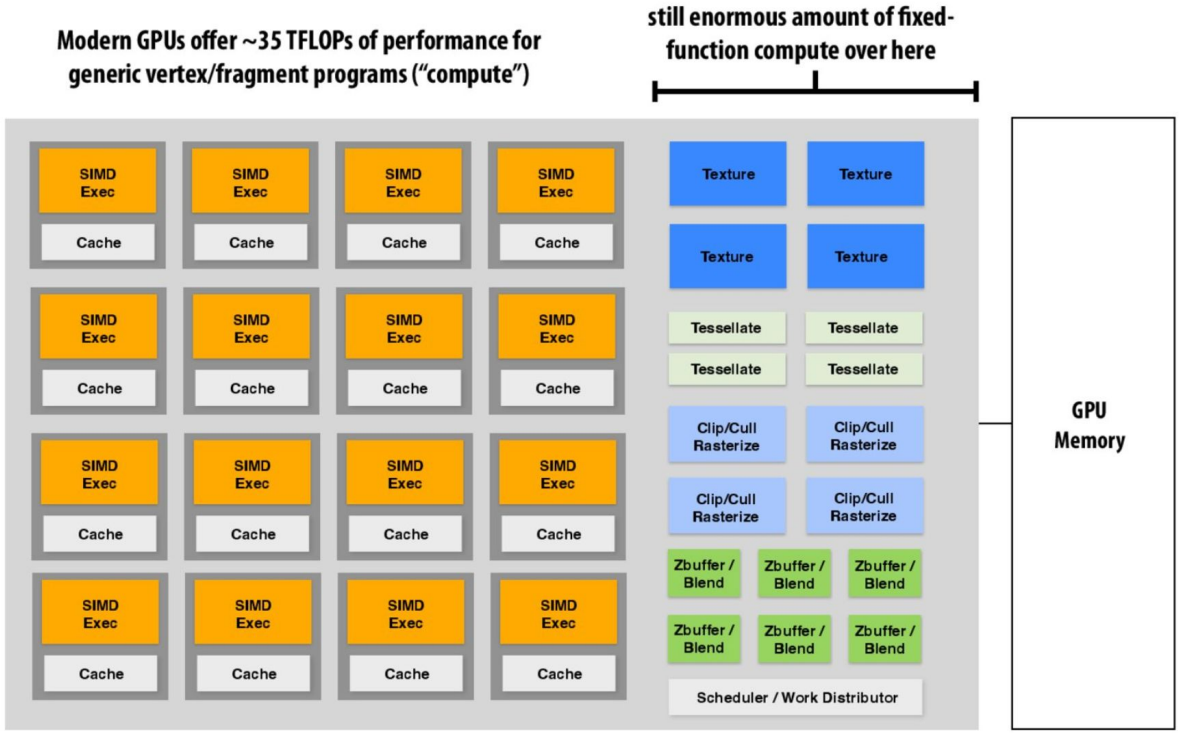
?

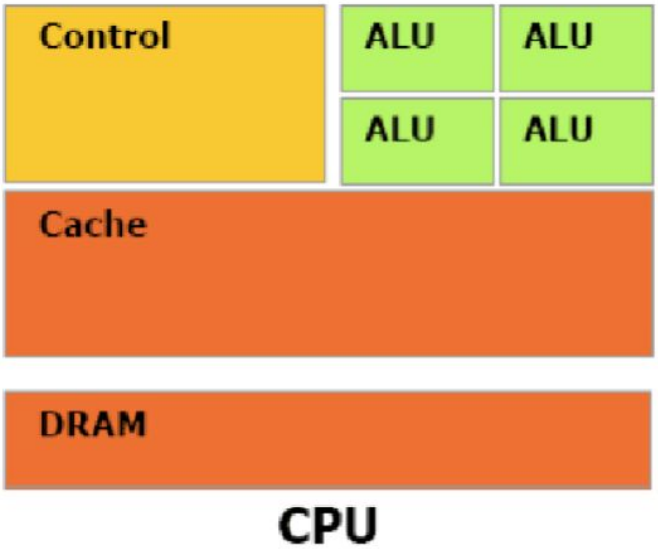
Algunas Fabricantes

Arquitectura

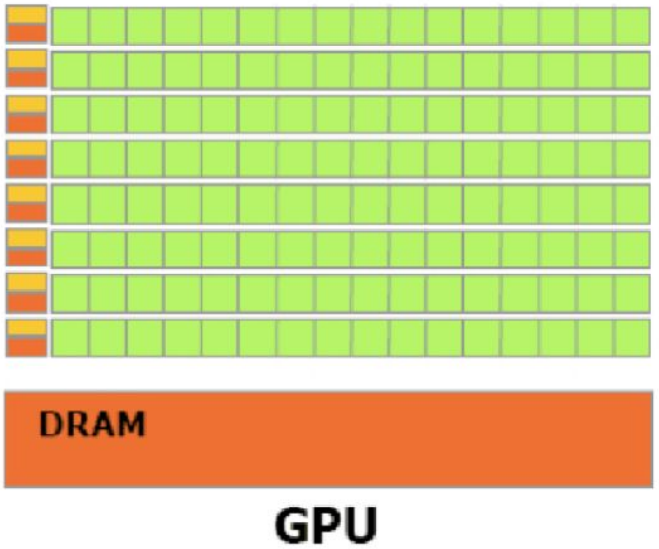
A diferencia de las primeras GPU, hoy son procesadores **programables**. Antes, tenían un **pipeline fijo** de funcionalidad limitada.

OJO: todavía incluyen co-procesadores para operaciones especializadas de gráficos.





CPU



GPU


NVIDIA

Arquitectura de Aplicación (general)

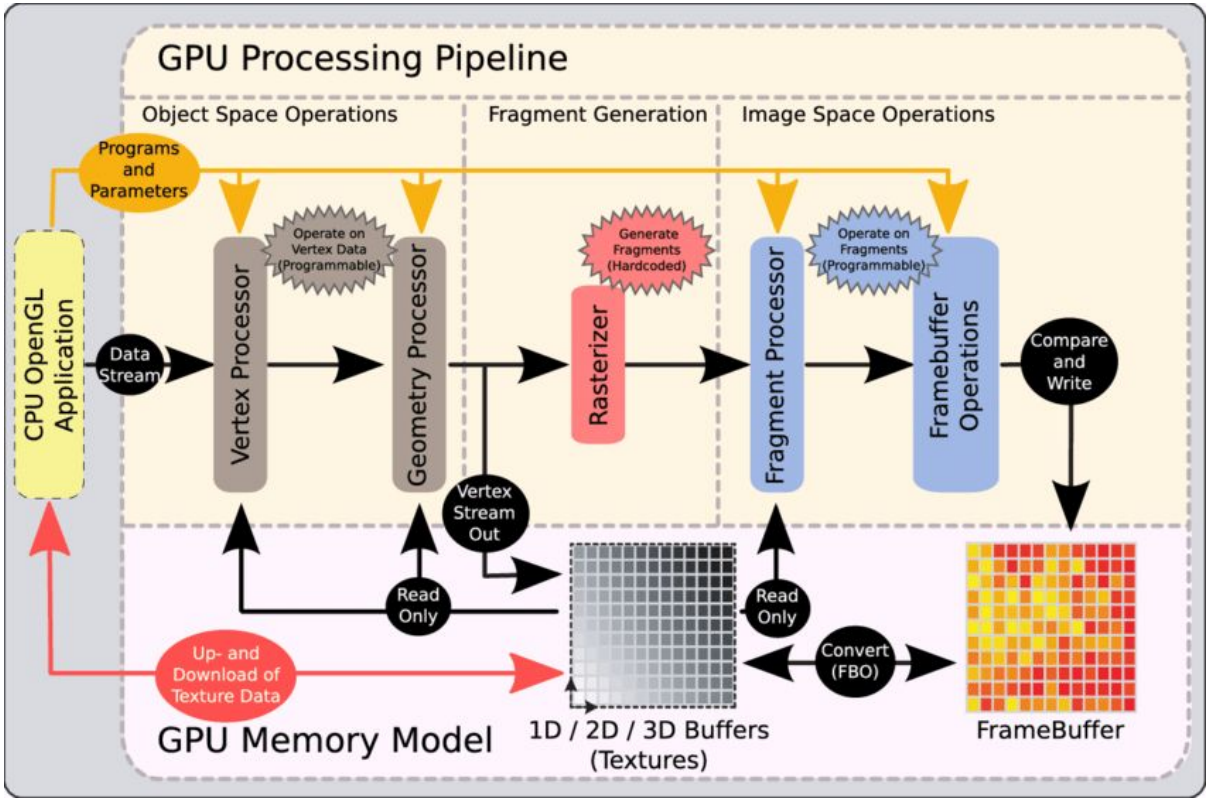
A la GPU le entregamos **datos**, que expresan nuestra escena.

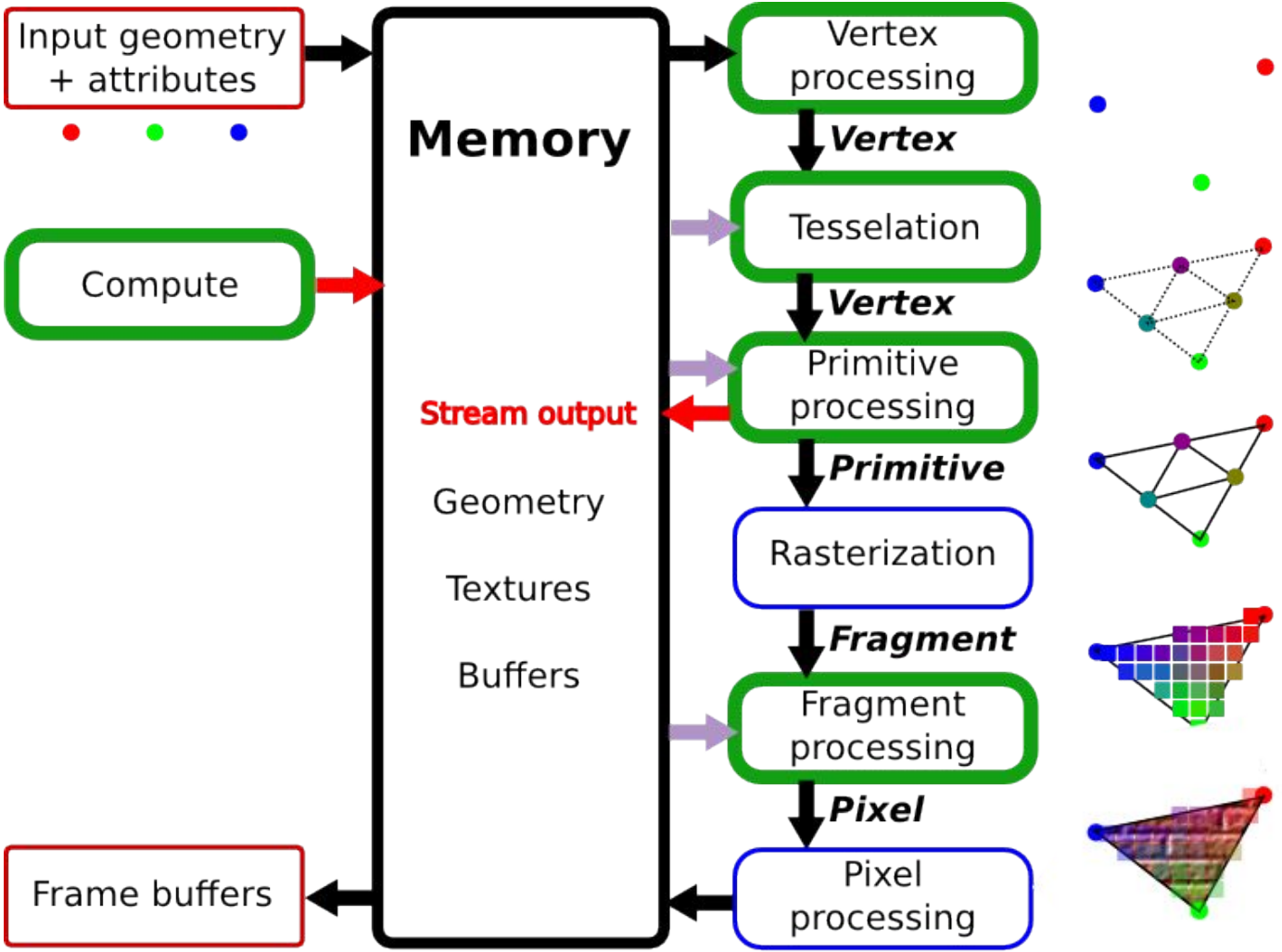
También podemos entregarle tres tipos de programas, cada uno con parámetros propios:

Vertex shader: opera sobre los vértices de nuestra escena (ej: hacer cálculos en vértices)

Geometry shader: opera sobre los elementos de la escena. Puede generar vértices y primitivas (ej: suavizar una superficie)

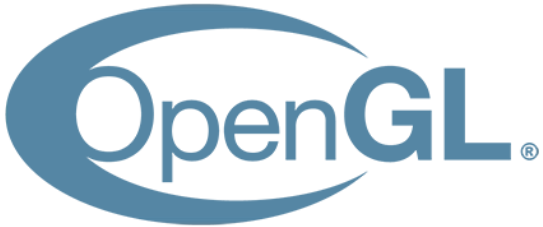
Fragment shader: determinar el color de un pixel en función del contenido de la escena y de las texturas disponibles.




Input geometry
+ attributes
Compute
Memory
Stream output
Geometry
Textures
Buffers
Frame buffers
Vertex processing
Vertex
Tesselation
Vertex
Primitive processing
Primitive
Rasterization
Fragment
Fragment processing
Pixel
Pixel processing


No Lighting
Per-Vertex Lighting
Per-Pixel Lighting
Unreal Engine
1
2
3
Copyright © NVIDIA Corporation 2006
Unreal © Epic

APIs Gráficas



OpenGL

Es una **especificación** de API para generar gráficos.

Su objetivo es **abstraer la generación de imágenes 3D**.

No es código abierto: lo abierto es la especificación.

Existen **implementaciones** de OpenGL. En Linux, Mesa 3D: https://mesa3d.org/

Una implementación de OpenGL no necesariamente usa la GPU. ¡Ojo con eso!

El **driver** de una tarjeta de video implementa la especificación.

Historia: proviene de Silicon Graphics a inicios de los 90. GL viene de Graphics Library.

Versiones de OpenGL

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! →

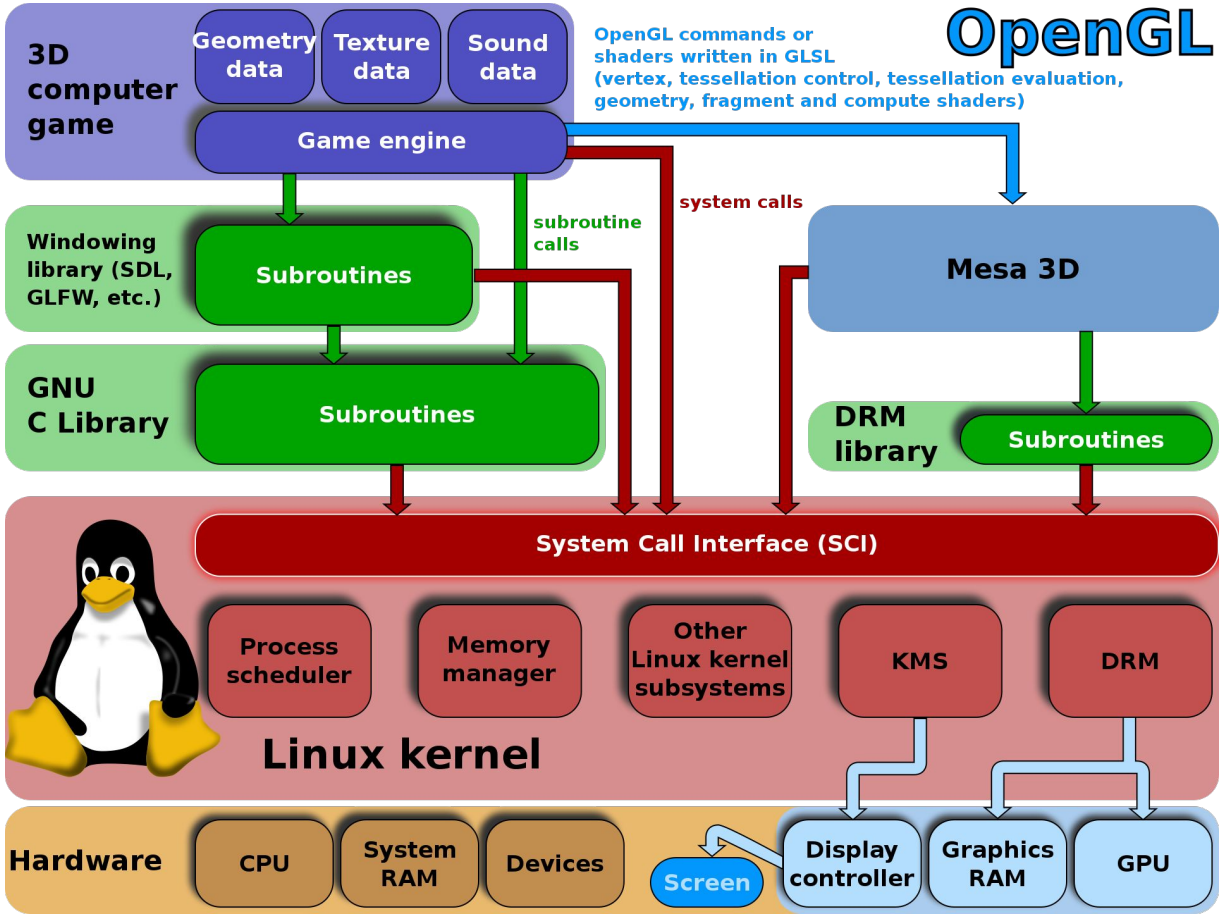
Se introduce modelo de deprecación →

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

Una implementación de OpenGL como Mesa 3D se encarga solamente de la generación de gráficos.

Todo lo demás es externo.

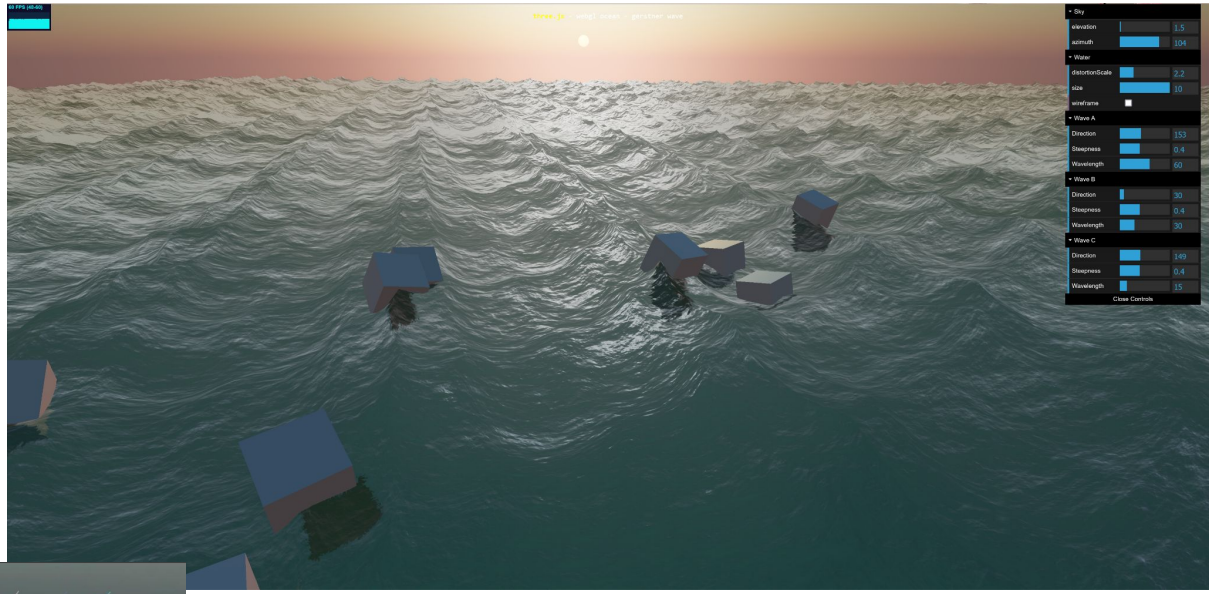
Existen bibliotecas como Qt, SDL y Pyglet (la que usaremos en el curso) para encargarse de lo demás.



Shaders

Programas que permiten manipular vértices, geometría y pixeles. Permiten **generar efectos gráficos no disponibles en el pipeline fijo**.

Se implementan en un lenguaje estilo-C llamado **GLSL** (OpenGL) o **HLSL** (DirectX).



https://raw.githack.com/Sean-Bradley/three.js/gerstner-waves/examples/webgl_shaders_ocean_gerstner.html

Vertex Shader

Se ejecuta en la etapa inicial del pipeline.

Calcula la posición y los atributos de cada vértice. Atributos incluyen color, coordenadas de textura, vector normal, etc.

La posición resultante debe estar en coordenadas del volumen de vista.

```
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

Este programa se ejecuta para cada pixel de cada objeto en la escena. Es decir, después de la **rasterización**.

No todos los pixeles terminarán en la pantalla :o **¿Por qué?**

La entrada son los atributos de los vértices, pero interpolados (ej., usando coordenadas baricéntricas).

El resultado es el **color del pixel** o incluso **un pixel descartado**. También puede generar **información auxiliar** que puede ser leída por otro fragment shader (ej., **profundidad**).

```
fragment_shader = """
    #version 130

    in vec3 fragColor;
    out vec4 outColor;

    void main()
    {
    outColor = vec4(fragColor, 1.0f);
    }
    """
```

Shader Program

El “programa” es la compilación de todos los shaders que utilizaremos, en conjunto con el pipeline fijo.

```
# 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)
```

Estructuras de Datos

Muy lindos los shaders, pero no hemos hablado del inicio del pipeline: la escena.

¿Cómo se la entregamos a OpenGL?

La respuesta: en **buffers** y **arrays**.

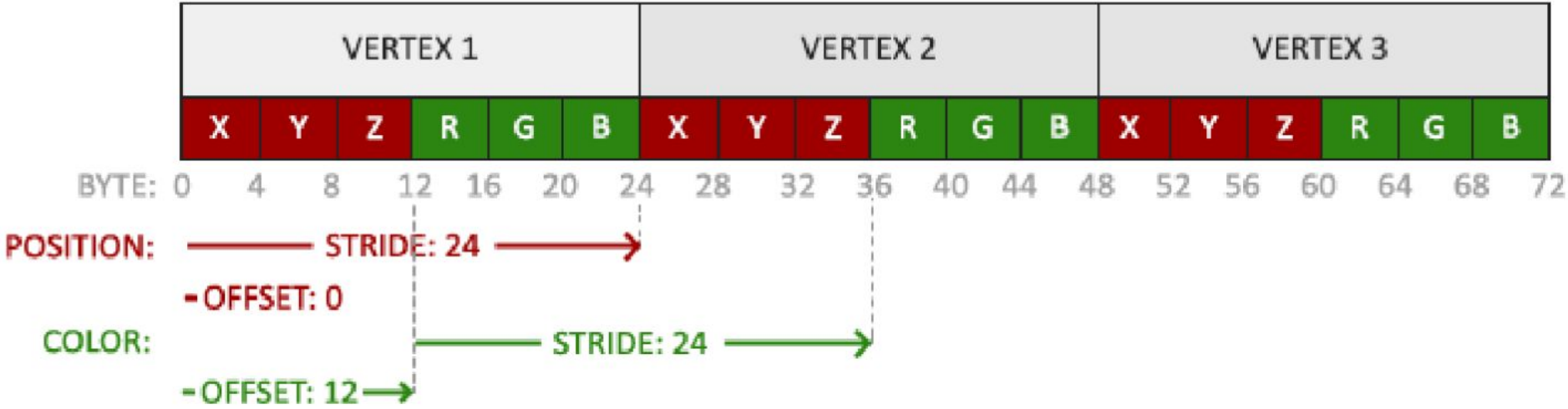
Vertex Buffer Object (VBO): datos asociados a vértices.

Element Buffer Object (EBO): índices de los elementos en VBOs que forman las primitivas gráficas.

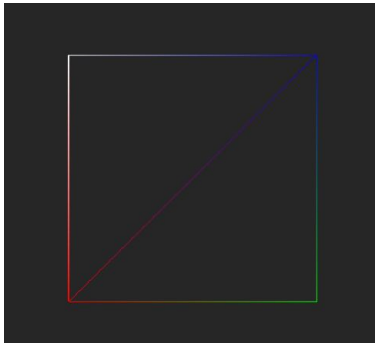
Vertex Array Object (VAO): información que permite interpretar los VBO.

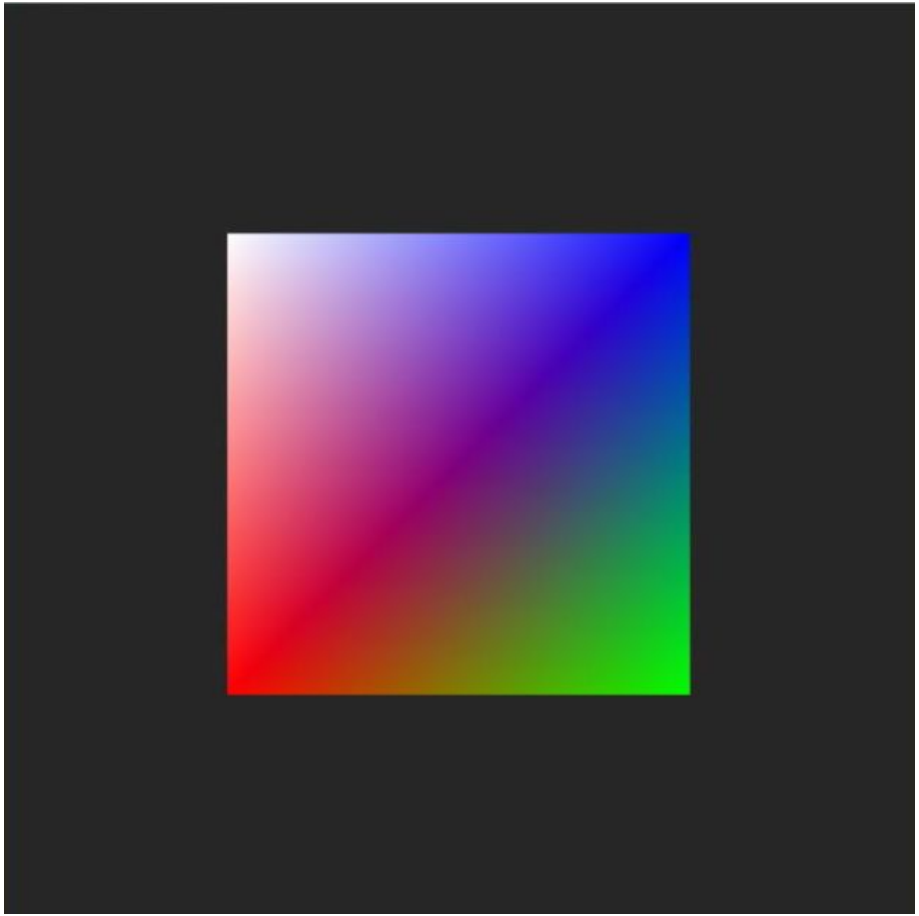
(sí, son **punteros** 🤪)

```
# VAO, VBO and EBO and for the shape
vao = glGenVertexArrays(1)
vbo = glGenBuffers(1)
ebo = glGenBuffers(1)
```


VERTEX 1
VERTEX 2
VERTEX 3
X Y Z R G B X Y Z R G B X Y Z R G B
BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72
POSITION:
STRIDE: 24
OFFSET: 0
COLOR:
STRIDE: 24
OFFSET: 12

```
# 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)
```



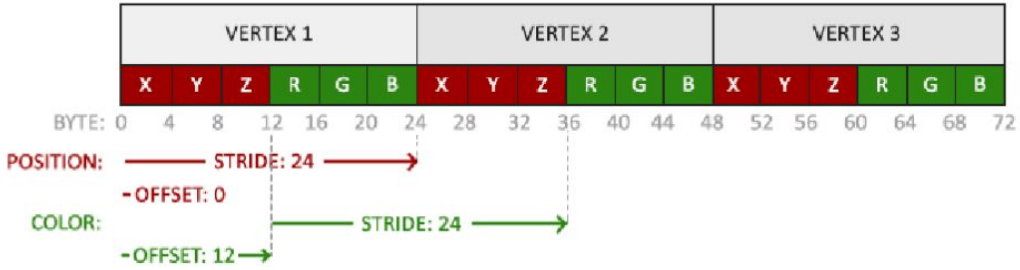


```
# Vertex data must be attached to a Vertex Buffer Object (VBO)
glBindBuffer(GL_ARRAY_BUFFER, vbo)
glBufferData(GL_ARRAY_BUFFER, len(vertexData) * SIZE_IN_BYTES, vertexData, GL_STATIC_DRAW)

# Connections among vertices are stored in the Elements Buffer Object (EBO)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, len(indices) * SIZE_IN_BYTES, indices, GL_STATIC_DRAW)
```

32 bits en bytes → 4

Que tan frecuentemente cambiará esta información? →
- GL_STATIC_DRAW
- GL_STREAM_DRAW
- GL_DYNAMIC_DRAW

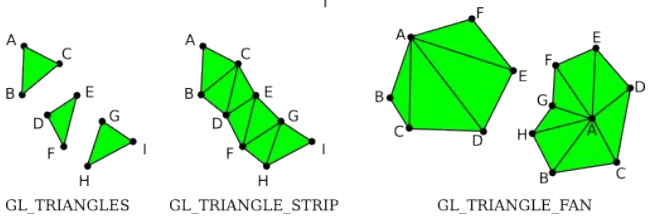
OpenGL funciona como una máquina de estados: debemos decir cuál es el buffer con el que trabajaremos, y luego podemos hacer operaciones sobre él.

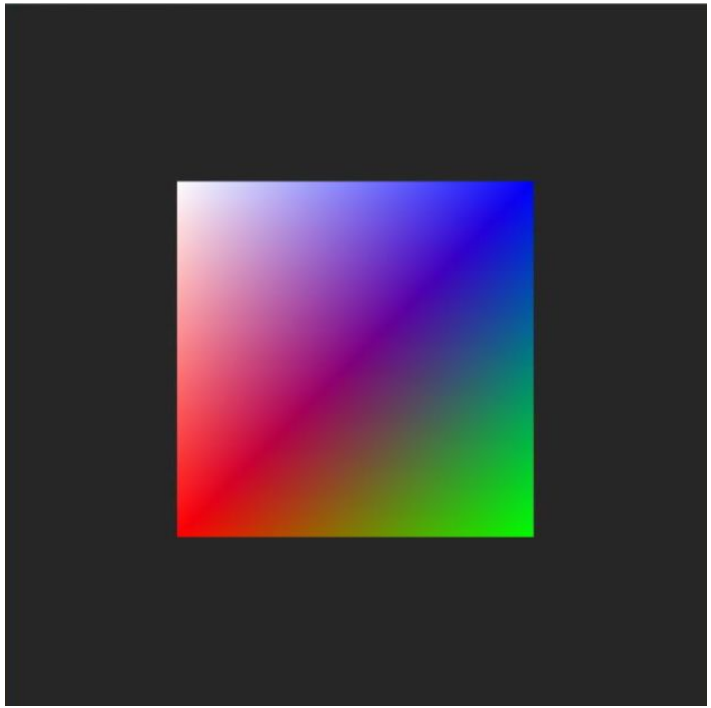



```
# Binding the proper buffers
glBindVertexArray(vao)
glBindBuffer(GL_ARRAY_BUFFER, vbo)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)

# 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)
```

Necesitamos entregarle la **estructura** de nuestros datos. **¿Cómo sabe nuestro vertex shader cuáles valores son de color y cuáles de posición?**

```
glBindVertexArray(vao)
glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT, None)
```

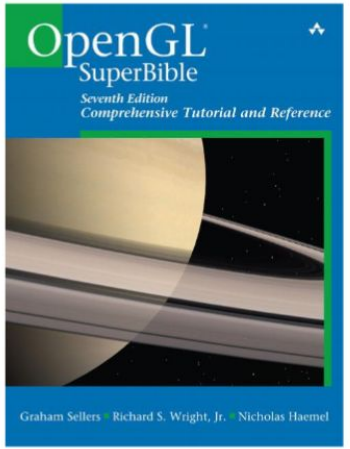




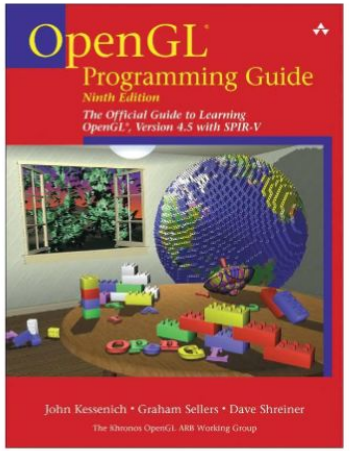
OJO: **Existen distintas maneras de graficar triángulos en OpenGL**. Usualmente trabajaremos con **GL_TRIANGLES**, pero en usos avanzados se pueden aprovechar los otros modos (mejor relación triángulos/vértices).

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



¿Preguntas?

¿Dudas de la tarea?