

Introducción a OpenGL 4.x



<http://www.anandtech.com>

Bibliografía:

- Capítulo 1; pp. 99-117 y 249-255 superbiblia 7ª ed.



OpenGL

- ◎ Es un estándar que define una API multiplataforma para el desarrollo de aplicaciones gráficas 2D y 3D
 - Se usa en ordenadores, consolas, móviles, etc.
- ◎ Es una librería orientada al desarrollo de aplicaciones interactivas, donde prima la velocidad frente al fotorrealismo
- ◎ Hace uso del hardware gráfico disponible
- ◎ Fue publicado por Silicon Graphics en 1992, y en la actualidad está controlado por el consorcio Khronos
 - <http://www.khronos.org>

OpenGL cumplió 30 años en 2022



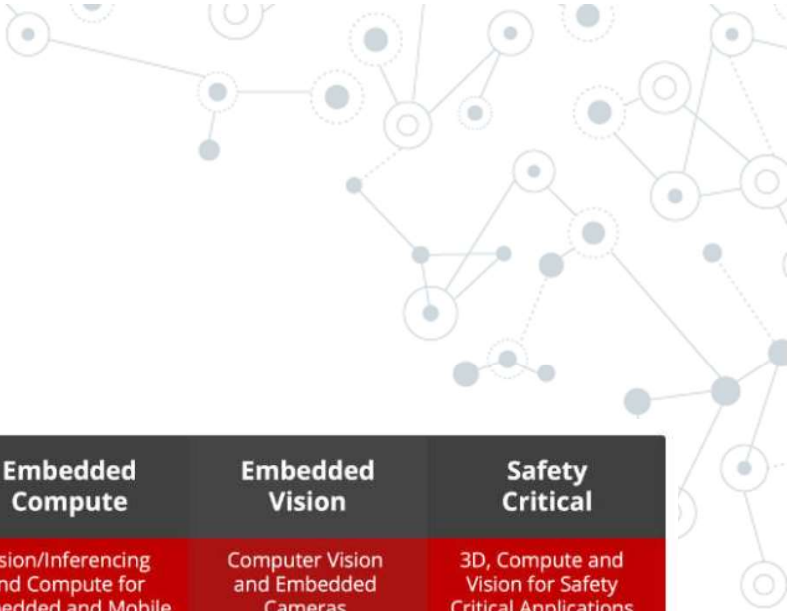
Khronos
































<https://www.khronos.org/assets/uploads/apis/memberSlide.zip>

OpenGL

Otras librerías de la familia



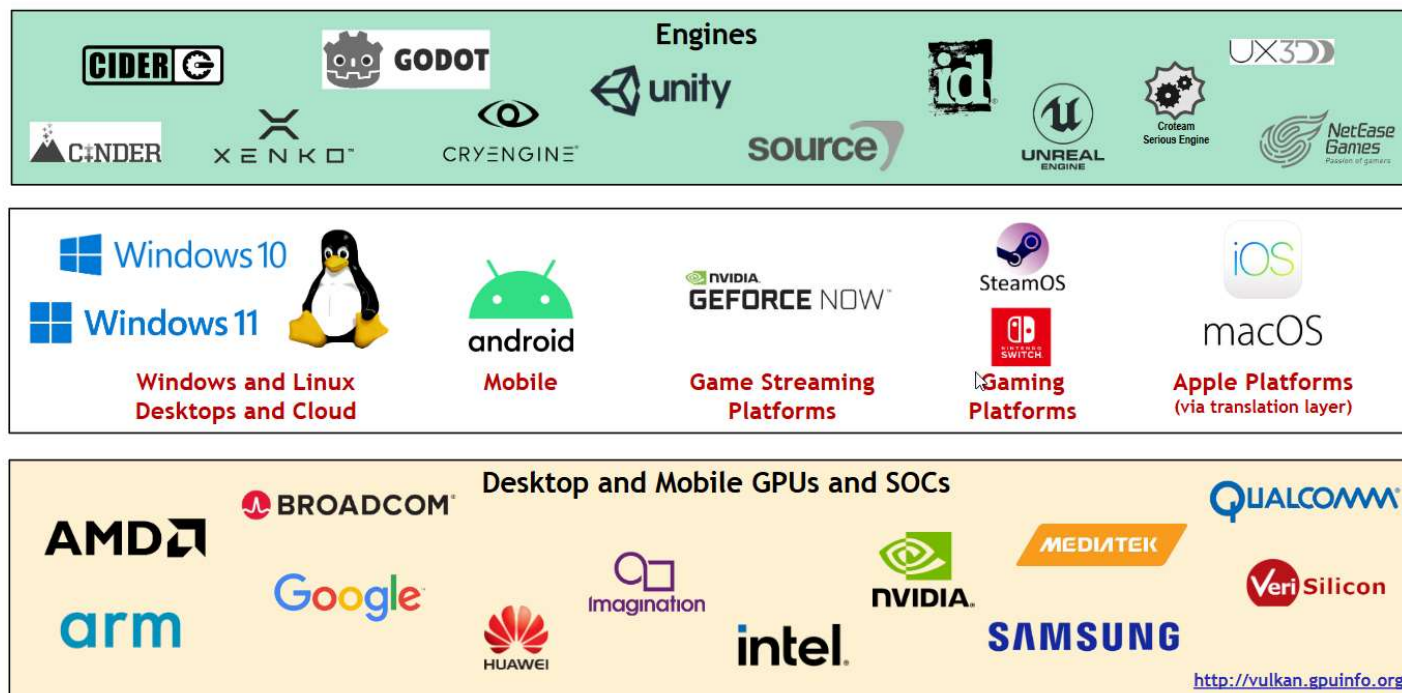
Performance 3D	Pervasive 3D	Extended Reality	Performance Compute	Embedded Compute	Embedded Vision	Safety Critical
Interactive 3D on Desktop and Mobile	Widespread use of 3D on web, desktop and mobile	Virtual and Augmented Reality	Parallel Computing on HPC and Desktop	Vision/Inferencing and Compute for Embedded and Mobile	Computer Vision and Embedded Cameras	3D, Compute and Vision for Safety Critical Applications
   	    	   	   	    	  	   

<https://www.khronos.org/about/>

OpenGL

Otras librerías de la familia

Vulkan Adoption



Note: The version of Vulkan available will depend on platform and vendor

This work is licensed under a Creative Commons Attribution 4.0 International License

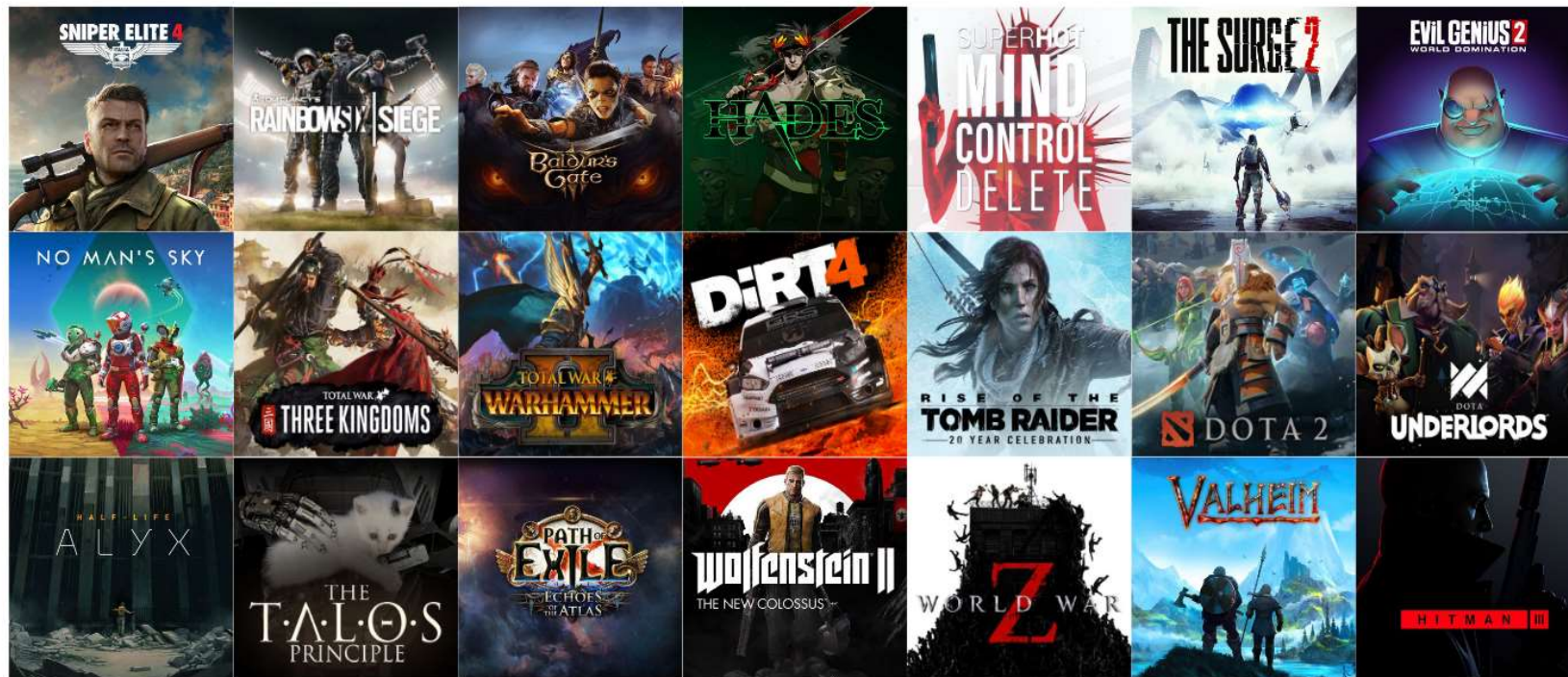
© The Khronos® Group Inc. 2023 - Page 6

https://www.khronos.org/assets/uploads/developers/presentations/Vulkan_BOF_SIGG2023_Khronos.pdf

OpenGL

Otras librerías de la familia

Vulkan Games Shipping on Desktop



Over 160 Vulkan Titles shipping across PC, Linux, Stadia, and MacOS with Molten VK

This work is licensed under a Creative Commons Attribution 4.0 International License

© The Khronos® Group Inc. 2021 - Page 5

<https://www.khronos.org/assets/uploads/developers/presentations/Vulkan-Update-SIGGRAPH-Aug21.pdf>

OpenGL

Otras librerías de la familia

Vulkan Games Shipping on Mobile



Vulkan's lower CPU overhead enables better performance and/or lower power

Vulkan is the default renderer for mobile projects on Unity

This work is licensed under a Creative Commons Attribution 4.0 International License

© The Khronos® Group Inc. 2021

<https://www.khronos.org/assets/uploads/developers/presentations/Vulkan-Update-SIGGRAPH-Aug21.pdf>

OpenGL

Historia de las versiones de OpenGL

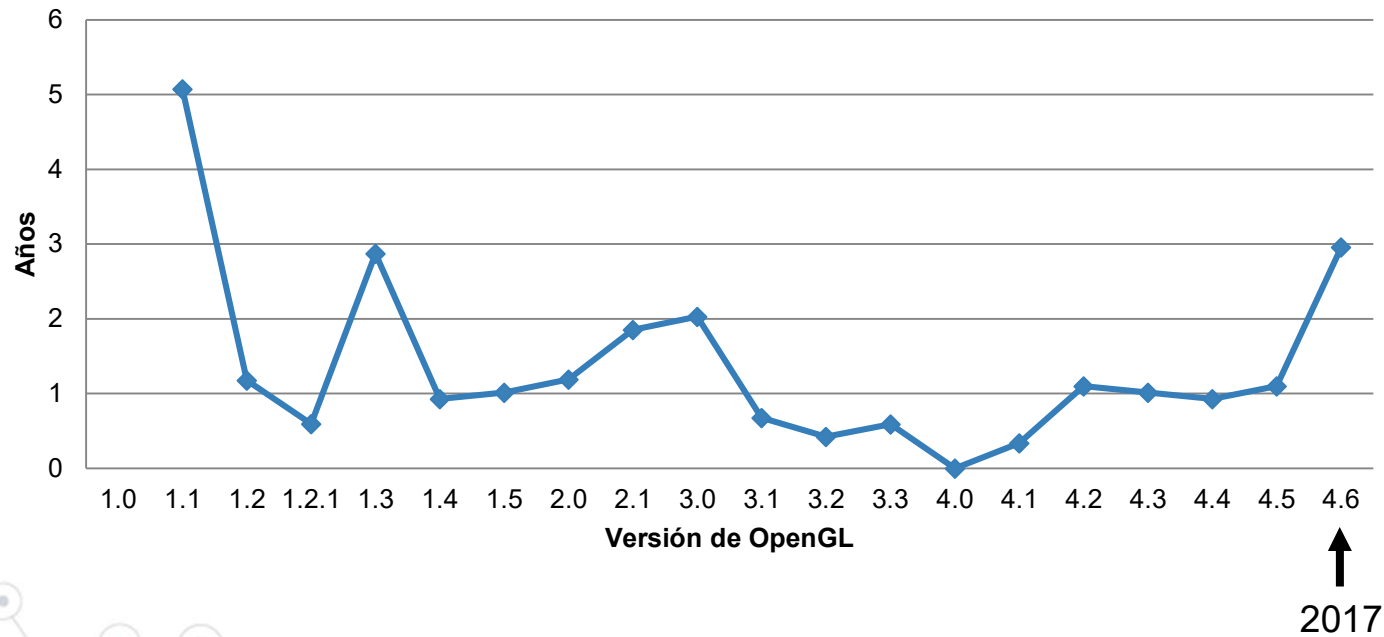
OpenGL	Fecha
1.0	Enero 1992
1.1	Enero 1997
1.2	Marzo 1998
1.2.1	Oct. 1998
1.3	Agosto 2001
1.4	Julio 2002
1.5	Julio 2003
2.0	Sept. 2004
2.1	Agosto 2006
3.0	Agosto 2008
3.1	Marzo 2009

OpenGL	Fecha
3.2	Agosto 2009
3.3	Marzo 2010
4.0	Marzo 2010
4.1	Julio 2010
4.2	Agosto 2011
4.3	Agosto 2012
4.4	Julio 2013
4.5	Agosto 2014
4.6	Julio 2017

OpenGL

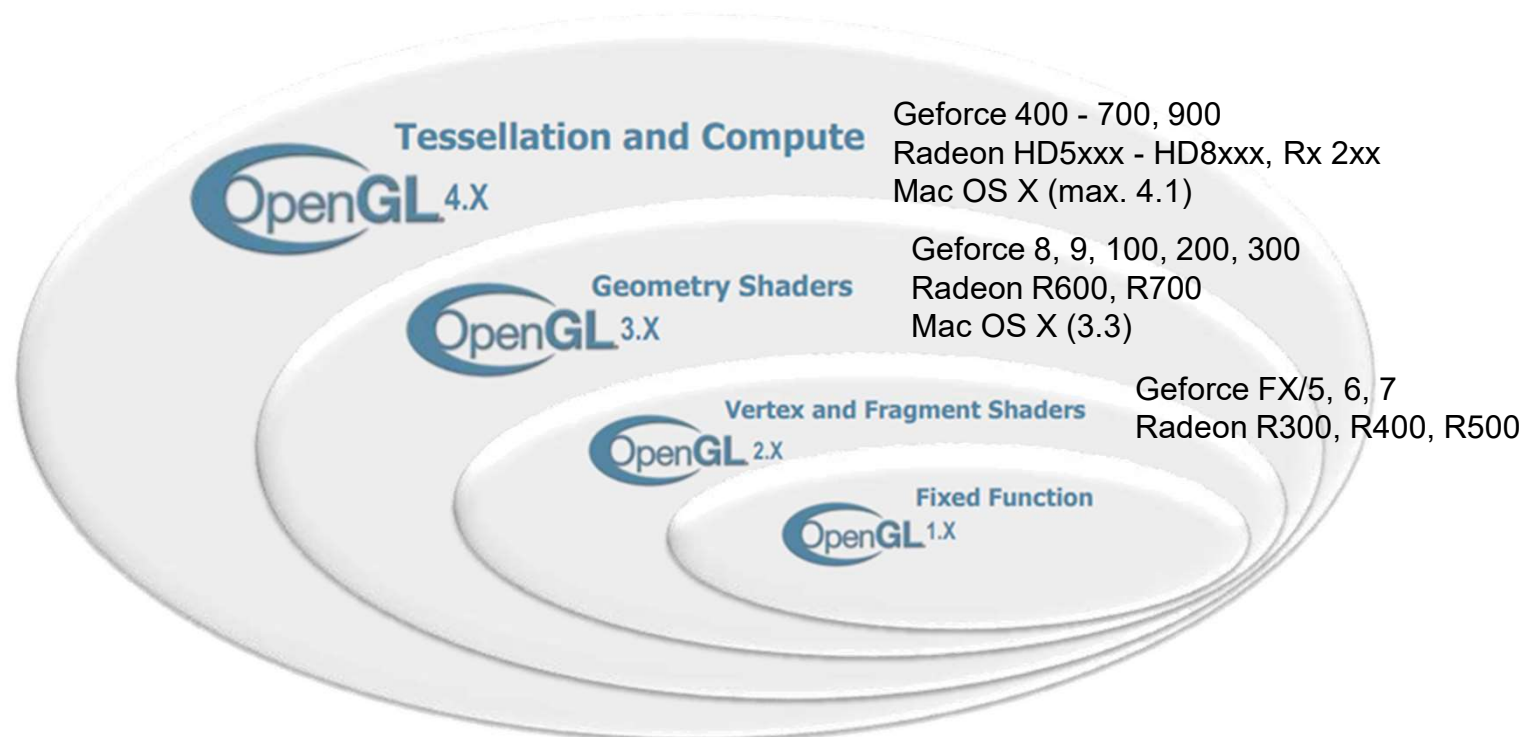
Historia de las versiones de OpenGL

Tiempo transcurrido desde la última actualización



OpenGL

Características del hardware



http://en.wikipedia.org/wiki/Nvidia_gpu

http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units

http://en.wikipedia.org/wiki/Comparison_of_Intel_graphics_processing_units

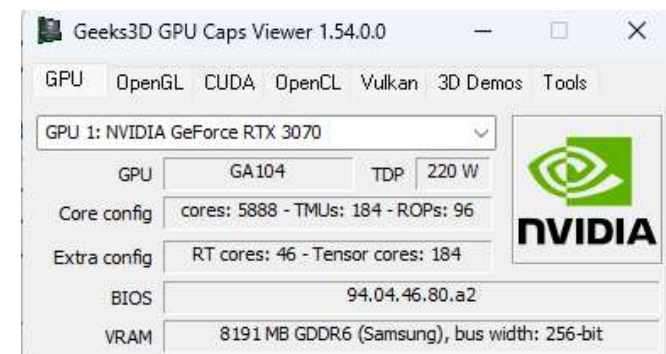
<http://support.apple.com/kb/HT5942>

OpenGL

Características del hardware

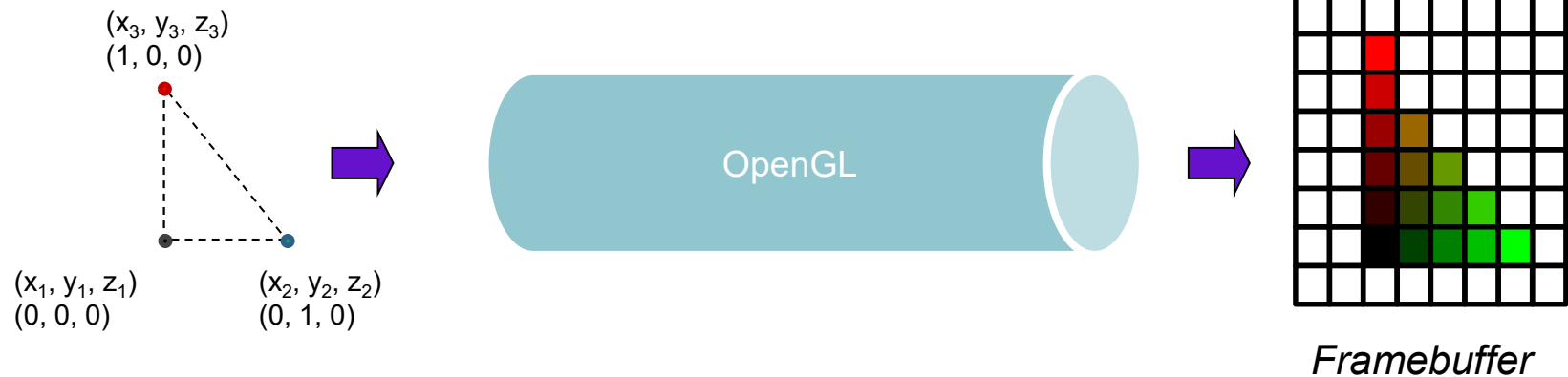
◎ Una GPU (Graphics Processing Unit) moderna está compuesta por un número de pequeños procesadores (*shader cores*)

- Adreno 730: 768 (Galaxy S22)
- GeForce GTX 1060(N): 1280
- Radeon RX Vega 64: 4096
- GeForce RTX 2060: 1920
- GeForce Titan RTX: 4608
- Radeon VII: 3840
- Radeon RX 7900 XTX: 6144
- GeForce RTX 4090: 9728



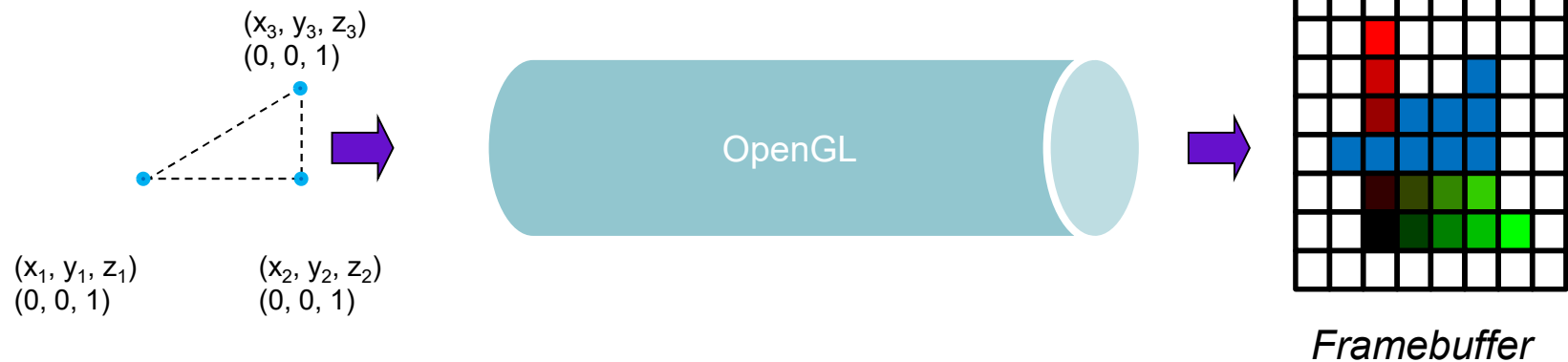
La tubería de OpenGL

- El funcionamiento interno de OpenGL es parecido al de una cadena de montaje



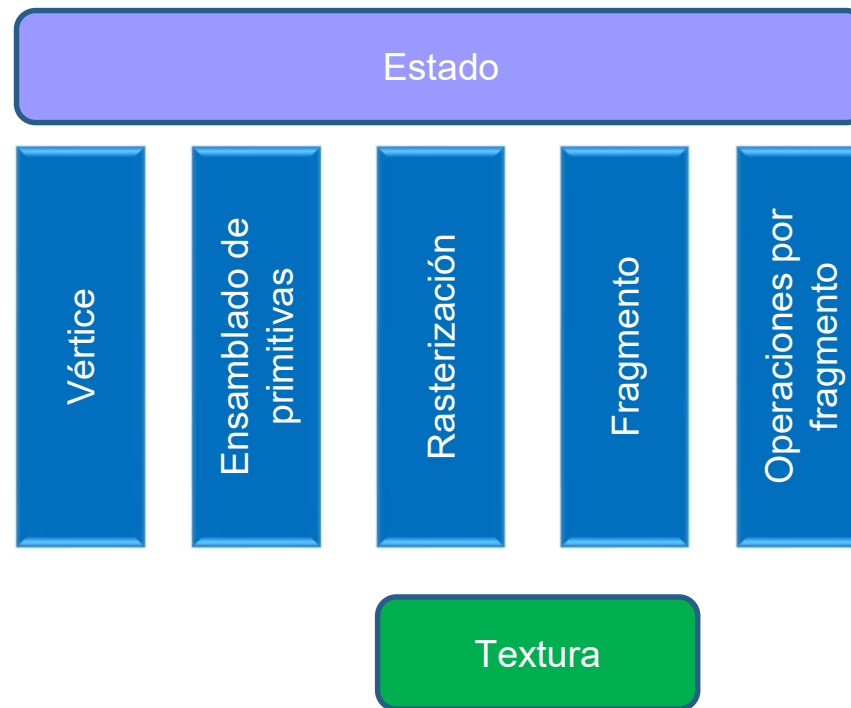
La tubería de OpenGL

- El funcionamiento interno de OpenGL es parecido al de una cadena de montaje



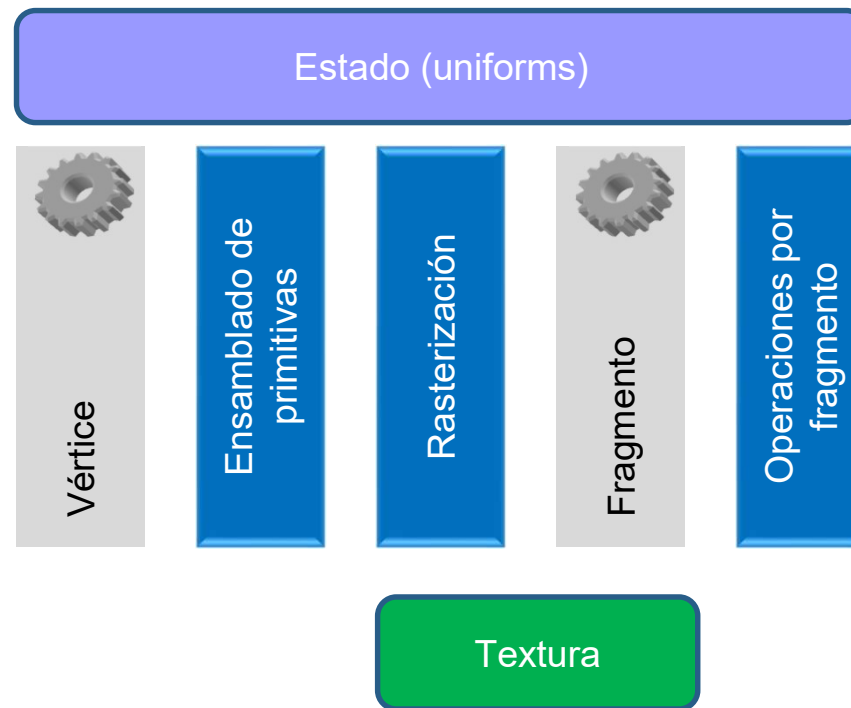
Evolución de la tubería

© OpenGL 1.x



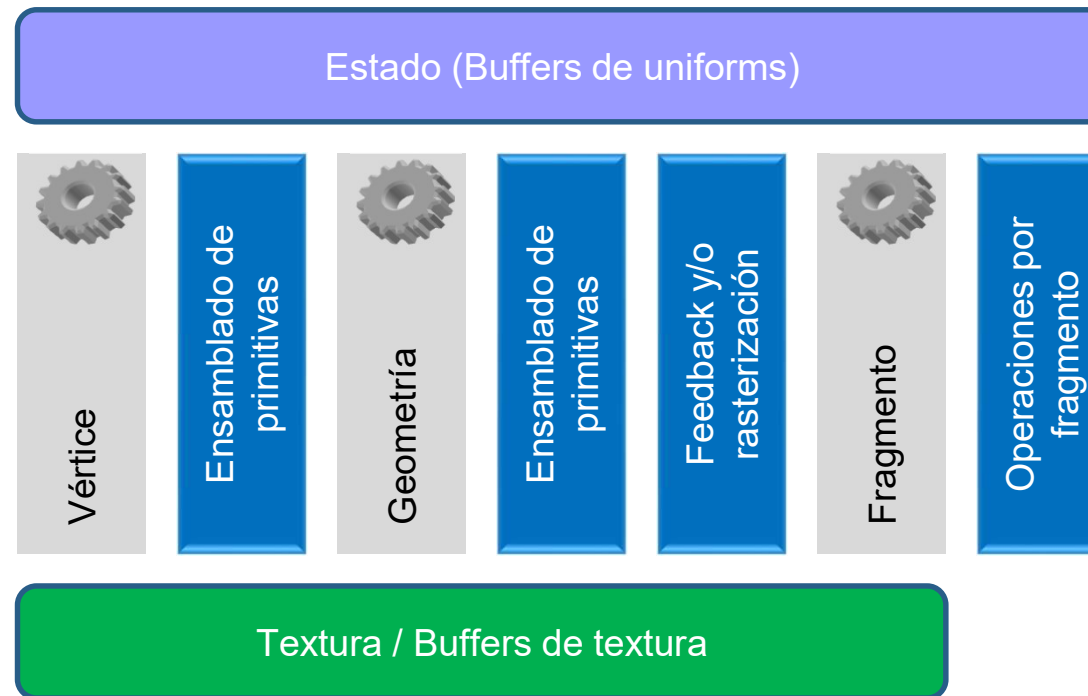
Evolución de la tubería

© OpenGL 2.x



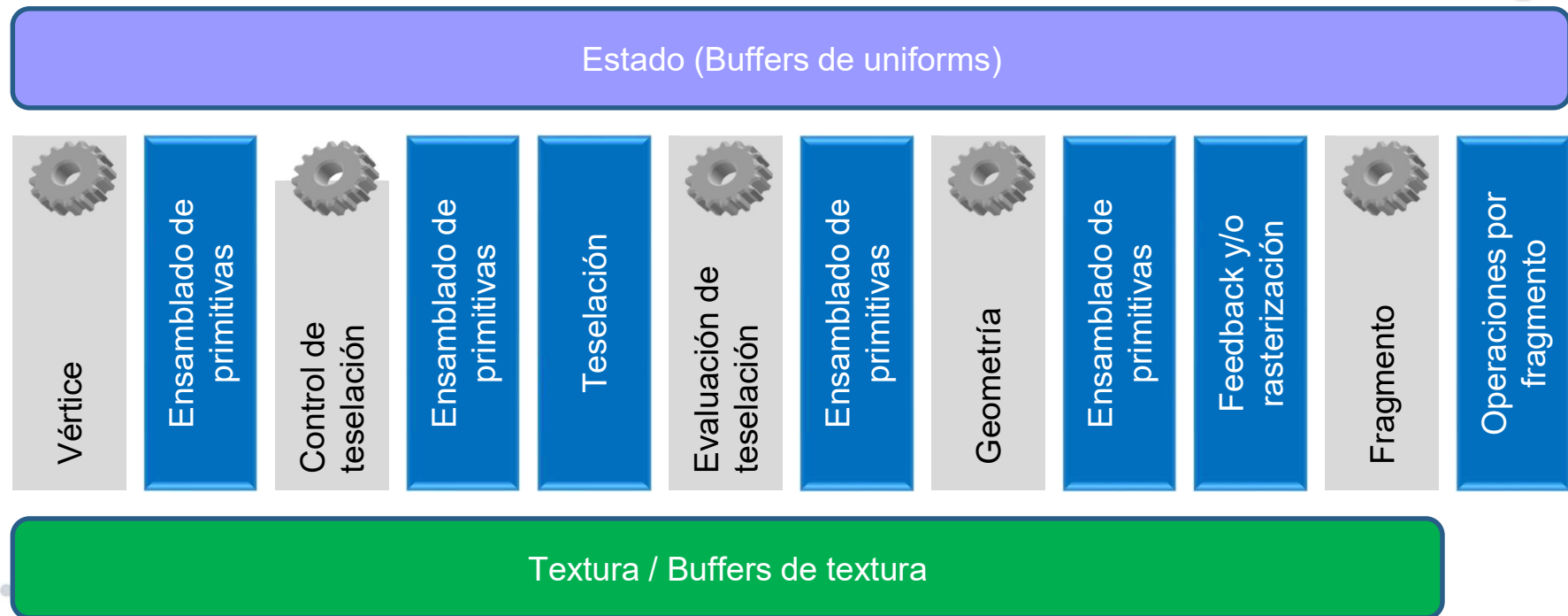
Evolución de la tubería

© OpenGL 3.x



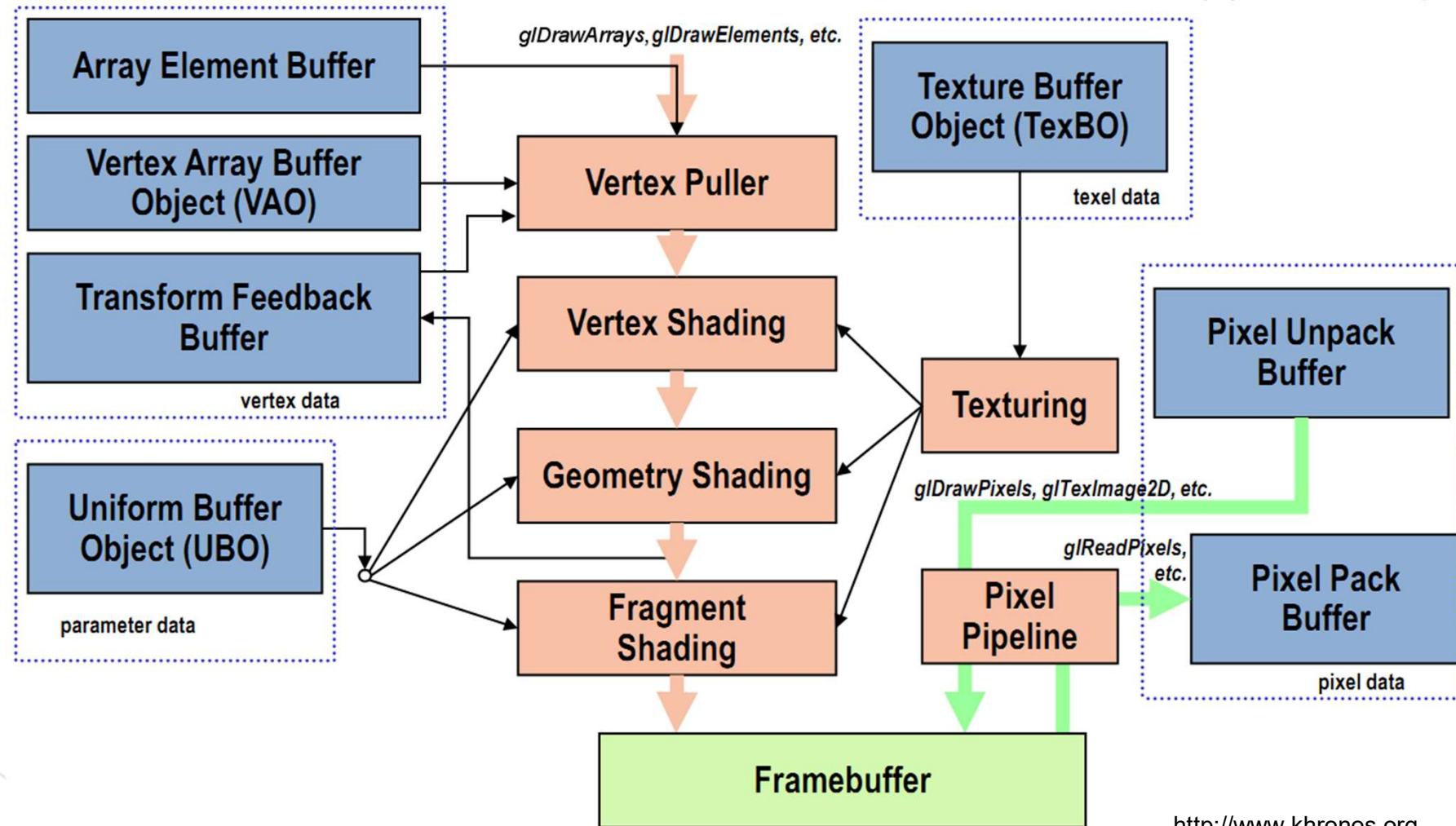
Evolución de la tubería

© OpenGL 4.x



Evolución de la tubería

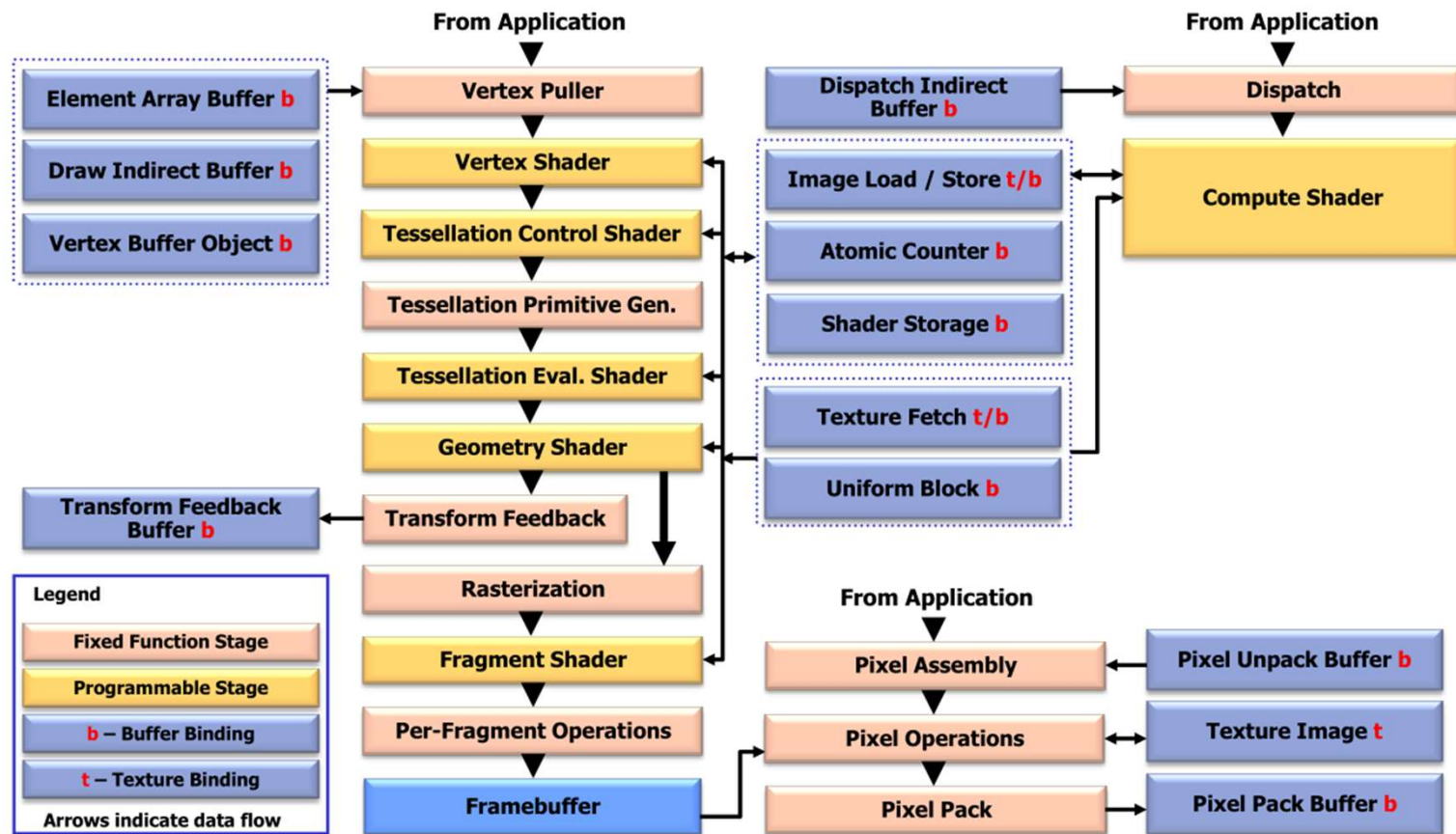
OpenGL 3



Evolución de la tubería

OpenGL 4.3

OpenGL 4.3 Pipelines



Funcionalidades de OpenGL

- ⦿ Antes de OpenGL 3.0, siempre se mantenía toda la funcionalidad anterior
- ⦿ A partir de OpenGL 3.0, se empezó a marcar funcionalidad como *deprecated*, de forma que versiones posteriores podrían eliminarla
- ⦿ Para gestionar la dualidad de capacidades se han definido dos perfiles:
 - *Core*: es la funcionalidad que debe estar en cualquier implementación de OpenGL
 - *Compatibility*: el resto de la funcionalidad implementada en el driver que da soporte a versiones anteriores

Funcionalidades de OpenGL

◎ Extensiones:

- Permiten a los fabricantes añadir nuevas funciones a OpenGL para aprovechar nuevas funciones del hardware
- Cada extensión tiene un nombre único, prefijado por el fabricante que la definió:
 - ◎ SGI_, ATI_, AMD_, NV_, IBM_, WGL_, ...
 - ◎ EXT_: implementado por varios fabricantes
 - ◎ ARB_: aprobado por la *Architecture Review Board*
- Ejemplos: **GL_NV_evaluators**, **GL_ATI_element_array**, **GL_ARB_sample_shading**
- <http://www.opengl.org/registry>

Funcionalidades de OpenGL

- ◎ ¿Y cuál de todas las versiones usaremos?
 - OpenGL 4.X Core
 - ◎ La versión 4.0 se publicó en 2010
- ◎ ¿Y en el mundo “real”?
 - <http://store.steampowered.com/hwsurvey>
 - <http://stats.unity3d.com>
 - Los motores se adaptan al hardware disponible
 - ◎ Activando o desactivando efectos que requieran una versión determinada
 - ◎ Sustituyendo efectos no disponibles en GPU por implementaciones en CPU
 - Requisitos mínimos
 - Aún hay mucho código legado que usa la tubería fija

Diseño de aplicaciones OpenGL

- ◎ El objetivo es minimizar el número de llamadas a OpenGL
 - Cada llamada recorre un largo camino hasta la tarjeta
 - El driver debe comprobar los parámetros de cada llamada
 - A menudo se envía la misma información a través del bus (p.e., la geometría de un objeto no deformable)

Avanzado

Approaching Zero Driver Overhead in OpenGL. NVIDIA. GDC 2014: <http://gdcvault.com/play/1020791/>

Diseño de aplicaciones OpenGL



Tampoco tenemos control sobre el momento en el que se manda una orden a la tarjeta

La transferencia se inicia si:

- el *buffer* del *driver* se llena
- *swap buffer*
- *glFlush*
- *glFinish (bloqueante)*




Diseño de aplicaciones OpenGL

- ◎ Para minimizar el tráfico en el bus, OpenGL (moderno) obliga a almacenar la información gráfica (los modelos y texturas) en memoria de la GPU
- ◎ Los *Buffer Objects* de OpenGL permiten:
 - almacenar información en la GPU
 - transferir información entre distintos puntos de la tubería, sin intervención de la CPU
 - hacer *rendering* fuera de la ventana
 - mover información entre la CPU y la GPU (aunque este proceso es mucho más lento que la transferencia entre buffer objects)

Diseño de aplicaciones OpenGL

- ◎ Puede haber cualquier número de BO en la GPU
- ◎ Pueden almacenar muchos tipos de información: datos de vértice, píxeles, texturas, salida de shaders, etc.
- ◎ ¡Cuidado!
 - es necesario controlar la memoria disponible
 - para el proceso de copia CPU-GPU-CPU se pasa por el bus

Buffer Objects

- 
- © Ciclo de vida de un BO:
1. Crear identificadores de *buffer object*
 2. Vincular un *buffer object* a un punto de vinculación
 3. Reservar espacio para los datos, especificando el tipo de información a almacenar
 4. Escribir, usar, leer...
 5. Desvincular el buffer object
 6. Liberar el buffer object

Buffer Objects

1. Crear identificadores de buffer object

☐ **void glGenBuffers(GLsizei n, GLuint *buffers)**

☒ donde:

☐ n: número de identificadores a generar

☐ buffer: vector para almacenarlos

☒ Ej:

```
GLuint bo;  
glGenBuffers(1, &bo);
```

Buffer Objects

2. Vincular un buffer object

- Activar un *buffer* al que harán referencia las operaciones siguientes
- **void glBindBuffer(GLenum target, GLuint buffer)**
- donde:
 - *target*: es el punto de vinculación donde conectarse
 - *buffer*: identificador de buffer, o 0 para desvincular (paso 5)

Buffer Objects

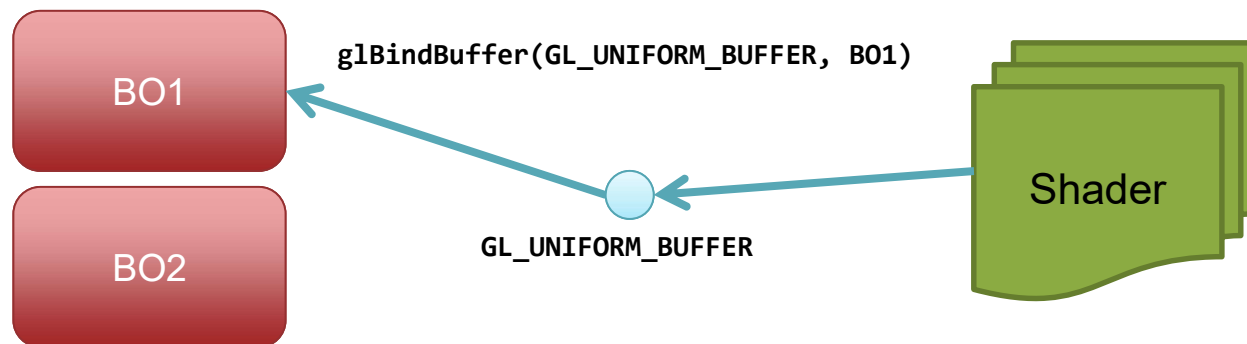
◎ Puntos de vinculación



Buffer Objects

◎ Puntos de vinculación

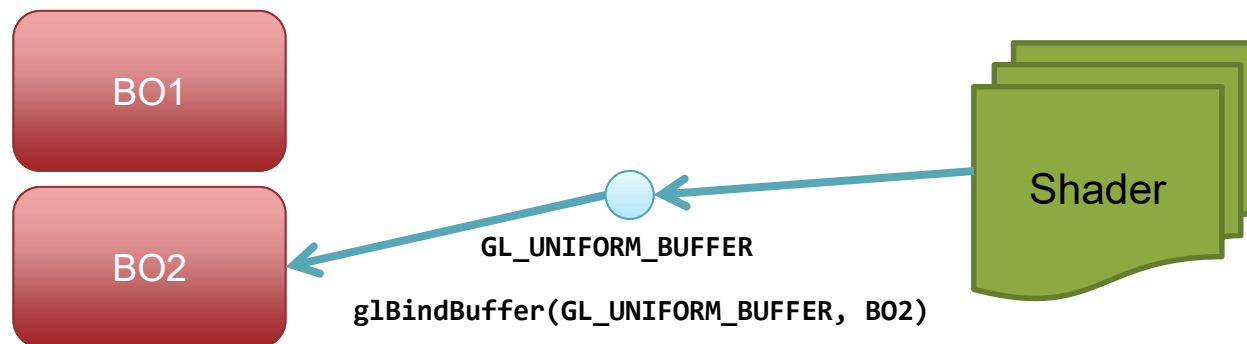
- GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_TEXTURE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, GL_UNIFORM_BUFFER...



Buffer Objects

◎ Puntos de vinculación

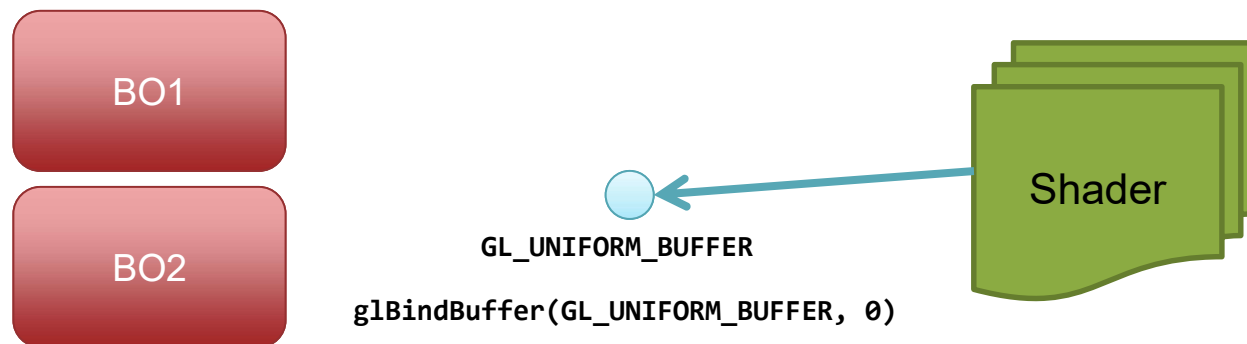
- GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_TEXTURE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, GL_UNIFORM_BUFFER...



Buffer Objects

◎ Puntos de vinculación

- GL_ARRAY_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_TEXTURE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, GL_UNIFORM_BUFFER



Buffer Objects

3. *Reservar espacio para los datos*

- Después de vincular un *buffer object*, hay que reservar espacio para los datos (*data store*)
- **`void glBufferData(GLenum target, GLsizei size, const GLvoid *data, GLenum usage)`**
 - donde:
 - target: igual que antes
 - size: tamaño del buffer en bytes
 - data: puntero a los datos en memoria de CPU o NULL, para reservar memoria no inicializada
 - usage: cómo se va a utilizar la información
- Si el buffer tenía memoria asociada, la libera

Buffer Objects

3. *Reservar espacio para los datos*

- ¿Qué se puede hacer con un buffer?
 - Dibujar (los datos se usarán para dibujar la escena)
 - Leer (los datos se copiarán a memoria de CPU)
 - Copiar (los datos se leerán desde un buffer OpenGL, y luego se usarán para dibujar)
- El resultado de la llamada (consultable desde **glGetError**)
 - **GL_OUT_OF_MEMORY**: si no hay bastante memoria libre en la GPU

Buffer Objects

- Uso del B.O.

Datos generados por	Datos usados por			
	App, poco	GL, poco	App, mucho	GL, mucho
	App, una vez		GL_STREAM_DRAW	GL_STATIC_DRAW
	GL, una vez	GL_STREAM_READ	GL_STREAM_COPY	GL_STATIC_READ
	App, muchas			GL_DYNAMIC_DRAW
	GL, muchas		GL_DYNAMIC_READ	GL_DYNAMIC_COPY

Esta información ayuda a OpenGL a seleccionar el tipo de memoria donde crear el buffer. No limita el uso del mismo

Buffer Objects

- ◎ Cada llamada a `glBufferData` destruye el buffer anterior y copia todos los datos de nuevo
- ◎ Para actualizar parte de un B.O.
 - `void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const Glvoid *data)`
donde:
 - ◎ **target**: igual que antes
 - ◎ **offset**: dirección de comienzo de la zona a actualizar
 - ◎ **size**: tamaño de la zona a actualizar
 - ◎ **data**: dirección del buffer con la nueva información en memoria del cliente

Buffer Objects

6. Liberando VBO

- Cuando ya no se necesite el *buffer*, se debe liberar:
- **`void glDeleteBuffers(GLsizei n, GLuint *buffers)`**
- donde:
 - **`n`**: número de buffers a liberar
 - **`buffers`**: vector con identificadores de BO

NEW

Novedades en GL 4.4 y GL 4.5:

- B.O. inmutables (**`glBufferStorage`**)
- Bindless B.O. (**`glNamedBufferStorage`**, **`glNamedBufferSubdata`**)

Dibujando con OpenGL

Primitivas

OpenGL 3:

- ☒ `GL_POINTS`

- ☒ `GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP`

- ☒ `GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN`

- ☒ ~~`GL_QUADS, GL_QUAD_STRIP`~~

- ☒ ~~`GL_POLYGON`~~

- ☒ `GL_LINES_ADJACENCY, GL_TRIANGLES_ADJACENCY,
GL_LINE_STRIP_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY`

OpenGL 4:

- ☒ `GL_PATCHES`

Dibujando con OpenGL

Puntos

- ⦿ Cada vértice se muestra como un punto en la ventana
- ⦿ Se puede establecer su tamaño:
 - Por programa:
 - ⦿ `glPointSize(GLfloat size) // En la aplicación`
 - En un shader:
 - ⦿ `glEnable(GL_PROGRAM_POINT_SIZE); // En la aplicación`
 - ⦿ `gl_PointSize = 2.5; // En el shader`
- ⦿ Por defecto, siempre aparecen cuadrados
- ⦿ La perspectiva no afecta a su tamaño



Dibujando con OpenGL

Puntos

- © Consultar los tamaños soportados por el HW:
`GLfloat sizes[2], step;`
`glGetFloatv(GL_POINT_SIZE_RANGE, sizes); // min, max`
`glGetFloatv(GL_POINT_SIZE_GRANULARITY, &step); // cambio más pequeño`

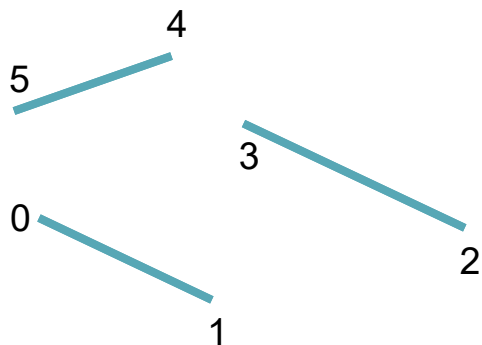
Dibujando con OpenGL

Líneas

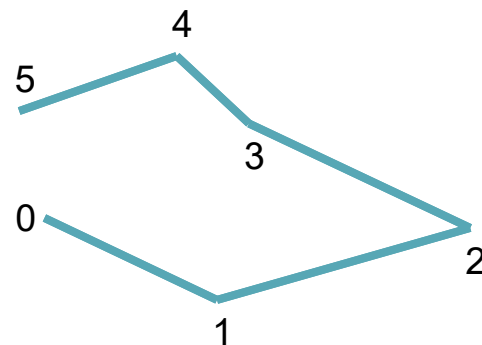
Primitivas:

- **GL_LINES**: dibuja un segmento cada par de vértices
- **GL_LINE_STRIP**: dibuja una polilínea
- **GL_LINE_LOOP**: dibuja una polilínea cerrada

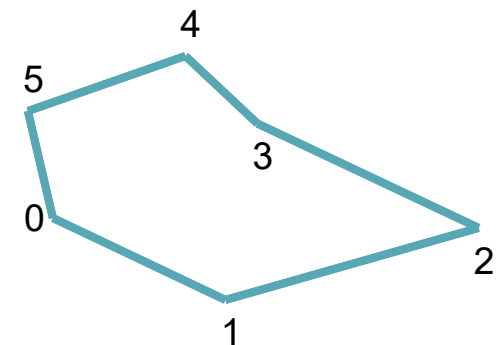
GL_LINES



GL_LINE_STRIP



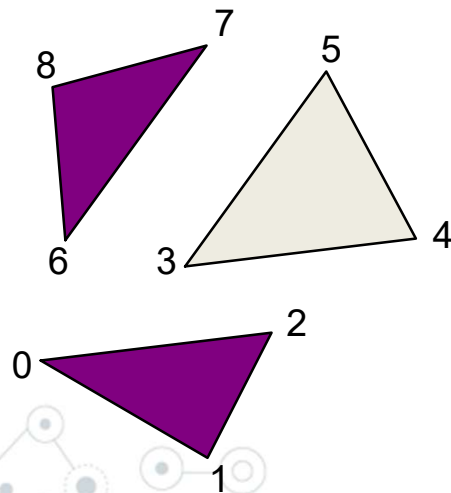
GL_LINE_LOOP



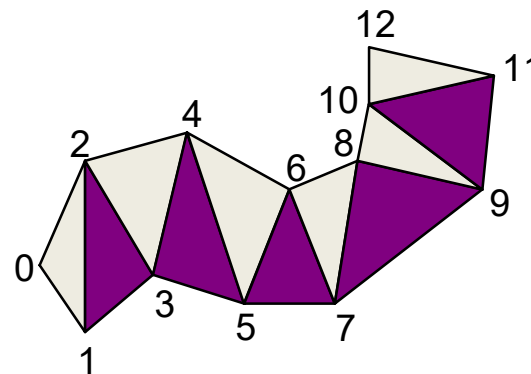
Dibujando con OpenGL

Triángulos

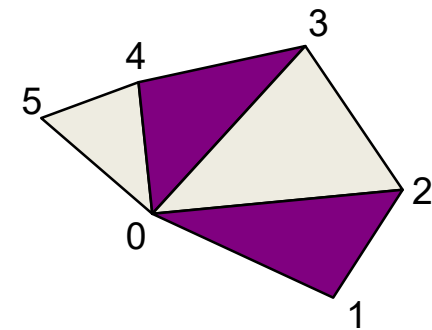
- Por defecto, los vértices de la cara frontal se definen en sentido antihorario
 - El sentido de la cara frontal se puede definir explícitamente con `glFrontFace(GL_CW | GL_CCW)`
- Las cintas (`GL_TRIANGLE_STRIP`) y los abanicos (`GL_TRIANGLE_FAN`) son más eficientes que los triángulos aislados para definir superficies



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Dibujando con OpenGL

🎯 Funciones de dibujo

- Las siguientes funciones enviarán a OpenGL la orden de dibujar una o varias primitivas
- Sólo indican qué vértices usar. La información de los mismos ya debe estar en la GPU (en un BO)
- Hay dos formas de especificar los vértices de una primitiva:
 - 🎯 una porción contigua de un array de vértices,
 - 🎯 un array de índices

Dibujando con OpenGL

Dibujo secuencial

Dibujar GL_TRIANGLES con los siguientes vértices:

Vértices

7 (0,1,1)
3 (0,0,1)
2 (1,0,1)
7 (0,1,1)
2 (1,0,1)
6 (1,1,1)

Dibujo indexado

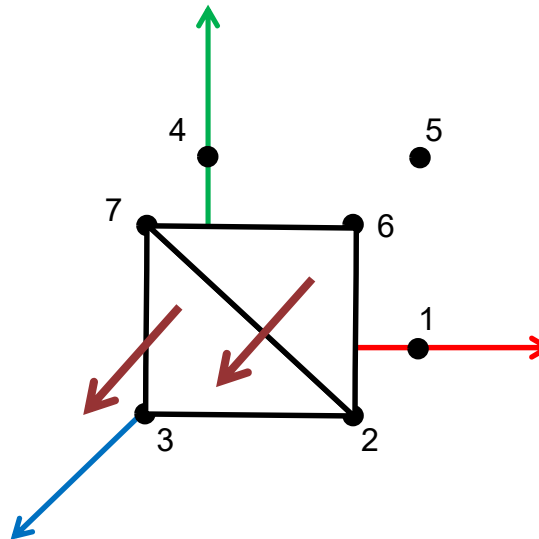
Dibujar GL_TRIANGLES con los índices:

Vértices

0 (0,0,0)
1 (1,0,0)
2 (1,0,1)
3 (0,0,1)
4 (0,1,0)
5 (1,1,0)
6 (1,1,1)
7 (0,1,1)

Índices

{7,3,2,7,2,6}



Dibujando con OpenGL

Funciones de dibujo secuencial

- ◎ Funciones:
 - **void glDrawArrays(GLenum mode, GLint first, GLsizei count)**
 - ◎ donde:
 - **mode**: tipo de primitiva (**GL_TRIANGLES**, **GL_LINES**...)
 - **first**: índice del primer vértice a dibujar
 - **count**: cantidad de vértices a dibujar
 - **void glMultiDrawArrays(mode, GLint *first, GLsizei *count, GLsizei primcount)**
 - ◎ donde:
 - **first**: array con el índice del primer vértice de cada primitiva
 - **count**: array con la cantidad de vértices de cada primitiva
 - **primcount**: cantidad de primitivas (y tamaño de **first** y **count**)

Los datos están
vinculados en:

GL_ARRAY_BUFFER

```
GLint first[] {10, 20, 30};  
GLsizei count[] {5, 6, 7};  
glMultiDrawArrays(GL_TRIANGLE_STRIP, first, count, 3);
```

Dibujando con OpenGL

Funciones de dibujo indexado

⦿ Funciones:

- `void glDrawElements(mode, count, GLenum type, void *indices)`

⦿ donde:

- **mode**: tipo de primitiva (`GL_TRIANGLES`, `GL_LINES` ...)
- **count**: cantidad de índices
- **type**: tipo de los índices (`GL_UNSIGNED_BYTE` | `GL_SHORT` | `GL_INT`)
- **indices**: puntero al primer índice. ¡Cuidado! ¡¡No es un puntero a memoria del cliente, sino un desplazamiento desde el inicio de un BO en GPU!!

Los vértices están vinculados en: `GL_ARRAY_BUFFER` y los índices, en: `GL_ELEMENT_ARRAY_BUFFER`

Dibujando con OpenGL

Funciones de dibujo indexado

⦿ Funciones:

- **void glDrawRangeElements(mode, GLuint start, GLuint end, count, type, indices)**
 - ⦿ donde:
 - **mode, count, type, indices:** igual que antes
 - **start, end:** mínimo y máximo índice que puede aparecer dentro de índices
 - ⦿ Aporta más información a OpenGL sobre la información que se utilizará, para que pueda optimizar el acceso a la memoria

Dibujando con OpenGL

Funciones de dibujo indexado

⦿ Funciones:

- `void glMultiDrawElements(mode, GLsizei *count, type, const void **indices, GLsizei primcount)`

⦿ donde:

- **mode, type:** igual que antes
- **count:** vector de *primcount* elementos con el número de índices de cada primitiva
- **indices:** vector de punteros al primer índice de cada primitiva
- **primcount:** número de primitivas

⦿ Equivale a:

```
for (i=0; i<primcount; i++) {  
    if (count[i] > 0)  
        glDrawElements(mode, count[i], type, indices[i]);  
}
```

Dibujando con OpenGL

Resumen de funciones de dibujo

	No indexado	Indexado
Directo	<code>glDrawArrays</code>	<code>glDrawElements</code> , <code>glDrawElementsBaseVertex</code> , <code>glDrawRangeElements</code> , <code>glDrawRangeElementsBaseVertex</code>
Indirecto	<code>glDrawArraysIndirect</code>	<code>glDrawElementsIndirect</code>
Multi	<code>glMultiDrawArrays</code>	<code>glMultiDrawElements</code> , <code>glMultiDrawElementsBaseVertex</code>
Multi+indirecto	<code>glMultiDrawArraysIndirect</code>	<code>glMultiDrawElementsIndirect</code>
Instanciado	<code>glDrawArraysInstanced</code> <code>glDrawArraysInstancedBaseInstance</code>	<code>glDrawElementsInstanced</code> , <code>glDrawElementsInstancedBaseVertex</code> , <code>glDrawElementsInstancedBaseInstance</code> , <code>glDrawElementsInstancedBaseVertexBaseInstance</code>

Dibujando con OpenGL

Reinicio de primitivas

- ⦿ En las funciones de dibujo indexado, se puede introducir un índice especial para marcar el fin de una primitiva, y el inicio de otra
 - Funciona para cintas y abanicos de triángulos, y para polilíneas y polilíneas cerradas
- ⦿ Para activar esta funcionalidad hay que:

```
glEnable(GL_PRIMITIVE_RESTART);
glPrimitiveRestartIndex(0xFFFF);
glDrawElements(GL_TRIANGLE_STRIP, ...);
glDisable(GL_PRIMITIVE_RESTART);
```
- ⦿ En el ejemplo anterior, cada vez que OpenGL se encuentre el índice 65535 (0xFFFF), terminará la cinta actual y empezará una nueva en el siguiente índice

Dibujando con OpenGL

VBO y VAO

- ⦿ La aplicación debe proporcionar a OpenGL la información necesaria para dibujar las primitivas
 - Vértices, normales, colores, coordenadas de textura y cualquier otra información
- ⦿ Cada vértice, aparte de la posición, lleva asociado otros *atributos*, que se almacenan en uno o varios *Vertex Buffer Objects* (VBO)
- ⦿ OpenGL obliga a permitir como mínimo 16 atributos por vértice
- ⦿ Pueden ser floats, ints o boolean, y siempre se almacenan como vectores 4D de floats
- ⦿ Si falta algún componente, se aumenta a (X, 0, 0, 1)



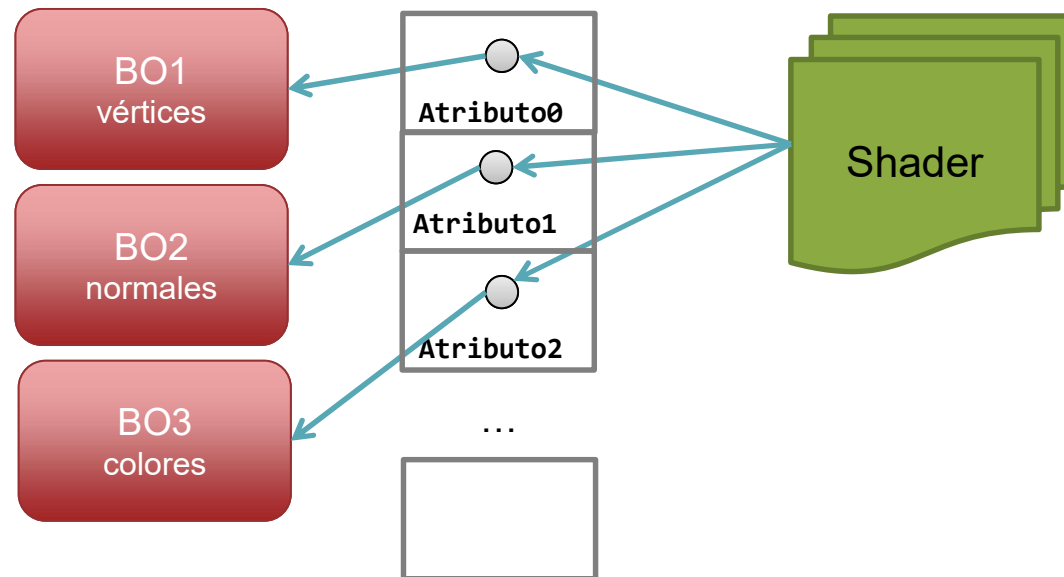
Dibujando con OpenGL

VBO y VAO

- ⦿ Obtener el número máximo de atributos en la máquina:
`glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &m)`
 - ⦿ Cada atributo tiene un índice, que empieza en 0
 - ⦿ Se asocian a variables del shader
 - ⦿ Sólo accesibles por el shader de vértice
-
- ⦿ Base de datos de características de GPU:
<http://opengl.gpuinfo.org/>

Dibujando con OpenGL

VBO y VAO



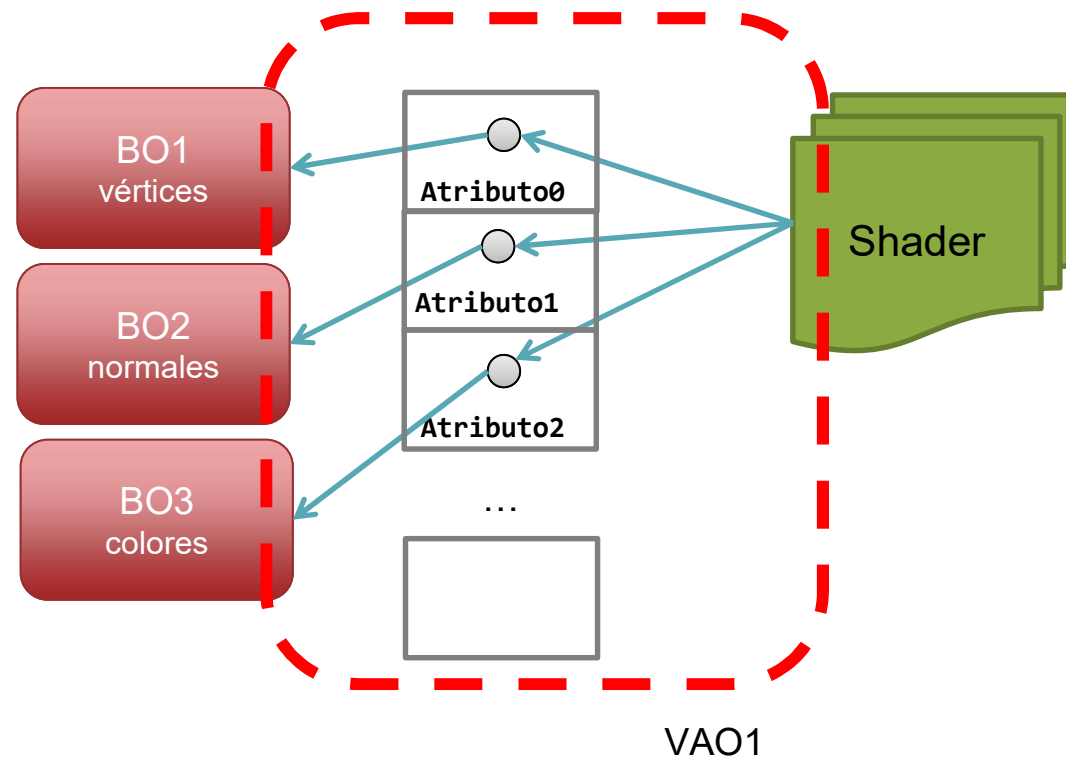
Dibujando con OpenGL

VBO y VAO

- ⦿ Como un modelo puede usar varios VBOs, OpenGL introdujo los *Vertex Array Objects* (VAO) para gestionar el estado de las vinculaciones de una primitiva
- ⦿ Los VAO permiten cambiar con una sola llamada todos los enlaces necesarios entre un shader, puntos de vinculación y VBO
- ⦿ Es obligatorio vincular un VAO antes de vincular los atributos con los VBO
- ⦿ En general, habrá un VAO por cada modelo de la escena

Dibujando con OpenGL

VBO y VAO



Dibujando con OpenGL

VBO y VAO

Generación de identificadores de VAO

- **void glGenVertexArrays(GLsizei n, GLuint *arrays)**
 - **n, arrays:** igual que glGenBuffers

Vinculación de un VAO

- **void glBindVertexArray(GLuint array)**
 - **array:** identificador del VAO a vincular, o 0 para desvincular el VAO actual

Liberación de un VAO

- **void glDeleteVertexArrays(GLsizei n, GLuint *arrays)**
 - Si el VAO está vinculado, se eliminará y no habrá ningún VAO vinculado


Dibujando con OpenGL

VBO y VAO

- ◎ Para almacenar atributos de vértice en un BO, hay que vincularlo a:
 - **GL_ARRAY_BUFFER**: para atributos de vértice (posiciones, colores, normales, etc)
 - **GL_ELEMENT_ARRAY_BUFFER**: para índices
- ◎ Una vez vinculado, se pueden usar las funciones vistas en el apartado de BO para rellenar los buffers


Dibujando con OpenGL

VBO y VAO

- 
- El siguiente paso es decir a OpenGL cómo interpretar los datos del VBO
 - `void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer)`
 - donde:
 - index**: índice de atributo
 - size**: 1, 2, 3, 4
 - type**: tipo de los datos apuntados por **pointer**: `GL_BYTE`, `GL_SHORT`, `GL_INT` (y sus versiones *unsigned*), `GL_FLOAT`, `GL_DOUBLE`...
 - normalized**: para enteros, si `GL_TRUE`, convertir a `[0..1]` o `[-1..1]`

Dibujando con OpenGL

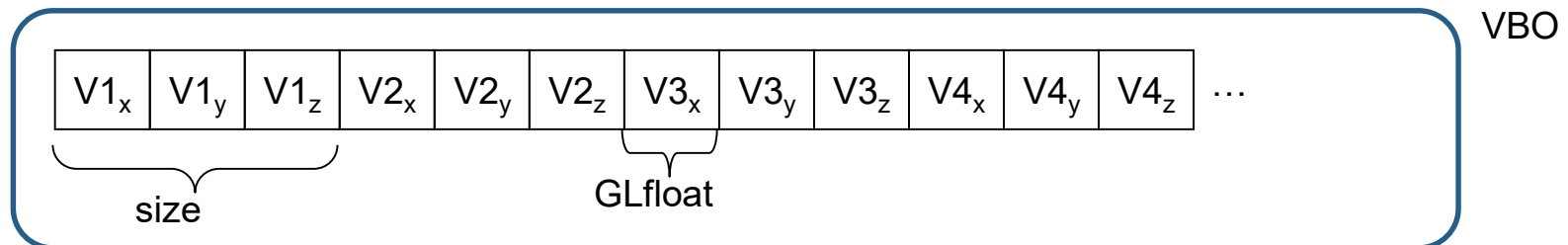
VBO y VAO

- 
- El siguiente paso es decir a OpenGL cómo interpretar los datos del VBO
 - `void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer)`
 - donde:
 - stride**: distancia en bytes entre los datos de dos vértices consecutivos, o 0 si están contiguos
 - pointer**: desplazamiento desde el inicio del VBO
 - `glVertexAttribPointer` siempre almacena floats en el BO. Usa `glVertexAttribIPointer` para almacenar ints, y `glVertexAttribLPointer` para almacenar doubles

Dibujando con OpenGL

VBO y VAO

- Organización en memoria de los atributos
 - Un VBO por atributo

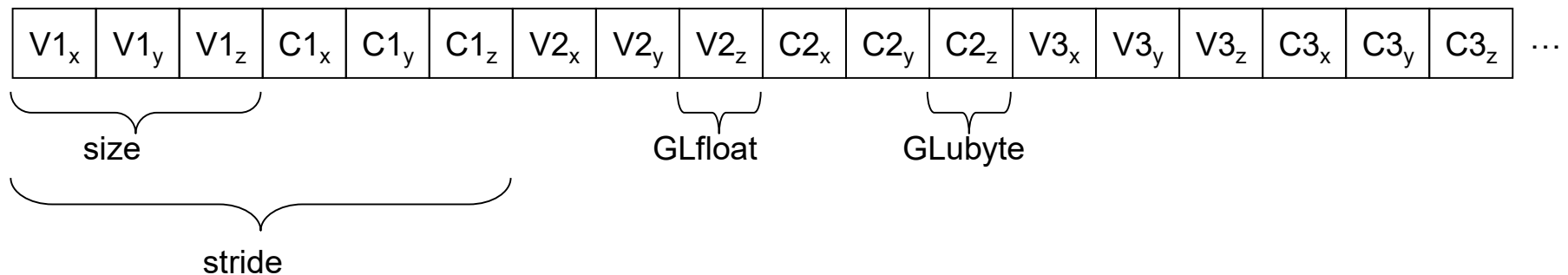


```
glVertexAttribPointer(index, 3, GL_FLOAT, GL_FALSE, 0, (const GLvoid *)0)  
glVertexAttribPointer(index, size, type, normalized, stride, pointer)
```

Dibujando con OpenGL

VBO y VAO

- Organización en memoria de los atributos
 - Dos atributos en un VBO



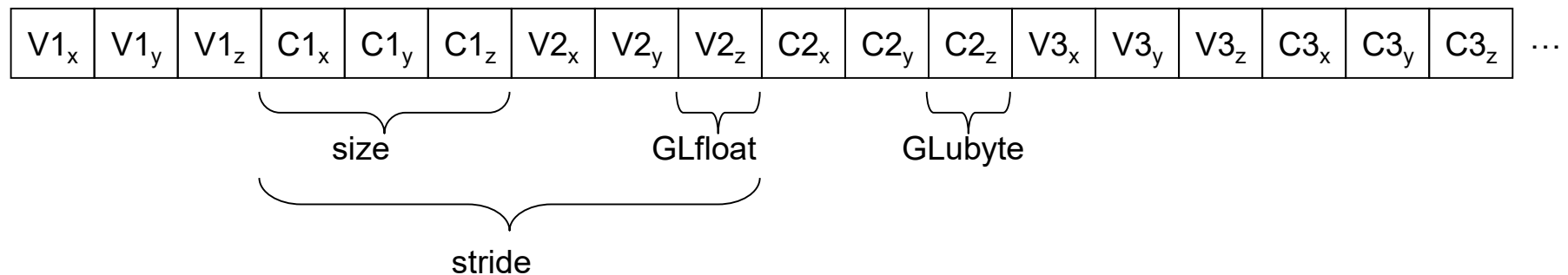
```
glVertexAttribPointer(indexV, 3, GL_FLOAT, GL_FALSE,  
    3*sizeof(GLfloat)+3*sizeof(GLubyte), (const GLvoid *)0)
```

```
glVertexAttribPointer(index, size, type, normalized, stride, pointer)
```

Dibujando con OpenGL

VBO y VAO

- Organización en memoria de los atributos
 - Dos atributos en un VBO



```
glVertexAttribPointer(indexC, 3, GL_UNSIGNED_BYTE, GL_TRUE,  
    3*sizeof(GLfloat)+3*sizeof(GLubyte), (const GLvoid *) (3*sizeof(GLfloat)))
```

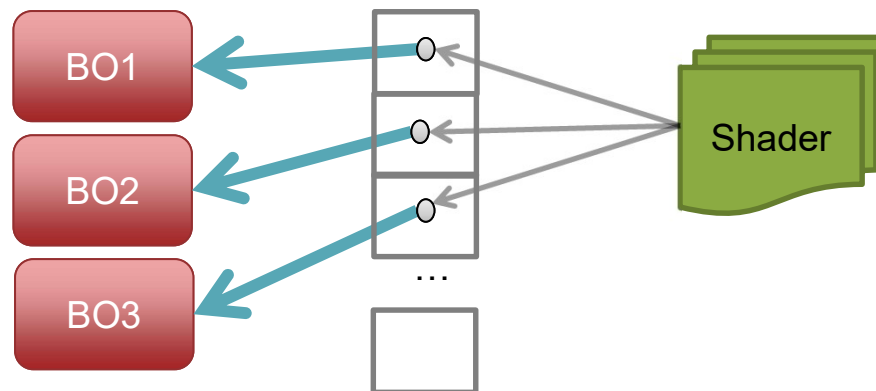
```
glVertexAttribPointer(index, size, type, normalized, stride, pointer)
```

Dibujando con OpenGL

VBO y VAO

⦿ Aparte de vincular un atributo con un VBO, hay que activar el uso de dicho atributo:

- **void glEnableVertexAttribArray(GLuint index)**
 - ⦿ **index:** índice del atributo que se leerá desde un VBO.
- **void glDisableVertexAttribArray(GLuint index)**
 - ⦿ **index:** índice del atributo en el que se usará un valor estático



Dibujando con OpenGL

VBO y VAO

- ◎ Para especificar un valor de atributo que usarán todos los vértices (atributo de vértice estático):
 - Deshabilitar el uso del array (`glDisableVertexAttribArray`)
 - Establecer el valor con alguna función de la familia `glVertexAttrib*`.
 - Ejemplos:
 - ◎ `glVertexAttrib4f(indexC, 1.0, 0.0, 0.0, 1.0);`
 - ◎ `glVertexAttrib4fv(indexC, &color[0]);`

Dibujando con OpenGL

VBO y VAO

- Los índices usados por `glDrawElements` se extraerán del VBO vinculado al punto de vinculación **`GL_ELEMENT_ARRAY_BUFFER`**
- No hay funciones para especificar el tipo de los índices ni el tamaño. Se hace en la llamada de dibujo:
 - `glDrawElements(GL_LINE_STRIP, 100, GL_UNSIGNED_SHORT, (const GLvoid *)0)`

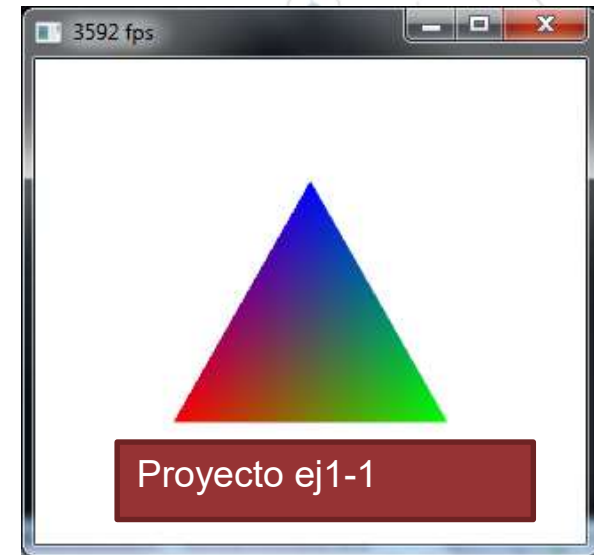
Dibujando con OpenGL

Ejemplo dibujo secuencial

```
GLfloat vertices[] = {-0.5, -0.5, 0.0,
                      0.5, -0.5, 0.0,
                      0.0, 0.5, 0.0};
GLfloat colores[] = {1.0, 0.0, 0.0,
                    0.0, 1.0, 0.0,
                    0.0, 0.0, 1.0};

GLuint vao, vbos[2];
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glGenBuffers(2, vbos);
glBindBuffer(GL_ARRAY_BUFFER, vbos[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (const void *)0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, vbos[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(colores), colores, GL_STATIC_DRAW);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, (const void *)0);
glEnableVertexAttribArray(2);
```



Dibujando con OpenGL

Ejemplo dibujo secuencial

Para dibujar el modelo:

```
glBindVertexArray(vao);  
glDrawArrays(GL_TRIANGLES, 0, 3);
```



Dibujando con OpenGL

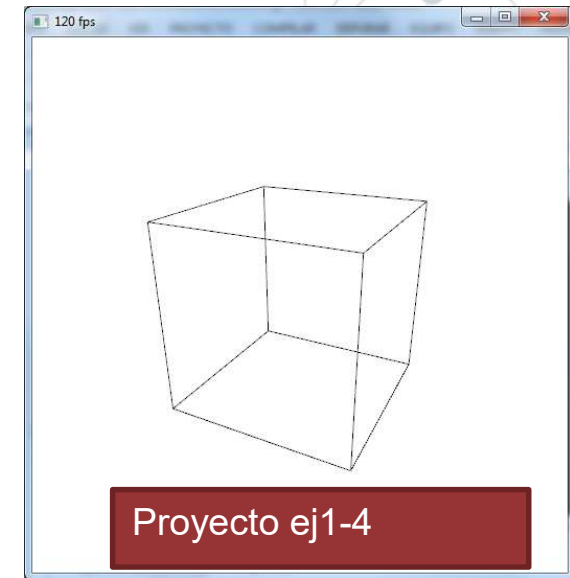
Ejemplo dibujo indexado

```
// Definición de los vértices e índices del modelo
GLfloat vertices[][3] = {
    -0.5f, 0.5f, 0.5f,  0.5f, 0.5f, 0.5f, ...};
GLushort indices[] = { 0, 4, 0, 3, 0, 1, ...};

glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glGenBuffers(2, vbos);
glBindBuffer(GL_ARRAY_BUFFER, vbos[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (const void *)0);
glEnableVertexAttribArray(0);

// Para el color, establecemos el atributo genérico 2 al color negro
glDisableVertexAttribArray(2);
glVertexAttrib4f(2, 0.0, 0.0, 0.0, 1.0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbos[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```



Dibujando con OpenGL

Ejemplo dibujo indexado

Para dibujar el modelo:

```
glBindVertexArray(vao);  
glDrawElements(GL_LINES, 24, GL_UNSIGNED_SHORT, 0);
```

