h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

# Informática visual

## Objetos gráficos y su programación

Universidad de Darmstadt

Prof. Dr. Elke Hergenröther
Björn Frömmer
Prof. Dr. Benjamin Meyer

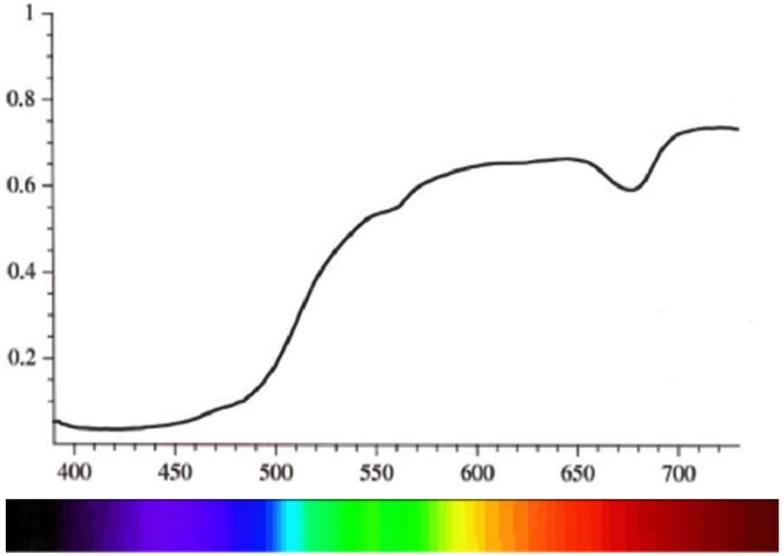
# CAPÍTULO 3

## Colores y primitivas

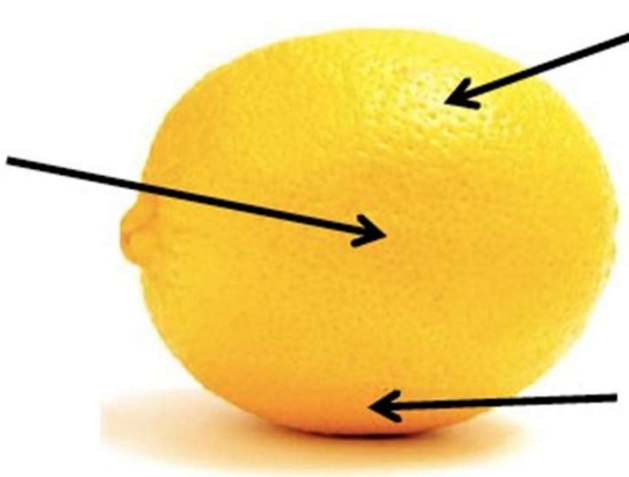
# Espacios de

- ¿Qué es el color y cómo se almacena?
- Física: Luz reflejada

Luz azul absorbida

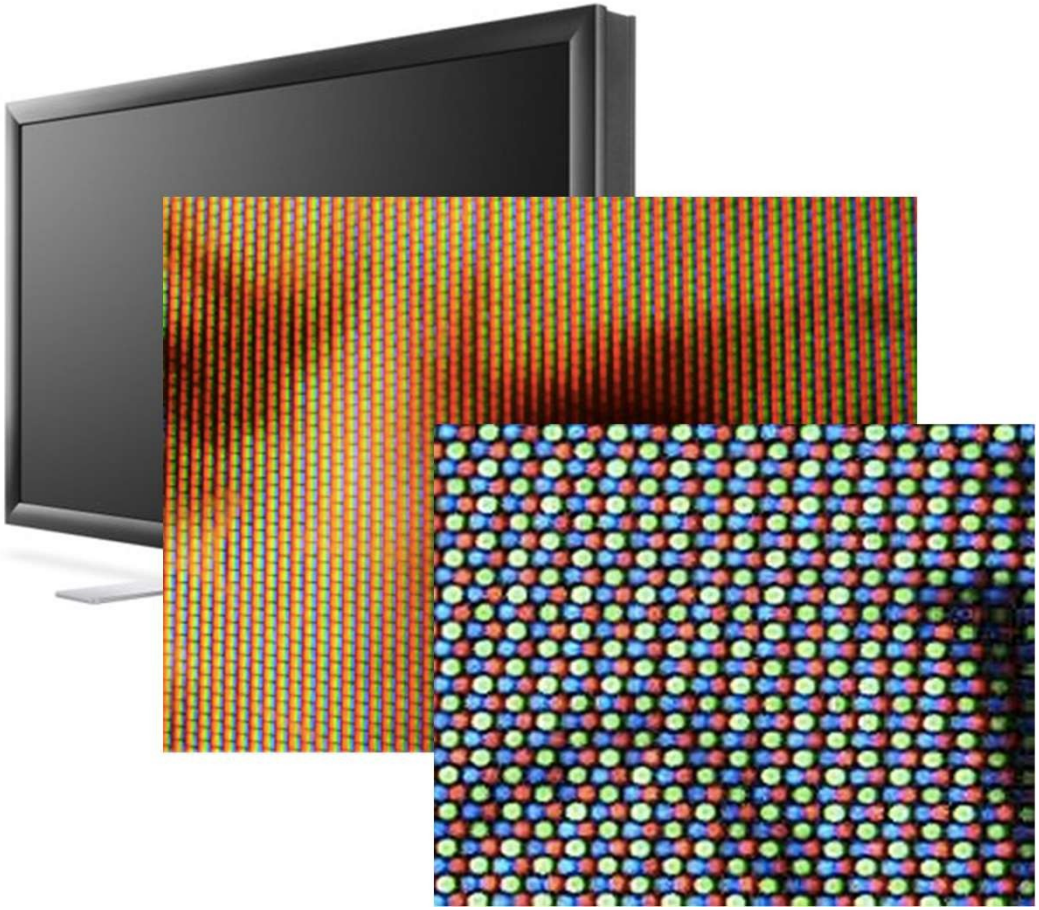


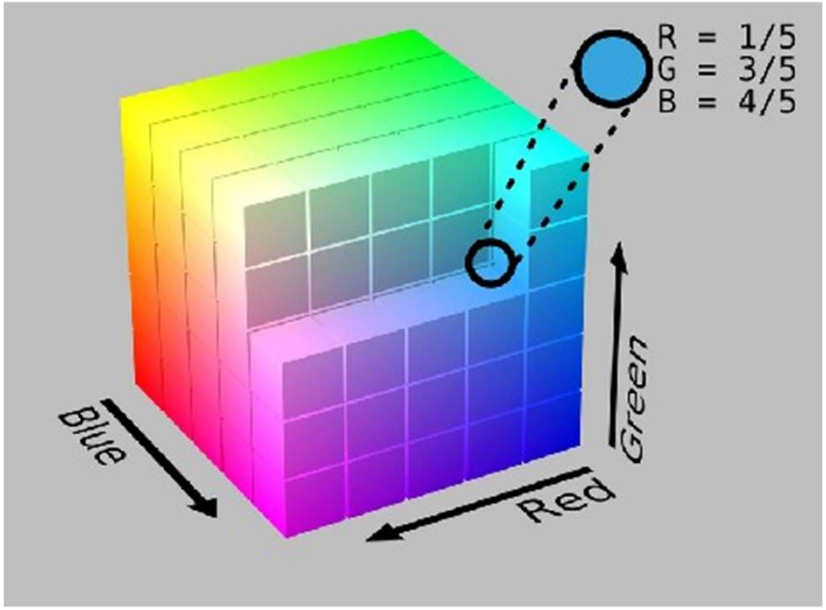

Aspectos especulativos destacados



Zonas oscurecidas

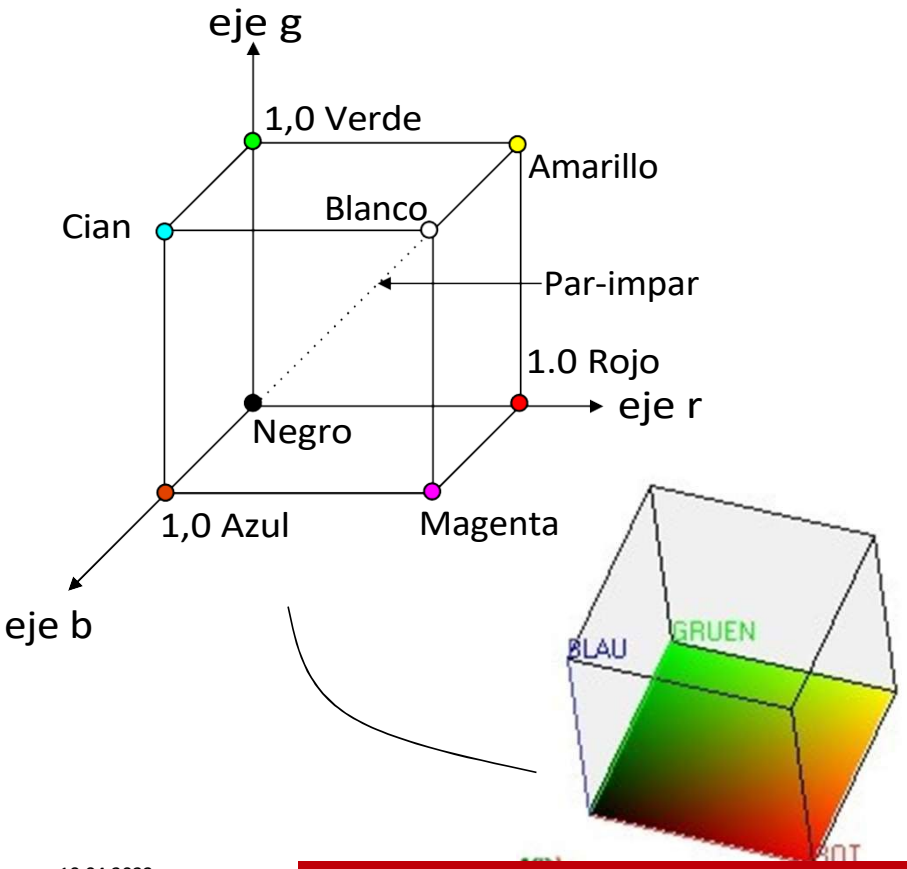
# Espacios de






Espacio de color RGB

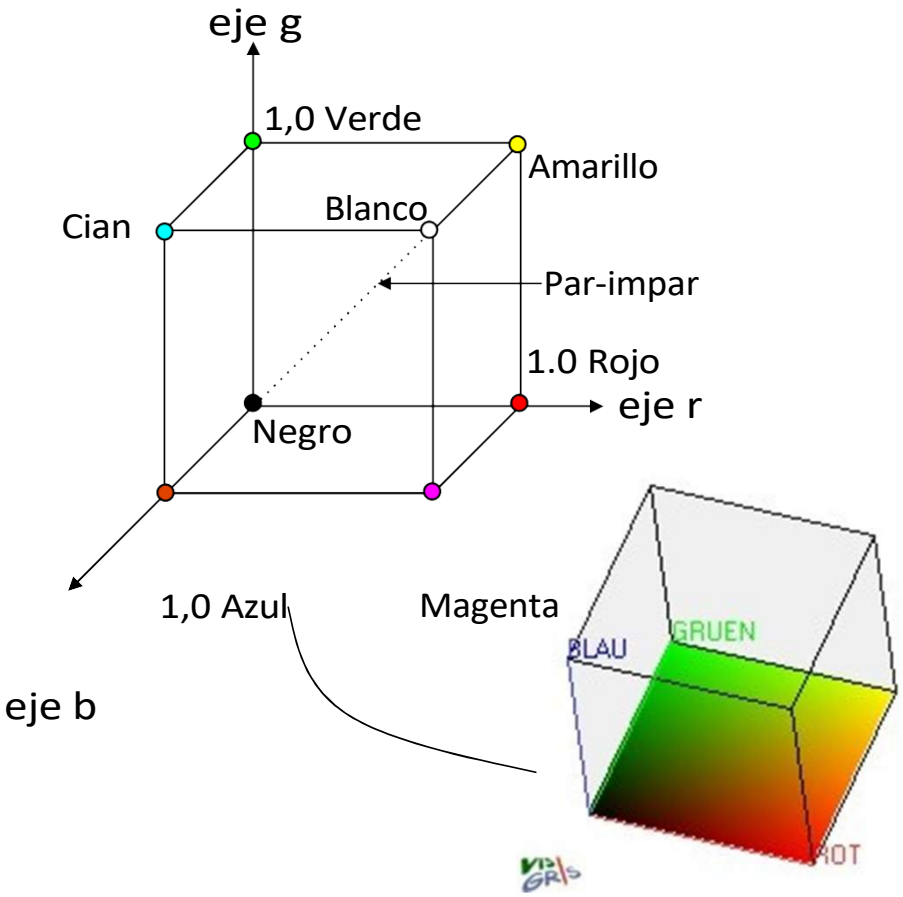
# El espacio de color




## General:

- Espacio de color 3D
- Los valores de las coordenadas deben estar comprendidos entre 0 y 1.
- Color en la superficie del cubo o en su interior
- El color se describe mediante un vector 3D:
  - Color = $[r, g, b]^t$
  - Rojo = $[1, 0, 0]^t$
  - Origen: Negro $[0, 0, 0]^t$
- brillo mínimo: $[0, 0, 0]^t_{RGB}$
- **¿brillo máximo?**

# El espacio de color




## Composición de un color:

mezcla aditiva de

colores amarillo= rojo

+ verde

$= [1,0,0]^t + [0,1,0]^t$

$= [1,1,0]^t$

Blanco= Rojo + Verde + Azul

$= [1,0,0]^t + [0,1,0]^t + [0,0,1]^t$

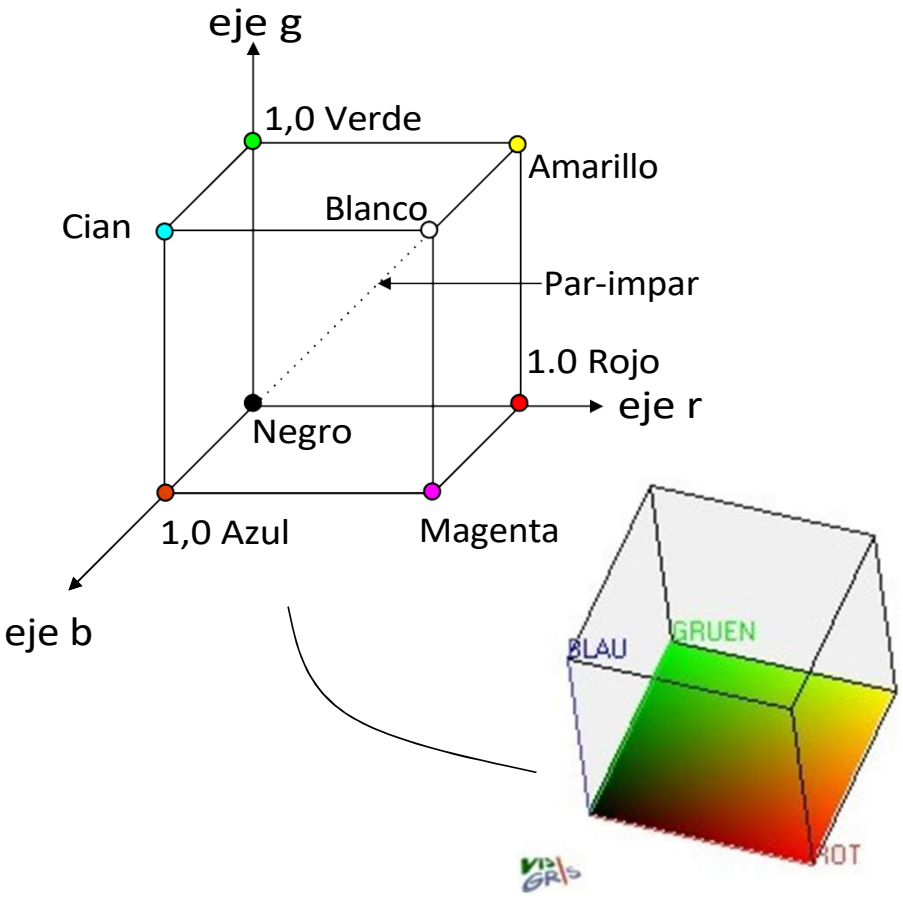
$= [1,1,1]^t$

Amarillo (medio $[0,5,0,5,0]^t$

claro) = Cian = $[0,1,1]^t$

Magenta = $[1,0,1]^t$

# El espacio de color



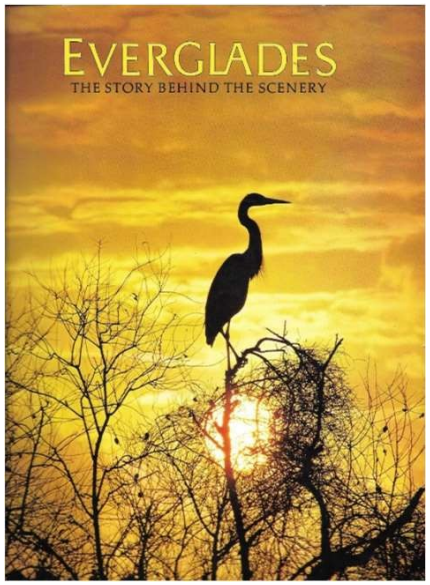

## Propiedades del modelo RGB

- El aspecto del color viene determinado principalmente por el componente o componentes más importantes.
- Si todos los componentes del color tienen el mismo valor, se trata de un tono gris (**acromático**).
- **No** se tiene en cuenta en el modelo:
  - Diferencias de luminosidad en los tonos azules y verdes: véase percepción de los colores.

# Ejercicio: Modelo de color RGB

La imagen se dividió en un canal rojo, uno verde y uno azul.

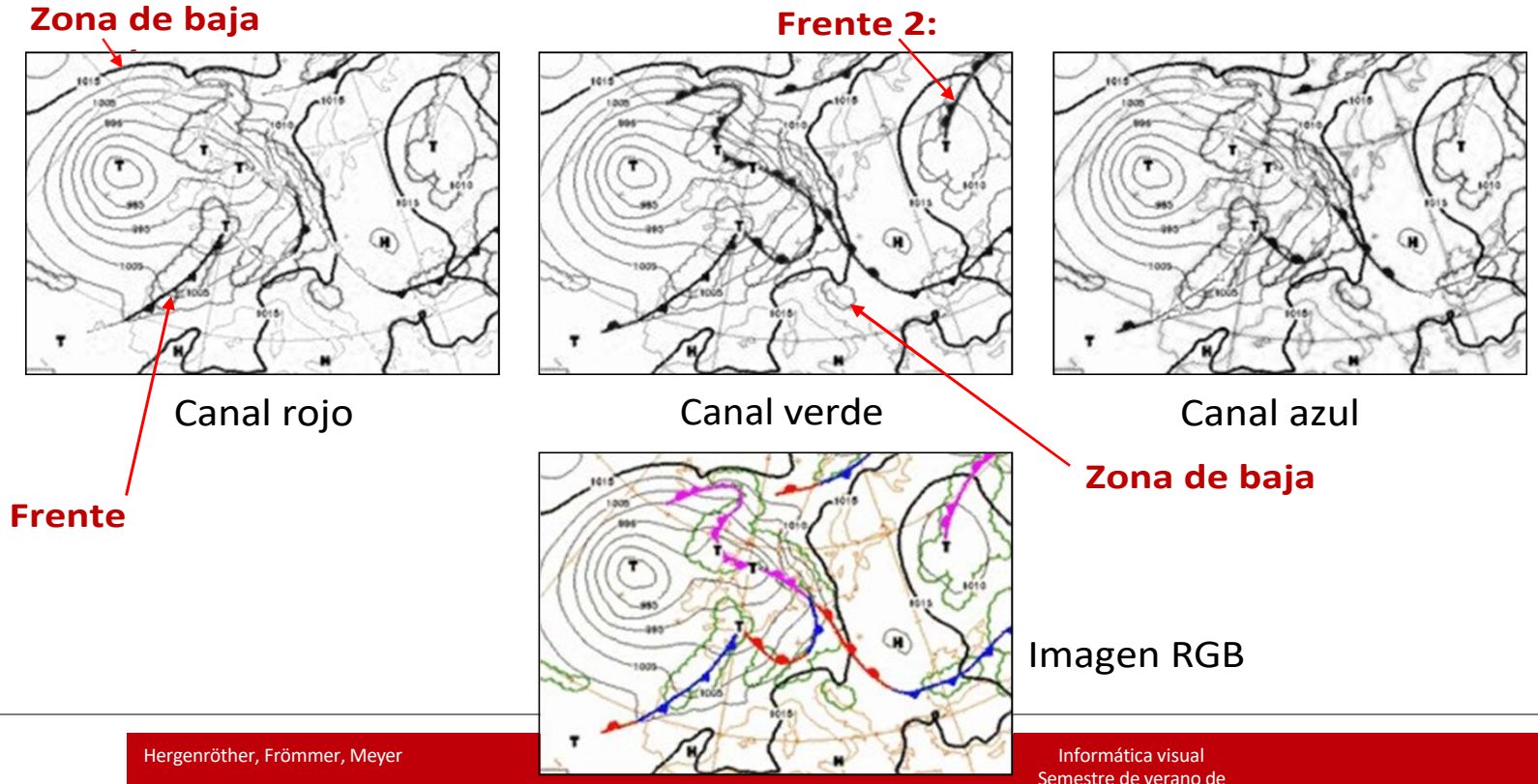
- ¿De qué color es la inscripción "EVERGLADES"?
- ¿De qué colores son el cielo, el sol y el pájaro?

# Ejercicio: Modelo de color RGB

La imagen se dividió en un canal rojo, uno verde y uno azul.

- ¿De qué color son los frentes marcados y las zonas de bajas presiones?



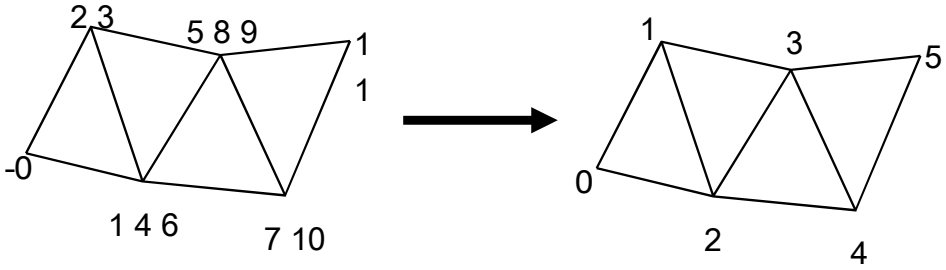
# Ampliación del sistema RGB a RGBA

Imagen en color de 32 bits:

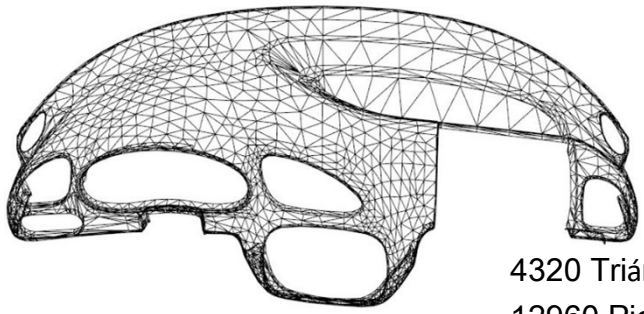
| Sample Length: | 8 | | | | | | | | 8 | | | | | | | | 8 | | | | | | | | 8 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Channel Membership: | Alpha | | | | | | | | Red | | | | | | | | Green | | | | | | | | Blue | | | | | | | |
| Bit Number: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

# Tiras triangulares / Tiras triangulares

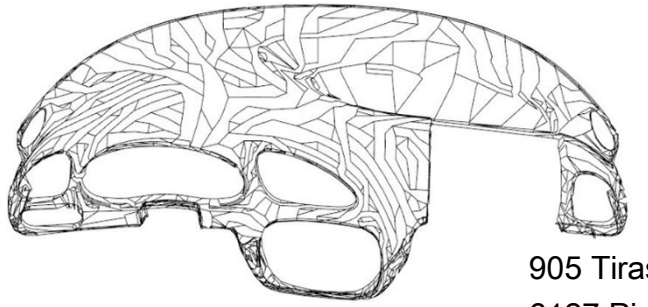
- El objetivo es crear el menor número posible de elementos/verticales.
- Los puntos de esquina pueden reciclarse mediante bandas conectadas.




*Número de puntos sin rayas y con rayas*



4320 Triángulos
12960 Piedras angulares

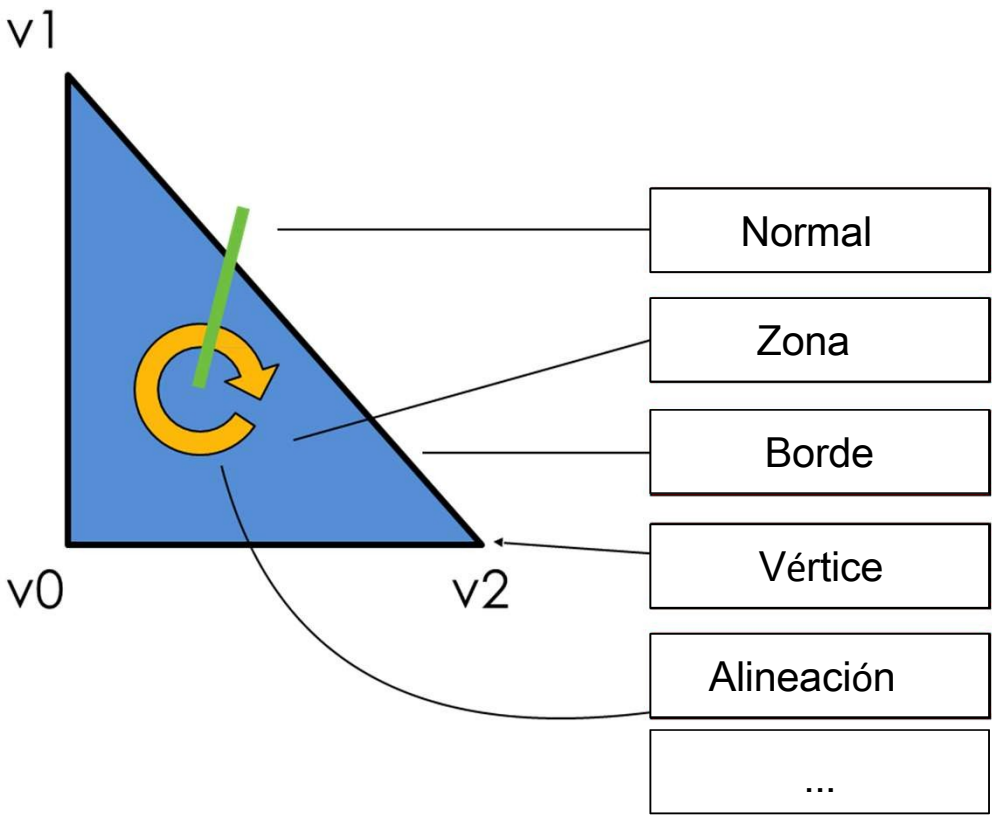


905 Tiras
6127 Piedras angulares

# Descripción de la escena

- ¿Qué es una primitiva? Consiste en
  - Puntos de esquina/vértices
  - Bordes
  - Zonas
  - Alineación
  - Normal

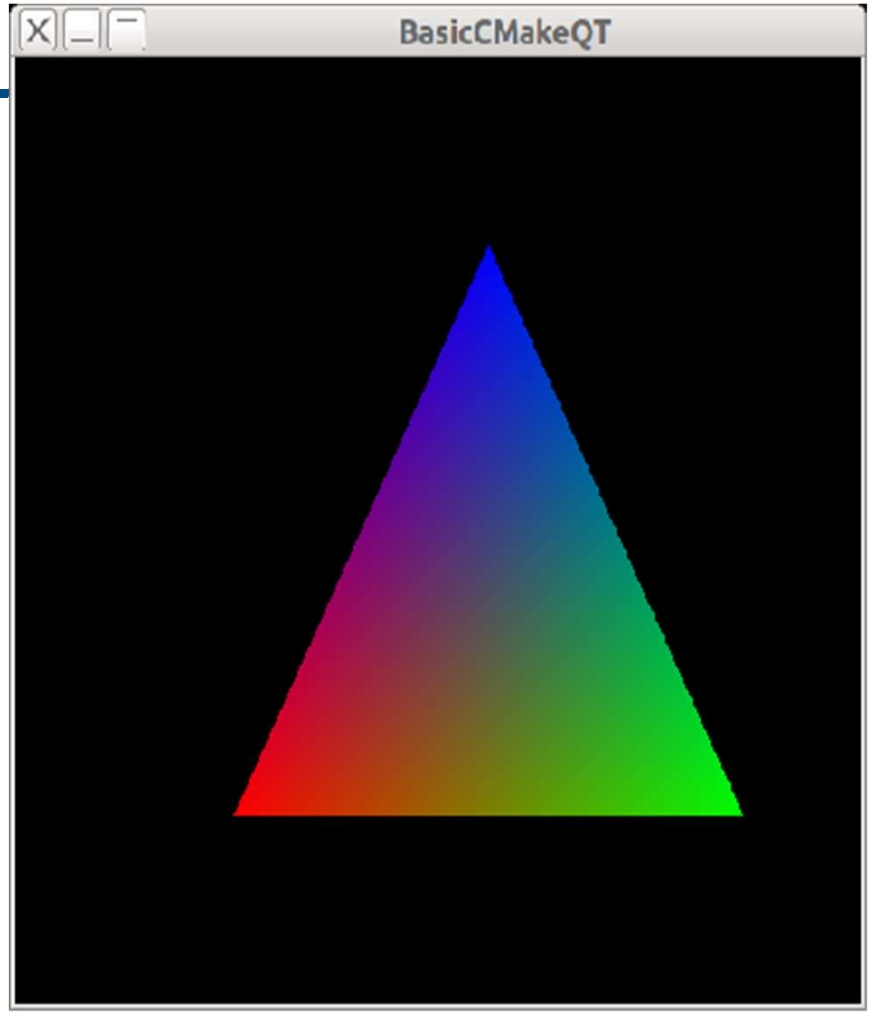
- **Pero todo esto puede deducirse de los vértices.**



# Descripción de la escena II

## Ejemplo:

- Guardar vértices
  - Coordenadas 2D y colores RGB
  - (5D total)



# CAPÍTULO 4

## OpenGL

# Información general

- OpenGL (Open Graphics Library) es una especificación de una API para gráficos 3D
- OpenGL especifica (de forma estandarizada) unos 250 comandos
- La implementación de los comandos se encuentra en los controladores de la tarjeta gráfica
  - A continuación, la tarjeta gráfica ejecuta los comandos
  - o en la CPU
- OpenGL es un sistema de renderizado, no un programa de modelado: los modelos complejos deben construirse a partir de primitivas gráficas sencillas.

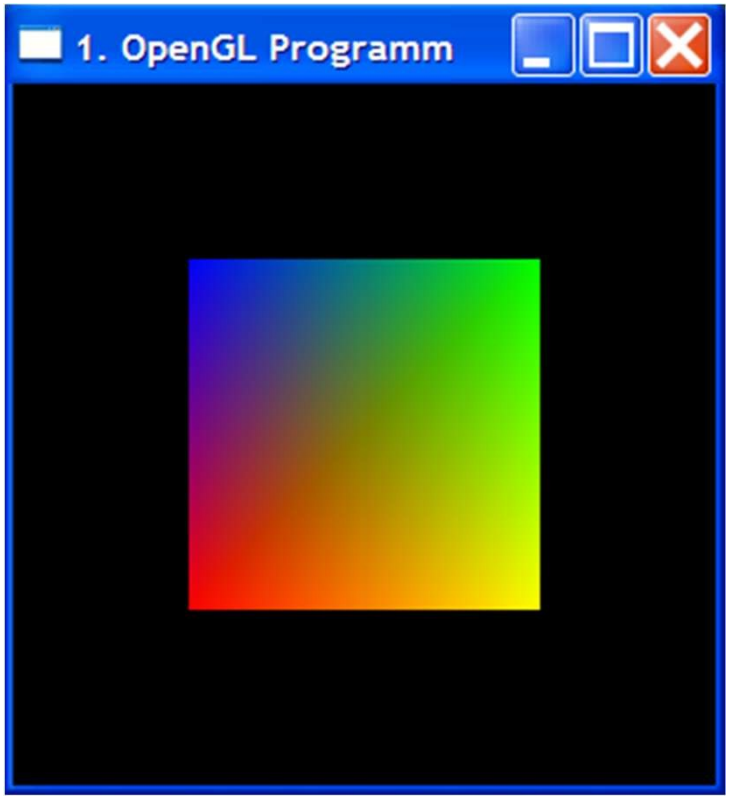
# Información general sobre OpenGL II

- OpenGL es una **máquina de estados:**
  - Las funciones cambian el estado interno o lo utilizan para la representación.
  - Esto significa que, una vez encendido, el estado respectivo permanece activo hasta que se apaga de nuevo o se conmuta.
- OpenGL es muy "**explícito**":
  - Lo que no se ha activado explícitamente permanece desactivado.
  - Ejemplo: No sirve de nada fijar la transparencia si no se ha dicho explícitamente que se calculen las transparencias.
- Utilizamos **GLFW** (Graphics Library Framework), una biblioteca multiplataforma de código abierto para OpenGL.
  - Más GLEW como extensión para determinadas funciones adicionales
  - La única alternativa: GLUT (OpenGL Utility Toolkit), o **FreeGlut** como desarrollo posterior.

# Código heredado (OpenGL 1.x)

```
glBegin( GL_POLYGON );

        // Estado-Máquina: Si sólo se especifica el
           primer valor de color tiene
        // todos los puntos de esquina siguientes el
           mismo valor de color

        glColor4f( 1., 0., 0., 1.);
        glVertex3f( -0.5, -0.5, 0);

        glColor4f( 1., 1., 0., 1.); // Amarillo
        glVertex3f( 0.5, -0.5, 0);

        glColor4f( 0., 1., 0., 1.);
        glVertex3f( 0.5, 0.5, 0);

        glColor4f( 0., 0., 1., 1.);
        glVertex3f( -0.5, 0.5, 0);

glEnd();
```

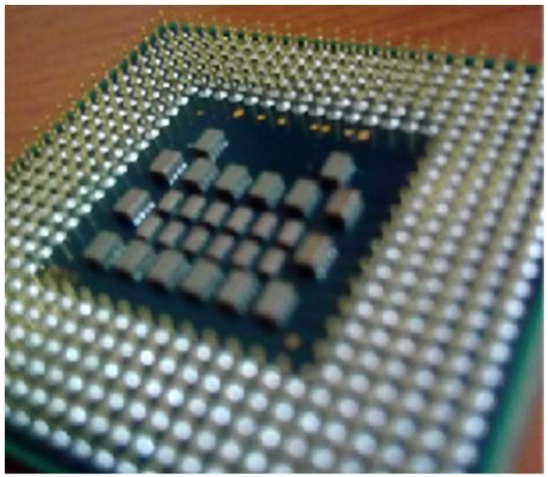
Salida del programa OpenGL:




La carga de datos de la CPU a la GPU es muy lenta.

# Interacción CPU y GPU

- **Hoy**: OpenGL ofrece funciones dirigidas tanto a la CPU como a la GPU



©Lamproslefteris

Objetivo: Cargar una gran cantidad de datos en la GPU. Acelera el proceso enormemente.

Carga de datos en la GPU

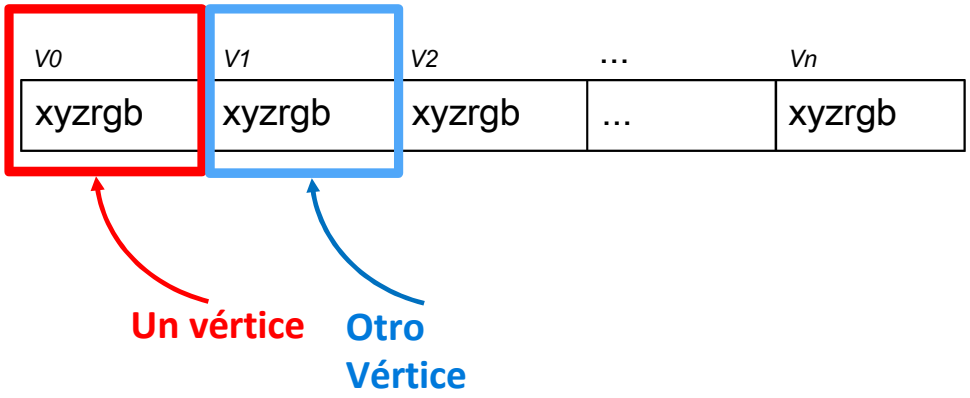
©Advanced Micro Devices (AMD)

- Creación/manipulación de datos
- Lógica del programa

- Tratamiento de datos
- Generación de imágenes

# Creación de datos

- La geometría se almacena como una matriz de datos contigua
  - **Intercalados**: todos los datos de un *Vértice Vi* se suceden




- En código fuente:

```
float vertices[] = {-0.5, -0.5, 0.0, 0.0, 1.0, // Aquí en 2D, xyrgb
                     0.5, -0.5, 0.0, 0.0, 1.0,
                     0.5, 0.5, 0.0, 1.0, 0.0,
                     0.0, 1.0, 1.0, 0.0, 0.0,
                     -0.5, 0.5, 0.0, 1.0, 0.0};
```

# El objeto Vertex Buffer

- Los datos geométricos deben cargarse en la memoria de la GPU
  - Para ello, OpenGL proporciona búferes de memoria especiales, los llamados "búferes de memoria". **Objetos de búfer de vértices (VBO)**
- Los objetos de la GPU pueden referenciarse mediante identificadores (únicos)
- La transferencia de datos a la GPU (casi) siempre se realiza en tres pasos:
  - Generar un ID
  - Activación de la memoria intermedia correspondiente (véase "Máquina de estados")
  - Cargar los datos

# VBOs en OpenGL (visión

## Generar ID

- `void glGenBuffers(GLsizei n, GLuint * buffers);`
  - crea y devuelve un (o n) ID de búfer, que se almacenan en `búferes`
  - `n:` Especifica el número de objetos de búfer que se generarán, normalmente 1.

## Activar el búfer correcto

- `void glBindBuffer(GLenum target, GLuint bufferID);`
  - `target:` Tipo de buffer a describir
    - `GL_ARRAY_BUFFER:` Buffer con los datos reales de la geometría
    - `GL_ELEMENT_ARRAY_BUFFER:` Buffer con índices a otro VBO
  - `bufferID:` El ID del VBO generado previamente con glGenBuffers.

## Borrar datos

- `glDeleteBuffers(GLuint bufferID);`
  - Borrar un buffer con el ID `bufferID`

# VBOs en OpenGL (visión

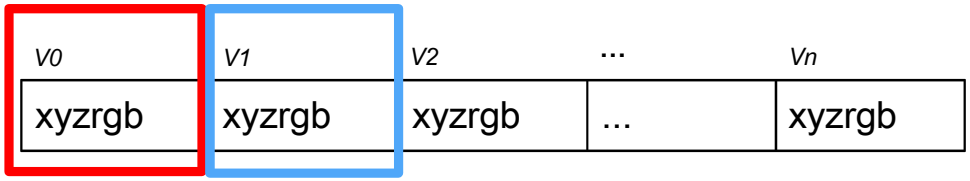
## Carga de datos en la GPU

- `glBufferData(GLenum target, GLsizeiptr size, const void * data, GLenum usage);`
  - `target`: Tipo de buffer en el que se va a escribir (como para `glBindBuffer`).
    - `GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, ...`
  - `tamaño`: tamaño de los datos a escribir **en bytes** (tamaño de float = 4 bytes, RGB => 12 bytes)
  - `data`: puntero al primer elemento de los datos que se van a cargar
  - `uso`: tipo de acceso posterior (previsto) a los datos
    - `GL_STATIC_DRAW`: inicializado una vez, renderizado frecuentemente
    - `GL_DYNAMIC_DRAW`: `se` modifica y renderiza con frecuencia
    - `GL_STREAM_DRAW`: raramente cambiado o renderizado
    - …

# VBO en OpenGL

**Pregunta:** ¿Qué aspecto tienen los datos de los VBO?

- OpenGL es estúpido. Sólo ve un montón de memoria asignada




*VBO con datos de vértices*

- Se necesita una explicación adicional sobre cómo interpretar los datos.
  - se crea un **objeto de atributo de vértice** para cada objeto que se va a renderizar
  - Aquí se almacenan los punteros a los distintos atributos (posición, color, etc.).

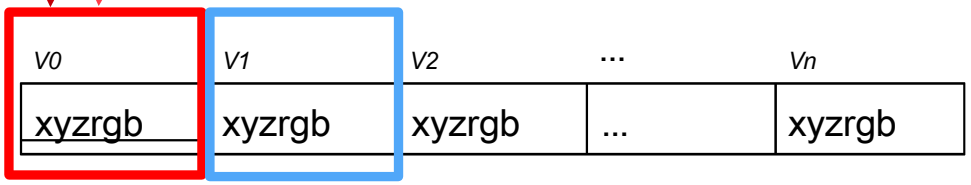
# Objeto matriz de vértices (VAO)

**Cada puntero de atributo describe un Atributo de vértice**

| Índice de atributos | Descripción |
|---|---|
| 0 | "3 coordenadas del tipo GL_FLOAT por vértice". |
| 1 | "3 coordenadas del tipo GL_FLOAT por vértice". |
| ... | ... |

**1. Atributo: Posición del vértice**

**2. Atributo : color rgb**




**Pero cuidado: La interpretación (coordenadas, color, etc.) se determinará más adelante.**

# VAOs en

## Generar ID (análogo a VBO)

- `void glGenVertexArrays(GLsizei n, GLuint * arrays);`
  - crea y devuelve un (o n) array ID, que se almacenan en `buffers`
  - `n`: Especifica el número de objetos de búfer que se generarán, normalmente 1.

## Activar VAO correcto

- `void glBindVertexArray(GLuint vaoID);`
  - `vaoIO`: El ID del VAO generado previamente con glGenVertexArrays.
  - Sólo puede haber un VAO activo a la vez.

## Borrar VAO

- `void glDeleteVertexArrays(GLuint vaoID);`
  - `vaoIO`: El ID del VAO generado previamente con glGenVertexArrays.

# VAOs en OpenGL

**Define los atributos de los vértices** (indica a OpenGL dónde encontrar cada dato).

- `glVertexAttribPointer(GLuint index, GLuint size, GLenum type, GLboolean normalized, GLsizei stride, const void * offset);`
  - `index`: ID definido por el usuario del atributo (necesario más adelante en la GPU)
  - `tamaño`: número de "coordenadas" por vértice, normalmente 3
  - `type`: `Tipo de` datos, por ejemplo GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
  - `normalizado`:
    - falso: Se proporcionan valores en coma flotante, no se requiere normalización
    - true: los valores proporcionados se asignan al rango [0,1] para datos sin signo y [-1,1] para datos con signo.
  - `stride`: Tamaño de un vértice **en bytes**.
  - `offset`: desplazamiento de memoria **en bytes en el** que comienza el atributo de vértice en la matriz activa (0 para el primer atributo)
- glVertexAttribPointer se refiere siempre al **VBO (y VAO) activo en ese momento.**

# VAOs en OpenGL

## Activar y desactivar atributos individuales

- `glEnableVertexAttribArray(índice GLuint);`
- `glDisableVertexAttribArray(índice GLuint);`
  - El índice corresponde al índice autoseleccionado en `glVertexAttribPointer(GLuint index, ...)`
- Por defecto, todos los atributos están desactivados.
- Ejemplo de aplicación:
  - En el mismo VAO pueden combinarse dos modos de renderizado: uno para visualizar la posición (por ejemplo, para la localización de averías) y otro para la visualización en color puro.

# Ejemplo de código

## Crear VBO para posición y color

```
float[] vértices = ... // almacena los datos de tus vértices
// en este ejemplo 3 float para la posición seguidos de 3 float para el
color GLuint vaoID, vboID;

// generar y activar VBO y cargar datos //
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// generar y activar VAO //
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);

// describir VBO en el VAO //
glVertexAttribPointer(0, 3, GL_FLOAT, false, 24, 0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, false, 24, (void*)12);
glEnableVertexAttribArray(1);
```

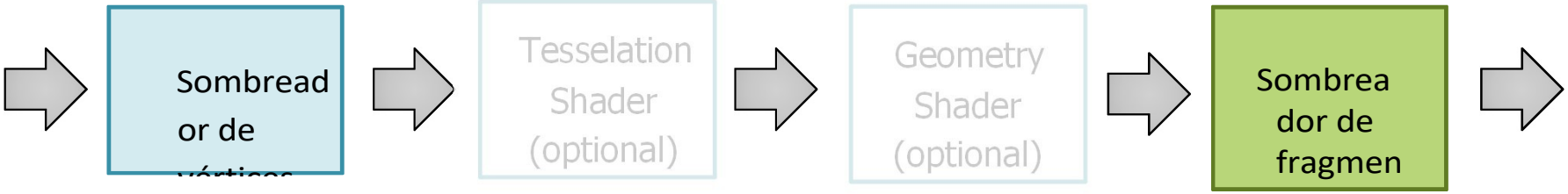
**Se definen dos punteros de atributos, las posiciones de los vértices reciben el ID 0 y los colores el ID 1 en el VAO.**

**Fíjate en el desplazamiento y en el extraño tipo de datos.**

# Tratamiento de datos

- Los datos cargados pueden seguir manipulándose antes de la renderización.
- **Ventaja:** los cálculos se realizan en paralelo por vértice o por píxel.

- Los cálculos deseados se implementan en los llamados **shaders**, pequeños fragmentos de programa que pueden ejecutarse directamente en la GPU.



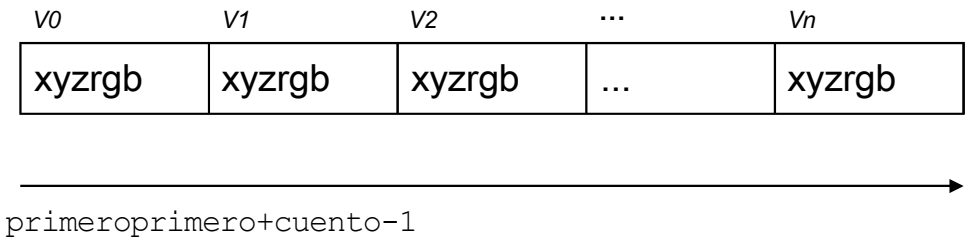

- Cada sombreador tiene una tarea definida con precisión:
  - **Vertex Shader:** Calcula la **posición final de** cada vértice en la imagen de salida.
  - **Fragment Shader:** Calcula el **color de cada píxel** de la imagen de salida.

# Generación de imágenes

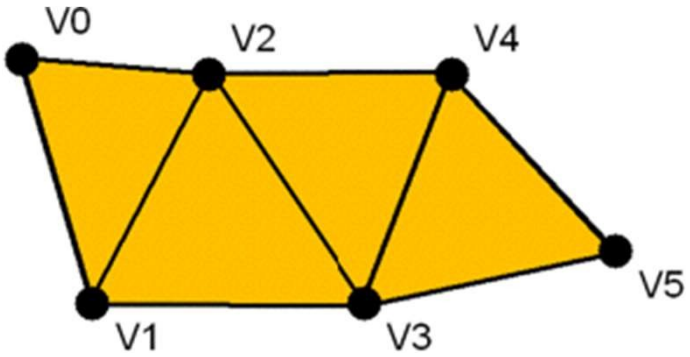
## Renderizado de un objeto de matriz de vértices (y todos los VBO conectados)

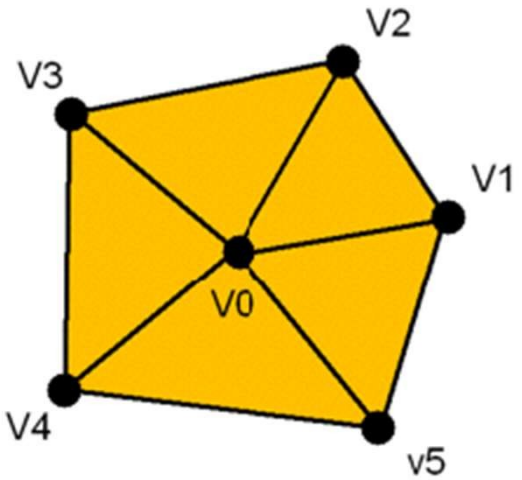
- `glDrawArrays(GLenum mode, GLint first, GLsizei count);`
  - Recorre toda la VAO, empieza `primero por` la primitiva y dibuja la primera `primero + recuento - 1` primitivo
  - `mode`: tipo de primitivas a dibujar, por ejemplo `GL_TRIANGLES`

- Inflexible, pero rápido



# Objetos del búfer de índice



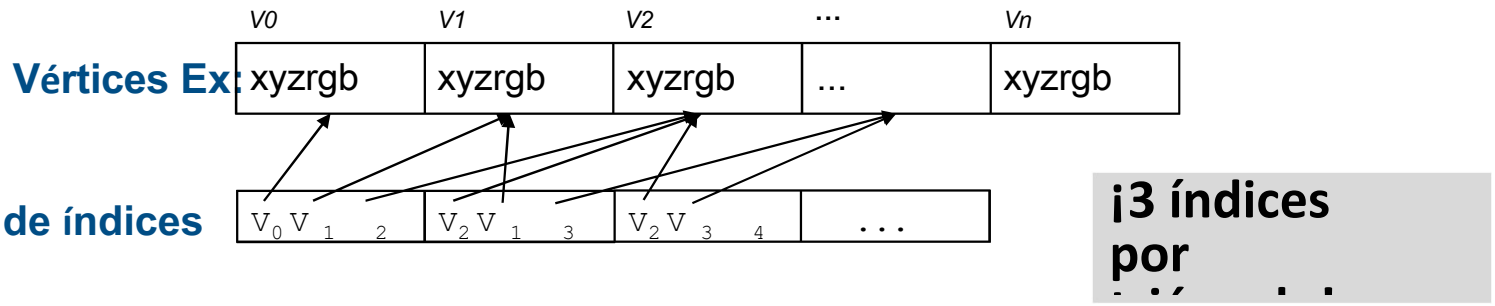




- Los vértices suelen ser compartidos por triángulos (véase la diapositiva *Tiras de triángulos*)
- Reutilización de vértices mediante indexación
- Se necesita un búfer de matriz de elementos adicional
  - `glGenBuffers(1, &iboID);`
  - `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboID);`
  - `glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(data), &data, GL_STATIC_DRAW);`

# Objetos del búfer de

## Desreferenciación mediante índices

- `glDrawElements(GLenum mode, GLsizei count, GLenum type, const void* indices);`
  - `mode:` primitiva de dibujo, por ejemplo `GL_TRIANGLES`
  - `count:` Número de elementos (es decir, índices, ¡no triángulos completos! ☐ Tamaño de la matriz de índices)
  - `type:` Tipo de datos en los índices, `GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT`
  - `índices:` Puntero al principio de la matriz de índices activos, normalmente 0.



# Objetos del búfer de

## Diferencia entre glDrawArrays() y glDrawElements()

- `glDrawArrays()`: Enfoque de fuerza bruta

- `glDrawElements()`: Más flexible (y a veces más rápido)
  - Reutilización de vértices para reducir la transferencia de datos
  - La adaptación del IBO permite renderizar sólo partes de un objeto

# Ejemplo de código

## Crear VBO para posición, con un objeto de memoria intermedia de índice

```
// dado: array de vértices y array de índices //
float[] vértices = ...
int[] índices = ...
GLuint vaoID, vboID;

// configurar VBO //
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// configurar VAO //
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 12, 0);
glEnableVertexAttribArray(0);

// configurar IBO //
GLuint iboID;
glGenBuffers(1, &iboID);    //sólo funciona después de glGenVertexArrays();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

# Ejemplo de código

### Renderizado con IBO

```
void setup(){
        // dado: array de vértices y array de índices
        //
        // configurar VAO //
        // configurar VBO //
        // configurar IBO //
}

void render(){
        // activar VAO //
        glBindVertexArray(vaoID);
        // llamada a render //
        glDrawElements(GL_TRIANGLES, count, GL_UNSIGNED_INT, 0);

        // los buenos programadores deberían
        restablecer // glBindVertexArray(0);
}
```

**Los VBO se activan implícitamente**

**Número de índices, ¡no de primitivas!**

**Especificar primitivas y tipo de datos (del IBO)**