



UNIVERSIDAD
DE GRANADA

Informática Gráfica: Teoría. Tema 1. Introducción.

Carlos Ureña

2022-23

Grado en Informática y Matemáticas

Grado en Informática y Administración y Dirección de Empresas

Dpt. Lenguajes y Sistemas Informáticos

ETSI Informática y de Telecomunicación

Universidad de Granada

Teoría. Tema 1. Introducción.

Índice.

1. Introducción
2. El proceso de visualización
3. La librería OpenGL (y GLFW). Visualización.
4. Programación básica del cauce gráfico
5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Sección 1. Introducción.

- 1.1. Concepto y metodologías
- 1.2. Aplicaciones.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 1. Introducción

Subsección 1.1.

Concepto y metodologías.

El término *Informática Gráfica*

El término **Informática Gráfica** (traducción del término inglés *Computer Graphics*) designa, en un sentido amplio a

El campo de la Informática dedicado al estudio de los algoritmos, técnicas o metodologías destinados a la creación y manipulación computacional de contenido visual digital.

En este curso introductorio nos centraremos esencialmente en:

Técnicas para el diseño e implementación de programas interactivos para visualización 3D y animación de modelos de caras planas y jerárquicos.

Áreas científicas implicadas

La Informática Gráfica puede considerarse un campo multidisciplinar que hace uso de otras disciplinas, quizás las más destacadas sean:

- ▶ Programación orientada a objetos y programación concurrente.
- ▶ Ingeniería del software.
- ▶ Geometría computacional.
- ▶ Hardware (hardware gráfico, dispositivos de interacción).
- ▶ Matemática aplicada (métodos numéricos).
- ▶ Física (óptica, dinámica).
- ▶ Psicología y medicina (percepción visual humana)

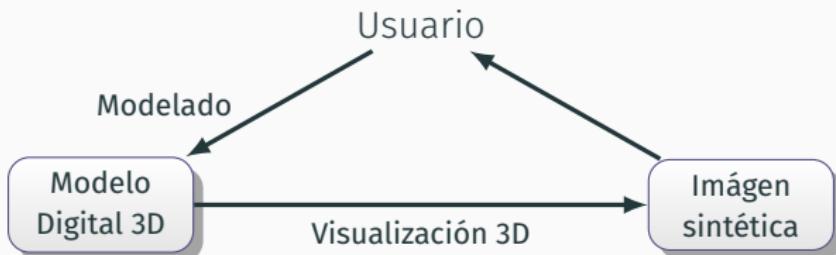
en aplicaciones específicas, se usan otros campos de la Informática en particular o la Ciencia en general (p.ej. para desarrollo de videojuegos se usan también técnicas de Inteligencia Artificial).

Informática Gráfica 3D interactiva

Los elementos esenciales de una aplicación gráfica son:

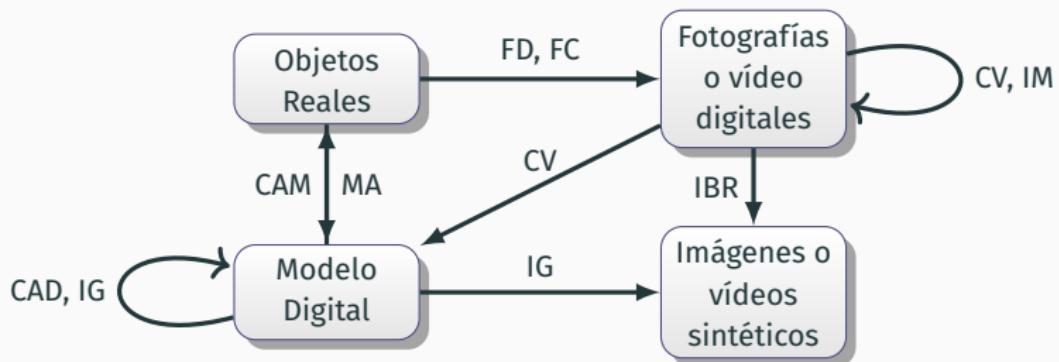
- ▶ **Modelos digitales** de objetos reales, ficticios o de datos
- ▶ **Imágenes o vídeos digitales** que se usan para visualizar dichos objetos.

En las aplicaciones interactivas 3D, los usuarios modifican los modelos 3D y reciben retroalimentación inmediata:



Informática Gráfica y Computación Visual

La Informática Gráfica se enmarca en el área de la **Computación Visual**, que incluye además otras tecnologías:



FD	Fotografía Digital
CV	Visión por Ordenador
CAD	Diseño Asistido por Ord.
MA	Adquisición de Modelos

FC	Fotografía Computacional
IBR	Rendering Basado en Imág.
CAM	Fabric. Asistida por Ord.
IM	Tratamiento de Imágenes

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 1. Introducción

Subsección 1.2.

Aplicaciones..

Aplicaciones

Las aplicaciones son muy numerosas e invaden actualmente muchos aspectos de la interacción y uso de ordenadores. Podríamos destacar algunas (dejando, seguramente, muchas fuera)

- ▶ Videojuegos para ordenadores, consolas y dispositivos móviles.
- ▶ Producción de animaciones, películas y efectos especiales.
- ▶ Diseño en general y diseño industrial.
- ▶ Modelado y visualización en Ingeniería y Arquitectura.
- ▶ Simuladores, juegos serios, entrenamiento y aprendizaje.
- ▶ Visualización de datos.
- ▶ Visualización científica y médica.
- ▶ Arte digital.
- ▶ Patrimonio cultural y arqueología.

Videojuegos



Fotograma del videojuego *Watch Dogs* de Ubisoft.

Vídeo: <http://www.youtube.com/watch?v=kPYgXvgS6Ww>

Realidad Aumentada (AR)



Imagen:

☞ <http://technomarketer.typepad.com/technomarketer/2009/04/firstlook-augmented-reality.html>

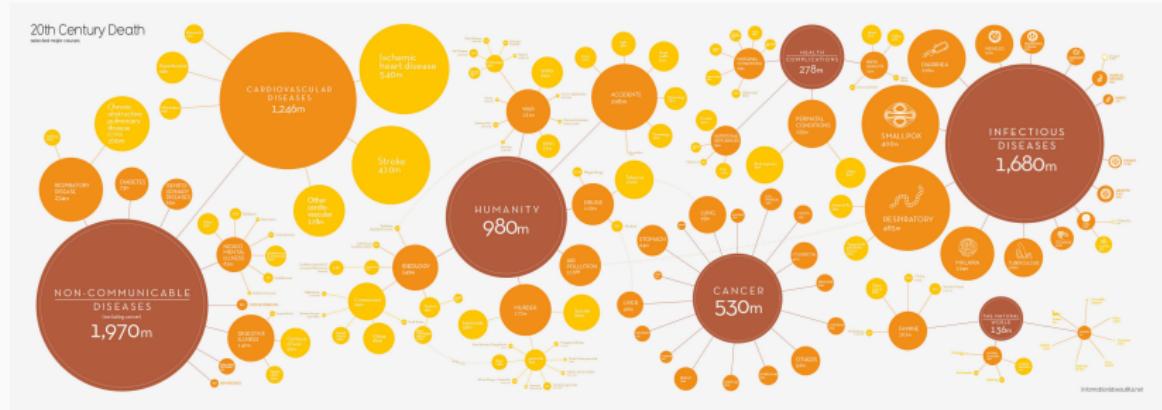
Películas y animaciones generadas por ordenador



Image courtesy of Digic Pictures © 2013 Ubisoft Entertainment. All rights reserved. Watch Dogs and Ubisoft, and the Ubisoft logo are trademarks of Ubisoft Entertainment in the US and/or other countries.

Fotograma del tráiler cinematográfico del videojuego Watch Dogs. Imagen creada por Digic Pictures para Ubisoft, usando Arnold de Solid Angle.
Img: <http://www.fxguide.com/featured/the-state-of-rendering-part-2/#arnold>.
Vídeo: <http://www.youtube.com/watch?v=xLLHYBlyBb8>

Visualización de datos



Frecuencia de causas de muerte en el siglo XX:

☞ <http://www.informationisbeautiful.net/visualizations/20th-century-death/>

Visualización en Medicina



Obtenido del sitio web *MIT Technology Review*

☞ <http://www.technologyreview.com/view/428134/the-future-of-medical-visualisation/>

Cirugía asistida con Realidad Aumentada

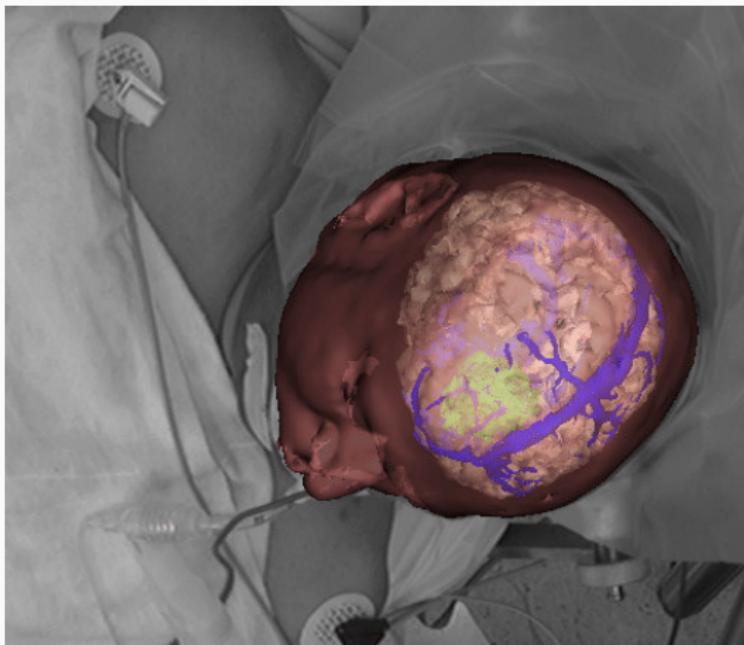
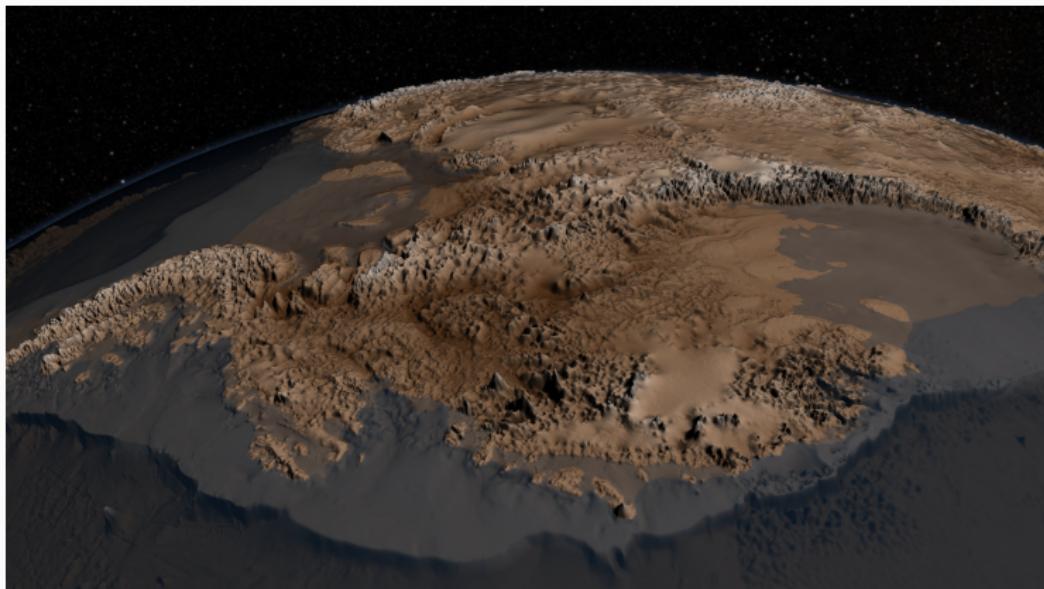


Imagen creada por Christopher Brown, Universidad de Rochester:
 <http://www.cs.rochester.edu/u/brown/projects.html>

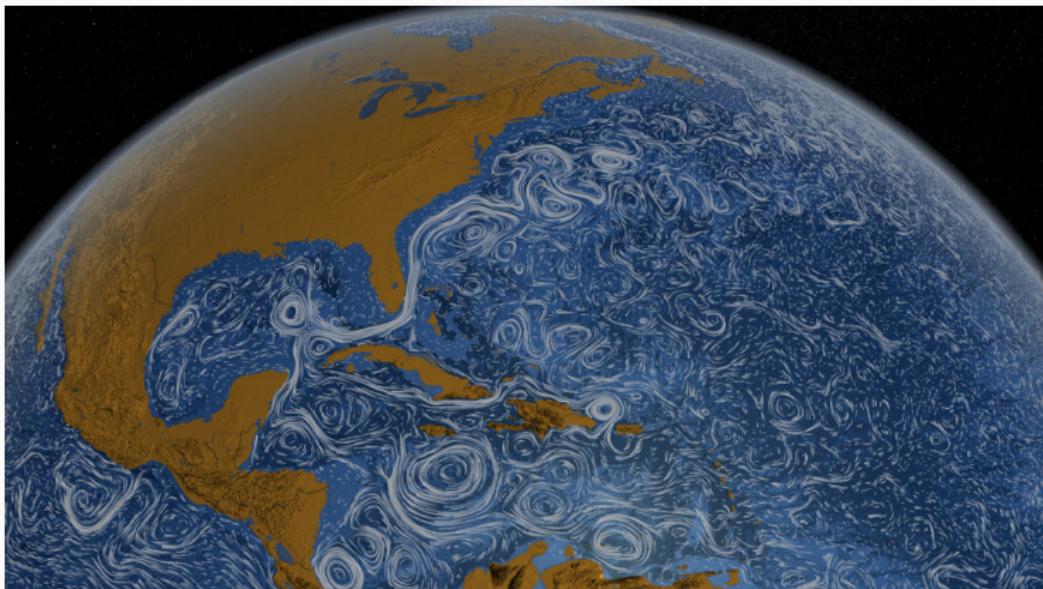
Visualización científica (geología)



Visualización de la topografía del suelo de la Antártica (NASA):

☞ <http://svs.gsfc.nasa.gov/vis/a000000/a004000/a004060/index.html>

Visualización científica (climatología)



Visualización de las corrientes oceánicas (NASA):

☞ <http://www.nasa.gov/topics/earth/features/perpetual-ocean.html>

Simuladores y entrenamiento



Simulador de conducción de Mercedes-Benz:

Imagen:  <http://mercedesbenzblogphotodb.wordpress.com/2010/10/06/>

Patrimonio histórico



Fotografía (izquierda) y visualización 3D de un modelo (derecha).
Proyecto *The Digital Michelangelo*, de la Universidad de Standford.

☞ <http://graphics.stanford.edu/projects/mich/>

Sección 2. El proceso de visualización.

- 2.1. Programas gráficos: interactivos versus off-line
- 2.2. El proceso de visualización
- 2.3. Rasterización y ray-tracing.
- 2.4. El cauce gráfico en rasterización
- 2.5. Las APIs de rasterización
- 2.6. El cauce gráfico en GPUs

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.1.

Programas gráficos: interactivos versus off-line.

Programas gráficos

Un programa gráfico es un programa (o parte de un programa o sistema) que

- ▶ Almacena una estructura de datos que constituye un **modelo** computacional de determinada información.
- ▶ Produce una salida constituida (principalmente) por una o varias imágenes.
- ▶ Las imágenes típicas son **imágenes raster**, constituidas por un array de pixels discretos, cada uno con un color RGB.
- ▶ Existen otros tipos de salidas gráficas, la más frecuentes son las **imágenes vectoriales** (p.ej.: archivos `.svg`).
- ▶ Los programas gráficos pueden ser: **interactivos** o **no interactivos**

Programas gráficos interactivos

Un programa gráfico **interactivo** es un programa que:

- ▶ Visualiza en una ventana gráfica una imagen que constituye una representación visual del modelo.
- ▶ Procesa acciones del usuario (llamadas **eventos**), que se traducen en modificaciones del modelo.
- ▶ Cada vez que el modelo es modificado, se vuelve a visualizar, de forma **interactiva**, lo que significa que desde que el usuario produce el evento hasta que puede observar la imagen actualizada pasan tiempos del orden de decenas de milisegundos como mucho.

Este esquema es el que se usa típicamente en aplicaciones de simuladores, diseño asistido por computador, videojuegos, realidad virtual y realidad aumentada.

Programas gráficos no interactivos

Un programa gráfico **no interactivo** es un programa que:

- ▶ Produce una o varias imágenes (vídeos) a partir del modelo, imágenes que quedan almacenadas en almacenamiento masivo.
- ▶ El proceso de producción de cada imagen tiene una duración que puede ir desde unos segundos hasta varias horas.
- ▶ El usuario solo especifica el modelo y los parámetros de visualización.
- ▶ El usuario no interviene de ninguna forma durante el intervalo de tiempo en el que se producen las imágenes.

Este esquema es el que se usa típicamente en las aplicaciones de síntesis de imágenes para películas y efectos especiales, o en aplicaciones de simulación que requieren tiempos de cálculo altos.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.2.

El proceso de visualización.

El proceso de visualización 3D: entradas (1/2)

El proceso de visualización produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

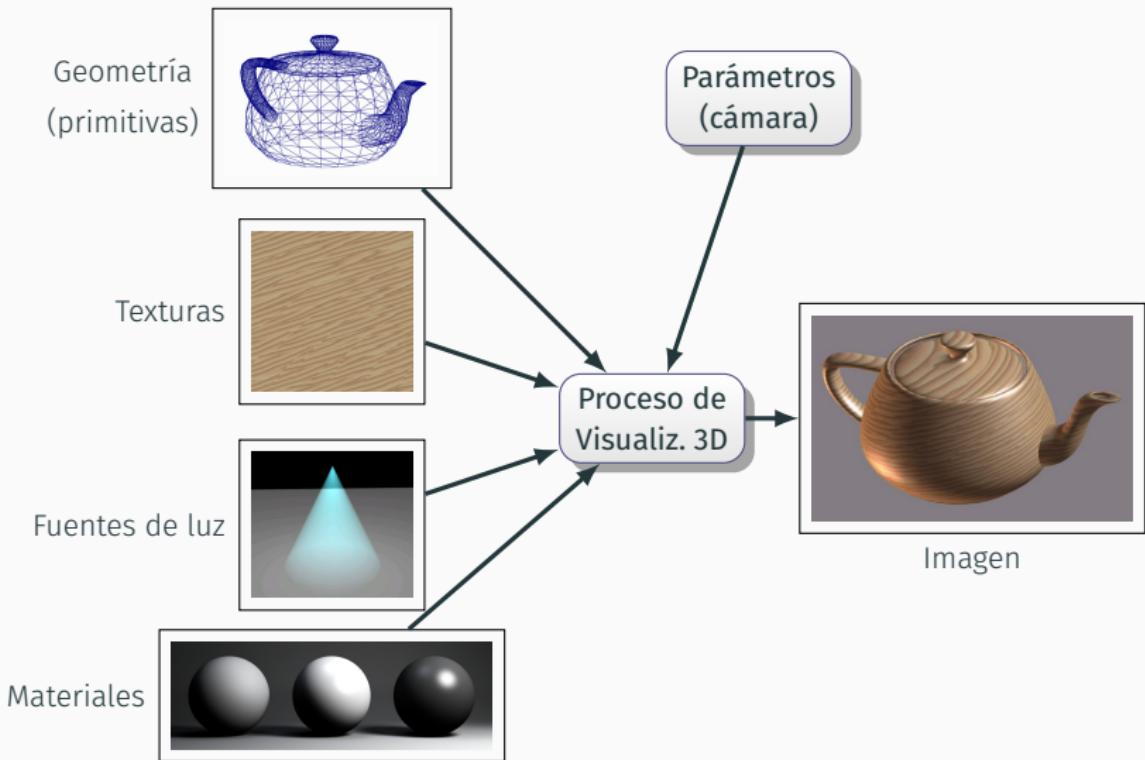
- ▶ **Modelo de escena:** estructura de datos en memoria que representa lo que se quiere ver, esta formado por varias partes:
 - ▶ **Modelo geométrico:** conjunto de **primitivas** (típicamente polígonos planos), que definen la forma de los objetos a visualizar
 - ▶ **Modelo de aspecto:** conjunto de parámetros que definen el aspecto de los objetos: tipo de material, color, texturas, fuentes de luz

El proceso de visualización 3D: entradas (2/2)

El proceso de visualización produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

- ▶ **Parámetros de visualización:** es un conjunto amplio de valores que determinan como se visualiza la escena en la imagen, algunos elementos esenciales son:
 - ▶ **Cámara virtual:** posición, orientación y ángulo de visión del observador ficticio que vería la escena como aparece en la imagen
 - ▶ **Viewport:** Resolución de la imagen, y, si procede, posición de la misma en la ventana.

El proceso de visualización 3D: esquema



Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.3.

Rasterización y ray-tracing..

Visualización basada en *rasterización*

En este curso nos centramos en los algoritmos relacionados con la visualización basada en **rasterización** (*rasterization*).

```
inicializar el color de todos los pixels  
para cada primitiva  $P$  del modelo a visualizar  
    encontrar el conjunto  $S$  de pixels cubiertos por  $P$   
    para cada pixel  $q$  de  $S$ :  
        calcular el color de  $P$  en  $q$   
        actualizar el color de  $q$ 
```

- ▶ Las **primitivas** son los elementos más pequeños que pueden ser visualizados (típicamente triángulos en 3D, aunque también pueden ser otros: polígonos, puntos, segmentos de recta, círculos, etc...)
- ▶ La complejidad en tiempo es claramente del orden del número de primitivas (n) por el número de pixels (p) (tiempo en $O(n \cdot p)$)

Visualización basada en Ray-Tracing

Existen otras posibilidades de esquema para el proceso visualización. En esta otra clase de algoritmos, los dos bucles de antes se intercambian:

```
inicializar el color de todos los pixels  
para cada pixel  $q$  de la imagen a producir  
    calcular  $T$ , el conjunto de primitivas que cubren  $q$   
    para cada primitiva  $P$  del conjunto  $T$   
        calcular el color de  $P$  en  $q$   
        actualizar el color de  $q$ 
```

- ▶ Cuando se trata de visualización 3D, la implementación de esta esquema se conoce como algoritmo de **Ray-tracing**.
- ▶ Se puede optimizar para lograr complejidad en tiempo del orden del número de pixels por el logaritmo del número de primitivas. Esto requiere el uso de **indexación espacial**, para el cálculo de T en cada pixel (tiempo en $O(p \log n)$)

Comparativa: rasterización versus ray-tracing (1/2)

En este curso nos centraremos en la rasterización 3D, y veremos una introducción a ray-tracing en el último tema.

Rasterización

- ▶ Las **unidades de procesamiento gráfico** (GPUs) son un hardware diseñado originalmente para ejecutar la rasterización de forma eficiente en tiempo.
- ▶ El método de rasterización es preferible para **aplicaciones interactivas**, y es el que se usa actualmente para **videojuegos, realidad virtual y simulación**, asistido por GPUs.

Comparativa: rasterización versus ray-tracing (2/2)

En este curso nos centraremos en la rasterización 3D, y veremos una introducción a ray-tracing en el último tema.

Ray-tracing

- ▶ El método de Ray-tracing y sus variantes suele ser más lento, pero consigue resultados más realistas cuando se pretende reproducir ciertos efectos visuales.
- ▶ Las variantes y extensiones de Ray-tracing son preferibles para síntesis de imágenes *off-line* (no interactivas), y es el que se usa actualmente para **producción de animaciones y efectos especiales** en películas o anuncios.
- ▶ En los últimos años (2018 en adelante) han aparecido arquitecturas de GPUs con aceleración por hardware para Ray-Tracing, lo que está llevando a implementar algunos videojuegos usando Ray-Tracing.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.4.

El cauce gráfico en rasterización.

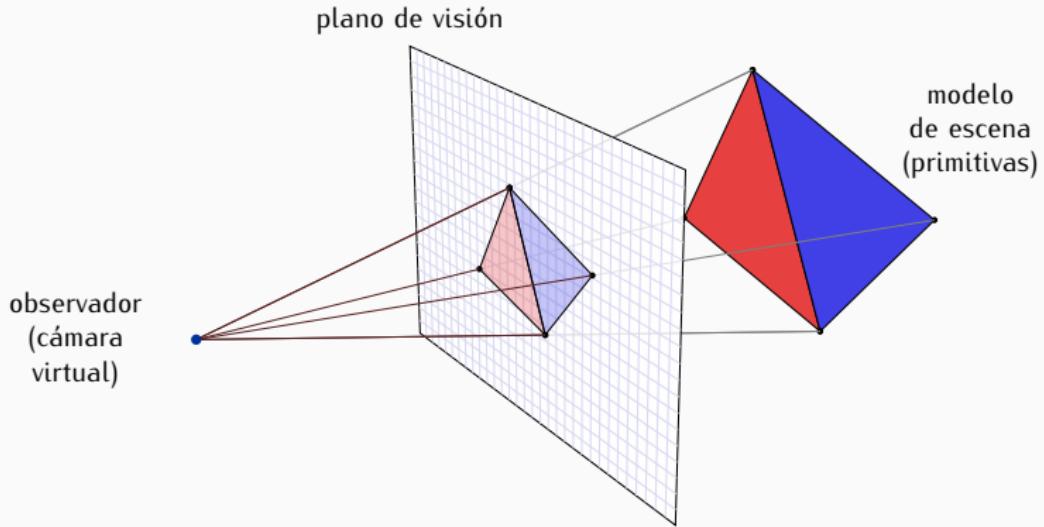
El cauce gráfico: entradas y salidas

El **cauce gráfico** es el conjunto de etapas de cálculo que permiten la síntesis de imágenes por rasterización:

- ▶ Las entradas al cauce gráfico se denominan **primitivas**, una primitiva es una forma visible que no se puede descomponer en otros más simples, en rasterización típicamente las primitivas son: triángulos, segmentos de líneas o puntos (en 2D o en 3D)
- ▶ Un **vértice** es un punto del espacio 2D o 3D, extremo de una arista de un triángulo, o de un segmento de recta, o donde se dibuja un punto. Una o varias primitivas se especifican mediante una **lista de coordenadas de vértices**, más alguna información adicional.
- ▶ El cauce escribe en el **framebuffer**, que es una zona de memoria donde se guardan uno o varios arrays con los colores RGB de los pixels de la imagen (y alguna información adicional). Está conectado al monitor.

Transformación y proyección

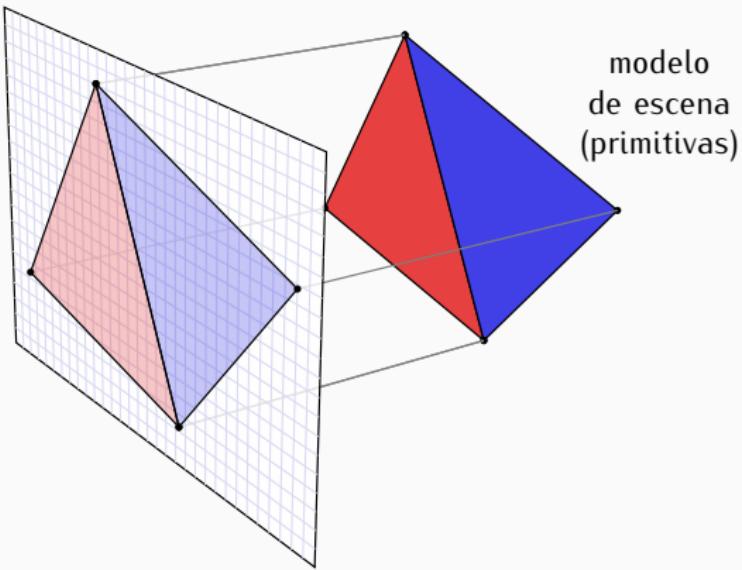
Cada primitiva se sitúan en su lugar en el espacio, y se encuentra su proyección en un plano imaginario (**plano de visión, viewplane**) situado entre el **observador** y la escena (las primitivas):



Proyección paralela

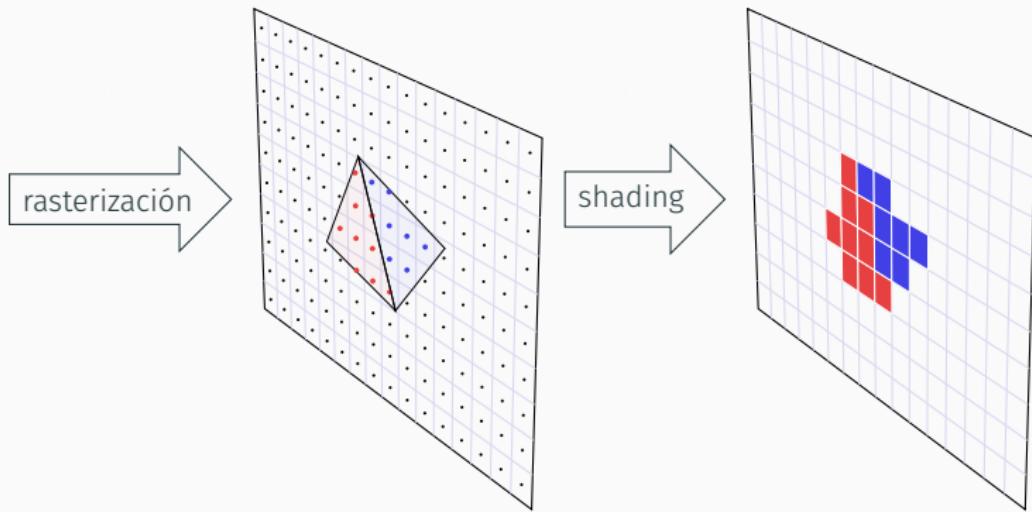
La proyección puede ser **perspectiva** (como en la transparencia anterior), o **paralela**, como aparece aquí:

plano de visión



Rasterización y sombreado

- ▶ **Rasterización:** para cada primitiva, se calcula qué pixels tienen su centro cubierto por ella.
- ▶ **Sombreado:** (*shading*) se usan los atributos de la primitiva para asignar color a cada pixel que cubre.



Sombreado: básico versus avanzado

El proceso de sombreado puede simplemente asignar un color *plano* a cada polígono (izquierda) o bien incluir cálculos avanzados con *iluminación y texturas* (derecha)



Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

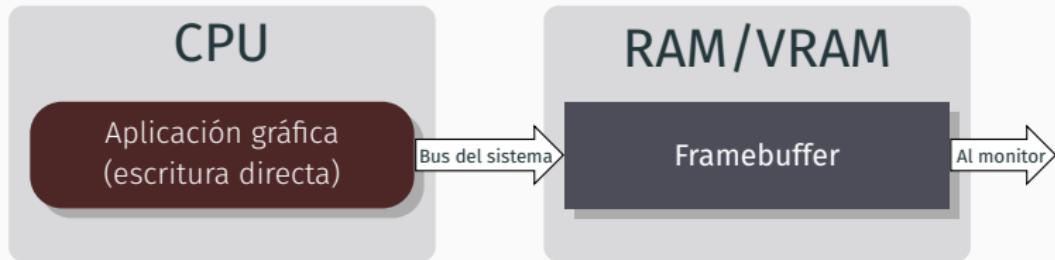
Sección 2. El proceso de visualización

Subsección 2.5.

Las APIs de rasterización.

Aplicaciones de escritura directa

Inicialmente (años 70-80), las aplicaciones gráficas escribían directamente en la memoria de vídeo (VRAM)

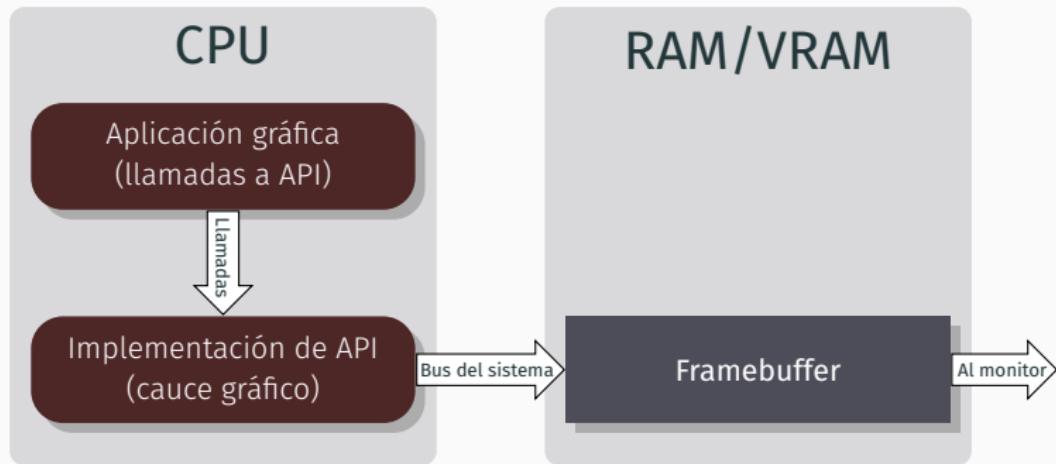


Desventajas

- ▶ La escritura en el framebuffer a través del bus del sistema es lenta, y se realiza pixel a pixel.
- ▶ Solución no portable entre arquitecturas hardware o software.
- ▶ Una aplicación gráfica no puede coexistir con otras

Uso de APIs gráficas

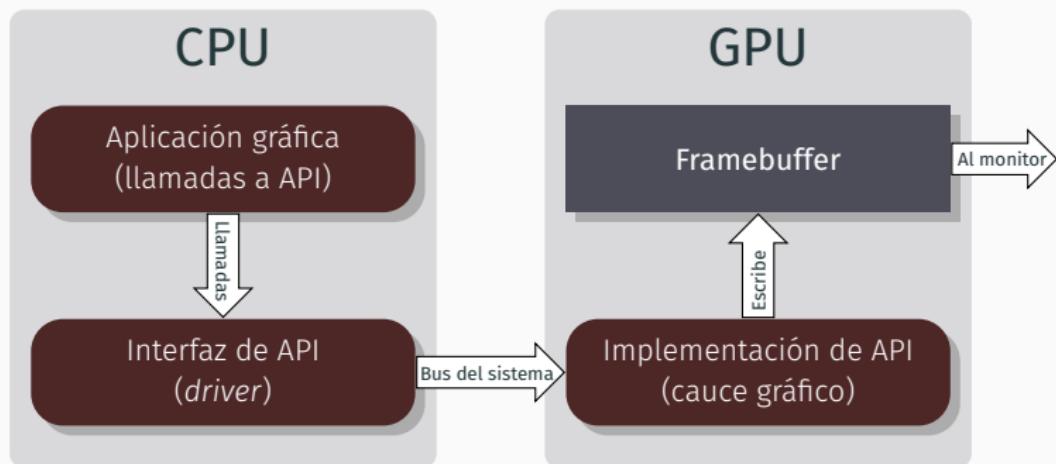
El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad** y **acceso simultáneo**



- ▶ La escritura en el *framebuffer* a través del bus del sistema sigue siendo lenta.

Uso de APIs y hardware gráfico (GPUs)

El uso de **GPUs** (Unidades de Procesamiento Gráfico, *Graphics Processing Units*) **aumenta la eficiencia** ya que ejecutan el cauce y reducen el tráfico a través del bus del sistema (se envía menos información de más alto nivel).



APIs de rasterización en GPUs: las primeras APIs

Las dos primeras APIs existentes son estas:

- ▶ **OpenGL** (1992): diseñada por el consorcio *Khronos group* (formado por múltiples empresas y organismos). Implementada por los principales fabricantes de GPUs, para distintas plataformas hardware/software.
- ▶ **DirectX** (hasta la versión 11, incluida) (1995): diseñada por Microsoft para las plataformas *Windows* y *XBox*, hay implementaciones de los fabricantes de GPUs.

Hay dos APIs adicionales, basadas en OpenGL

- ▶ **OpenGL ES** (2003): subconjunto de OpenGL, orientado a dispositivos móviles. Tiende a converger con OpenGL.
- ▶ **WebGL** (2011): basada en OpenGL ES, diseñada para programas Javascript ejecutándose en navegadores.

APIs de rasterización en GPUs: APIs modernas

En la actualidad se han diseñado varias APIs orientadas a maximizar la eficiencia mediante un uso exhaustivo de las capacidades de paralelismo y concurrencia avanzadas de las GPUs y las CPUs actuales.

- ▶ **Metal** (2014): diseñada e implementada exclusivamente por Apple para macOS, iOS y tvOS.
- ▶ **DirectX 12** (2015): basada en DirectX, pero mucho más eficiente.
- ▶ **Vulkan** (2016): sucesora de OpenGL, inspirada en DirectX 12 y Metal, también diseñado por *Khronos group*.

Estas APIs son de más bajo nivel que las anteriores (los programas son más complejos), pero a cambio se puede aprovecha mejor el hardware.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.6.

El cauce gráfico en GPUs.

Etapas del cauce gráfico (1/2)

El cauce gráfico tiene estas etapas:

1. **Procesado de vértices:** parte de una secuencia de vértices (puntos del espacio) y produce una secuencia de primitivas (puntos, segmentos o triángulos). Tiene estas sub-etapas:
 - 1.1. **Transformación:** los vértices de cada **primitiva** son transformados en diversos pasos hasta encontrar su proyección en el plano de la imagen. Es realizado por un sub-programa llamado **Vertex Shader** (modificable por el programador, o *programable*).
 - 1.2. **Teselación y nivel de detalle:** transformaciones adicionales avanzadas, realizadas por varios programas, entre ellos el **geometry shader** (*programable*). No lo vamos a estudiar.

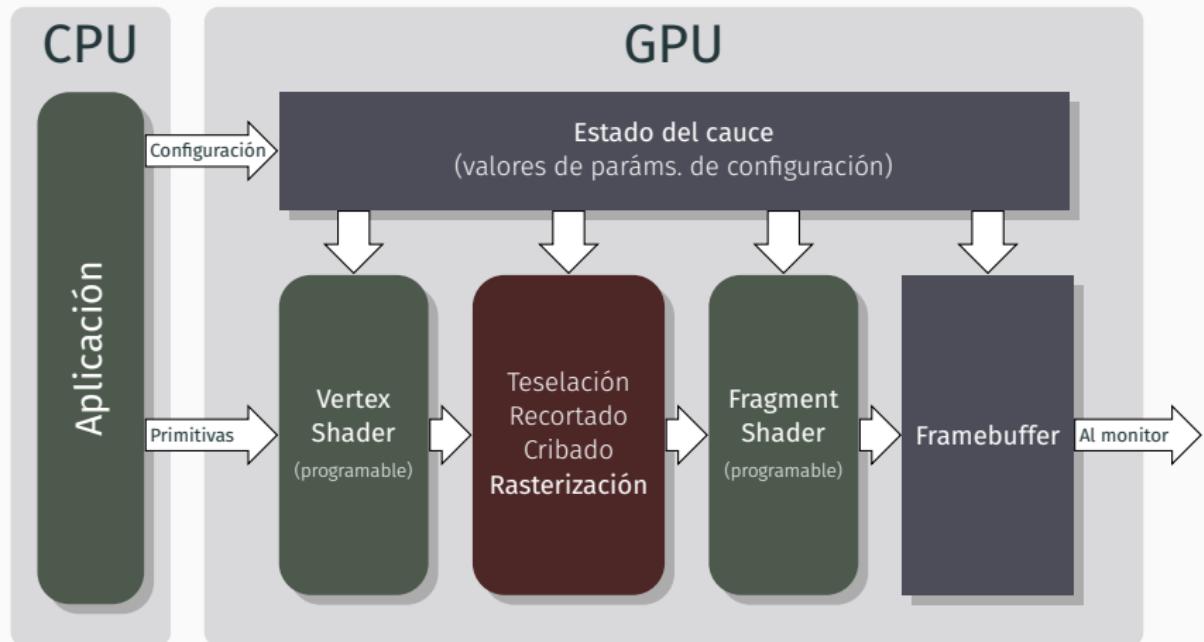
Etapas del cauce gráfico (2/2)

El cauce gráfico tiene estas etapas:

2. **Post-procesado de vértices y montaje de primitivas** incluye varios cálculos como el *recortado (clipping)* y el *cribado de caras (face culling)*, ninguno de ellos programable.
3. **Rasterización (rasterization)** cada primitiva es *rasterizada* (discretizada), y se encuentran los pixels que cubre en la imagen de salida, no es programable.
4. **Sombreado (shading)**: en cada pixel cubierto se calcula el color que se le debe asignar. Se realiza por un programa llamado **fragment shader o pixel shader**, programable.

Esquema simplificado del cauce gráfico en una GPU

DFD simplificado de una aplicación gráfica y el cauce en GPU



Tipos de cauce gráfico: funcionalidad fija o programable

Respecto de la posibilidad de programar partes del cauce:

- ▶ Las primeras APIs no ofrecían la posibilidad de programar el cauce. Se dice que incorporan un **cauce de funcionalidad fija**.
- ▶ Al extenderse el uso de GPUs de complejidad creciente, se da la posibilidad de que los programadores puedan escribir código (con limitaciones) de determinadas partes del cauce.
Inicialmente los *vertex shaders* y los *fragment shaders* o *pixel shaders*).
- ▶ Se dice que se usa un **cauce de funcionalidad programable**, o simplemente **cauce programable**. Esto ocurre a partir del año 2000 aproximadamente.

Evolución del cauce programable

A lo largo de los años y hasta la actualidad, se incrementa la programabilidad del cauce:

- ▶ Se usan lenguajes de alto nivel estandarizados (GLSL, HLSL, Metal Shading Language).
- ▶ Se pueden programar más etapas del cauce (*tesselation shaders, geometry shaders, mesh shaders, etc....*)
- ▶ Se incorporan GPUs programables en toda clase de dispositivos: ordenadores portátiles y dispositivos móviles
- ▶ Se usan las GPUs para cálculo de propósito general (simulación y AI, principalmente)

Sección 3.

La librería OpenGL (y GLFW). Visualización..

- 3.1. La API OpenGL.
- 3.2. Programación y eventos en GLFW
- 3.3. Tipos de primitivas.
- 3.4. Atributos de vértices
- 3.5. Modos de envío.
- 3.6. Almacenamiento de vértices y atributos.
- 3.7. Envío de vértices y atributos.
- 3.8. Estado de OpenGL y visualización de un frame

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.1.

La API OpenGL..

La API OpenGL



- ▶ OpenGL es la **especificación** de un conjunto de funciones útil para visualización 2D/3D basada en rasterización (un documento con: funciones, sus parámetros y comportamiento).
- ▶ Permite la rasterización de primitivas de bajo nivel (polígonos, líneas segmentos), de forma eficiente y portable.
- ▶ **OpenGL ES** (*OpenGL for Embedded Systems*): variante de OpenGL para dispositivos móviles y consolas.
- ▶ **GLSL** (*GL Shading Language*): lenguaje de programación de *shaders* que se usa con OpenGL.

Características de OpenGL

- ▶ Existen implementaciones de la API para las principales plataformas (Windows, MacOS, Linux, Android, iOS,...) y lenguajes de programación (C/C++, Java, Python,...)
- ▶ OpenGL hace que las aplicaciones sean independientes del hardware.
- ▶ Para gestionar ventanas y eventos de entrada se deben usar librerías auxiliares, que pueden o no ser dependientes del entorno hardware/software.
- ▶ Utiliza las capacidades de aceleración de las tarjetas gráficas (GPUs).
- ▶ Hay muchas bibliotecas de más alto nivel sobre OpenGL (p.ej., OSG, *Open Scene Graph*).

Historia de OpenGL.

- ▶ Se origina a finales de los 80 a partir de la librería IRIS GL de *Silicon Graphics* (primera empresa fabricante de hardware gráfico comercial).
- ▶ En 1992 Silicon Graphics crea el *OpenGL Architectural Review Board* (ARB), con otras empresas. Es el comité encargado de diseñar las versiones de OpenGL, la primera versión se define ese año.
- ▶ En 2003 se diseña y publica la primera versión de OpenGL ES.
- ▶ En 2018 se publica la versión 3.2 de OpenGL ES y la versión 4.6 de OpenGL. A partir de aquí no hay nuevas versiones de OpenGL (se trabaja en la librería **Vulkan**, más avanzada).
- ▶ En la actualidad la labor de desarrollo, estándarización y publicación de OpenGL y Vulkan está en manos del consorcio **Khronos Group** (khronos.org).

La librería GLFW

OpenGL no incluye funcionalidad para la gestión de ventanas y eventos de entrada. Se necesitan otras librerías para esta labor. Dos posibilidades son usar la librerías GLUT o GLFW. Usaremos **GLFW** ([glfw.org](https:// glfw.org)):

- ▶ Es una librería *open source*, con *bindings* para C/C++, Go, Java, Python, etc...
- ▶ Es portable (hay implementaciones en Linux, macOS y Windows)
- ▶ Permite creación y cierre de ventanas en las cuales se hace visualización con OpenGL.
- ▶ Permite gestión de eventos de entrada (leer de teclado y ratón).

Los nombres de las funciones de GLFW comienzan con `glfw`.

Repositorio público con ejemplo en OpenGL

Los trozos de código de ejemplo en esta sección están basados en el código fuente C++11 que se encuentra en una carpeta de un repositorio público de github:

☞ github.com/carlos-urena/opengl3-minimo

- ▶ Se puede compilar en Linux, Windows o macOS.
- ▶ Se dan instrucciones y archivos de compilación en Linux y macOS.
- ▶ Incluye un ejemplo mínimo que visualiza dos triángulos en 2D.
- ▶ Es idóneo para escribir y probar el código que forma parte de las respuestas a muchos problemas de la relación de problemas (se recomienda hacer una copia para cada problema).

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.2.

Programación y eventos en GLFW.

Eventos y sus tipos

En las aplicaciones interactivas, un **evento** es la ocurrencia de un suceso relevante para la aplicación, hay varios **tipos de eventos**, entre otros cabe destacar estos:

- ▶ **Teclado:** pulsación o levantado una tecla, de tipo carácter o de otras teclas.
- ▶ **Ratón:** pulsación o levantado de botones del ratón, movimiento del ratón, movimiento de la rueda del ratón para scroll.
- ▶ **Cambio de tamaño:** cambio de tamaño de alguna ventana de la aplicación

Los eventos permiten a la aplicación responder de forma más o menos inmediata a las acciones del usuario, es decir, permiten interactividad.

Funciones gestoras de eventos (*callbacks*)

Las **funciones gestoras de eventos** (FGE) (*event managers*, o *callbacks*), son funciones del programa que se invocan cuando ocurre un evento de un determinado tipo.

- ▶ El programa establece que tipos de eventos se quieren gestionar y que funciones lo harán.
- ▶ Tras invocar a una de estas funciones, se dice que el correspondiente evento ya ha sido **procesado o gestionado**.
- ▶ Para cada tipo de evento, la función que lo gestione debe aceptar unos determinados parámetros. Por ejemplo:
 - ▶ Tecla que ha sido pulsada o levantada
 - ▶ Nueva posición del ratón tras moverse
 - ▶ Botón del ratón que ha sido pulsado o levantado
 - ▶ Nuevo tamaño de la ventana

Estructura de un programa (1/2)

El texto de un programa típico con OpenGL/GLFW tiene varias partes:

- ▶ Variables, estructuras de datos y definiciones globales.
- ▶ Código de las funciones gestoras de eventos.
- ▶ Código de inicialización:
 - ▶ Creación y configuración de la ventana (o ventanas) donde se visualizan las primitivas,
 - ▶ Establecimiento de las funciones del programa que actuarán como gestoras de eventos.
 - ▶ Configuración inicial de OpenGL, si es necesario.
- ▶ Función de visualización de un frame o cuadro.
- ▶ **Bucle principal** (gestiona eventos y visualiza frames)

Estructura del programa. (2/2)

Por todo lo dicho, la estructura o esquema de un programa sencillo sería esta:

```
void VisualizarFrame( ) // visualiza la imagen (un cuadro o frame)
{
    .... }

void FGE_CambioTamano( GLFWwindow* ventana, int nuevoAncho, int nuevoAlto )
{
    .... }

void FGE_PulsarLevantarTecla( GLFWwindow* ventana, int tecla, .... )
{
    .... }

void FGE_PulsarLevantarBotonRaton( GLFWwindow* ventana, int boton, .... )
{
    .... }

void Inicializa(GLFW( int argc, char * argv[] )
{
    .... }

void Inicializa_OpenGL( )
{
    .... }

void BucleEventos(GLFW()
{
    .... }

int main( int argc, char *argv[] )
{
    Inicializa(GLFW(argc,argv) ; // crea una ventana
    Inicializa_OpenGL() ;           // inicializa estado del cauce
    BucleEventos(GLFW() ;           // ejecuta el bucle (ver más abajo)
    glfwTerminate();                // cerrar la ventana
}
```

Bucle principal o de gestión de eventos

Una aplicación OpenGL/GLFW ejecuta un **bucle principal** o **bucle de gestión de eventos** (en GLFW, el programador debe implementarlo explicitamente):

- ▶ GLFW mantiene una **cola de eventos**: es una lista (FIFO) con información de cada evento que ya ha ocurrido pero que no ha sido gestionado aún por la aplicación.
- ▶ En cada iteración se espera hasta que hay al menos un evento en la cola, entonces:
 1. Se extrae el siguiente evento de la cola: si hay designada una función gestora para ese tipo de evento, se ejecuta dicha función.
 2. Si la ejecución de la función ha cambiado el modelo de escena o algún parámetro, se visualiza un cuadro nuevo.
- ▶ El bucle termina típicamente cuando en alguna función gestora se ordena cerrarla (p.ej.: al pulsar la tecla ESC)

Código de inicialización de GLFW (1/2)

Se ejecuta una vez al inicio de la aplicación, **antes** de cualquier orden OpenGL. Usa **ventana_tam_x** y **ventana_tam_y** (tamaño de ventana)

```
void Inicializa(GLFW( int argc, char * argv[] )  
{  
    // intentar inicializar, terminar si no se puede  
    if ( ! glfwInit() )  
    { cout << "Imposible inicializar GLFW. Termino." << endl ;  
        exit(1) ;  
    }  
  
    // especificar que función se llamará ante un error de GLFW  
    glfwSetErrorCallback( ErrorGLFW );  
  
    // crear la ventana (var. global ventana_glfw), activar el rendering context  
    ventana_glfw = glfwCreateWindow( ventana_tam_x, ventana_tam_y,  
                                    "Practicas IG (19-20)", nullptr, nullptr );  
    glfwMakeContextCurrent( ventana_glfw ); // necesario para OpenGL  
  
    ....  
}
```

Código de inicialización de GLFW (2/2)

Una vez creada la ventana, se deben especificar los nombres de las funciones de nuestro programa que deben ser llamadas cuando ocurre un evento (funciones FGE)

```
void Inicializa(GLFW( int argc, char * argv[] )  
{  
    ....  
  
    // definir cuales son las funciones gestoras de eventos...  
    glfwSetWindowSizeCallback ( ventana_glfw, FGE_CambioTamano );  
    glfwSetKeyCallback      ( ventana_glfw, FGE_PulsarLevantarTecla );  
    glfwSetMouseButtonCallback( ventana_glfw, FGE_PulsarLevantarBotonRaton );  
    glfwSetCursorPosCallback ( ventana_glfw, FGE_MovimientoRaton );  
    glfwSetScrollCallback    ( ventana_glfw, FGE_Scroll );  
}
```

Bucle principal en GLFW (sin animaciones)

```
void BucleEventos(GLFW*)
{
    redibujar_ventana = true ; // dibujar la ventana la primera vez
    terminar_programa = false ; // activar para terminar (p.ej. con tecla ESC)
    while ( ! terminar_programa )
    {
        if ( redibujar_ventana ) // si ha cambiado algo y es necesario redibujar
        { VisualizarFrame(); // dibujar la escena
            redibujar_ventana = false; // evitar que se redibuje continuamente
        }
        glfwWaitEvents(); // esperar evento y llamar FGE (si hay alguna)
        terminar_programa = terminar_programa || glfwWindowShouldClose( glfw_window ) ;
    }
}
```

- ▶ **redibujar_ventana** y **terminar_programa** son variables lógicas globales.
- ▶ Las F.G.E. las ponen a **true** cuando se quiera refrescar (redibujar) la ventana o acabar la aplicación, respectivamente.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.3.

Tipos de primitivas..

Especificación de primitivas

En OpenGL (y en todas las librerías con el mismo propósito), cada primitiva o conjunto de primitivas se especifica mediante una secuencia ordenada de coordenadas de **vértices**:

- ▶ Un vértice es un punto de un espacio afín 3D.
- ▶ Se representa en memoria mediante una tupla de coordenadas en algún marco de coordenadas de dicho espacio afín.
- ▶ Puede tener asociados otros valores, llamados **atributos** (p.ej. un color).

Existen distintos tres tipos de primitivas: **puntos**, **segmentos** y **triángulos**:

- ▶ Por tanto, además de la secuencia de vértices, es necesario tener información acerca de que tipo de primitiva representa dicha secuencia.

Tipos de primitivas: puntos y segmentos

Una lista de n coordenadas de vértices (con $n \geq 1$) puede usarse para codificar puntos o segmentos. Más en concreto, puede codificar:

- ▶ n puntos aislados (n arbitrario) (constante OpenGL: **GL_POINTS**)
- ▶ uno o varios segmentos de recta, en concreto:
 - ▶ $n/2$ segmentos independientes (n par) (**GL_LINES**)
 - ▶ $n - 1$ segmentos formando una polilínea abierta ($n \geq 2$) (**GL_LINE_STRIP**).
 - ▶ n segmentos formando una polilínea cerrada ($n \geq 3$) (**GL_LINE_LOOP**).

Las constantes indicadas están definidas en OpenGL y se traducen en enteros que identifican cada tipo de primitiva en los programas.

Tipos de primitivas: triángulos

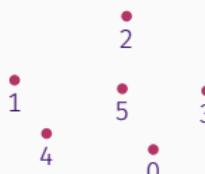
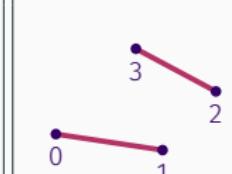
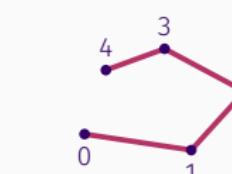
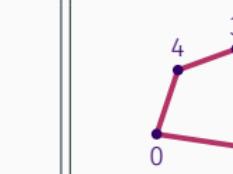
Una lista de n coordenadas de vértices también puede codificar uno o varios triángulos, en concreto, puede codificar:

- ▶ $n/3$ triángulos (n múltiplo de 3) (**GL_TRIANGLES**)
- ▶ $n - 2$ triángulos compartiendo aristas (**tira de triángulos**), cada triángulo comparte dos vértices con el anterior ($n \geq 3$) (**GL_TRIANGLE_STRIP**).
- ▶ $n - 1$ triángulos compartiendo un vértice (**abanico de triángulos**) todos los triángulos comparten el primer vértice, y cada triángulo comparte dos vértices con el anterior ($n \geq 3$) (**GL_TRIANGLE_FAN**).

En las primeras versiones de OpenGL se contemplaban primitivas de tipo cuadrilátero y polígono, pero en las versiones actuales estas primitivas no se contemplan.

Primitivas de tipo puntos y segmentos.

Las coordenadas $(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1})$ forman puntos, segmentos y polilíneas (abiertas o cerradas). Aquí se ilustran varios ejemplos:

Puntos GL_POINTS	Segmentos GL_LINES	Polilínea abierta GL_LINE_STRIP	Polilínea cerrada GL_LINE_LOOP
 A set of six red dots representing points. They are labeled with integers: 1, 2, 3, 4, 5, and 0. Point 1 is at the bottom left, point 2 is above it, point 3 is to the right of 2, point 4 is below 1, point 5 is between 1 and 4, and point 0 is below 5.	 Two red line segments. The first segment connects point 0 at the bottom left to point 1 at the bottom right. The second segment connects point 1 to point 2 at the bottom right.	 An open polygonal chain consisting of five red line segments. The vertices are labeled 0, 1, 2, 3, and 4. The segments connect vertex 0 to 1, 1 to 2, 2 to 3, 3 to 4, and 4 back to 0.	 A closed polygonal loop consisting of five red line segments. The vertices are labeled 0, 1, 2, 3, and 4. The segments connect vertex 0 to 1, 1 to 2, 2 to 3, 3 to 4, and 4 back to 0.

Triángulos delanteros y traseros. Cribado.

Cada primitiva de tipo triángulo (también llamada **cara**, *face*) es clasificada por OpenGL como **delantera** o **trasera**:

- ▶ Será **delantera** si sus vértices se visualizan en pantalla en el sentido contrario de las agujas del reloj.
- ▶ Será **trasera** si sus vértices se visualizan en pantalla en el sentido de las agujas del reloj

Este es el comportamiento por defecto (se puede cambiar).

- ▶ OpenGL puede ser configurado para no visualizar las caras traseras o no visualizar las delanteras (se llama hacer **cribado de caras**, *face culling*).
- ▶ Por defecto, el cribado está deshabilitado (todas se ven)

Esta clasificación tiene utilidad especialmente en visualización 3D.

Modo de visualización de triángulos.

En el caso de las primitivas de tipo triángulos, OpenGL puede visualizarlos de varias formas, según el valor de un parámetro de configuración en el estado de OpenGL, que se llama el **modo de visualización de polígonos**, y que permite seleccionar una de estas opciones:

- ▶ **modo puntos**: cada triángulo se visualiza como un punto en cada vértice (constante OpenGL **GL_POINT**).
- ▶ **modo líneas**: cada triángulo se visualiza como una polilínea cerrada (un segmento por cada arista) (**GL_LINE**)
- ▶ **modo relleno**: cada triángulo se visualiza relleno de color (plano, degradado, textura, etc...) (**GL_FILL**)

El modo de visualización de polígonos se puede cambiar en cualquier momento.

Primitivas tipo triángulos (no adyacentes)

La visualización de una secuencia de n vértices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_{n-1}$ (que codifica $n/3$ triángulos) depende del modo de visualización de polígonos.



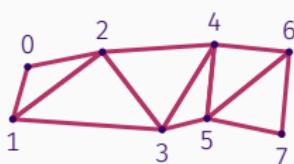
Primitivas tipo triángulos (adyacentes)

Los triángulos comparten algunos vértices
(lo vemos con el *modo de polígonos* fijado a líneas):

Tira de triángulos **GL_TRIANGLE_STRIP**

Polígonos:

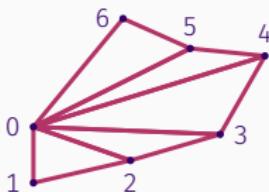
$(0, 1, 2), (2, 1, 3), (2, 3, 4),$
 $(4, 3, 5), (4, 5, 6), (6, 5, 7), \dots$



Abanico de triángulos **GL_TRIANGLE_FAN**

Polígonos:

$(0, 1, 2), (0, 2, 3), (0, 3, 4),$
 $(0, 4, 5), (0, 5, 6), \dots$



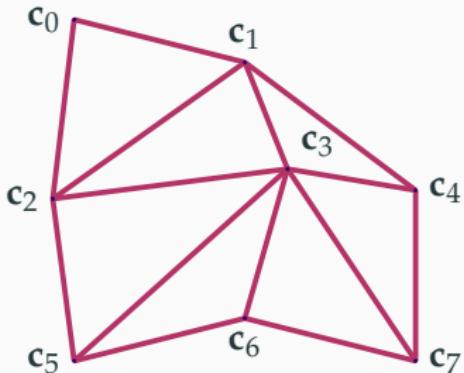
Polígonos de más de tres vértices

Respecto a la posibilidad de visualizar primitivas de más de tres vértices:

- ▶ En versiones de OpenGL anteriores a la 3.0 existían las primitivas tipo cuadrilátero y polígono
- ▶ Las constantes eran: **GL_POLYGON**, **GL_QUADS** y **GL_QUAD_STRIP**.
- ▶ En la versión 3.0 de OpenGL (2008), se declararon *obsoletas* este tipo de primitivas, y en posteriores versiones se eliminaron
- ▶ En OpenGL moderno, para visualizar estas primitivas en modo relleno hay que descomponerlas en triángulos.
- ▶ En cualquier caso, en versiones antiguas de OpenGL lo que se hacía internamente es descomponerlas en triángulos.

Problema de vértices replicados

Muchas veces necesitamos usar unas mismas coordenadas para varios vértices, p.ej. si queremos visualizar estos 7 triángulos:



Si usamos **GL_TRIANGLES**, la secuencia de coords. de vértices es esta:

$$\{ \quad \mathbf{c_0, c_2, c_1, c_1, c_2, c_3,} \\ \mathbf{c_1, c_3, c_4, c_2, c_5, c_3,} \\ \mathbf{c_3, c_5, c_6, c_3, c_6, c_7,} \\ \mathbf{c_3, c_7, c_4} \}$$

Supone emplear más memoria y/o tiempo para visualizar del necesario. En este ejemplo necesitamos una secuencia de 21 coordenadas de vértices, de las cuales solo hay 8 distintas (p.ej., las coordenadas $\mathbf{c_3}$ aparecen repetidas 6 veces)

Secuencias indexadas

Las APIs de rasterización permiten especificar una secuencia de vértices (con repeticiones) a partir de una secuencia de vértices únicos:

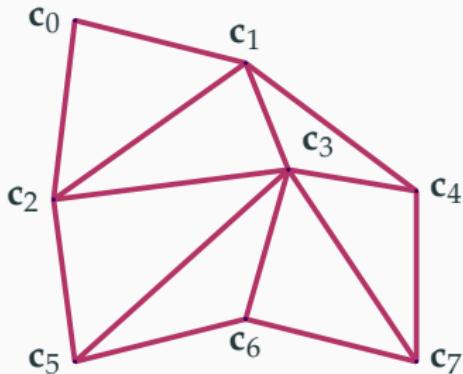
- ▶ Se parte de una secuencia V_n de n coordenadas arbitrarias de vértices $V_n = \{\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}\}$.
- ▶ Se usa una secuencia I_m de m **índices** $I_m = \{i_0, i_1, \dots, i_{m-1}\}$ donde cada valor i_j es un entero entre 0 y $n - 1$ (ambos incluidos). Puede tener índices repetidos.
- ▶ La secuencia de vértices V_n y la de índices determinan otra secuencia S_m de m vértices:

$$S_m = \{ \mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_{m-1}} \}$$

que tiene las mismas coordenadas de vértices de V_n pero en el orden especificado por los índices en I_m .

Ejemplo de secuencia indexada

En este ejemplo que hemos visto antes



Haríamos:

$$V_8 = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$$

$$I_{21} = \{0, 2, 1, 1, 2, 3, 1, 3, 4, 2, 5, 3, 3, 5, 6, 3, 6, 7, 3, 7, 4\}$$

En este ejemplo, cada tres índices consecutivos forman un triángulo.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.4.

Atributos de vértices.

Atributos de vértices

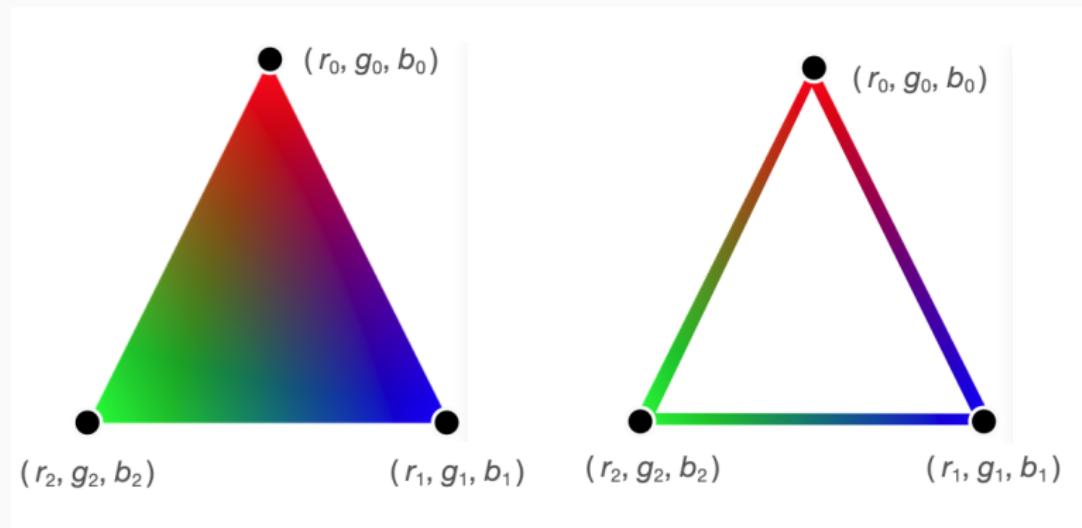
Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se pueden opcionalmente usar otros atributos, por ejemplo:

- ▶ El **color** del vértice (una terna RGB con valores entre 0 y 1).
- ▶ La **normal**: una vector unitario con tres coordenadas reales, determina la orientación de la superficie de un objeto en el punto donde está el vértice. Se usa para iluminación.
- ▶ Las **coordenadas de textura**: típicamente un par de valores reales, que se usan para determinar que punto de la textura se fija al vértice (lo veremos)

En el cauce programable moderno de OpenGL se pueden definir tantos atributos como queramos. Nosotros veremos como usar estos tres junto con la posición, usando llamadas de OpenGL 3.3

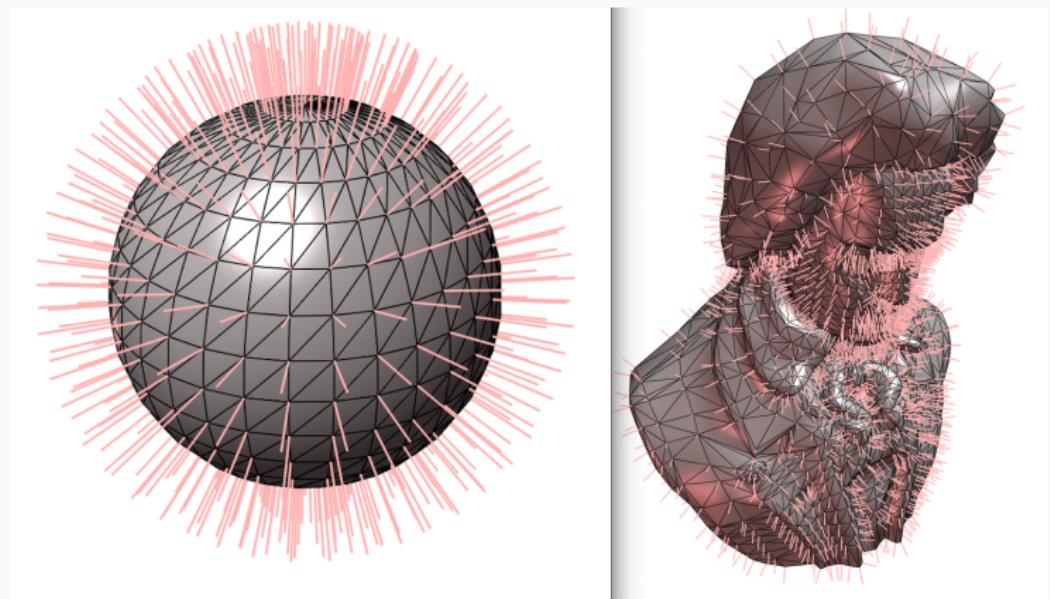
Atributos: colores de vértices

Es posible asignar un color a cada vértice, es una terna RGB con tres reales (r, g, b) , (con valores entre 0 y 1) o bien una cuádrupla RGBA (RGB+transparencia). En el interior (o en las aristas) del polígono se usa interpolación para calcular el color de cada pixel.



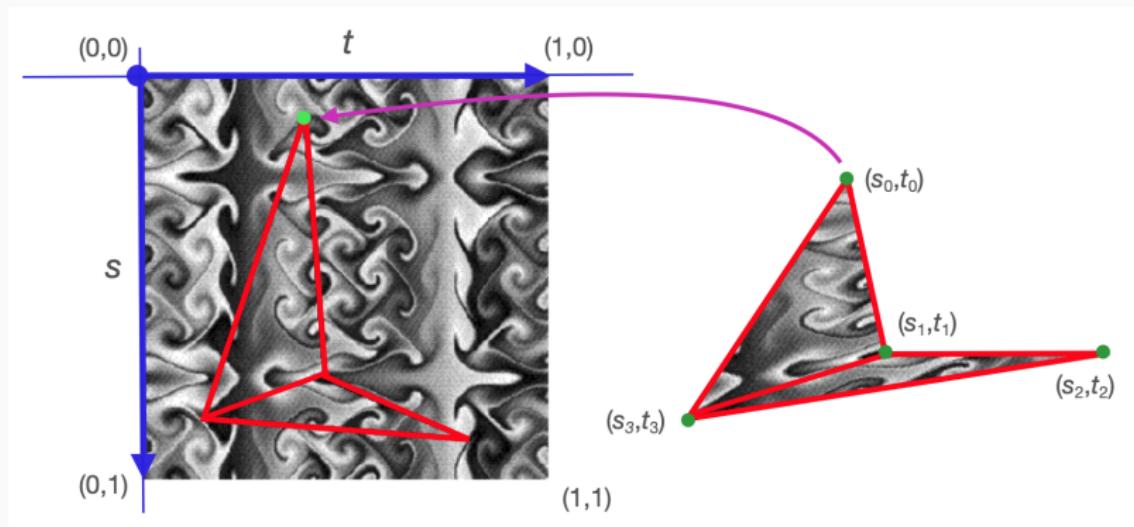
Atributos: normales

En visualización 3D, a cada vértice se le puede asociar un vector de 3 componentes (x, y, z) (su vector **normal**) que determina la orientación de la superficie en ese vértice y sirve para hacer el sombreado y la iluminación:



Atributos: coordenadas de textura

Para usar imágenes (texturas) en lugar de colores , podemos asociar a cada vértice un par de reales (s, t) (sus **coordenadas de textura**), típicamente en $[0, 1]^2$. Esto determina como se aplica la imagen (a la izquierda) a las primitivas (a la derecha):



Definición de valores de atributos

En OpenGL a cada vértice **siempre** se le asocia una tupla por cada atributo.

- ▶ Es decir, todo vértice tiene siempre asociado una posición, un color, una normal y unas coordenadas de textura (u otros atributos definidos por la aplicación).
- ▶ Según la configuración del cauce, algunos atributos serán usados o no. P.ej., si un objeto no tiene textura, no se usarán sus coordenadas de textura, o si no está activada la iluminación, no se usará la normal.
- ▶ Podemos definir único valor de un atributo para todos los vértices de una primitiva, o bien especificar un valor para cada vértice.

El valor de cada atributo está definido en cada pixel donde se proyecta la primitiva. Estos valores se calculan durante la rasterización usando **interpolación**.

Índices de atributos

A cada atributo que queramos usar hay que asignarle un valor entero (≥ 0) para identificarlo:

- ▶ El índice 0 debe usarse siempre para las coordenadas de vértice (es el único atributo requerido para visualizar cualquier cosa, el resto son opcionales).
- ▶ En cada aplicación hay que establecer un convenio claro de asignación de índices a atributos. En adelante (en las prácticas y ejemplos) usaremos este convenio:
 - ▶ Índice 1: colores.
 - ▶ Índice 2: normales.
 - ▶ Índice 3: coordenadas de textura.
- ▶ Es conveniente, por claridad, definir constantes con nombres descriptivos para los índices de atributos, y para el total de atributos que se van a usar.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.5.

Modos de envío..

Modos de envío

Hay varios formas de visualizar secuencias de vértices y sus atributos:

- ▶ Envío en **modo inmediato**: cada vez que queremos visualizar, se envían los atributos e índices a la GPU por el bus del sistema.
De dos formas:
 - ▶ Usando una llamada a una función por cada vértice o atributo.
 - ▶ Usando una única llamada para enviar tablas completasEste modo de visualización es muy ineficiente en tiempo (requiere transferir muchos datos por cada cuadro o frame), y no se usa en OpenGL moderno.
- ▶ Envío en **modo diferido**: los datos de la secuencia de vértices se envían a la GPU una sola vez. Es el modo que usaremos.

Envío en Modo Diferido

Por eso actualmente se usa el **modo diferido**:

- ▶ La información sobre primitivas (la secuencia de vértices) se envía una única vez a la GPU. Requiere reservar memoria en la GPU y transferir los datos.
- ▶ Cada vez que se visualizan las primitivas, se hace con una única llamada, OpenGL lee las tablas de la memoria de la GPU, en lugar de la memoria de la aplicación.
- ▶ Los accesos a memoria en la GPU son mucho más rápidos que las transferencias por el bus del sistema.

Las zonas de memoria en GPU con información de las primitivas se llaman *Vertex Buffer Objects* (VBOs)

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.6.

Almacenamiento de vértices y atributos..

Almacenamiento de vértices y atributos: AOS y SOA (1/2)

Cuando usamos arrays o tablas de coordenadas y atributos en memoria (ya sea memoria principal o la memoria de la GPU), tenemos dos opciones:

- ▶ **Array de estructuras (*Array Of Structures*, AOS):** se usa un array o vector, donde cada entrada contiene las coordenadas de un vértice y todos sus atributos.
- ▶ **Estructura de arrays (*Structure Of Arrays*, SOA):** se usa una estructura con varios (punteros a) arrays de número de elementos. Uno de ellos contiene las coordenadas y los otros contienen cada uno una tabla de atributos (colores, normales, coordenadas de textura).

Nosotros usaremos la opción SOA (estructura de arrays), ya que permite almacenar únicamente las tablas de atributos necesarias en cada caso. **Los índices siempre están contiguos** en su propia tabla.

Almacenamiento de vértices y atributos: AOS y SOA (2/2)

En la opción AOS, hay un array de estructuras (una por vértice)

verts. = { $\underbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{n_{x0}, n_{y0}, n_{z0}}_{\text{normal 0}}, \underbrace{s_0, t_0}_{\text{cc.t. 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{r_1, g_1, b_1}_{\text{color 1}}, \dots, \underbrace{s_{n-1}, t_{n-1}}_{\text{cc.t. n-1}}$ } vértice 0

En la opción SOA, hay una estructura con (punteros a) varios arrays:

posiciones $\rightarrow \{ \underbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \dots, \underbrace{x_{n-1}, y_{n-1}, z_{n-1}}_{\text{posic. n - 1}} \}$

colores $\rightarrow \{ \underbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{r_1, g_1, b_1}_{\text{color 1}}, \dots, \underbrace{r_{n-1}, g_{n-1}, b_{n-1}}_{\text{color n - 1}} \}$

normales $\rightarrow \{ \underbrace{n_{x0}, n_{y0}, n_{z0}}_{\text{normal 0}}, \underbrace{n_{x1}, n_{y1}, n_{z1}}_{\text{normal 1}}, \dots, \underbrace{n_{x,n-1}, n_{y,n-1}, n_{z,n-1}}_{\text{normal n - 1}} \}$

cc.textura $\rightarrow \{ \underbrace{u_0, v_0}_{\text{cc.t. 0}}, \underbrace{u_1, v_1}_{\text{cc.t. 1}}, \dots, \underbrace{u_{n-1}, v_{n-1}}_{\text{cc.t. n - 1}} \}$

(algunos de los punteros pueden ser nulos, excepto las posiciones)

Almacenamiento de vértices en la aplicación

En este ejemplo la secuencia de vértice tiene tablas de colores, normales, y coordenadas de textura (en general, alguna podría no estar).

AOS

```
struct
{  float posicion  [3],
    color        [3],
    normal       [3],
    coord_text[2];
}
secuenciaAOS[ num_vertices ];
```

SOA

```
struct
{  float posiciones[ num_vertices*3 ],
    colores      [ num_vertices*3 ],
    normales     [ num_vertices*3 ],
    coord_text[ num_vertices*2 ];
}
secuenciaSOA ;
```

- ▶ Los índices (si hay) están en una tabla independiente (`unsigned indices[num_indices]`, por ejemplo).
- ▶ En este ejemplo, `num_vertices` (y `num_indices`) deben ser constantes conocida en tiempo de compilación.

Almacenamiento de secuencias de vértices en la aplicación.

Usaremos C++11 (y una biblioteca para tuplas) de datos para gestionar el almacenamiento de las tablas de atributos e índices de una secuencia de vértices en la aplicación:

- ▶ Cada tupla de coordenadas, u otros atributos (color, normal, etc...) de un vértice se representa usando tipos de datos para tuplas o pequeños vectores.
- ▶ Cada tupla contiene 2,3 o 4 valores reales (pueden ser **float** o **double**).
- ▶ Usamos una librería de tuplas que tiene los tipos de datos con nombres **Tuplant**, donde n es la longitud de la tupla (2,3 o 4) y t es el tipo de los valores (f para **float** y d para **double**).
- ▶ Para las tablas usamos vectores de la librería estándard de C++ (tipo **std::vector**), ya que tienen tamaño variable y pueden inicializarse con una simple asignación.

Declaraciones de tablas

Un caso típico son tipos flotantes con coordenadas, colores y normales de longitud 3, y cc. de textura de longitud 2. Podemos entonces declarar las tablas de esta forma:

```
#include <tuplasg.h>    // para los tipos tuplas: Tupla3f, Tupla2f, etc....  
std::vector<Tupla3f>    posiciones; // coordenadas de pos. de vértices  
std::vector<Tupla3f>    colores;    // colores de vértices  
std::vector<Tupla3f>    normales;   // normales de vértices  
std::vector<Tupla2f>    coord_text; // coords. de text. de vérts.  
std::vector<unsigned int> indices; // índices
```

- ▶ La tabla de vértices no puede estar vacía, y si la secuencia es indexada, la tabla de índices tampoco.
- ▶ Cada tabla de atributos o bien está vacía, o bien tiene tantas tuplas como la de vértices.
- ▶ La cuenta de entradas se obtiene con el método **size** de **std::vector**.
- ▶ El puntero al primer valor se obtiene con el método **data** de **std::vector**.

Almacenamiento de tablas en GPU: *Vertex Buffer Objects*

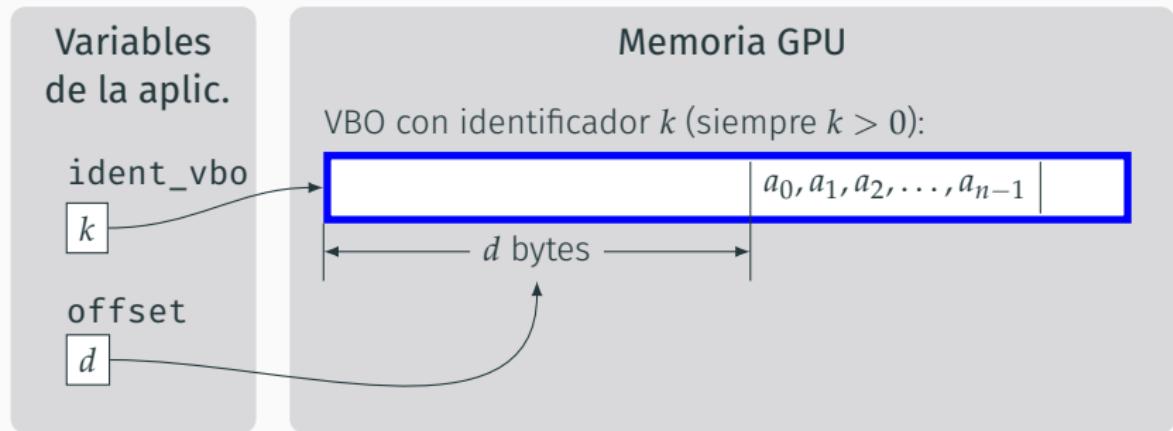
El modo diferido requiere reservar memoria en la GPU, para ello se usan los *Vertex Buffer Objects* (VBOs).

Un *Vertex Buffer Object* es una secuencia de bytes contiguos (un bloque de memoria) en la memoria de la GPU. Dicho bloque contiene una o varias tablas con coordenadas, colores u otros atributos de vértices.

- ▶ El uso de ese bloque de memoria se hace exclusivamente a través de llamadas a OpenGL (decimos que una VBO está *gestionado por OpenGL*),
- ▶ Cada VBO tiene un valor entero único (mayor que cero) que denominamos **nombre** (**name**) o **identificador** de VBO. Es de tipo **GLuint** (equivale a **unsigned int**).
- ▶ Un VBO puede tener atributos o puede tener índices, pero no ambos mezclados (los llamamos **VBOs de atributos** y **VBOs de índices**, respectivamente).

Tablas en VBOs: identificador y offset

El identificador del VBO (**ident_vbo**) y el offset (**offset**) deben almacenarse en la memoria RAM (como variables de la aplicación), de forma que podamos acceder a los datos en el VBO:



Parámetros descriptores de una tabla de atributo (1/2)

Una tabla de atributos se puede describir usando un conjunto de valores o metadatos relativos a la propia tabla. A ese conjunto lo llamamos **parámetros descriptores**, son estos:

1. Índice de atributo o *index* (**GLuint ind_atributo**): índice asociado al atributo (≥ 0). Por claridad, nosotros usaremos constantes con nombre descriptivos:

```
constexpr GLuint
    ind_atrib_posiciones = 0,    ind_atrib_colores      = 1 ,
    ind_atrib_normales   = 2,    ind_atrib_coord_text = 3 ,
    numero_atributos_cauce = 4 ;
```

2. Tipo de valores o *type* (**GLenum tipo_valores**): codifica el tipo de los valores, puede valer **GL_FLOAT** para **float** o **GL_DOUBLE** para **double** (generalmente en las prácticas y ejemplos usaremos datos de tipo **float**).

Parámetros descriptores de una tabla de atributo (2/2)

3. Número de valores por tupla o size

(**GLint num_vals_tupla**): número de valores reales por vértice, en general puede ser de 1 a 4 para las tablas de atributos y 1 en tablas de índices. En nuestro caso será 3 para las posiciones y resto de atributos, excepto para las coordenadas de textura, que vale 2.

4. Número de tuplas (número de vértices) o count

(**GLsizei num_tuplas**): número de tuplas de datos, coincide con el número vértices, ya que debe haber exactamente una tupla por vértice.

5. Puntero a datos o data (**const void * datos**): puntero a la dirección de memoria del programa donde está el primer byte de la tabla. No puede ser nulo y se usa simplemente para lectura.

Otros parámetros de una tabla de atributo (1/2)

Hay un par de valores que habría que usar en general, pero los cuales valen ambos cero siempre en los ejemplos y en las prácticas:

- ▶ Longitud de paso o stride (**GLsizei long_paso**): distancia en bytes entre los atributos de un vértice y el siguiente, cuando se usan AOS y hay atributos adicionales a las posiciones. Para SOA puede valer 0, así que siempre usaremos 0.
- ▶ Desplazamiento u offset (también llamado *pointer*) (**const void * desplazamiento**): número de bytes que hay en el VBO entre el inicio del VBO y el inicio de esta tabla. Puesto que siempre alojaremos una única tabla en cada VBO, este valor siempre será 0.

Por claridad, usaremos dos constantes (**long_paso** y **desplazamiento**), definidas como 0.

Otros parámetros de una tabla de atributo (2/2)

Estos dos parámetros adicionales se pueden calcular a partir de los anteriores:

- ▶ Número de bytes por valor (`unsigned num_bytes_valor`), se puede calcular con la función `sizeof` de C++, exclusivamente en base a `tipo_valores` (> 0).
- ▶ Tamaño en bytes o size (`GLsizeiptr tamaño_en_bytes`) tamaño de la tabla completa, se calcula como producto de `num_bytes_valor`, `num_vals_tupla` y `num_tuplas_ind`.

Parámetros descriptores de una tabla de índices

En el caso de una tabla de índices, sus parámetros descriptores son:

1. Tipo de los índices o *type* (**GLenum tipo_indices**): se usan tipos enteros sin signo, puede valer: **GL_UNSIGNED_BYTE** para **unsigned char**, **GL_UNSIGNED_SHORT** para **unsigned short** o **GL_UNSIGNED_INT** para **unsigned**. En las prácticas y ejemplos usaremos **unsigned**.
2. Número de índices o *count* (**GLsizei num_indices**): número total de índices en la tabla.
3. Puntero a datos o *data* (**void * indices**): puntero a la dirección de memoria del programa donde está el primer byte con los índices (todos consecutivos).

Otros parámetros de una tabla de índices

Además de los anteriores, hay otros parámetros (que dependen de los otros o que son cero):

- ▶ Número de bytes por índice (`unsigned num_bytes_indice`), se puede calcular con la función `sizeof` de C++, exclusivamente en base a `tipo_indices` (> 0).
- ▶ Tamaño en bytes o size (`GLsizeiptr tamaño_en_bytes`) tamaño de la tabla completa, se calcula como producto de `num_bytes_indice` y `num_indices`.
- ▶ Desplazamiento u offset (también llamado *pointer*) (`const void * desplazamiento`): mismo significado que en la tabla de atributos, y por tanto siempre es cero.

Almacenamiento de secuencias de vértices en GPU

Antes de poder visualizar una secuencia de vértices es necesario especificar a OpenGL todos los datos relativos a dicha secuencia, a saber:

- ▶ Parámetros descriptores de la tabla de coordenadas de posición.
- ▶ Para cada atributo (adicional a las posiciones) gestionado por los shaders:
 - ▶ Un valor lógico que indica si esta secuencia de vértices tiene asociada una tabla de este atributo.
 - ▶ Si la tiene, decimos que esa tabla de un atributo está habilitada (*enabled*).
 - ▶ Si no la tiene, está deshabilitada (*disabled*), y todos los vértices toman el mismo valor para ese atributo.
 - ▶ Si está habilitada, parámetros descriptores de esa tabla.
 - ▶ Si la secuencia es indexada, parámetros descriptores de la tabla de índices.

Vertex Array Objects (VAO) en OpenGL

Para gestionar todos los datos relativos a una secuencia de vértices, las APIs de rasterización usan determinadas estructuras de datos. En el caso de OpenGL, se llaman *Vertex Array Objects*:

Un **Vertex Array Object (VAO)** es una estructura de datos, gestionada por OpenGL, que almacena toda la información sobre una secuencia de vértices: la localización y el formato de las coordenadas, otros atributos, e índices.

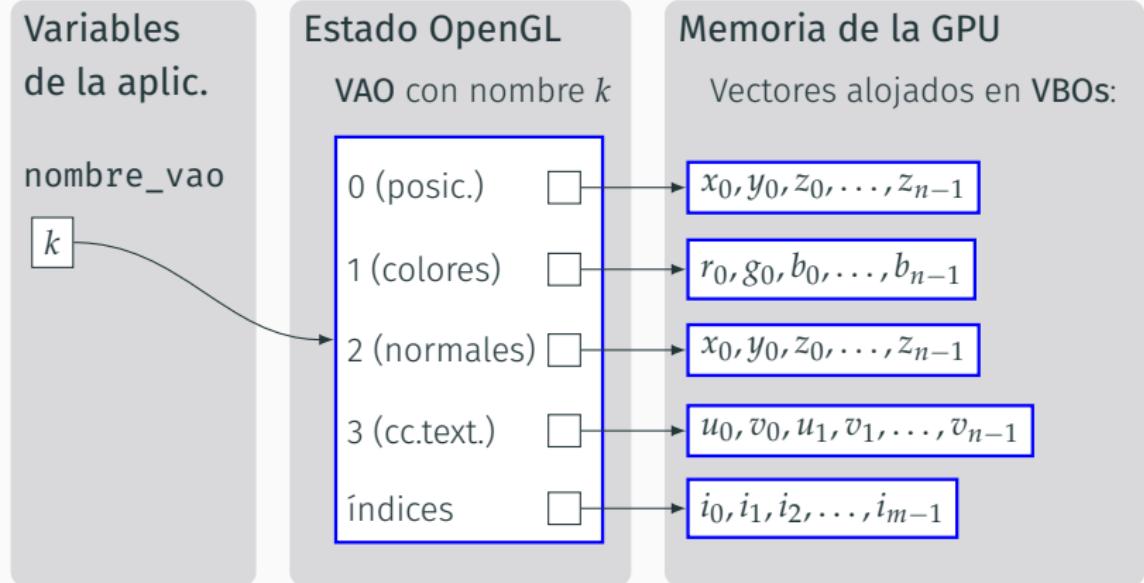
- ▶ Una aplicación debe crear una de estas estructuras por cada secuencia de vértices que quiera visualizar.
- ▶ Cada VAO forma parte del estado de OpenGL, y se usa o actualiza exclusivamente mediante llamadas a OpenGL (no directamente).

Vertex Array Objects (VAOs) de OpenGL

En el estado de OpenGL se puede guardar información sobre múltiples secuencias de vértices que forman la escena que queremos visualizar:

- ▶ Cada VAO se identifica por un entero (no negativo) único (es el *nombre* o *identificador* del VAO).
- ▶ Siempre hay un VAO activo y en uso. Inicialmente es el VAO con nombre 0.
- ▶ Para visualización en modo inmediato (OpenGL antiguo) se usaba siempre el VAO con identificador 0, el **VAO por defecto** (creado y activo al inicio), requiere modificarlo para cada secuencia distinta que se quiera visualizar. En nuestro caso, nunca usamos el VAO 0 (no es posible en OpenGL moderno).
- ▶ Para visualización en modo diferido con OpenGL moderno, se usa **VAOs creado por la aplicación y alojados en la GPU** (tienen identificador mayor que 0, puede haber uno por cada secuencia de vértices).

Esquema de un VAO



Los VBOs de atributos e índices son opcionales (pueden no estar habilitados en un VAO), excepto las coordenadas de posición, que son obligatorias en todos los VAO.

Operaciones sobre VAOs

Para operar con un VAO se pueden usar las siguientes funciones:

- ▶ **glGenVertexArrays**: crea uno o varios VAO
- ▶ **glBindVertexArray**: activa un VAO ya creado
- ▶ **glDeleteVertexArray**: destruye uno o varios VAOs
- ▶ **glVertexAttribPointer**: especifica los parámetros descriptores de una tabla de atributos dentro del VAO activo (dado su índice).
- ▶ **glEnableVertexAttribArray**,
glDisableVertexAttribArray: habilita o deshabilita el uso de una tabla de atributos concreta en el VAO activo (dado su índice).

Si una tabla de atributos está deshabilitada al visualizar un VAO, todos los vértices usarán el *valor actual* de dicho atributo. Ese *valor actual* se puede cambiar con la función **glVertexAttrib** (antes de visualizar).

Operaciones sobre VBOs en el VAO activo

OpenGL tiene siempre un VBO activo, inicialmente es el VBO con nombre 0 (VBO por defecto, no usable), y se debe cambiar para operar con otro VBOs. Para poder crear y poblar VBOs dentro del VAO activo, se puede usar:

- ▶ **glGenBuffers** crea uno o varios VBOs, se obtiene el identificador nuevo de cada uno.
- ▶ **glBindBuffer** activa un buffer ya creado, usando su identificador.
- ▶ **glBufferData** reserva memoria para el VBO activo y transfiere un bloque de bytes desde la memoria de la aplicación (RAM) hacia dicha memoria (los contenidos previos del VBO, si había alguno, se pierden).
- ▶ **glSubBufferData** permite actualizar un bloque de bytes dentro del VBO (no lo usamos).

Creación y activación de un VBO de atributo (1/3)

Esta función crea un VBO de atributos, dados los parámetros descriptores de la tabla. También transfiere los datos a la GPU y fija los parámetros en OpenGL.

```
GLenum CrearVBOAtrib( GLuint ind_atributo, GLenum tipo_datos,
                      GLint num_vals_tupla, GLsizei num_tuplas,
                      const GLvoid * datos )
{
    // 1. comprobar integridad los parámetros:
    assert( datos != nullptr );
    assert( num_vals_tupla > 0 ); assert( num_tuplas > 0 );
    assert( ind_atributo < numero_atributos_cause );
    assert( tipo_datos == GL_FLOAT || tipo_datos == GL_DOUBLE );

    // 2. calcular parámetros no independientes
    const GLsizei
        num_bytes_valor = (tipo_datos == GL_FLOAT) ? sizeof(float)
                                                    : sizeof(double),
        tamano_en_bytes = num_tuplas * num_vals_tupla * num_bytes_valor;

    ...
}
```

Creación y activación de un VBO de atributos (2/3)

```
GLenum CrearVBOAtrib( ..... )
{
    .....

    // 3. crear y activar VBO (vacío por ahora)
    GLenum nombre_vbo = 0;
    glGenBuffers( 1, &nombre_vbo ); assert( 0 < nombre_vbo );
    glBindBuffer( GL_ARRAY_BUFFER, nombre_vbo );

    // 4. transfiere datos desde aplicación al VBO en GPU
    glBufferData( GL_ARRAY_BUFFER, tamano_en_bytes, datos, GL_STATIC_DRAW );

    // 5. establece formato y dirección de inicio en el VBO
    // ('long_paso' y 'desplazamiento' son 0)
    glVertexAttribPointer( ind_atributo, num_vals_tupla, tipo_datos,
                          GL_FALSE, long_paso, desplazamiento );

    // 6. habilita la tabla, desactiva VBO
    glEnableVertexAttribArray( ind_atributo );
    glBindBuffer( GL_ARRAY_BUFFER, 0 );

    // 7. devolver el nombre o identificador de VBO
    return nombre_vbo ;
}
```

Creación y activación de un VBO de atributo (3/3)

Por comodidad, podemos usar versiones de **CrearVBOAtrib** que aceptan vectores de tuplas de 2 o 3 flotantes:

```
GLenum CrearVBOAtrib( unsigned ind_atributo,
                      const std::vector<Tupla3f> & tabla )
{
    return CrearVBOAtrib( ind_atributo, GL_FLOAT, 3,
                          tabla.size(), tabla.data() );
}
GLenum CrearVBOAtrib( unsigned ind_atributo,
                      const std::vector<Tupla2f> & tabla )
{
    return CrearVBOAtrib( ind_atributo, GL_FLOAT, 2,
                          tabla.size(), tabla.data() );
}
```

Creación y activación de un VBO de índices (1/3)

Esta función crea y deja activado un VBO de índices,

```
GLEnum CrearVBOInd(  GLenum tipo_indices, GLint num_indices,
                      const void * indices )

{
    // 1. comprobar la integridad de los parámetros
    assert( num_indices > 0 );
    assert( indices != nullptr );
    assert( tipo_indices == GL_UNSIGNED_BYTE || 
            tipo_indices == GL_UNSIGNED_SHORT || 
            tipo_indices == GL_UNSIGNED_INT      );

    // 2. calcular cuantos bytes ocupa cada índice y la tabla completa
    const GLint num_bytes_indice =
        (tipo_indices == GL_UNSIGNED_BYTE ) ? sizeof(char) : 
        (tipo_indices == GL_UNSIGNED_SHORT) ? sizeof(unsigned short) :
                                            sizeof(unsigned int) ;
    const GLsizei tamano_en_bytes = num_bytes_indice * num_indices ;

    ....
}
```

Creación y activación de un VBO de índices (2/3)

```
GLenum CrearVBOInd( ..... )
{
    ....
    // 3. crear y activar el VBO
    GLenum nombre_vbo = 0;
    glGenBuffers( 1, &nombre_vbo ); assert( 0 < nombre_vbo );
    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, nombre_vbo );

    // 4. copiar los índices desde la aplicación al VBO en la GPU
    glBufferData( GL_ELEMENT_ARRAY_BUFFER, tamano_en_bytes, indices,
                  GL_STATIC_DRAW );

    // 5. hecho, devolver el nombre o identificador de
    return nombre_vbo ;
}
```

Al acabar esta función, el VBO queda como buffer de índices actual de OpenGL (es el designado por la constante **GL_ELEMENT_ARRAY_BUFFER**). Esto es necesario para visualizar después.

Creación y activación de un VBO de índices (3/3)

Por comodidad, podemos usar una versión de `CrearActVBOInd` que acepta un vector de `unsigned`

```
GLenum CrearVBOInd( const std::vector<unsigned> & indices )
{
    return CrearVBOInd( GL_UNSIGNED_INT,
                        indices.size(), indices.data() );
}
```

También otra que acepta vectores de tuplas de 3 enteros, esta se usará más adelante para *mallas indexadas de triángulos*:

```
GLenum CrearVBOInd( const std::vector<Tupla3u> & indices )
{
    return CrearVBOInd( GL_UNSIGNED_INT,
                        3*indices.size(), indices.data() );
}
```

Creación y activación de un VAO

Esta función crea un VAO, lo activa como VAO actual, y devuelve el nombre:

```
GLenum CrearVAO()
{
    GLenum nombre_vao = 0 ;
    glGenVertexArrays( 1, &nombre_vao ); // generar nuevo nombre
    glBindVertexArray( nombre_vao ); // activar VAO
    return nombre_vao ;
}
```

- ▶ El VAO queda activado tras la llamada (hasta que se haga **glBindVertexArray(0)** o se active otro VAO).
- ▶ Después de creado un VAO, podemos crear VBOs que queden asociados al VAO activo actualmente.
- ▶ Un mismo VBO con una tabla de atributos o índices puede usarse en más de un VAO.

Creación de VAOs para tablas de tuplas

En un caso general, y para las tablas que codifican una secuencia de vértices, podríamos hacer:

```
nombre_vao = CrearVAO(); // el VAO queda activado

CrearVBOAtrib( ind_atrib_posiciones, posiciones ); // crear VBO pos.
if (indices.size() >0) CrearVBOInd( indices );
if (colores.size() >0) CrearVBOAtrib( ind_atrib_colores, colores );
if (normales.size() >0) CrearVBOAtrib( ind_atrib_normales, normales );
if (coord_text.size()>0) CrearVBOAtrib( ind_atrib_coord_text, coord_text);
```

Este código debe ejecutarse:

- ▶ Despues de haber incializado OpenGL y haberse creado la ventana (se dice que *ya existe un contexto de OpenGL*).
- ▶ Antes de visualizar la secuencia por primera vez.

Para cumplir esto se puede hacer la creación del VAO inmediatamente antes de la primera visualización.

Problema 1.1.

Escribe el código que genera una tabla de coordenadas de posición de vértices con las coordenadas de los vértices de un polígono regular de n lados o aristas (es una constante del programa), con centro en el origen y radio unidad.

Los vértices se almacenan como flotantes de doble precisión (**double**), con 2 coordenadas por vértice. Escribe el código que crea el correspondiente VAO a esta secuencia de vértices.

En estos problemas, puedes usar las funciones **CrearVBOAtrib**, **CrearVBOInd** y **CrearVAO**.

(el enunciado continua en la siguiente transparencia)

Problema 1.1. (continuación)

El valor de n (> 2) es un parámetro (un entero sin signo). La tabla sería la base para visualizar el polígono (únicamente las aristas), considerando la tabla de vértices como una **secuencia de vértices no indexada**.

Escribe dos variantes del código, de forma que la tabla se debe diseñar para representar el polígono usando distintos tipos de primitivas:

- (a) tipo de primitiva **GL_LINE_LOOP**.
- (b) tipo de primitiva **GL_LINES**.

Problemas: generación de triángulos de un polígono regular

Problema 1.2.

Escribe el código para generar una tabla de vértices y una tabla de índices, que permitiría visualizar el mismo polígono regular del problema anterior pero ahora como un conjunto de n triángulos iguales rellenos que comparten un vértice en el centro del polígono (el origen). Usa ahora datos de simple precisión **float** para los vértices, con tres valores por vértice, siendo la Z igual a 0 en todos ellos.

La tabla está destinada a ser visualizada con el tipo de primitiva **GL_TRIANGLES**.

Escribe dos variantes del código: una variante (a) usa una secuencia no indexada (con $3n$ vértices), y otra (b) usa una secuencia indexada (sin vértices repetidos), con $n + 1$ vértices y $3n$ índices.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.7.

Envío de vértices y atributos..

Funciones de visualización en modo diferido

Para visualizar una secuencia de vértices en el modo *diferido* se usan exclusivamente las funciones **glDrawArrays** (no indexado) y **glDrawElements** (array indexado).

- ▶ Antes de la primera visualización, **una única vez**, debemos almacenar las secuencias de coordenadas, atributos e índices (si procede) en uno o varios VBOs dentro de un VAO específico.
- ▶ En cada visualización solo es necesario activar el VAO (con **glBindVertexArray**) y visualizar con una única llamada (con **glDrawArrays** o **glDrawElements**).
- ▶ Ambas funciones visualizan la secuencia **de forma asíncrona**: durante la llamada la visualización se pone en marcha, pero no necesariamente ha acabado cuando termina de ejecutarse la función.

La función `glDrawArrays`

La función **glDrawArrays** requiere que haya un VAO activado (distinto del VAO 0), y visualiza una secuencia de vértices usando las tablas de atributos habilitadas en dicho VAO. Tiene estos parámetros:

```
glDrawArrays( tipo_primitiva, primero, num_tuplas );
```

- ▶ **tipo_primitiva** es alguna de las constantes asociadas a los tipos de primitivas que ya conocemos.
- ▶ **primero** es el índice del primer vértice a visualizar, puede ser > 0 , pero en nuestro caso será siempre 0
- ▶ **num_tuplas** es el número de vértices que queremos visualizar, en nuestro caso siempre visualizamos todos los vértices del VAO, luego es el número de vértices de la secuencia.

La función `glDrawElements`

La función **`glDrawElements`** sirve para visualizar secuencia indexadas, además de usar las tablas de atributos habilitadas, asume que hay activada una tabla de índices, y la usa:

```
glDrawElements( tipo_primitiva, num_indices, tipo_indices, desplazamiento);
```

- ▶ **`tipo_primitiva`** es alguna de las constantes asociadas a los tipos de primitivas que ya conocemos.
- ▶ **`num_indices`** es el número de índices que queremos visualizar, en nuestro caso siempre serán todos los índices de la secuencia.
- ▶ **`tipo_indices`** tipo de los índices (en general).
- ▶ **`desplazamiento`** distancia en bytes desde el inicio del VBO hasta el primer índice a visualizar, en nuestro caso siempre será 0.

Creación y visualización de VAOs

Por tanto, para visualizar una secuencia de vértices genérica, únicamente debemos de activar el VAO y después usar **glDrawArrays** o **glDrawElements**. Para ello usamos una variable (permanente, no local) con el nombre del VAO (**nombre_vao**), inicializada a 0:

```
if ( nombre_vao == 0 ) // si VAO aun no se ha creado
{
    nombre_vao = CrearVAO(); // crear el VAO con las coordenadas de vértices
    CrearVBOAtrib( ind_atrib_posiciones, .... ); // añadir VBO de posiciones
    ..... // añadir otros VBOs de atributos o índices (si hay)
    if (es una secuencia indexada )
        CrearVBOInd( .... );
}
else // si el VAO ya está creado
    glBindVertexArray( nombre_vao ); // activar el VAO

if (es una secuencia indexada)
    glDrawElements( tipo_primitiva, num_indices, tipo_indices, 0); //desplz==0
else
    glDrawArrays( tipo_primitiva, 0, num_tuplas ); // primero == 0
```

Problema: visualización de polígono regular (1/2)

Problema 1.3.

Crea una copia del repositorio `opengl3-minimo`, y modifica el código de ese repositorio para incluir una nueva función para visualizar las aristas y el relleno de polígono regular de n lados (donde n es una constante de tu programa), usando las tablas, VBOs y VAOs de coordenadas que codifican dicho polígono regular, según se describe en:

- ▶ el enunciado del problema 1.1 (variante (a), con **GL_LINE_LOOP**) para las aristas,
- ▶ el enunciado del problema 1.2 (variante (b), secuencia indexada) para el relleno.

(el enunciado continua en la siguiente transparencia)

Problema: visualización polígono regular (2/2)

Problema 1.3. (continuación)

El polígono se verá relleno de un color plano y las aristas con otro color (también plano), distintos ambos del color de fondo. Debes usar un VAO para las aristas y otro para el relleno.

No uses una tabla de colores de vértices para este problema, en su lugar usa la función **glVertexAttrib** para cambiar el color antes de dibujar.

Incluye todo el código en una función de visualización (nueva), esa función debe incluir tanto la creación de tablas, VBOs y ambos VAOs (en la primera llamada), como la visualización (en todas las llamadas).

Problema: visualización de polígono regular con VAO único

Problema 1.4.

Repite el problema anterior (problema 1.3), pero ahora intenta usar el mismo VAO para las aristas y los triángulos rellenos. Para eso puedes usar una única tabla de $n + 1$ posiciones de vértices, esa tabla sirve de base para el relleno, usando índices. Para las aristas, puedes usar **GL_LINE_LOOP**, pero teniendo en cuenta únicamente los n vértices del polígono (sin usar el vértice en el origen).

Problema: visualización de polígono regular con colores

Problema 1.5.

Repite el problema anterior (problema 1.4), pero ahora el relleno, en lugar de hacerse todo del mismo color plano, se hará mediante interpolación de colores (las aristas siguen estando con un único color). Para eso se debe generar una tabla de colores de vértices, inicializada con colores aleatorios para cada uno de los $n + 1$ vértices.

Ten en cuenta que para visualizar las aristas, debes de deshabilitar antes el uso de la tabla de colores.

Problema: visualización de polígono regular con colores

Problema 1.6.

Repite el problema anterior (problema 1.5), pero ahora, para guardar las posiciones y los vértices, se usará una estructura tipo AOS, es decir, se usa un array de estructuras o bloques de datos, en cada estructura o bloque se guardan 3 flotantes para la posición y 3 flotantes para el color RGB.

La estructura completa se debe alojar en un único VBO de atributos con todos los flotantes de las posiciones y colores, así que no uses las funciones dadas para creación de VAOs y VBOs (asumen una tabla por VBO con estructura SOA).

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.8.

Estado de OpenGL y visualización de un frame.

Introducción

En este apartado veremos como cambiar diversas variables o parámetros (forman parte del estado de OpenGL) que determinan como se visualizan las primitivas:

- ▶ Las variables del estado de OpenGL se modifican o leen mediante funciones OpenGL, nunca directamente.
- ▶ En el cauce programable, eso incluye los parámetros *uniform*
- ▶ Muchos parámetros no se modifican (se usan con el valor inicial por defecto)
- ▶ Algunos parámetros bastará con darles valor al inicio, una sola vez.
- ▶ Si un parámetro se modifica durante la visualización de cada cuadro, hay que fijarlo a un valor conocido al inicio de cada cuadro.

También veremos un ejemplo de la función de visualización de un cuadro (**VisualizarFrame**)

Cambio del modo de visualización de polígonos (1/2)

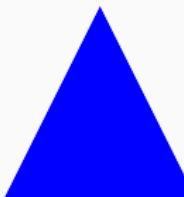
El modo de visualización de polígonos se cambia usando la llamada:

```
glPolygonMode( GL_FRONT_AND_BACK, nuevo_modo )
```

- ▶ *nuevo_modo* es un valor de tipo **GLenum** (un entero) que puede valer alguna de estas tres constantes:
 - ▶ **GL_POINT** se visualizan únicamente los vértices como puntos.
 - ▶ **GL_LINE** se visualizan únicamente las aristas como segmentos.
 - ▶ **GL_FILL** se visualizan el triángulo relleno del color actual.
- ▶ El valor inicial es **GL_FILL**.
- ▶ En OpenGL 2.1 o anteriores, también es posible usar **GL_FRONT** y **GL_BACK** en lugar de **GL_FRONT_AND_BACK**. Permite seleccionar el modo exclusivamente para las caras delanteras, o exclusivamente para las traseras.

Color de vértices y modos de sombreado

Si el cauce gráfico está preparado, OpenGL permite usar dos **modos de sombreado**:



- ▶ **Modo plano** (izq.): se asigna a toda la primitiva un color plano, igual al color del último vértice que forma la primitiva.
- ▶ **Modo de interpolación (suave)** (der.): se hace una interpolación lineal de las componentes RGB del color, usando los colores de todos los vértices.

Los *shaders* del repositorio `opengl3-minimo`, permiten cambiar entre ambos modos (por defecto se hace sombreado suave).

Cambio del modo de sombreado

En `opengl3-minimo`, el modo de sombreado plano se puede activar y desactivar modificando un parámetro de los shaders (de nombre `u_usar_color_plano`), de tipo lógico, cuando vale `true` se activa el sombreado plano, y cuando vale `false` se activa interpolación.

Para cambiarlo usamos:

```
glUniform1i( loc_usar_color_plano, nuevo_valor );
```

Donde `nuevo_valor` puede ser `GL_TRUE` o `GL_FALSE`.

La variable `loc_usar_color_plano` es una variable entera de la aplicación que sirve para identificar el parámetro `u_usar_color_plano` de los shaders.

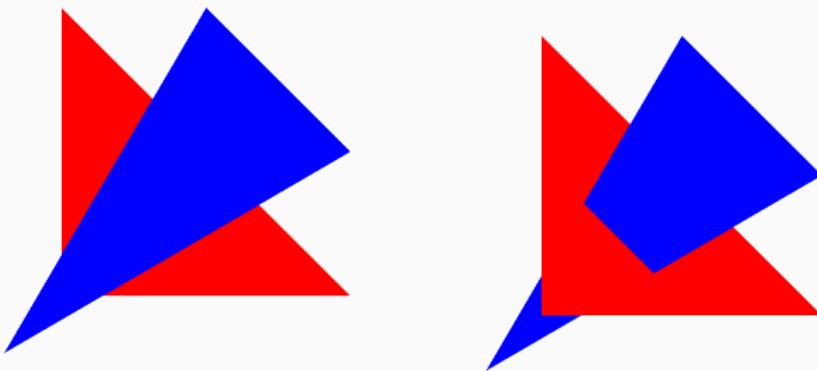
Eliminación de partes ocultas (EPO) con Z-buffer

OpenGL usa las coordenadas Z de los vértices para calcular (por interpolación) la profundidad en Z en cada pixel de cada primitiva visualizada (Z es la dirección perpendicular a la pantalla).

- ▶ Existe un buffer (llamado **Z-buffer**) donde se guarda la coordenada Z de lo que hay dibujado en cada pixel. Esto permite hacer el **test de profundidad** (*depth test*).
- ▶ Permite dibujar primitivas en 2D o 3D con posibles ocultaciones entre ellas.
- ▶ Inicialmente (por defecto) en un pixel, una primitiva *A* con una Z menor estará por delante de otra *B* con una Z mayor (*A* oculta a *B*).
- ▶ Esto puede activarse o desactivarse, con **glEnable** y **glDisable**, usando **GL_DEPTH_TEST** como argumento. Inicialmente, **está desactivado**.

Ejemplo de EPO con Z-buffer.

Se visualiza en primer lugar el triángulo rojo y luego el azul. A la izquierda está deshabilitado el test de profundidad, y a la derecha está habilitado:



Hay que recordar activar este test, y, al limpiar la pantalla, limpiar también el Z-buffer.

Otros parámetros de visualización

OpenGL guarda (dentro de su **estado** interno) varios atributos que se usarán para la visualización de primitivas o para su operación en general. Entre otros muchos, podemos destacar estos:

- ▶ Aspecto de las primitivas:
 - ▶ **Ancho** (en pixels) de las lineas (real), con la función **glLineWidth**.
 - ▶ **Ancho** (en pixels) de los puntos (real), con la función **glPointSize**.
- ▶ Otros atributos:
 - ▶ **Color** que será usado cuando se limpie la ventana (antes de dibujar) (RGBA), con la función **glClearColor**.

Lectura del estado de OpenGL

Muchísimas variables internas del estado de OpenGL pueden leerse usando las funciones `glGet`, `glIsEnabled` u otras, por ejemplo:

- ▶ Rango de anchos de línea permitidos por la aplicación:

```
GLfloat rango_lineas[2] ; // contendrá mínimo y máximo  
glGetFloatv( GL_ALIASED_LINE_WIDTH_RANGE, rango_lineas );
```

- ▶ Si está habilitado el test de profundidad o no:

```
GLboolean test_habilitado = glIsEnabled( GL_DEPTH_TEST );
```

- ▶ El valor actual de algún atributo de vértices, identificado por su índice de atributo

```
GLfloat valor_actual[4];  
glGetVertexAttribfv( indice, GL_CURRENT_VERTEX_ATTRIB, valor_actual );
```

Inicialización de OpenGL

Los valores de los atributos pueden cambiarse en cualquier momento. En nuestro ejemplo sencillo, lo haremos una vez al inicializar OpenGL:

```
void Inicializa_OpenGL( )
{
    // comprobar si el flag de error de OpenGL ya estaba activiado (si estaba aborta)
    assert( glGetError() == GL_NO_ERROR );
    // establecer color de fondo: (1,1,1) (blanco)
    glClearColor( 1.0, 1.0, 1.0, 1.0 );
    // establecer color inicial para todas las primitivas, hasta que se cambie
    glVertexAttrib3f( ind_atrib_colores, 0.7, 0.2, 0.4, 1.0 );
    // establecer ancho de líneas o segmentos (en pixels)
    GLfloat rango[2]; glGetFloatv( GL_ALIASED_LINE_WIDTH_RANGE, rango );
    glLineWidth( GL_ALIASED_LINE_WIDTH_RANGE, std::min( 2.1, rango[1] ) );
    // habilitar eliminación de partes ocultas usando el Z-buffer
    glEnable( GL_DEPTH_TEST );
    // comprobar si ha habido algún error en esta función
    assert( glGetError() == GL_NO_ERROR );
}
```

Definición del *viewport*

La función **glViewport** permite establecer que parte de la ventana será usada para visualizar. Dicha parte (llamada **viewport**) es un bloque rectangular de pixels.

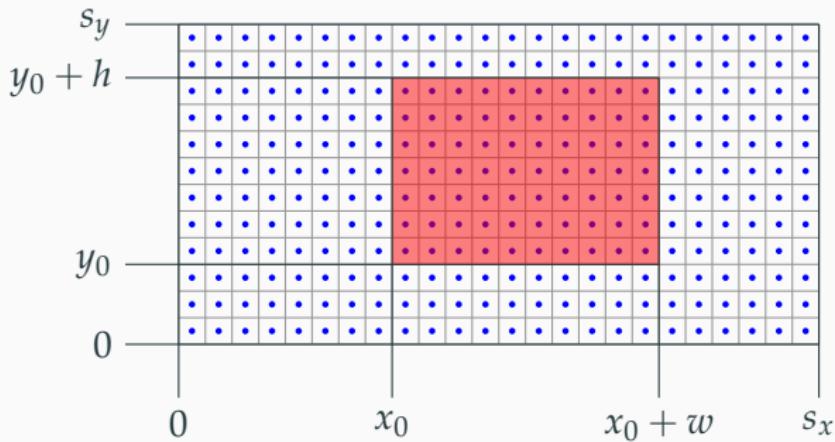
```
glViewport( izqui, abajo, ancho, alto ) ;
```

Los parametros de la función (todo enteros, no negativos) son los siguientes (en orden)

- ▶ **izqui** (x_0): número de columna de pixels donde comienza (la primera por la izquierda es la cero)
- ▶ **abajo** (y_0): número de la fila de pixels donde comienza (la primera por abajo es la cero)
- ▶ **ancho** (w): número total de columnas de pixels que ocupa.
- ▶ **alto** (h): número total de filas de pixel que ocupa.

El viewport y la ventana como rejillas de pixels

La ventana puede considerarse un bloque rectangular de pixels, cada uno con un punto central (llamado **centro del pixel**) a un cuadrado (llamado **área del pixel**), dentro está otro rectángulo que es el viewport (en rojo):



La función gestora del cambio de tamaño de ventana

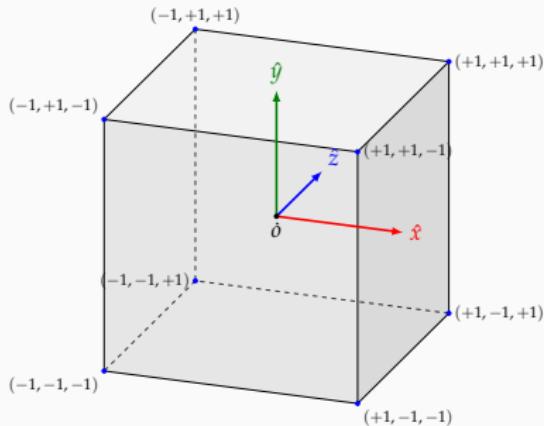
El evento de cambio de tamaño de la ventana se produce siempre una vez tras crear la ventana, y además siempre después de que se cambie su tamaño.

- ▶ Por lo tanto, podemos situar en la correspondiente función gestora una llamada a **glViewport** para establecer el rectángulo de dibujo. En nuestro ejemplo sencillo, dicho rectángulo puede ocupar toda la ventana:

```
void FGE_CambioTamano( int nuevoAncho, int nuevoAlto )
{
    glViewport( 0, 0, nuevoAncho, nuevoAlto );
}
```

Región visible inicial

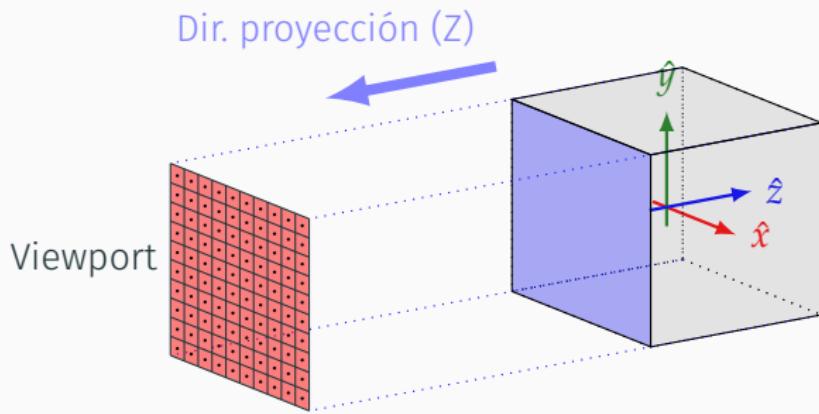
La **región visible** en pantalla es (initialmente) un cubo de lado 2 y centro en el origen (ocupa el intervalo $[-1, 1]$ en los tres ejes):



Las primitivas o partes de primitivas fuera de esta región no se dibujan (quedan *descartadas* o *recortadas*).

Región visible y proyección sobre el viewport

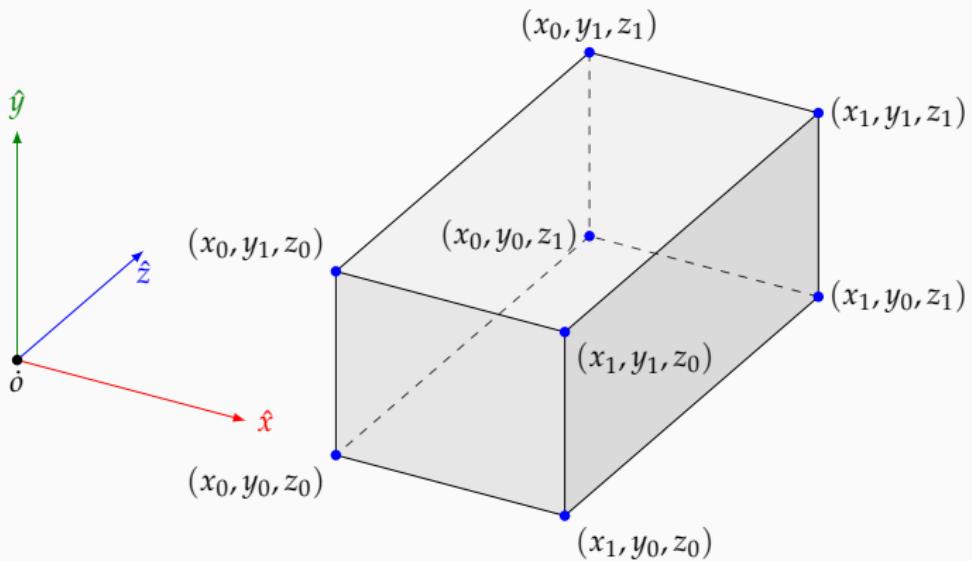
Inicialmente, OpenGL usa una proyección paralela al eje Z, hacia la rama negativa de dicho eje. Podemos imaginar los pixels de *viewport* sobre la *cara delantera* (en azul) del cubo:



Nota: si se activa EPO, las primitivas con Z menor ocultan a las primitivas con Z mayor.

Región visible arbitraria

La zona visible original (un cubo) puede cambiarse a cualquier región de posición y tamaño arbitrarios (un **ortoedro**, *cuboid*), región que estará entre x_0 e x_1 en X, entre y_0 e y_1 en Y, y entre z_0 y z_1 en Z:



Cambiar región visible: la *matriz de proyección*

OpenGL mantiene una matriz P de 4×4 valores reales (llamada **matriz de proyección**), que se aplica a las coordenadas de todos los vértices que envía la aplicación, antes de visualizar.

Para cambiar la zona visible hay que fijar la matriz P a estos valores:

$$P = \begin{pmatrix} s_x & 0 & 0 & -c_x \cdot s_x \\ 0 & s_y & 0 & -c_y \cdot s_y \\ 0 & 0 & s_z & -c_z \cdot s_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde:

$$\begin{array}{lcl} s_x & = & 2/(x_1 - x_0) \\ s_y & = & 2/(y_1 - y_0) \\ s_z & = & 2/(z_1 - z_0) \end{array} \quad \begin{array}{lcl} c_x & = & (x_0 + x_1)/2 \\ c_y & = & (y_0 + y_1)/2 \\ c_z & = & (z_0 + z_1)/2 \end{array}$$

Cambiar la matriz de proyección: llamadas

Para realizar este cambio, podemos definir una matriz de 16 floats:

```
const GLfloat matriz_proyeccion[16] =  
{  sx,  0,  0, -cx*sx,  
  0,  sy,  0, -cy*sy,  
  0,  0,  sz, -cz*sz,  
  0,  0,  0,  1  
};
```

Para cambiar la zona visible, debemos de usar una variable (**loc_proyeccion**) con un entero que identifica a la matriz de proyección en el cauce programable, y usar la función **glUniformMatrix4fv**:

```
glUniformMatrix4fv( loc_proyeccion, 1, GL_TRUE, matriz_proyeccion );
```

Esta variable se puede usar en el ejemplo **opengl3-minimo**.

Problema: definición de la región visible sin deformaciones

Problema 1.7.

Modifica el código del ejemplo **opengl3-minimo** para que no se introduzcan deformaciones cuando la ventana se redimensiona y el alto queda distinto del ancho. El código original del repositorio presenta los objetos deformados (escalados con distinta escala en vertical y horizontal) cuando la ventana no es cuadrada, ya que visualiza en el viewport (no cuadrado) una cara (cuadrada) del cubo de lado 2.

Para evitar este problema, en cada cuadro se deben de leer las variables que contienen el tamaño actual de la ventana y en función de esas variables modificar la zona visible, que ya no será siempre un cubo de lado 2 unidades, sino que será un ortoedro que contendrá dicho cubo de lado 2, pero tendrá unas dimensiones superiores a 2 (en X o en Y, no en ambas), adaptadas a las proporciones de la ventana (el ancho en X dividido por el alto en Y es un valor que debe coincidir en el ortoedro visible y en el viewport).

La función de redibujado

La visualización de primitivas debe hacerse exclusivamente una vez por cada iteración del bucle principal, en la función **VisualizarFrame** (o en otras funciones llamadas desde la misma).

- ▶ Esta función comienza con una llamada a **glClear** para restablecer el color de todos los pixels de la imagen.
- ▶ Dentro de dicha función, pueden enviarse un número arbitrario de primitivas.
- ▶ Cada vez que OpenGL termina de recibir una primitiva, se envía a través del cauce gráfico para ser visualizada, de forma **asíncrona** con la aplicación.
- ▶ Al terminar de enviar las primitivas, es necesario llamar a la función **glfwSwapBuffers**. Esto **espera a que se rasterizen las primitivas** en el *framebuffer* y después **se visualiza en la ventana la imagen** ya creada en dicho *framebuffer*.

Ejemplo de función de redibujado

Un ejemplo sencillo para la función de redibujado es esta:

```
void VisualizarFrame()
{
    // comprobar si ha habido error, restablecer variable de error
    CError();
    // configurar estado de OpenGL (cámara, modo de polígonos, etc..)
    ....
    // limpiar la ventana: limpiar colores y limpiar Z-búffer
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    // envío de secuencias de vértices (dibujamos varios objetos)
    ....
    // visualización de la imagen creada
    glfwSwapBuffers() ;
    // comprobar si ha habido error en esta función
    CError();
}
```

Detección de errores de OpenGL

Las funciones OpenGL pueden activar una variable de estado con un código de error, que en condiciones normales vale **GL_NO_ERROR**

- ▶ La función que lee ese código de error es **glGetError()**:
 - ▶ Devuelve el valor de esa variable
 - ▶ Pone la variable interna a **GL_NO_ERROR**
- ▶ Se puede abortar el programa cuando hay error usando:

```
assert( glGetError() == GL_NO_ERROR );
```

(usar **assert** tiene la ventaja de que se imprime el número de línea y el nombre del archivo)

- ▶ Para depurar programas se puede comprobar el error antes y después de cada trozo de código donde se sospeche que hay un error.
- ▶ Para aislar la llamada errónea se insertan comprobaciones dentro del trozo de código.

La macro CError

En las prácticas, para verificar los errores y obtener un mensaje descriptivo se usa **CError()**:

```
#define CError() CompruebaErrorOpenGL(__FILE__,__LINE__)
void CompruebaErrorOpenGL( const char * nomArchivo, int linea )
{
    const GLint codigoError = glGetError() ;
    if ( codigoError != GL_NO_ERROR )
    { cout
        << "Detectado error de OpenGL. Programa abortado." << endl
        << " archivo (linea) : " << QuitarPath(nomArchivo) <<"(" <<linea << ")" << endl
        << " código error      : " << ErrorCodeString( codigoError ) << endl
        << " descripción       : " << ErrorDescr( codigoError ) << "." << endl
        << endl << flush ;
        exit(1);
    }
}
```

(necesita las funciones **QuitarPath**, **ErrorCodeString** y **ErrorDescr**, ver archivo **ig-aux.cpp**).

Documentación on-line sobre OpenGL y GLFW

- ▶ Páginas de referencia:
 - ▶ OpenGL, ver.2.1 registry.khronos.org/OpenGL-Refpages/gl2.1/
 - ▶ OpenGL+GLSL, ver.4.5 registry.khronos.org/OpenGL-Refpages/gl4/
- ▶ OpenGL Programming Guide (*the red book*)
 - ▶ OpenGL 4.5: <http://www.opengl-redbook.com/>
- ▶ Registry (*documentos de especificación oficiales de OpenGL*):
 - ▶ Actuales (ver 4.6):
registry.khronos.org/OpenGL/index_gl.php#apispecs
 - ▶ Versiones anteriores:
registry.khronos.org/OpenGL/index_gl.php#oldspecs
- ▶ Librería GLFW (*documentación, código fuente, binarios*)
 - ▶ Sitio web: www.glfw.org
 - ▶ Documentación: www.glfw.org/documentation.html

Sección 4.

Programación básica del cauce gráfico.

- 4.1. El cauce gráfico. Tipos. Shaders.
- 4.2. Estructura de los *shaders*. Ejemplos.
- 4.3. Creación y ejecución de programas.
- 4.4. Funciones auxiliares.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 4. Programación básica del cauce gráfico

Subsección 4.1.

El cauce gráfico. Tipos. Shaders..

Transformación y sombreado

Hay (entre otros) dos pasos importantes del cauce gráfico de OpenGL que (usualmente) se ejecutan en la GPU:

1. Transformación:

En esta etapa se parte de las coordenadas de un vértice que se especifican en la aplicación, y se calculan las coordenadas (normalizadas) de su proyección en la ventana.

2. Sombreado:

El cálculo del color de un pixel (una vez que se ha determinado que una primitiva se proyecta en dicho pixel). Por lo visto hasta ahora, esto se hace simplemente asignando un color prefijado al pixel (pero es usualmente más complicado).

entre ambas etapas se sitúa la rasterización y el recortado de polígonos.

Los shaders. Tipos.

Los **shaders** son programas que se ejecutan en las diversas etapas del cauce gráfico. Hay de varios tipos (en distintas etapas), pero nos centramos en estos dos:

1. **Procesador de vértices (vertex shader):** programa encargado de la transformación de coordenadas.
 - ▶ Se ejecuta una vez para cada vértice en el VAO a visualizar.
 - ▶ Produce como resultado las **coordenadas normalizadas del vértice en la ventana** y opcionalmente otros atributos.
2. **Procesador de fragmentos (píxeles) (fragment shader):** subprograma encargado del sombreado.
 - ▶ Se ejecuta cada vez que se determina que una primitiva se proyecta en un pixel de la ventana.
 - ▶ Produce como resultado el **color del pixel**.

Tipos de cauces gráficos:

Hay dos tipos de cauce gráfico:

- ▶ **Cauce de funcionalidad fija** (*fixed function pipeline*):
 - ▶ Se usan shaders predefinidos en OpenGL (fijos).
 - ▶ Solo disponible hasta OpenGL 3.0 (o en versiones posteriores con el *compatibility profile*)
- ▶ **Cauce programable** (*programmable pipeline*):
 - ▶ El programador de la aplicación especifica el código fuente de los shaders, que se escribe en el lenguaje llamado **GLSL** (parecido a C).
 - ▶ Los shaders se compilan y enlazan en tiempo de ejecución (OpenGL incorpora un compilador/enlazador de GLSL).
 - ▶ Es más **flexible**: se puede escribir código arbitrario para funciones no previstas en el cauce fijo.
 - ▶ Es más **eficiente**: no obliga a ejecutar código innecesario para aplicaciones específicas.

Etapas del cauce gráfico y shaders (1/2)

El cauce gráfico completo incluye otras etapas y tipos de shaders (opcionales). Aquí vemos la lista de etapas en secuencia:

1. Procesamiento de vértices *Vertex Processing*:
 - 1.1. Lectura de vértices y atributos de los VAOs.
 - 1.2. Procesado de cada vértice: **Vertex Shader** (programable, obligatorio).
 - 1.2. Teselado (*Tesselation*): *Tesselation Shaders* (programables, opcionales).
 - 1.3. Procesado de primitivas: *Geometry shader* (programable, opcional).
 2. Post-procesado de vértices: ensamblado de primitivas, recortado, división de perspectiva, transformación de viewport (no programable).
- (sigue).

Etapas del cauce gráfico y shaders (2/2)

En este punto se obtienen las primitivas visibles ya proyectadas en el viewport (ordenadas en unidades de pixels), y se puede proceder a su *rasterizado* (o *scan conversión*). Las etapas restantes son:

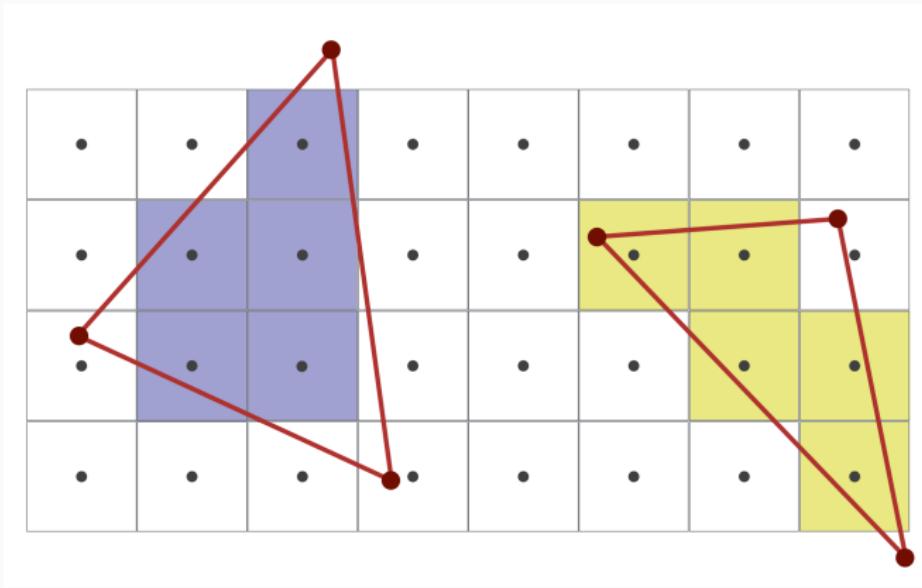
3. Rasterizado e interpolación de atributos (no programable).
4. Procesado de fragmentos: **Fragment Shader** (programable, obligatorio).
5. Procesado de muestras: etapas adicionales que se ejecutan después (algunas antes) del procesado de fragmentos, incluyendo, por ejemplo, el test de profundidad (no programable).

Más info en el sitio web del consorcio Khronos:

☞ https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

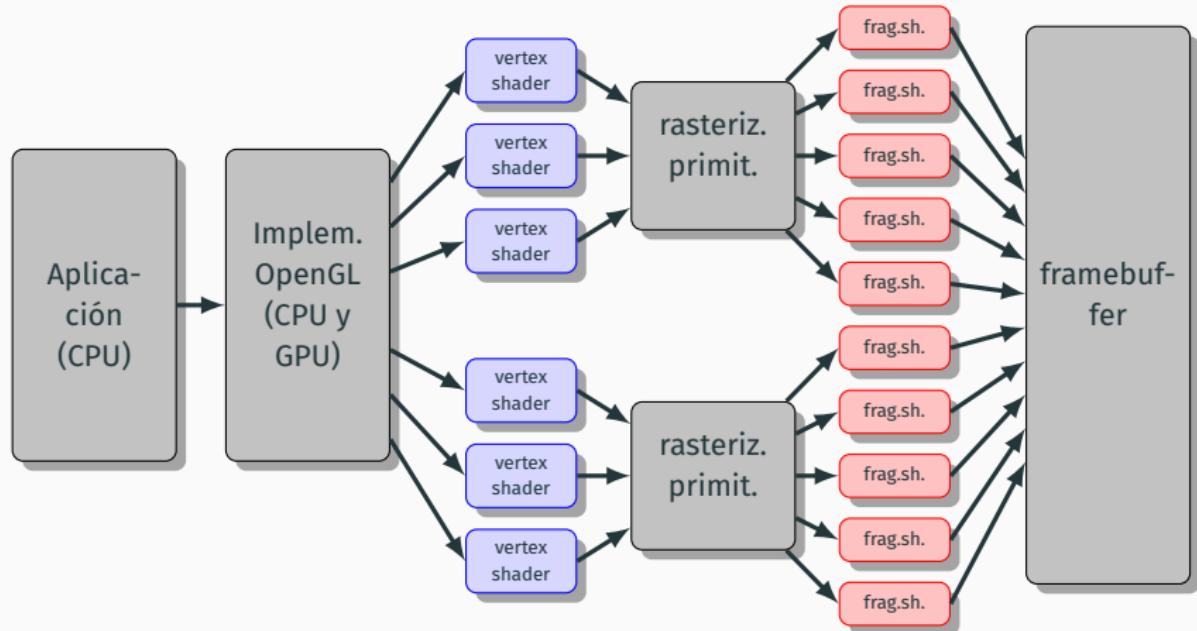
Ejemplo: Visualización de 2 triángulos

En este ejemplo, tenemos 6 vértices que definen 2 triángulos y que cubren 6 pixels (cada triángulo cubre 5 pixels):



Cauce gráfico: DFD simplificado

Para el ejemplo anterior, el DFD de la rasterización sería así:



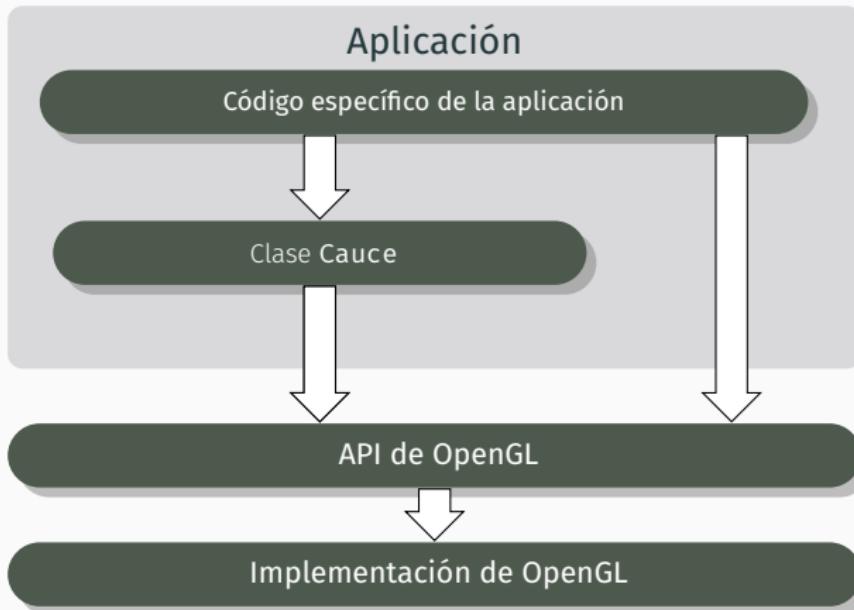
Implementación de cauce programable

Una aplicación puede usar el cauce fijo o el cauce programable:

- ▶ Algunas órdenes de OpenGL para configurar el cauce fijo difieren de las usadas para el cauce programable
- ▶ Algunas órdenes de OpenGL son iguales para ambos tipos de cauce.
- ▶ Para las prácticas, usaremos la clase **Cauce** para inicializar y configurar los parámetros del cauce programable.
- ▶ En la inicialización de la instancia de **Cauce** (en el constructor), se compilan los shaders.
- ▶ Una vez inicializado, se usan métodos para configurar los parámetros de la instancia.

Clase para interfaz de acceso al cauce

Se usa el objeto **Cauce** para algunas cosas y la API para otras:



Esto permite abstraernos de los detalles de los *shaders*.

Ejemplos de métodos de configuración del cauce

A modo de ejemplo, vemos como se cambia una variable de estado del cauce que contiene el color a usar para las visualizaciones posteriores (si el VAO no tiene colores), es decir, es el valor actual del atributo de color del vértice:

```
void Cauce::fijarColor( const float r, const float g, const float b )
{
    color = { r,g,b } ; // registra color en el objeto cauce
    glVertexAttrib3f( ind_atrib_colores, r, g, b ); // cambia valor atributo
}
```

Otro ejemplo es el método que sirve para activar o desactivar la iluminación (cambia un parámetro de los shaders):

```
void Cauce::fijarEvalMIL( const bool nue_eval_mil )
{
    eval_mil = nue_eval_mil ; // registra valor en el objeto Cauce.
    glUseProgram( id_prog ); // activa el programa
    glUniform1ui( loc_eval_mil, eval_mil ); // cambia parámetro de los shaders
}
```

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 4. Programación básica del cauce gráfico

Subsección 4.2.

Estructura de los *shaders*. Ejemplos..

Creación y uso de objetos programa

Necesitamos como mínimo, un texto fuente del **vertex shader** y otro del **fragment shader**:

- ▶ Los fuentes de los dos shaders deben estar almacenados en memoria en variables de tipo **char *** (vectores de caracteres o cadenas, acabados en 0).
- ▶ Es conveniente guardarlos en archivos en el sistema de archivos (extensión **.glsl**) y leerlos al inicio del programa.
- ▶ Los dos shaders deben compilarse usando llamadas a OpenGL (puede haber errores al compilar).
- ▶ Una vez compilados correctamente, los dos shaders se enlazan, creándose un **objeto programa** (*program object*).
- ▶ Cada objeto programa tiene asociado un identificador (o nombre), es un entero positivo único.

Elementos del fuente de los shaders: declaraciones

El código fuente de una shader tiene **declaraciones** de:

- ▶ Parámetros **uniform**: valores proporcionados por la aplicación (son constantes para cada secuencia de vértices).
- ▶ Variables **varying**: valores calculados el vertex shader en cada vértice, y legibles (interpolados) por el fragment shader en cada pixel. Se declaran con **out** (en el vertex shader) o con **in** (en el fragment shader).
- ▶ Atributos de vértices: son variables de entrada en el vertex shader. Se declaran con **in** (también con **layout(location = *i*) in**, donde *i* es el índice del atributo).
- ▶ Función **main**: es la única función obligatoria.
- ▶ Funciones auxiliares: llamadas directa o indirectamente desde **main**.

Entradas y salidas según el tipo de shader.

Vertex Shaders (se ejecutan una vez por vértice)

- ▶ Entradas:
 - ▶ Parámetros **uniform**
 - ▶ Atributos del vértice (**layout .. in**): posición, color, etc..
- ▶ Salidas:
 - ▶ Variables **varying (out)**: para ser interpoladas.
 - ▶ Variable **gl_Position**: coordenadas transformadas del vértice.

Fragment Shaders (se ejecutan una vez por pixel)

- ▶ Entradas:
 - ▶ Parámetros **uniform**.
 - ▶ Variables **varying (in)**: ya interpoladas en el pixel.
- ▶ Salida:
 - ▶ Variable (**layout ... out**) con color del pixel.

Ejemplo de *shaders* mínimos: declaraciones

En el repositorio de github ([opengl3-minimo](#)) hay un ejemplo de shaders mínimos, válidos para visualizar polígonos rellenos o polilíneas. Se declaran en el propio programa C++, así:

```
// declaración de la cadena con el texto fuente del vertex shader
const char * fuente_vs = R"glsl(
    .....
)glsl";

// declaración de la cadena con el texto fuente del fragment shader
const char * fuente_fs = R"glsl(
    .....
)glsl";
```

- ▶ La funcionalidad es mínima: solo procesan la posición y el color.
- ▶ Se declaran dos variables (de tipo **char ***): **fuente_vs** y **fuente_fs**.
- ▶ Tener el fuente GLSL dentro del fuente C++ es sencillo pero puede ser difícil de encontrar los errores de compilación.

Ejemplo de *shaders* mínimos: Vertex Shader

```
#version 330 core

// Parámetros uniform
uniform mat4 u_mat_modelview;    // matriz de transformación de posiciones
uniform mat4 u_mat_proyeccion;   // matriz de proyección
uniform bool u_usar_color_plano; // 1 -> usar color plano, 0 -> usar interpolado

// Atributos de vértice
layout( location = 0 ) in vec3 atrib_posicion ; // atributo 0: posición
layout( location = 1 ) in vec3 atrib_color ;      // atributo 1: color RGB

// Variables 'varying'
out      vec3 var_color        ; // color RGB del vértice
flat out vec3 var_color_plano ; // idem (no se interpola)

// Función principal
void main()
{
    var_color      = atrib_color ;
    var_color_plano = atrib_color ;
    gl_Position    = u_mat_proyeccion * u_mat_modelview *
                      vec4( atrib_posicion, 1);
}
```

Ejemplo de *shaders* mínimos: Fragment Shader

```
#version 330 core

// Parámetros uniform
uniform bool u_usar_color_plano; // 1 -> usar color plano, 0 -> usar interpolado

// Variables 'varying'
in      vec3 var_color ;          // color interpolado en el pixel.
flat in vec3 var_color_plano ;   // color (plano) producido por el 'provoking vertex'

// Salida (color del pixel)
layout( location = 0 ) out vec4 out_color_fragmento ;

// Función principal
void main()
{
    if ( u_usar_color_plano )
        out_color_fragmento = vec4( var_color_plano, 1 );
    else
        out_color_fragmento = vec4( var_color, 1 );
}
```

Shaders más complejos

El código de las prácticas incluye shaders más complejos, que están diseñados para visualización 2D o 3D, incluyendo:

- ▶ Proyección perspectiva en 3D: usa una matriz de proyección que tiene en cuenta la perspectiva.
- ▶ Además de la matriz *modelview* para coordenadas, hay una versión de esa matriz para las normales.
- ▶ Incorporación de texturas: usa coordenadas de textura y consulta de la imagen de textura (función **texture** de GLSL).
- ▶ Simulación de iluminación en 3D: usa las normales, se debe evaluar un *modelo de iluminación local* (en la función **EvaluarMIL** y otras).

Los parámetros se configuran a través de la clase **Cauce**.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 4. Programación básica del cauce gráfico

Subsección 4.3.

Creación y ejecución de programas..

Identificación y activación de *shader programs*

- ▶ Cada **shader program** (o simplemente *programa*) se identifica en la aplicación con un valor entero (**GLuint**), que llamamos su **identificador**.
- ▶ El cauce de funcionalidad fija (si está disponible), se identifica con el identificador de programa 0
- ▶ Los programas creados por la aplicación tienen identificador mayor que cero.
- ▶ La función **glUseProgram** permite usar el identificador de un programa para activarlo (a partir de la llamada se usará el programa designado, el cual puede ser el del cauce de funcionalidad fija, si se usa un cero).

Funciones para compilar y enlazar shaders

Para usar un shader program en una aplicación, es necesario compilar sus dos shaders (el vertex y el fragment shader) por separado, y enlazar el programa completo, todo ello desde la propia aplicación (en *tiempo de ejecución* de la misma):

- ▶ Crear un shader (**glCreateShader**).
- ▶ Asociar su código fuente a un shader (**glShaderSource**).
- ▶ Compilar un shader (**glCompileShader**).
- ▶ Crear un programa (**glCreateProgram**).
- ▶ Asociar sus dos shader a un programa (**glAttachShader**).
- ▶ Enlazar un programa (**glLinkProgram**).
- ▶ Ver log de errores al compilar o enlazar.

El código para compilar y enlazar los shaders puede formar parte de la inicialización de OpenGL.

Compilar un shader (*vertex* o *fragment* shader)

Este método compila un vertex o fragment shader, dado el nombre del archivo y el tipo. Si hay errores, informe y aborta.

```
GLuint CompilarShader( const std::string& nombreArchivo,
                        GLenum tipoShader )
{
    GLuint idShader ; // resultado: identificador de shader

    // crear shader nuevo, obtener identificador (tipo GLuint)
    idShader = glCreateShader( tipoShader );

    // leer archivo fuente de shader en memoria, asociar fuente al shader, liberar memoria:
    const GLchar * fuente = leerArchivo( nombreArchivo );
    glShaderSource( idShader, 1, &fuente, nullptr );
    delete [] fuente ;
    fuente = nullptr ;

    // compilar y comprobar errores (si hay aborta)
    glCompileShader( idShader );
    VerErroresCompilar( idShader );

    // no ha habido errores: devolver identificador
    return idShader ;
}
```

Crear un programa (1/2): compilar y enlazar

El constructor de **Cauce** crea un programa y almacena el identificador en **id_prog**:

```
Cause::Cause()
{
    // inicializar los nombres de los archivos fuente:
    frag_fn = "cause33-frag.glsl" ;
    vert_fn = "cause33-vert.glsl" ;

    // crear y compilar shaders, crear el programa, guardar idents.
    id_frag_shader = CompilarShader( frag_fn, GL_FRAGMENT_SHADER );
    id_vert_shader = CompilarShader( vert_fn, GL_VERTEX_SHADER );
    id_prog        = glCreateProgram();

    // asociar shaders al programa
    glAttachShader( id_prog, id_frag_shader );
    glAttachShader( id_prog, id_vert_shader );

    // enlazar programa y ver errores
    glLinkProgram( id_prog );
    VerErroresEnlazar( id_prog );
    .....
}
```

Crear un programa (2/2): inicialización de uniforms.

Al enlazar un programa, OpenGL asocia un identificador o **localización (location)** entero a cada parámetro uniform. Esas localizaciones se deben usar para fijar valores de dichos parámetros.

- ▶ Debemos obtener y almacenar las localizaciones (con **glGetUniformLocation**).
- ▶ Debemos de dar valores iniciales con **glUniform**.

```
....  
// obtener localizaciones de params. uniforms  
loc_eval_mil    = glGetUniformLocation( id_prog, "eval_mil" );  
loc_sombr_plano = glGetUniformLocation( id_prog, "sombr_plano" );  
loc_eval_text   = glGetUniformLocation( id_prog, "eval_text" );  
.....  
// inicializar parámetros uniform a valores por defecto  
glUniform1i( loc_eval_mil, 0 );  
glUniform1i( loc_sombr_plano, 0 );  
glUniform1i( loc_eval_text, 0 );  
.....  
} // fin del constructor
```

Inicialización del cauce programable.

En la función de inicialización de OpenGL es necesario:

- ▶ Inicializar los punteros a funciones OpenGL de la versión 2.0 o posteriores (en este ejemplo lo hacemos con la librería GLEW)
- ▶ Invocar la creación, compilación y enlazado de shaders a usar

```
#include <GL/glew.h> // (innecesario en macOS)

void Inicializa_OpenGL()
{
    Inicializar_GLEW();
    // hacer el resto de inicializaciones (igual que antes)
    .....
    // compilar shaders, crear 'cauce'
    Cauce * cauce = new Cauce();
}
```

Según el entorno hardware/software, uno de los dos cauces podría no estar disponible.

Uso de un programa.

Lo usual es que durante la inicialización de OpenGL

- ▶ se creen los programas que se vayan a usar por la aplicación.

Para usar un programa ya durante la visualización de un cuadro, debemos de:

1. Activar el programa con **glUseProgram**. Se usa como parámetro el identificador de programa.
2. Fijar los valores de los parámetros *uniform* usando la familia de funciones **glUniform** (hay una versión por cada tipo de datos del correspondiente parámetro uniform).
3. Enviar las secuencias de vértices, lo cual provoca llamadas a los shaders en la GPU.

Lo usual es que todo esto se encapsule en clases que permitan portabilidad y sencillez. Nosotros usamos la clase **Cauce**, ya citada.

Uso de la clase Cauce y derivadas

Una vez creado un programa, para usarlo debemos activarlo, para ello usamos el método **activar**:

```
void Cauce::activar() { glUseProgram( id_prog ); }
```

Para fijar los parámetros, usamos métodos específicos que dan valor a los parámetros uniform. Por tanto, el programa puede quedar así:

```
void VisualizarEscena()
{
    // 'cauce' contiene un puntero al cauce actual
    cauce->activar();
    cauce->fijarParametro1( .... ); // (ejemplo)
    cauce->fijarParametro2( .... ); // (ejemplo)

    // enviar secuencias de vértices
    ....
}
```

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 4. Programación básica del cauce gráfico

Subsección 4.4.

Funciones auxiliares..

Verificar errores de compilación

Esta función verifica si hay errores al compilar, si es así escribe el log de errores y aborta:

```
void VerErroresCompilar( GLuint idShader )
{
    using namespace std ;
    const GLsizei maxt = 1024L*10L ;
    GLsizei tam ;
    GLchar buffer[maxt] ;
    GLint ok ;

    glGetShaderiv( idShader, GL_COMPILE_STATUS, &ok ); // ver si hay errores
    if ( ok == GL_TRUE ) // si la compilación ha sido correcta:
        return ;           // no hacer nada

    glGetShaderInfoLog( idShader, maxt, &tam, buffer); // leer log de errores
    cout << "error al compilar:" << endl
        << buffer << flush
        << "programa abortado" << endl << flush ;
    exit(1); // abortar
}
```

Verificar errores de enlazado

Esta función verifica si hay errores al compilar, si es así escribe el log de errores y aborta:

```
void VerErroresEnlazar( GLuint idProg )
{
    using namespace std ;
    const GLsizei maxt = 1024L*10L ;
    GLsizei tam ;
    GLchar buffer[maxt] ;
    GLint ok ;

    glGetProgramiv( idProg, GL_LINK_STATUS, &ok ); // ver si hay errores
    if ( ok == GL_TRUE ) // si el enlazado ha sido correcto:
        return ; // no hacer nada

    glGetProgramInfoLog( idProg, maxt, &tam, buffer); // leer log de errores
    cout << "error al enlazar:" << endl
        << buffer << flush
        << "programa abortado" << endl << flush ;
    exit(1); // abortar
}
```

Lectura de un archivo

Finalmente, para leer un archivo, se puede usar esta función:

```
char * LeerArchivo( const char * nombreArchivo )
{
    // intentar abrir stream, si no se puede informar y abortar
    ifstream file( nombreArchivo, ios::in|ios::binary|ios::ate );
    if ( ! file.is_open() )
    {   std::cout << "imposible abrir archivo para lectura ("
        << nombreArchivo << ")" << std::endl ;
        exit(1);
    }
    // reservar memoria para guardar archivo completo
    size_t numBytes = file.tellg();           // leer tamaño total en bytes
    char * bytes    = new char [numBytes+1]; // reservar memoria dinámica

    // leer bytes:
    file.seekg( 0, ios::beg );    // posicionar lectura al inicio
    file.read( bytes, numBytes ); // leer el archivo completo
    file.close();                // cerrar stream de lectura
    bytes[numBytes] = 0 ;         // añadir cero al final

    // devolver puntero al primer elemento
    return bytes ;
}
```

Inicialización de GLEW

En Linux y Windows es necesario leer punteros a las funciones de OpenGL nuevas de la versión 2.0 o posteriores. Para eso usamos la librería GLEW.

```
void Inicializa_GLEW()
{
#ifndef __APPLE__
    // hacer init de GLEW y comprobar errores
    GLenum codigoError = glewInit();
    if ( codigoError != GLEW_OK )
    {
        std::cout << "Imposible inicializar 'GLEW', mensaje: "
              << glewGetString(codigoError) << std::endl ;
        exit(1);
    }
#endif
}
```

(en macOS la función está vacía y no hace nada, es innecesario)

Sección 5.

Apéndice: puntos, vectores, marcos, coordenadas y matrices..

- 5.1. Puntos y vectores
- 5.2. Marcos de referencia y coordenadas
- 5.3. Coordenadas homogéneas
- 5.4. Operaciones entre vectores: producto escalar y vectorial
- 5.5. Transformaciones geométricas y afines
- 5.6. Matrices de transformación
- 5.7. Representación y operaciones con tuplas y matrices
- 5.8. Representación y operaciones sobre matrices.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.1.

Puntos y vectores.

Puntos y vectores

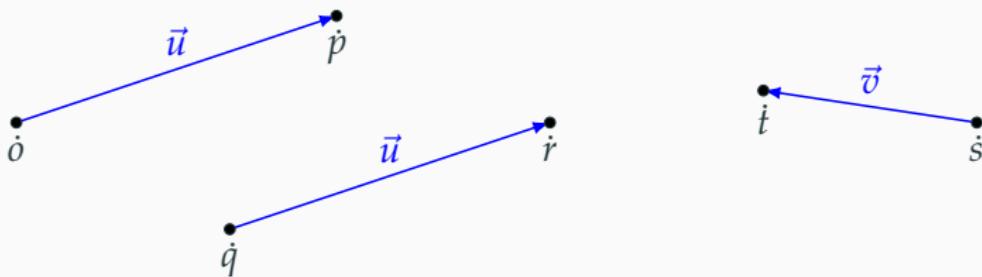
Los modelos 3D y 2D de objetos y figuras que vamos a representar se pueden construir cada uno de ellos en base a una conjunto abstracto (con estructura de **espacio afín**), cuyos elementos son puntos de un determinado espacio donde imaginamos el modelo. Cada uno de estos **puntos o localizaciones** los notaremos con un punto: \dot{p}, \dot{q}, \dots



- ▶ Cada modelo 2D o 3D tiene asociado su propio espacio de puntos.
- ▶ Como veremos, los modelos que vamos a visualizar y almacenar en memoria se basan en conjuntos finitos de vértices, y cada uno de ellos se asocia a uno de estos puntos.

Vectores

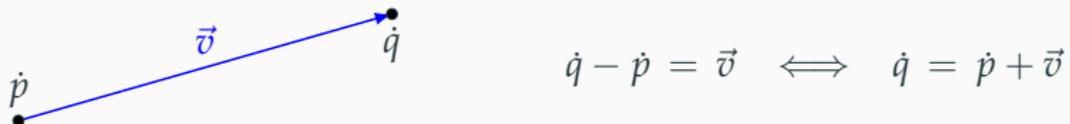
Además del conjunto de puntos, cada modelo tiene asociado un conjunto o espacio de vectores. Cada par de puntos del espacio tiene asociado un **vector** (o **vector libre**), los representamos con flechas \vec{u}, \vec{v}, \dots)



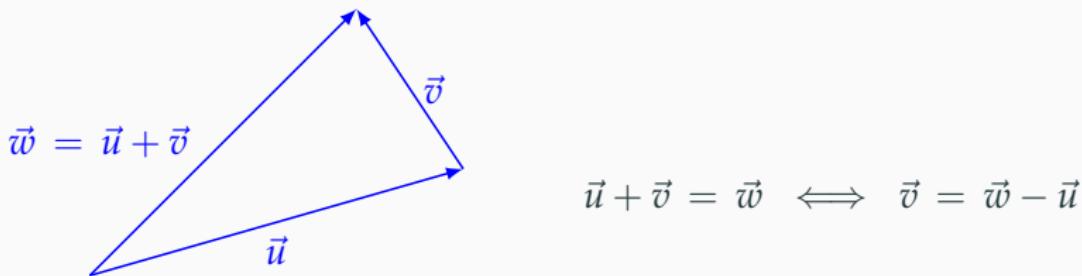
- ▶ Cada vector va asociado a la distancia y la dirección entre un punto y otro. El vector de \dot{p} a \dot{q} se escribe como $\dot{q} - \dot{p}$.
- ▶ Dos pares distintos de puntos pueden tener asociado el mismo vector (los pares \dot{o}, \dot{p} y \dot{q}, \dot{r} tienen ambos asociado el vector \vec{u}).
- ▶ En un espacio afín, los vectores forman un **espacio vectorial** asociado a dicho espacio afín.

Resta de puntos, suma de vectores

La diferencia de dos puntos produce el vector asociado a ambos. Por tanto, un punto cualquiera más un vector cualquiera produce otro punto.



Dos vectores \vec{u} y \vec{v} se pueden sumar entre si, produciendo otro vector $\vec{w} = \vec{u} + \vec{v}$, de forma que $\forall \dot{p}$, se cumple: $\dot{p} + \vec{w} = (\dot{p} + \vec{u}) + \vec{v}$.



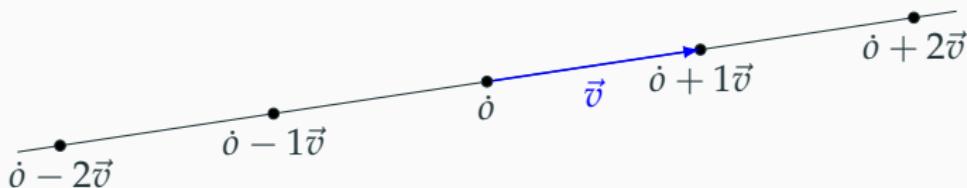
El **vector nulo** lo notamos como $\vec{0}$, y se define como $\vec{0} = \dot{p} - \dot{p}$ (para cualquier punto \dot{p}).

Producto de vectores y valores escalares

Un vector \vec{u} se puede multiplicar por un valor real s , produciendo otro vector $\vec{v} = s\vec{u}$, en la misma dirección de \vec{u} , pero de distinta longitud (cuando $s \neq 1$).



Como consecuencia, todos los puntos de la forma $\dot{o} + t\vec{v}$ (para todos los valores reales posibles de t) están en la recta que pasa por \dot{o} y es paralela a \vec{v}



Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

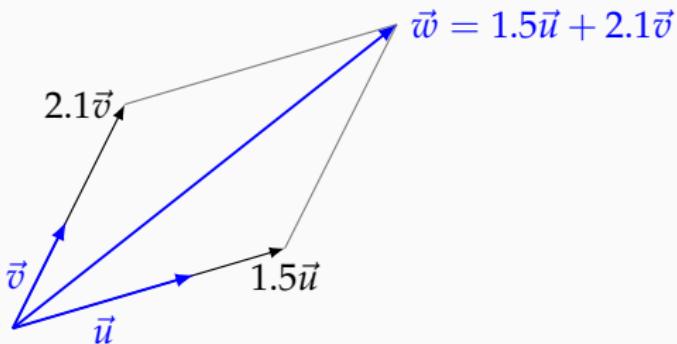
Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.2.

Marcos de referencia y coordenadas.

Bases de vectores

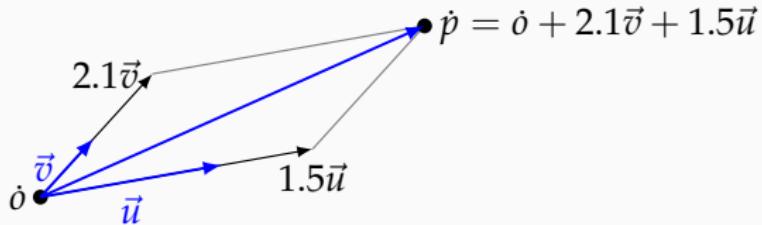
Usando dos vectores cualesquiera \vec{u} y \vec{v} del plano (no paralelos ni nulos), podemos escribir cualquier otro vector \vec{w} como una combinación lineal de ellos:



- ▶ El par de vectores $\{\vec{u}, \vec{v}\}$ forman una **base** de los vectores en 2D.
- ▶ Si $\vec{w} = a\vec{u} + b\vec{v}$, entonces al par de valores (a, b) se le llama **coordenadas** de \vec{w} respecto de la base $\{\vec{u}, \vec{v}\}$.
- ▶ El conjunto de vectores forma un **espacio vectorial** (en 3D, una base debe contener tres vectores).

Marcos de referencia y coordenadas

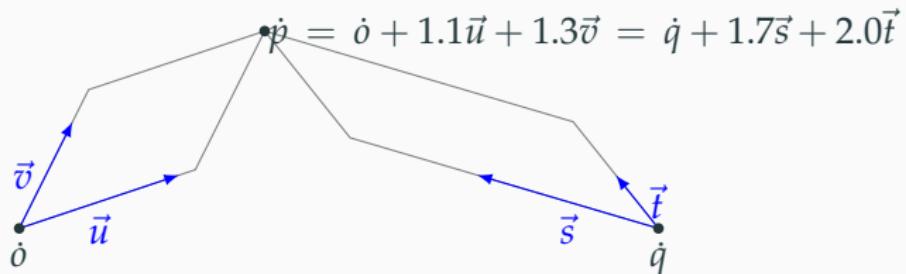
Si fijamos un punto \dot{o} (origen) y una base $\{\vec{u}, \vec{v}\}$, cualquier punto \dot{p} del plano se puede escribir como $\dot{p} = \dot{o} + a\vec{u} + b\vec{v}$:



- ▶ La terna $\mathcal{R} = [\vec{u}, \vec{v}, \dot{o}]$ forma un **marco de referencia** (*reference frame*) del plano 2D.
- ▶ Un marco sirve para **identificar puntos y vectores usando distancias (valores reales)**.
- ▶ Al par (a, b) se le llaman las **coordenadas** del punto \dot{p} en el marco de referencia \mathcal{R} .

Coordenadas y puntos

Un mismo punto (o un mismo vector) pueden tener distintas coordenadas en distintos marcos de referencia:



En general, un punto \dot{p} (o un vector \vec{v}) se puede identificar con sus coordenadas (usaremos el símbolo \equiv), es decir,

- \dot{p} tiene como coordenadas $(1.1, 1.3)$ en el marco $\mathcal{R} = [\vec{u}, \vec{v}, \dot{o}]$
- \dot{p} tiene como coordenadas $(1.7, 2.0)$ en el marco $\mathcal{S} = [\vec{s}, \vec{t}, \dot{q}]$

Unas coordenadas **no tienen significado** fuera del contexto de algún marco de referencia.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.3.

Coordenadas homogéneas.

Coordenadas homogéneas

En Informática Gráfica, la representación en memoria de las coordenadas de puntos y los vectores se hace usando las llamadas **coordenadas homogéneas** (su uso simplifica muchísimo los cálculos que se hacen con las coordenadas durante el cauce gráfico):

- ▶ A las tuplas de coordenadas se le añade una nueva componente (un valor real adicional), que se suele notar como w . Para los **puntos** siempre se hace $w = 1$. Para los **vectores**, siempre se hace $w = 0$.
- ▶ Por tanto, en 2D las tuplas tendrán tres componentes: (x, y, w) , y en 3D tendrán cuatro: (x, y, z, w) .
- ▶ La suma de punto y vector y la resta de dos vectores (usando coordenadas) se pueden seguir haciendo igual (ya que en w se hace: $1 + 0 = 1$ y $1 - 1 = 0$)
- ▶ El producto vectorial se hace ignorando la componente w .

Notación para tuplas de coordenadas

Usaremos vectores columna para escribir las coordenadas homogéneas de un punto o de un vector, es decir, las escribiremos en vertical, o bien en horizontal pero con el símbolo t para denotar transposición:

$$\mathbf{c} = (x, y, z, w)^t = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

nótese que hemos usado el símbolo en negrita **c** para denotar una tupla de coordenadas. Usaremos este tipo de símbolos (**a**, **b**, **c**, ...) para las tuplas de coordenadas homogéneas.

El uso de tuplas de coordenadas para puntos y vectores permite realizar en un programa operaciones con los mismos.

Coordenadas homogéneas de puntos

En un marco de referencia cualquiera $\mathcal{R} = [\vec{u}, \vec{v}, \vec{w}, \dot{o}]$, una tupla de coordenadas homogéneas $\mathbf{c} = (c_0, c_1, c_2, 1)^t$ representa un punto \dot{p} definido como:

$$\dot{p} = 1\dot{o} + c_0\vec{u} + c_1\vec{v} + c_2\vec{w}$$

(aquí hemos definido $1\dot{o} = \dot{o}$ (el mismo punto)). Con esto, la igualdad anterior se puede expresar de forma matricial:

$$\dot{p} = c_0\vec{u} + c_1\vec{v} + c_2\vec{w} + 1\dot{o} = [\vec{u}, \vec{v}, \vec{w}, \dot{o}] \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ 1 \end{pmatrix} = \mathcal{R} \mathbf{c}$$

de forma que se pueden relacionar explicitamente los puntos con sus coordenadas homogéneas, usando algún marco de referencia:

$$\dot{p} = \mathcal{R} \mathbf{c}$$

Coordenadas homogéneas de vectores

Lo anterior se puede aplicar a los vectores. En un marco de referencia cualquiera $\mathcal{R} = [\vec{u}, \vec{v}, \vec{w}, \dot{o}]$, una tupla de coordenadas homogéneas $\mathbf{d} = (d_0, d_1, d_2, 0)^t$ representa un vector \vec{s} definido como:

$$\vec{s} = d_0 \vec{u} + d_1 \vec{v} + d_2 \vec{w} + 0 \dot{o}$$

(aquí hemos definido $0\dot{o} = \vec{0}$ (el vector nulo)). Con esto, la igualdad anterior se puede expresar de forma matricial:

$$\vec{s} = 0\dot{o} + d_0 \vec{u} + d_1 \vec{v} + d_2 \vec{w} = [\vec{u}, \vec{v}, \vec{w}, \dot{o}] \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ 0 \end{pmatrix} = \mathcal{R} \mathbf{d}$$

la notación, por tanto, también permite relacionar los vectores con sus coordenadas en un marco (en este caso \vec{s} con \mathbf{d} en el marco \mathcal{R})

$$\vec{s} = \mathcal{R} \mathbf{d}$$

Operaciones usando coordenadas

Interpretar unas coordenadas en un marco es una operación lineal, ya que para cualquier tuplas \mathbf{u}, \mathbf{v} (con $w = 0$) y \mathbf{p}, \mathbf{q} (con $w = 1$), se cumple

$$\begin{aligned}\mathcal{R}(\mathbf{p} + \mathbf{u}) &= \mathcal{R}\mathbf{p} + \mathcal{R}\mathbf{u} & \mathcal{R}(a\mathbf{u} + b\mathbf{v}) &= a\mathcal{R}\mathbf{u} + b\mathcal{R}\mathbf{v} \\ \mathcal{R}(\mathbf{p} - \mathbf{q}) &= \mathcal{R}\mathbf{p} - \mathcal{R}\mathbf{q}\end{aligned}$$

En el contexto de un marco de referencia \mathcal{R} , el cálculo por un programa de operaciones entre vectores y puntos se puede realizar, por tanto, fácilmente usando sus coordenadas:

$$\begin{aligned}\mathcal{R}((u_0, u_1, u_2, 0)^t + (v_0, v_1, v_2, 0)^t) &= \mathcal{R}(u_0 + v_0, u_1 + v_1, u_2 + v_2, 0)^t \\ \mathcal{R}((p_0, p_1, p_2, 1)^t - (q_0, q_1, q_2, 1)^t) &= \mathcal{R}(p_0 - q_0, p_1 - q_1, p_2 - q_2, 0)^t \\ \mathcal{R}((p_0, p_1, p_2, 1)^t + (v_0, v_1, v_2, 0)^t) &= \mathcal{R}(p_0 + v_0, p_1 + v_1, p_2 + v_2, 1)^t \\ \mathcal{R}(a(u_0, u_1, u_2, 0)^t) &= \mathcal{R}(au_0, au_1, au_2, 0)^t\end{aligned}$$

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.4.

Operaciones entre vectores: producto escalar y vectorial.

El marco de referencia especial

En todo espacio de puntos o vectores (2D o 3D) que consideremos habrá un **marco de referencia especial** $\mathcal{E} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$, en ese marco por definición:

- ▶ los vectores \vec{x} , \vec{y} y \vec{z} tienen longitud unidad: por tanto estos vectores determinarán la longitud de todos los demás, es decir: definen la unidad de longitud en el espacio de coordenadas.
- ▶ los vectores \vec{x} , \vec{y} y \vec{z} son perpendiculares entre ellos dos a dos: por tanto, esos vectores forman ángulos de 90 grados, y determinan los angulos entre cualquiera dos vectores.
- ▶ A los vectores de longitud unidad los llamaremos **versores** o **vectores unitarios**. Se suelen escribir con un sombrero sobre ellos, por tanto el marco especial lo escribiremos como $\mathcal{E} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$

Más adelante se define formalmente el ángulo y la distancia de vectores.

Producto escalar y módulo de vectores en 2D o 3D

El **producto escalar** o **producto interno** (*inner product* o *dot product*) es una función que se aplica a dos vectores \vec{u} y \vec{v} y produce un valor real, que se nota como $\vec{u} \cdot \vec{v}$. Cumple estas dos propiedades:

- ▶ Conmutativa: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$
- ▶ Linealidad: $\vec{u} \cdot (a\vec{v} + b\vec{w}) = a(\vec{u} \cdot \vec{v}) + b(\vec{u} \cdot \vec{w}) \quad (\forall a, b \in \mathbb{R})$

Muchas funciones que cumplen estas condiciones. Para concretar a cual de ellas no referimos, usamos el marco especial (en). Se cumple:

- ▶ En 3D, el marco especial es $\mathcal{E} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$, se cumple:

$$\hat{x} \cdot \hat{x} = \hat{y} \cdot \hat{y} = \hat{z} \cdot \hat{z} = 1 \quad y \quad \hat{x} \cdot \hat{y} = \hat{y} \cdot \hat{z} = \hat{z} \cdot \hat{x} = 0$$

- ▶ En 2D, el marco especial es $\mathcal{E} = [\hat{x}, \hat{y}, \dot{o}]$, se cumple:

$$\hat{x} \cdot \hat{x} = \hat{y} \cdot \hat{y} = 1 \quad y \quad \hat{x} \cdot \hat{y} = 0$$

Longitud y perpendicularidad de vectores

El **módulo** (o norma) de un vector cualquiera \vec{u} se nota con $\|\vec{u}\|$, y es un valor real no negativo que se define como:

$$\|\vec{u}\| = \sqrt{\vec{u} \cdot \vec{u}}$$

Es fácil demostrar que se cumple

$$\|a\vec{u}\| = \|a\| \|\vec{u}\|$$

El módulo de un vector coincide con su **longitud**:

- ▶ Decimos que dos vectores \vec{u} y \vec{v} de longitud no nula son **perpendiculares** cuando $\vec{u} \cdot \vec{v} = 0$.
- ▶ Esto implica que al designar cual es el marco especial \mathcal{E} de un espacio afín estamos definiendo la unidad de longitud y la noción de perpendicularidad.

Producto vectorial o externo en 3D

El **producto vectorial o producto externo** (*cross product o vector product*) es una función que se aplica a dos vectores \vec{u} y \vec{v} (en 3D) y produce un tercer vector (perpendicular a \vec{u} y \vec{v}), que se nota como $\vec{u} \times \vec{v}$.

- ▶ Anticonmutativa: $\vec{u} \times \vec{v} = -\vec{v} \times \vec{u}$
- ▶ Linealidad: $\vec{u} \times (a\vec{v} + b\vec{w}) = a(\vec{u} \times \vec{v}) + b(\vec{u} \times \vec{w}) \quad (\forall a, b \in \mathbb{R})$

Puesto que muchas funciones distintas pueden cumplir estos axiomas, para definir bien el producto vectorial se establece además que en el marco especial $\mathcal{E} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$ se deben cumplir estas propiedades:

$$\hat{x} \times \hat{y} = \hat{z} \quad \hat{y} \times \hat{z} = \hat{x} \quad \hat{z} \times \hat{x} = \hat{y}$$

Marcos cartesianos en 2D

Sea $\mathcal{C} = [\vec{e}_x, \vec{e}_y, \dot{q}]$ un marco de referencia 2D tal que:

- ▶ Sus dos vectores *tienen longitud unidad*, es decir:

$$\vec{e}_x \cdot \vec{e}_x = \vec{e}_y \cdot \vec{e}_y = 1$$

- ▶ Sus dos vectores son *perpendiculares entre si*, es decir:

$$\vec{e}_x \cdot \vec{e}_y = 0$$

- ▶ La **orientación** es semejante a la de $\mathcal{E} = [\vec{x}, \vec{y}, \dot{o}]$, es decir:

$$\vec{e}_x \cdot \vec{x} = \vec{e}_y \cdot \vec{y}$$

En estas condiciones, decimos que \mathcal{C} es un marco de referencia **cartesiano** en 2D, y escribimos $\mathcal{E} = [\hat{e}_x, \hat{e}_y, \dot{o}]$ (el marco de referencia \mathcal{E} es cartesiano por definición).

Marcos cartesianos en 3D

Sea $\mathcal{C} = [\vec{e}_x, \vec{e}_y, \vec{e}_z, \dot{q}]$ un marco de referencia 3D cualquiera, tal que:

- ▶ Sus vectores *tienen longitud unidad*, es decir:

$$\vec{e}_x \cdot \vec{e}_x = \vec{e}_y \cdot \vec{e}_y = \vec{e}_z \cdot \vec{e}_z = 1$$

- ▶ Sus vectores son *perpendiculares dos a dos*, es decir:

$$\vec{e}_x \cdot \vec{e}_y = \vec{e}_y \cdot \vec{e}_z = \vec{e}_z \cdot \vec{e}_x = 0$$

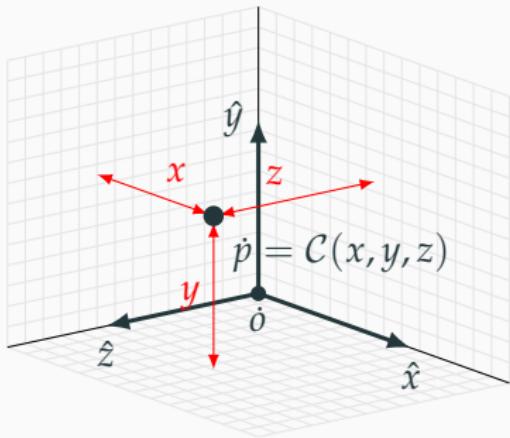
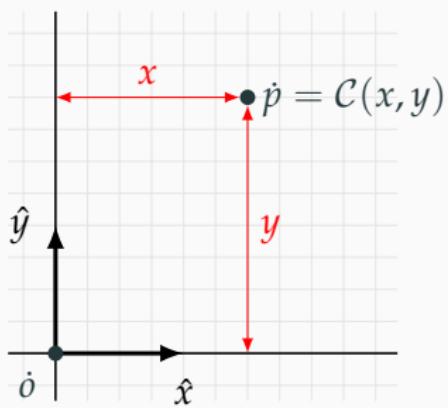
- ▶ La **orientación** es semejante a la de \mathcal{E} , es decir:

$$\vec{e}_x \times \vec{e}_y = \vec{e}_z \quad \vec{e}_y \times \vec{e}_z = \vec{e}_x \quad \vec{e}_z \times \vec{e}_x = \vec{e}_y$$

En estas condiciones, decimos que \mathcal{C} es un marco de referencia **cartesiano** en 3D, y escribimos $\mathcal{E} = [\hat{e}_x, \hat{e}_y, \hat{e}_z, \dot{o}]$.

Marcos y coordenadas cartesianas

En un marco cartesiano $\mathcal{C} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$, los versores son paralelos a tres líneas (que pasan por el origen, \dot{o}) que se suelen llamar **ejes de coordenadas**. A las coordenadas se les denomina **coordenadas cartesianas**



Las coordenadas cartesianas se pueden interpretar como distancias, medidas perpendicularmente a los planos definidos por dos versores (en 3D), o perpendicularmente al otro versor (en 2D).

Marcos ortogonales y ortonormales. Orientación.

- ▶ Un marco de referencia cuyos vectores son perpendiculares entre sí, pero no tienen necesariamente longitud unidad es un marco **ortogonal**
- ▶ Un marco ortogonal cuyos ejes son de longitud unidad es un marco **ortonormal**
- ▶ Un marco ortonormal $[\hat{e}_x, \hat{e}_y, \hat{e}_z, \dot{p}]$ puede tener la misma **orientación** que el marco $\mathcal{E} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$ u otra distinta (solo hay dos posibles orientaciones):
 - ▶ En 2D, los valores $\hat{e}_x \cdot \hat{x}$ y $\hat{e}_y \cdot \hat{y}$ pueden coincidir o bien pueden ser uno igual al otro negado. En \mathcal{E} coinciden.
 - ▶ En 3D, el vector $\hat{e}_x \times \hat{e}_y$ puede ser igual a \hat{z} o bien a $-\hat{z}$. En \mathcal{E} es \hat{z} .
- ▶ Un marco **ortonormal** cuya orientación coincide con la de \mathcal{E} es un **marco cartesiano**.

Calculo del producto escalar y el módulo

Se puede calcular fácilmente el producto escalar y el módulo de vectores usando sus coordenadas relativas a un marco cartesiano \mathcal{C} . Sean dos vectores $\vec{a} = \mathcal{C}(a_x, a_y, a_z, 0)^t$ y $\vec{b} = \mathcal{C}(b_x, b_y, b_z, 0)^t$:

- ▶ El producto escalar es la suma de los productos componente a componente:

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$$

(este valor sería el mismo si usasemos las coordenadas de cualquier otro marco cartesiano distinto de \mathcal{C}).

- ▶ Como consecuencia, el módulo se puede obtener como:

$$\|\vec{a}\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

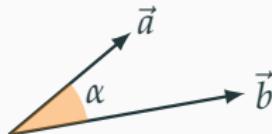
- ▶ El módulo de un vector coincide con su **longitud** en el espacio (ya que de los versores de \mathcal{E} dijimos que tenían longitud unidad por definición). El módulo, calculado así, es siempre el mismo en cualquier marco cartesiano.

Ángulo entre vectores

Dados dos vectores \vec{a} y \vec{b} (ninguno nulo)

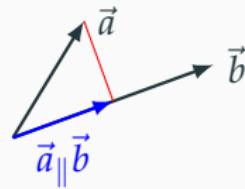
- El **ángulo α** entre \vec{a} y \vec{b} se define como el arco cuyo coseno es el producto escalar de $\vec{a}/\|\vec{a}\|$ y $\vec{b}/\|\vec{b}\|$. Se cumple:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos \alpha$$



- Si llamamos $\vec{a}_{\parallel} \vec{b}$ a la componente de \vec{a} paralela a \vec{b} , entonces:

$$\vec{a}_{\parallel} \vec{b} = \left(\frac{\vec{a} \cdot \vec{b}}{\vec{b} \cdot \vec{b}} \right) \vec{b}$$



- Dado un versor \hat{b} se cumple: $\vec{a}_{\parallel} \hat{b} = (\vec{a} \cdot \hat{b}) \hat{b}$
- Dados dos versores \hat{a} y \hat{b} se cumple: $\hat{a} \cdot \hat{b} = \cos \alpha$, donde α es el ángulo entre \hat{a} y \hat{b} .

Cálculo del producto vectorial

En un marco de referencia cartesiano cualquiera $\mathcal{C} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$, se pueden usar las coordenadas de dos vectores \vec{a} y \vec{b} para calcular las coordenadas del vector $\vec{a} \times \vec{b}$.

- ▶ A partir de los axiomas se puede demostrar que las coordenadas del producto vectorial se pueden obtener así:

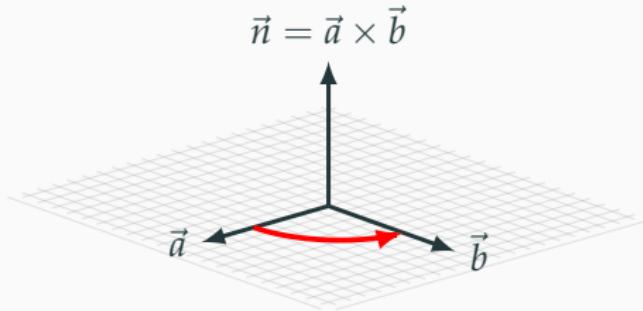
$$\mathcal{C} \begin{pmatrix} a_x \\ a_y \\ a_z \\ 0 \end{pmatrix} \times \mathcal{C} \begin{pmatrix} b_x \\ b_y \\ b_z \\ 0 \end{pmatrix} = \mathcal{C} \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \\ 0 \end{pmatrix}$$

- ▶ Esta propiedad se cumple siempre que \mathcal{C} sea cartesiano, ya que el producto vectorial es invariante entre marcos cartesianos.

Producto vectorial: dirección del vector

El producto vectorial constituye un método para obtener un vector perpendicular a otros dos vectores dados (no paralelos)

- El vector $\vec{a} \times \vec{b}$ es perpendicular al plano que forman \vec{a} y \vec{b} (y por lo tanto, perpendicular tanto a \vec{a} como a \vec{b})



La dirección de $\vec{n} = \vec{a} \times \vec{b}$ es la dirección en la que avanza un tornillo paralelo a \vec{n} cuando se gira desde \vec{a} hacia \vec{b} (si \mathcal{E} es a derechas)

Problemas: cálculo del producto escalar

Problema 1.8.

Demuestra que el producto escalar de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano como la suma del producto componente a componente, a partir de las propiedades que definen dicho producto escalar.

Problemas: cálculo del producto vectorial

Problema 1.9.

Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se indica en la transparencia anterior, a partir de las propiedades que definen dicho producto vectorial.

Problema 1.10.

Demuestra que el producto vectorial de dos vectores es perpendicular a cada uno de esos dos vectores.

Producto vectorial: longitud del vector

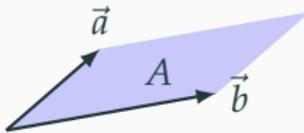
La longitud del vector obtenido como producto vectorial de otros dos está relacionada con el área entre esos otros dos vectores:

- ▶ La longitud de $\vec{a} \times \vec{b}$ es proporcional al seno del ángulo α entre \vec{a} y \vec{b}

$$\|\vec{a} \times \vec{b}\| = \|\vec{a}\| \|\vec{b}\| \sin \alpha$$

- ▶ Esa longitud es igual al área A del paralelepípedo formado por \vec{a} y \vec{b}

$$\|\vec{a} \times \vec{b}\| = A$$



- ▶ Por lo tanto dados dos versores \hat{a} y \hat{b} , se cumple:

$$\|\hat{a} \times \hat{b}\| = \sin \alpha$$

donde α es el ángulo entre \hat{a} y \hat{b} .

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.5.

Transformaciones geométricas y afines.

Transformación geométrica

Para la definición de modelos geométricos se usa el concepto de transformación geométrica

Una transformación geométrica T es una aplicación que asocia a cualquier punto \dot{p} de un espacio afín otro punto \dot{q} del mismo (u otro) espacio afín. Escribimos

$$\dot{q} = T(\dot{p})$$

decimos: \dot{q} es T aplicado a \dot{p} , o bien \dot{q} es la imagen de \dot{p} a través de T .

Las transformaciones geométricas se usan para diseñar modelos de objetos complejos en 3D.

Transformación de coordenadas

En un marco \mathcal{R} , una transformación T cambia las coordenadas de los puntos sobre los actua. Supongamos que $\dot{q} = T(\dot{p})$, entonces:

$$\dot{p} = \mathcal{R}(p_0, p_1, p_2, 1)^t \quad \text{se transforma en} \quad \dot{q} = \mathcal{R}(q_0, q_1, q_2, 1)^t$$

Para este marco \mathcal{R} , la transformación T viene determinada por tres funciones reales f_0 , f_1 y f_2 que producen las coordenadas del punto transformado en función de las originales:

$$\begin{aligned} q_0 &= f_0(p_0, p_1, p_2) \\ q_1 &= f_1(p_0, p_1, p_2) \\ q_2 &= f_2(p_0, p_1, p_2) \end{aligned}$$

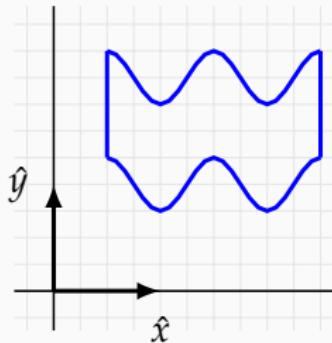
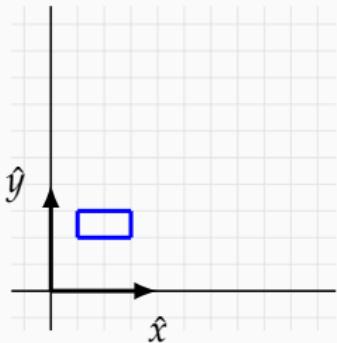
Lógicamente, para una única transformación T , las funciones f_0 , f_1 y f_2 dependen del sistema de referencia \mathcal{R} en uso.

Ejemplo de transformación en 2D

En un marco cartesiano $\mathcal{C} = \{\hat{x}, \hat{y}, \dot{o}\}$ en 2D, una transformación T podría ser la definida por estas expresiones:

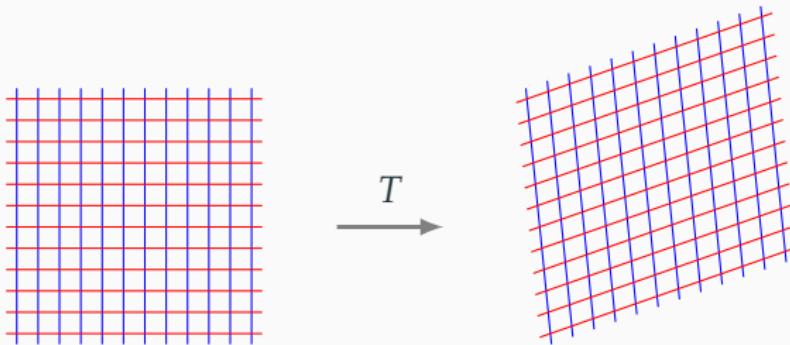
$$f_0(x, y) = 4x - 1 \quad f_1(x, y) = 2y + \frac{2 + \cos((8x - 2)\pi)}{4}$$

el efecto de T sobre los puntos de un polígono (el rectángulo azul) es el que se aprecia aquí:



Definición de transformación afín

Una **transformación afín** T es una transformación que conserva las líneas rectas, y aplica rectas paralelas en rectas paralelas (*conserva el paralelismo*). También se llaman **transformaciones lineales**:



Las transformaciones afines más comunes incluyen: traslaciones, rotaciones, escalados, reflexiones, cizallas y las combinaciones de estas.

Propiedades de las transformaciones afines

Una transformación afín T conserva el paralelismo, por tanto:

$$\dot{p} - \dot{q} = \dot{r} - \dot{s} \implies T(\dot{p}) - T(\dot{q}) = T(\dot{r}) - T(\dot{s})$$

Esto permite extender T a los vectores del espacio afín:

$$\vec{v} = \dot{p} - \dot{q} \implies T(\vec{v}) \equiv T(\dot{p}) - T(\dot{q})$$

Como consecuencia, podemos caracterizar las transformaciones afines:

Cualquier transformación será afín si y solo si se cumple, para cualquier punto \dot{p} , vectores \vec{u}, \vec{v} y reales a, b estas propiedades:

$$T(\dot{p} + \vec{u}) = T(\dot{p}) + T(\vec{u})$$

$$T(a\vec{u} + b\vec{v}) = aT(\vec{u}) + bT(\vec{v})$$

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

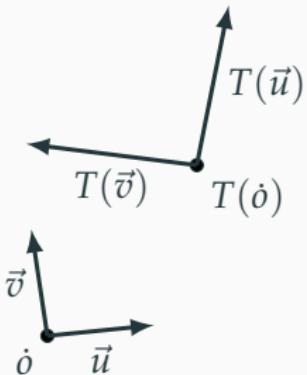
Subsección 5.6.
Matrices de transformación.

Transformación de marcos

Dado un marco de referencia $\mathcal{R} = [\vec{u}, \vec{v}, \vec{w}, \dot{o}]$ y una transf. afín T , definimos $T(\mathcal{R})$ como el marco \mathcal{R} transformado por T , es decir:

$$T(\mathcal{R}) = [T(\vec{u}), T(\vec{v}), T(\vec{w}), T(\dot{o})]$$

Consideramos las coordenadas de $T(\mathcal{R})$ en el marco \mathcal{R} (son 4 tuplas de valores reales: **a,b,c,d**)



$$\begin{aligned} T(\vec{u}) &= \mathcal{R}\mathbf{a} = \mathcal{R}(a_0, a_1, a_2, 0)^t \\ T(\vec{v}) &= \mathcal{R}\mathbf{b} = \mathcal{R}(b_0, b_1, b_2, 0)^t \\ T(\vec{w}) &= \mathcal{R}\mathbf{c} = \mathcal{R}(c_0, c_1, c_2, 0)^t \\ T(\dot{o}) &= \mathcal{R}\mathbf{d} = \mathcal{R}(d_0, d_1, d_2, 1)^t \end{aligned}$$

Transformación de coordenadas

Supongamos un punto $\mathcal{R}\mathbf{p} = \mathcal{R}(p_0, p_1, p_2, 1)^t$ y consideramos como se transforman sus coordenadas mediante T para obtener otro punto $\mathcal{R}\mathbf{q} = \mathcal{R}(q_0, q_1, q_2, 1)^t$:

$$\begin{aligned}\mathcal{R}\mathbf{q} = T(\mathcal{R}\mathbf{p}) &= T(p_0\vec{u} + p_1\vec{v} + p_2\vec{w} + \vec{o}) \\&= p_0T(\vec{u}) + p_1T(\vec{v}) + p_2T(\vec{w}) + T(\vec{o}) \\&= p_0\mathcal{R}\mathbf{a} + p_1\mathcal{R}\mathbf{b} + p_2\mathcal{R}\mathbf{c} + \mathcal{R}\mathbf{d} \\&= \mathcal{R}(p_0\mathbf{a} + p_1\mathbf{b} + p_2\mathbf{c} + \mathbf{1d})\end{aligned}$$

Luego se cumple $\mathcal{R}\mathbf{q} = \mathcal{R}(p_0\mathbf{a} + p_1\mathbf{b} + p_2\mathbf{c} + \mathbf{1d})$, lo cual implica que las coordenadas deben ser las mismas, es decir:

$$\mathbf{q} = p_0\mathbf{a} + p_1\mathbf{b} + p_2\mathbf{c} + \mathbf{1d}$$

Matriz de transformación de coordenadas (puntos)

Matricialmente:

$$\begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ 1 \end{pmatrix} = p_0 \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ 0 \end{pmatrix} + p_1 \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ 0 \end{pmatrix} + p_2 \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ 0 \end{pmatrix} + 1 \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ 1 \end{pmatrix}$$

o lo que es lo mismo:

$$\begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ 1 \end{pmatrix}$$

A la matriz 4x4 la llamamos M (en 2D es una matriz 3x3), vemos que esta matriz depende de \mathcal{R} y de T .

Matriz de transformación de coordenadas (vectores)

En el caso de un vector $\mathcal{R}(u_0, u_1, u_2, 0)^t$, al aplicarle la transformación afín T obtenemos otro vector $\mathcal{R}(v_0, v_1, v_2, 0)^t$.

- ▶ Aplicando un razonamiento similar al usado para los puntos, obtenemos una relación parecida entre las coordenadas de ambos vectores:

$$\begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ 0 \end{pmatrix}$$

- ▶ Se usa la misma matriz M , aunque el resultado es ahora independiente de la última columna de M , ya que las coordenadas de los vectores tienen w a 0 en lugar de a 1.

Matriz asociada a una transformación afín (3/3)

Es decir, para cada transformación afín T y marco de coordenadas \mathcal{R} existe una única matriz M tal que si $\mathcal{R}\mathbf{q} = T(\mathcal{R}\mathbf{p})$ entonces:

$$\mathbf{q} = M\mathbf{p}$$

Es decir: **toda transformación afín tiene asociada una matriz en cada marco de coordenadas.** Esa matriz determina como se transforman las coordenadas tanto de puntos como de vectores

- ▶ M permite obtener las coordenadas de los puntos transformados en términos de las coordenadas de los puntos originales (en ese marco)
- ▶ En 3D es una matriz 4×4 , mientras que en 2D será una matriz 3×3 .
- ▶ Permite implementar en un programa una transformación afín, especificando su matriz asociada.
- ▶ La última fila siempre es $0, 0, 0, 1$

Descomposición de una matriz

Multiplicar unas coordenadas por este tipo de matrices 4x4 es equivalente a multiplicar por una matriz 3x3 y aplicar una traslación después (la traslación no afecta a los vectores libres).

$$\begin{pmatrix} q_0 \\ q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} a_0 & b_0 & c_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} + \begin{pmatrix} d_0 \\ d_1 \\ d_2 \end{pmatrix}$$

o lo que es lo mismo, la matriz M se puede descomponer en una matriz R y una matriz de desplazamiento D :

$$\overbrace{\begin{pmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^M = \overbrace{\begin{pmatrix} 1 & 0 & 0 & d_0 \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 1 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^D \overbrace{\begin{pmatrix} a_0 & b_0 & c_0 & 0 \\ a_1 & b_1 & c_1 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^R$$

Ventajas del uso de coords. homogéneas

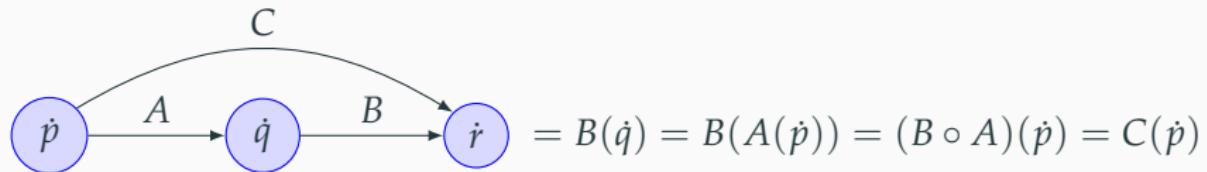
El uso de coordenadas homogéneas permite unificar la matriz R y la matriz D en una única matriz 4x4

- ▶ Simplifica los cálculos.
- ▶ Permite componer un número arbitrario de transformaciones en una única matriz.
- ▶ Los puntos y los vectores libres se tratan igual: en ambos casos hay que multiplicar una tupla por una matriz.
- ▶ Permite implementar eficiente la transformación de proyección (no lineal)

Composición e inversa

Una transformación C se puede obtener como **composición** de otras dos transformaciones A (primero) y B (después), escribimos

$$C = B \circ A:$$



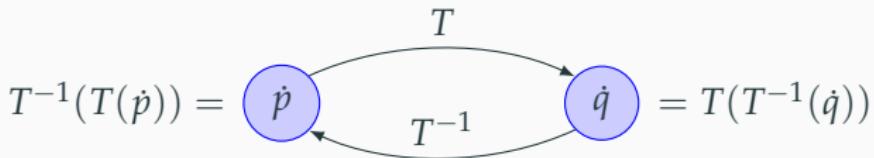
La composición es, en general, **no conmutativa**. Se puede extender a 3 o más transformaciones:

$$T_4(T_3(T_2(T_1(\dot{p})))) = (T_4 \circ T_3 \circ T_2 \circ T_1)(\dot{p})$$

la composición es **asociativa**

Transformación inversa

Una transformación biyectiva T siempre tiene una inversa T^{-1} . La transformación T^{-1} es la que *deshace* el efecto de T :



La composición de una transformación y su inversa (de las dos formas posibles) es la transformación identidad:

$$T^{-1} \circ T = T \circ T^{-1} = I$$

donde I es la transformación identidad.

Composición y producto de matrices.

Supongamos una secuencia T_1, T_2, \dots, T_n de n transformaciones afines. Consideraremos la transformación compuesta

$$U = T_n \circ T_{n-1} \circ \cdots \circ T_2 \circ T_1$$

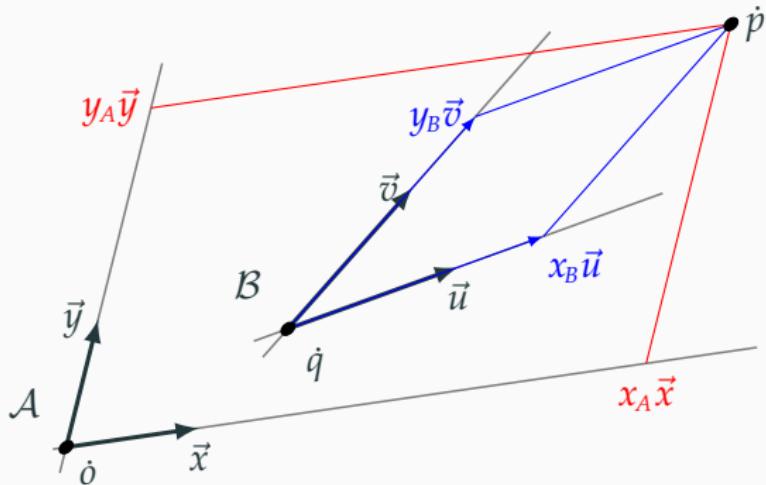
La matriz M_U asociada a U en un marco \mathcal{R} será el producto de las matrices M_i asociadas a T_i en ese marco: asociadas a cada una de las T_i :

$$M_U = M_n M_{n-1} \cdots M_2 M_1$$

(nótese que el producto de matrices es derecha a izquierda: a la izquierda aparecen las matrices que se aplican después). Esta propiedad es fundamental, pues **permite obtener matrices de transformaciones compuestas mediante multiplicación de matrices**.

Relación entre marcos arbitrarios

Suponemos dos marcos 2D cualesquiera $\mathcal{A} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$ y $\mathcal{B} = [\vec{u}, \vec{v}, \vec{w}, \dot{q}]$, y un punto $\dot{p} = \mathcal{B}(x_B, y_B, z_B, 1)^t = \mathcal{A}(x_A, y_A, z_A, 1)^t$



$$\dot{p} = \dot{o} + x_A \vec{x} + y_A \vec{y} + z_A \vec{z} = \dot{q} + x_B \vec{u} + y_B \vec{v} + z_B \vec{w}$$

Transformación de marcos de coordenadas

Supongamos que conocemos las coordenadas del marco \mathcal{B} en el marco \mathcal{A} (son los vectores columna $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$), entonces:

$$\begin{aligned}\vec{u} &= \mathcal{A}\mathbf{a} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}] (a_x, a_y, a_z, 0)^t = a_x \vec{x} + a_y \vec{y} + a_z \vec{z} + 0 \dot{o} \\ \vec{v} &= \mathcal{A}\mathbf{b} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}] (b_x, b_y, b_z, 0)^t = b_x \vec{x} + b_y \vec{y} + b_z \vec{z} + 0 \dot{o} \\ \vec{w} &= \mathcal{A}\mathbf{c} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}] (c_x, c_y, c_z, 0)^t = c_x \vec{x} + c_y \vec{y} + c_z \vec{z} + 0 \dot{o} \\ \dot{q} &= \mathcal{A}\mathbf{d} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}] (d_x, d_y, d_z, 1)^t = d_x \vec{x} + d_y \vec{y} + d_z \vec{z} + 1 \dot{o}\end{aligned}$$

esto se puede expresar matricialmente:

$$\mathcal{B} = [\vec{u}, \vec{v}, \vec{w}, \dot{q}] = [\vec{x}, \vec{y}, \vec{z}, \dot{o}] \begin{pmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathcal{A}M_{\mathcal{A}, \mathcal{B}}$$

Por tanto, podemos decir que: la matriz $4 \times 4 M_{\mathcal{A}, \mathcal{B}}$ transforma el sistema de referencia \mathcal{A} en el sistema de referencia \mathcal{B}

Descomposición de $M_{\mathcal{A},\mathcal{B}}$

La matriz $M_{\mathcal{A},\mathcal{B}}$ que acabamos de considerar se puede descomponer en el producto de una matriz $D_{\mathcal{A},\mathcal{B}}$ y otra matriz $R_{\mathcal{A},\mathcal{B}}$:

$$\overbrace{\begin{pmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{M_{\mathcal{A},\mathcal{B}}} = \overbrace{\begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{D_{\mathcal{A},\mathcal{B}}} \overbrace{\begin{pmatrix} a_x & b_x & c_x & 0 \\ a_y & b_y & c_y & 0 \\ a_z & b_z & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{R_{\mathcal{A},\mathcal{B}}}$$

- ▶ La matriz $R_{\mathcal{A},\mathcal{B}}$ no tiene términos de desplazamiento.
- ▶ La matriz $D_{\mathcal{A},\mathcal{B}}$ es la que produce un desplazamiento, de forma que el origen de \mathcal{A} (el punto \dot{o}) se lleva hasta el origen de \mathcal{B} (el punto \dot{q}), el vector de desplazamiento es

$$\dot{q} - \dot{o} = \mathcal{A}(d_x, d_y, d_z, 0)^t$$

Transformación de coordenadas

Consideramos el punto \dot{p} : sabemos que sus coordenadas resp. de \mathcal{A} son \mathbf{c}_A y respecto de \mathcal{B} son \mathbf{c}_B , es decir:

$$\dot{p} = \mathcal{A}\mathbf{c}_A = \mathcal{B}\mathbf{c}_B$$

puesto que $\mathcal{B} = \mathcal{A}M_{\mathcal{A},\mathcal{B}}$, podemos sustituir y reagrupar (por asociatividad) en la anterior igualdad:

$$\dot{p} = \mathcal{A}\mathbf{c}_A = \mathcal{B}\mathbf{c}_B = (\mathcal{A}M_{\mathcal{A},\mathcal{B}})\mathbf{c}_B = \mathcal{A}M_{\mathcal{A},\mathcal{B}}\mathbf{c}_B = \mathcal{A}(M_{\mathcal{A},\mathcal{B}}\mathbf{c}_B)$$

de donde se deduce que

$$\mathbf{c}_A = M_{\mathcal{A},\mathcal{B}} \mathbf{c}_B$$

es decir, la matriz $M_{\mathcal{A},\mathcal{B}}$ transforma coordenadas relativas a \mathcal{B} en coordenadas relativas a \mathcal{A}

Interpretación dual de las matrices

Todo lo anterior implica que dados un sistema de referencia cualquiera \mathcal{A} y una matriz cualquiera M , hay dos formas alternativas de interpretar que cosa es M :

- ▶ M es la matriz que convierte coordenadas de un punto \dot{p} en coordenadas de otro \dot{q} (ambas coordenadas relativas al mismo marco \mathcal{A}):

$$\dot{p} = \mathcal{A}\mathbf{c}_A \implies \dot{q} = \mathcal{A}(M\mathbf{c}_A)$$

- ▶ M es la matriz que transforma unas coordenadas \mathbf{c}_B relativas al marco $\mathcal{B} = \mathcal{A}M$ en otras coordenadas relativas al marco \mathcal{A} (ambas coordenadas del mismo punto \dot{p}):

$$\dot{p} = \mathcal{B}\mathbf{c}_B \implies \dot{p} = \mathcal{A}(M\mathbf{c}_B)$$

(igual se puede razonar acerca de vectores en lugar de puntos)

Transformación inversa. Descomposición.

Si se conocen las coordenadas \mathbf{c}_A y se quieren calcular las coordenadas relativas a \mathbf{c}_B , evidentemente debemos usar la matriz inversa, ya que :

$$\mathbf{c}_B = (M_{A,B})^{-1} \mathbf{c}_A$$

Lo mismo ocurre con los sistemas de referencia, es decir, podemos escribir:

$$\mathcal{A} = \mathcal{B} (M_{A,B})^{-1}$$

de cualquiera de estas dos igualdades se hace evidente que:

$$M_{A,B}^{-1} = M_{B,A}$$

Es decir, obviamente: la matriz que transforma el sistema de referencia \mathcal{B} en el sistema de referencia \mathcal{A} es la inversa de la que transforma \mathcal{A} en \mathcal{B}

Descomposición de la inversa

La descomposición de $M_{\mathcal{B},\mathcal{A}}$ es:

$$M_{\mathcal{B},\mathcal{A}} = M_{\mathcal{A},\mathcal{B}}^{-1} = (D_{\mathcal{A},\mathcal{B}} R_{\mathcal{A},\mathcal{B}})^{-1} = R_{\mathcal{A},\mathcal{B}}^{-1} D_{\mathcal{A},\mathcal{B}}^{-1}$$

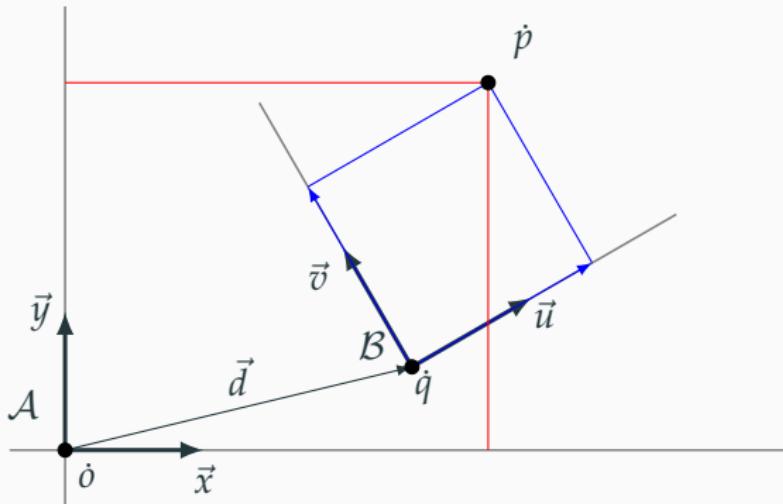
la matriz $D_{\mathcal{A},\mathcal{B}}^{-1}$ es un desplazamiento que lleva \dot{q} a \dot{o} , es decir, un desplazamiento por el vector $\dot{o} - \dot{q} = \mathcal{A}(-d_x, -d_y, -d_z, 0)^t$.

Matricialmente:

$$M_{\mathcal{B},\mathcal{A}} = \begin{pmatrix} a_x & b_x & c_x & 0 \\ a_y & b_y & c_y & 0 \\ a_z & b_z & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Relación entre marcos cartesianos

Supongamos ahora que \mathcal{A} y \mathcal{B} son dos marcos cartesianos, de nuevo se tiene $\mathcal{B} = \mathcal{A}M_{\mathcal{A},\mathcal{B}}$ y un punto cualquiera \dot{p} se puede expresar de dos formas:



$$\text{donde: } \vec{d} = \dot{q} - \dot{o}$$

Transformación inversa entre marcos cartesianos

La matriz $M_{\mathcal{A},\mathcal{B}}$ se puede descomponer en $D_{\mathcal{A},\mathcal{B}}R_{\mathcal{A},\mathcal{B}}$, ademas:

- ▶ Al ser ambos marcos cartesianos, la matriz $R_{\mathcal{A},\mathcal{B}}$ es una matriz **ortonormal**, es decir, las columnas son perpendiculares entre sí y de longitud unidad.
- ▶ $R_{\mathcal{A},\mathcal{B}}$ es una *rotación* que alinea los ejes.
- ▶ La inversa de $R_{\mathcal{A},\mathcal{B}}$ es su traspuesta, es decir: $R_{\mathcal{A},\mathcal{B}}^{-1} = R_{\mathcal{A},\mathcal{B}}^T$.

Matricialmente, por tanto:

$$M_{\mathcal{B},\mathcal{A}} = \begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

es decir, la transformación inversa entre marcos cartesianos es muy fácil de construir directamente a partir de las coordenadas de \mathcal{B} en \mathcal{A} .

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.7.

Representación y operaciones con tuplas y matrices.

Tuplas y matrices: el archivo `tup_mat.h`

Para representar en memoria tuplas de coordenadas y matrices, y operar con ellas, podemos usar el archivo `tup-mat.h`:

- ▶ Clases para tuplas (pequeños arrays) de 2,3 o 4 elementos, de tipos **float**, **double**, **int** o **unsigned**.
- ▶ Clases para matrices de 4x4 elementos, de tipo **float** o **double**.
- ▶ Operaciones sobre tuplas y matrices (sumar, restar, componer matrices, aplicar matrices a tuplas, etc...).
- ▶ Funciones para generar matrices usuales en IG.

Ejemplo de uso de las tuplas.

```
// Importar todas las definiciones:  
#include <tup_mat.h>  
using namespace tup_mat ;  
  
// Tuplas adecuadas para coordenadas de puntos, vectores o normales en 3D  
// también para colores (R,G,B)  
Tupla3f t1 ; // tuplas de tres valores tipo float  
Tupla3d t2 ; // tuplas de tres valores tipo double  
  
// adecuadas para la tabla de caras en mallas indexadas  
Tupla3i t3 ; // tuplas de tres valores tipo int  
Tupla3u t4 ; // tuplas de tres valores tipo unsigned  
  
// adecuadas para puntos o vectores en coordenadas homogéneas  
// también para colores (R,G,B,A)  
Tupla4f t5 ; // tuplas de cuatro valores tipo float  
Tupla4d t6 ; // tuplas de cuatro valores tipo double  
  
// adecuadas para puntos o vectores en 2D, y coordenadas de textura  
Tupla2f t7 ; // tuplas de dos valores tipo float  
Tupla2d t8 ; // tuplas de dos valores tipo double
```

Creación, consulta y modificación de tuplas.

Este código válido ilustra las distintas opciones:

```
float      arr3f[3] = { 1.0, 2.0, 3.0 } ;
unsigned   arr3i[3] = { 1, 2, 3 } ;

// declaraciones e inicializaciones de tuplas
Tupla3f  a( 1.0, 2.0, 3.0 ), b, c(arr3f) ; // b indeterminado
Tupla3i  d( 1, 2, 3 ), e, f(arr3i) ;           // e indeterminado

// accesos de solo lectura, usando su posición o índice en la tupla (0,1,2,...),
// o bien varias constantes predefinidas para coordenadas (X,Y,Z) o colores (R,G,B):
float x1 = a(0), y1 = a(1), z1 = a(2),    //
      x2 = a(X), y2 = a(Y), z2 = a(Z),    // apropiado para coordenadas
      re = c(R), gr = c(G), bl = c(B) ;  // apropiado para colores

// conversiones a punteros
float *      p1 = a ; // conv. a puntero de lectura/escritura
const float * p2 = b ; // conv. a puntero de solo lectura

// accesos de escritura
a(0) = x1 ;  c(G) = gr ;

// escritura en un 'ostream' (cout) (se escribe como: (1.0,2.0,3.0))
cout << "la tupla 'a' vale: " << a << endl ;
```

Operaciones entre tuplas y escalares.

En C++ se pueden sobrecargar los operadores binarios y unarios usuales (+ , - , etc...) para operar sobre las tuplas de valores reales:

```
// declaraciones de tuplas y de valores escalares
Tupla3f a,b,c ;
float s,l ;
// operadores binarios y unarios de suma/resta/negación
a = b+c ;
a = b-c ;
a = -b ;
// multiplicación y división por un escalar
a = 3.0f*b ;      // a = 3b
a = b*4.56f ;     // a = 4.56b
a = b/34.1f ;      // a = (1/34.1)b
// otras operaciones
s = a.dot(b)       ; // producto escalar (usando método dot)
s = a|b            ; // producto escalar (usando operador binario | )
a = b.cross(c)    ; // producto vectorial a = b × c (solo para tuplas de 3 valores)
l = a.lengthSq()   ; // l = ||a||2 (calcular módulo al cuadrado)
a = b.normalized(); // a = copia normalizada de b (b no cambia)
```

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.8.

Representación y operaciones sobre matrices..

Representación de transf. en memoria.

Para representar una matriz en memoria, es cómodo almacenar los 16 valores de forma contigua, y de tal manera que se puedan acceder usando el índice de fila y de columna. Para ello se puede usar el tipo de datos **Matriz4f**:

```
#include <matrixg.hpp>
// declaraciones de matrices
Matriz4f m,m1,m2,m3 ; float a,b,c ; unsigned f = 0 , c = 1 ;
// accesos (comprobados) de lectura (var = matriz(fila,columna))
a = m(1,2) ;
b = m(f,c) ;
// accesos (comprobados) de escritura (matriz(fila,columna) = expr)
m(1,2) = 34.6 ;
m(f,c) = 0.0 ;
// multiplicación o composición de matrices
m1 = m2*m3 ;
// multiplicación de matriz 4x4 por tupla de 4 floats (y de 3, añadiendo 1)
Tupla4f t, t4( 1.0,2.0,3.0,4.0 ) ; Tupla3f t3(1.0,1.0,3.0);
t = m2*t4 ; t = m2 * t3 ;
// conversión a puntero a 16 flotantes (float *) (formato OpenGL)
float * pm = m ; const float * pcm = m ;
// escritura en la salida estándar (varias líneas)
cout << m << endl ;
```

Matrices más usuales

También podemos construir funciones C++ para obtener las matrices más usuales:

```
// devuelve la matriz identidad
Matriz4f MAT_Ident( ) ;

// devuelven una matriz de traslación por dx,dy,dz (o d[X],d[Y],d[Z])
Matriz4f MAT_Traslacion( const float d[3] ) ;
Matriz4f MAT_Traslacion( const float dx, const float dy ,
                        const float dz ) ;

// devuelve una matriz de escalado por s_x,s_y,s_z
Matriz4f MAT_Escalado( const float sx, const float sy,
                      const float sz ) ;

// devuelve una matriz de rotación de eje arbitrario (ex,ey,ez)
Matriz4f MAT_Rotacion( const float ang_gra, const float ex,
                      const float ey, const float ez ) ;
```

Fin de la presentación.