



UNIVERSIDAD  
DE GRANADA

# Informática Gráfica: Teoría. Tema 1. Introducción.

---

Carlos Ureña

2021-22

**Grado en Informática y Matemáticas**  
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 2021-22.

**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)

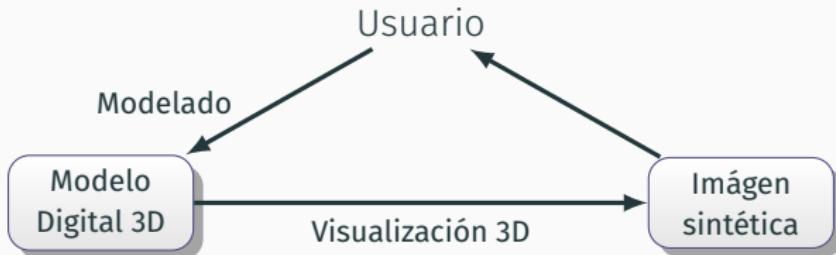
en aplicaciones específicas, se usan otros campos de la Informática en particular o la Ciencia en general (p.ej. para desarrollo de videojuegos se usan también técnicas de Inteligencia Artificial).

# Informática Gráfica 3D interactiva

Los elementos esenciales de una aplicación gráfica son:

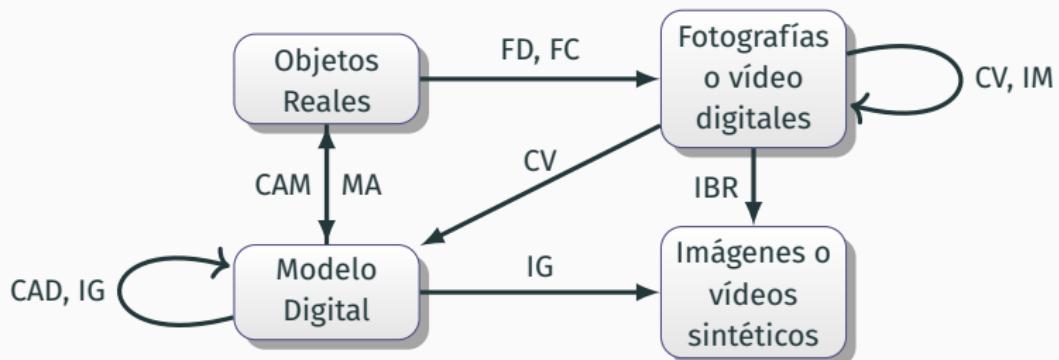
- ▶ **Modelos digitales** de objetos reales, ficticios o de datos
- ▶ **Imágenes o vídeos digitales** que se usan para visualizar dichos objetos.

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 2021-22.

**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>

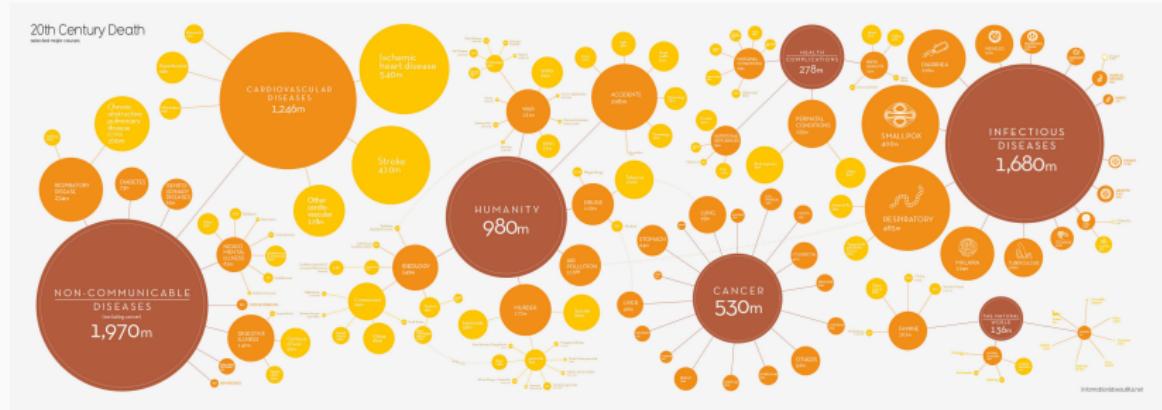
# Películas y animaciones generadas por ordenador



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

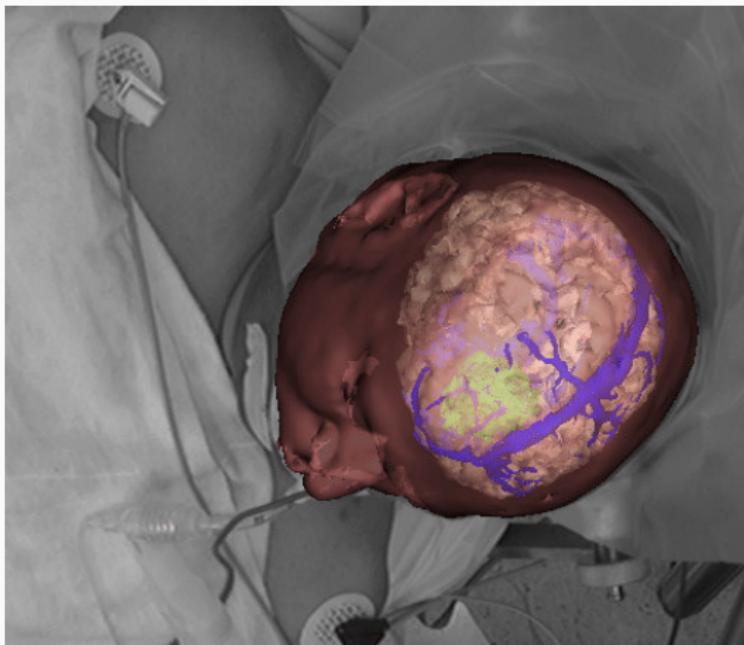
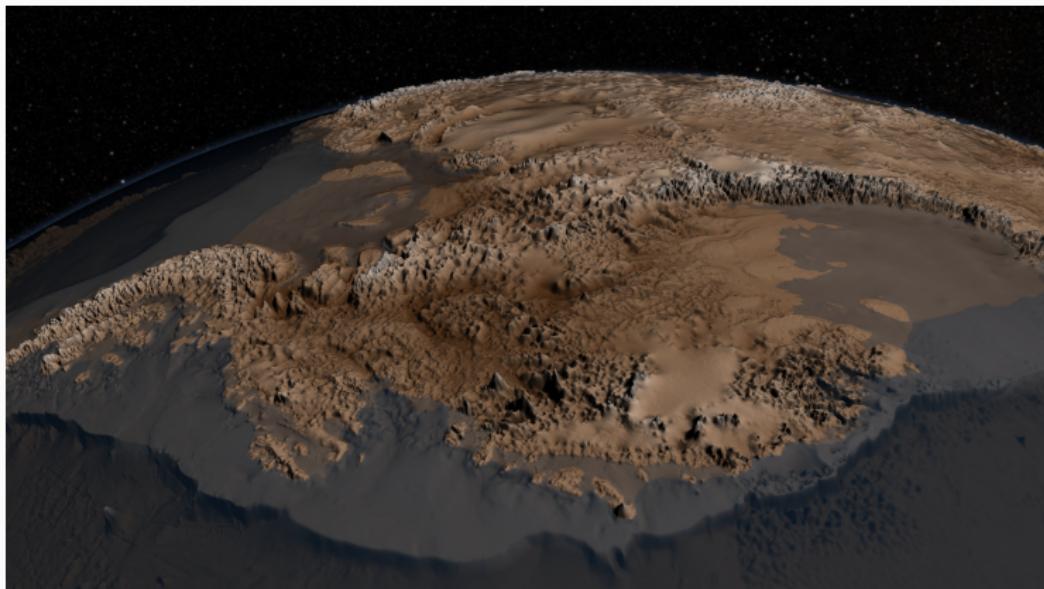


Imagen creada por Christopher Brown, Universidad de Rochester:  
 <http://www.cs.rochester.edu/u/brown/projects.html>

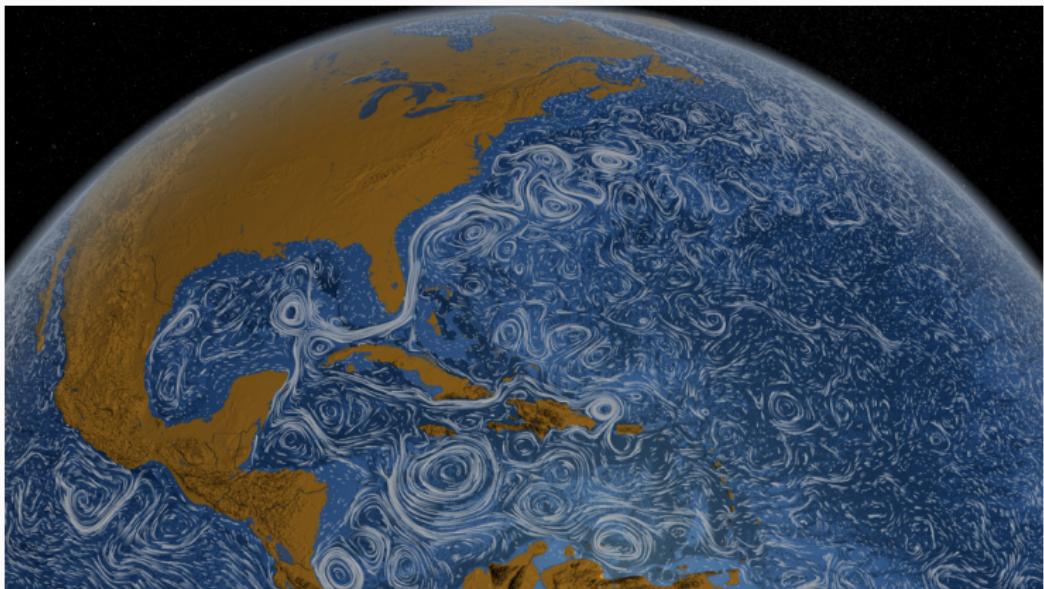
# Visualización científica (geología)



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/](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 2021-22.

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 2021-22.

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**:

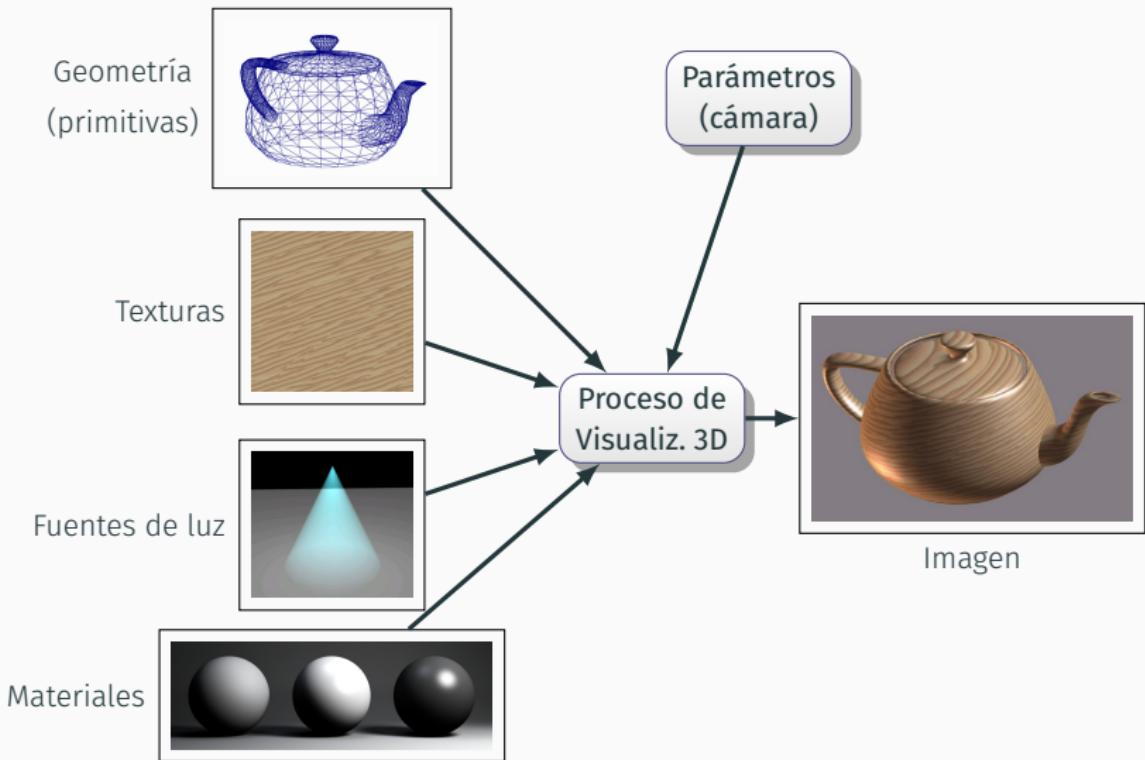
- ▶ **Modelo de escena:** estructura de datos en memoria que representa lo que se quiere ver, esta formado por varias partes:
  - ▶ **Modelo geométrico:** conjunto de **primitivas** (típicamente polígonos planos), que definen la forma de los objetos a visualizar
  - ▶ **Modelo de aspecto:** conjunto 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 2021-22.

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 2021-22.

Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.4.

El cauce gráfico en rasterización.

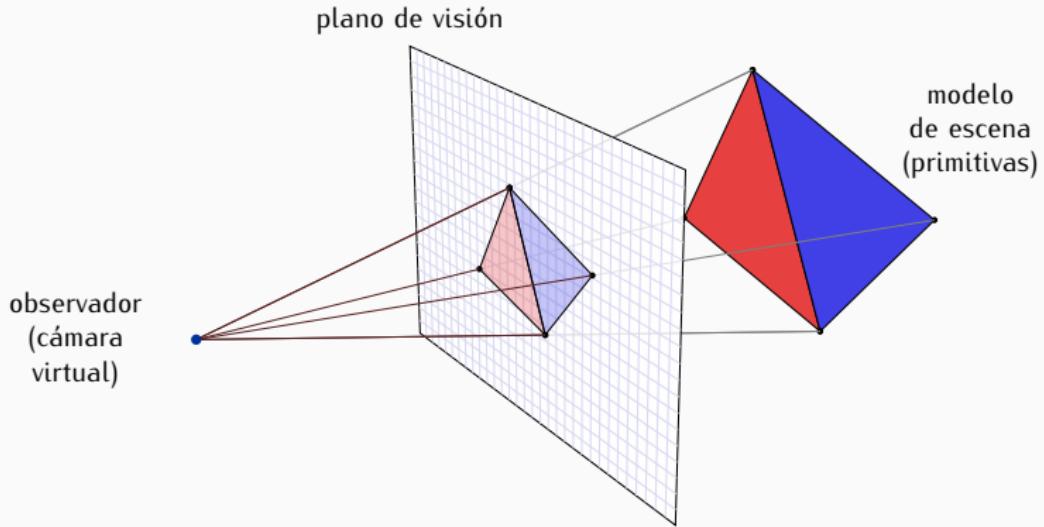
# El cauce gráfico: entradas y salidas

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

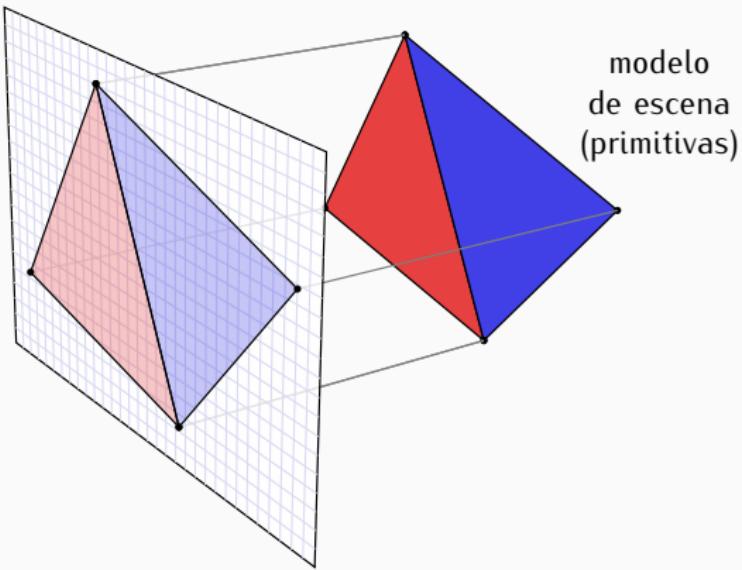
Cada primitiva se sitúan en su lugar en el espacio, y se encuentra su proyección en un plano imaginario (**plano de visión, viewplane**) situado entre el **observador** y la escena (las primitivas):



# Proyección paralela

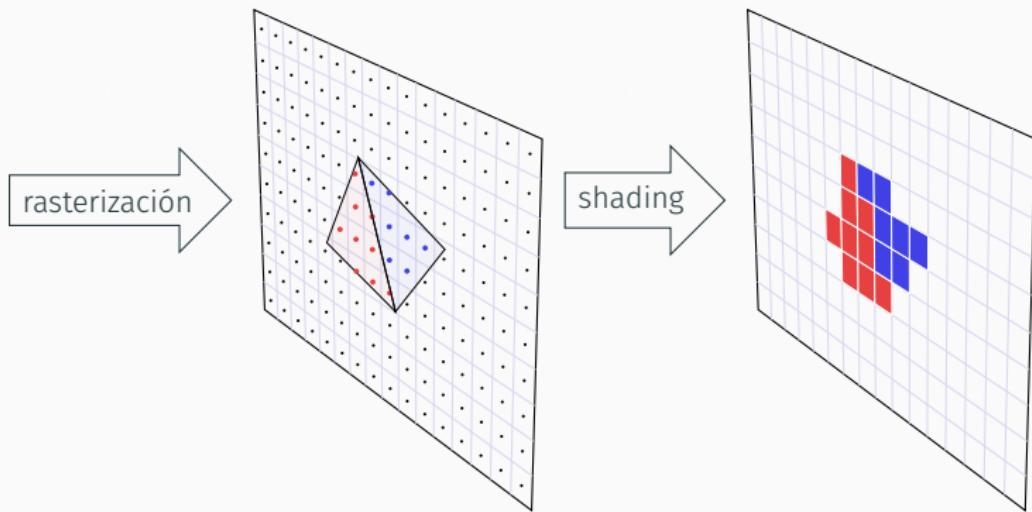
La proyección puede ser **perspectiva** (como en la transparencia anterior), o **paralela**, como aparece aquí:

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 2021-22.

Teoría. Tema 1. Introducción.

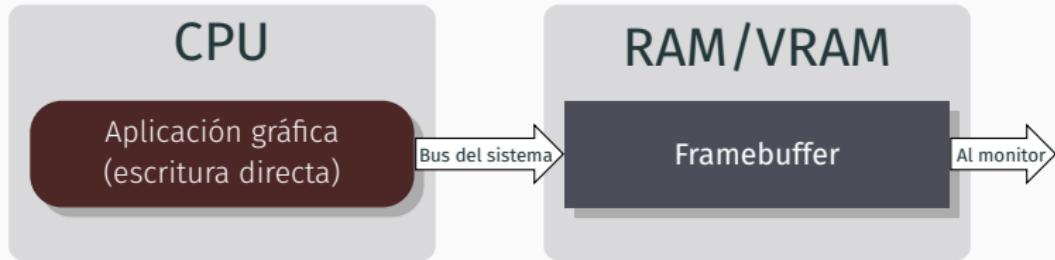
Sección 2. El proceso de visualización

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)

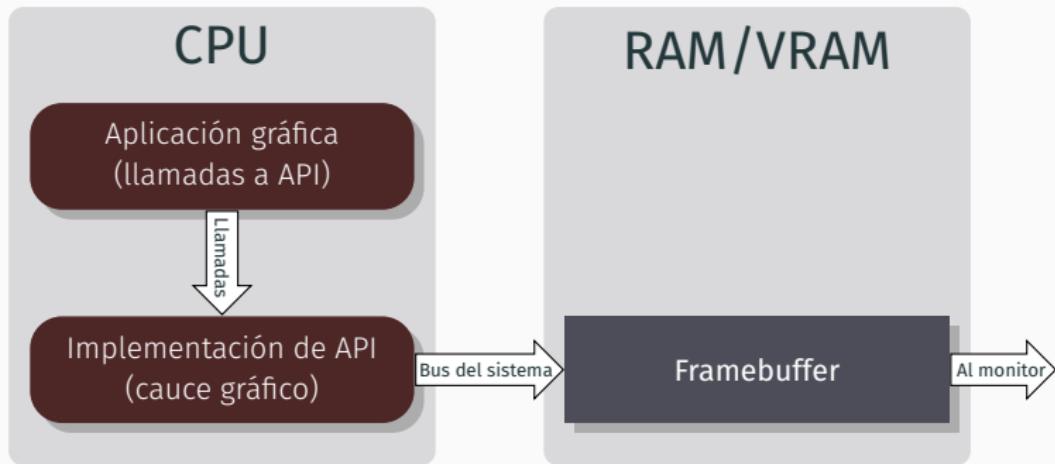


## Desventajas

- ▶ 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

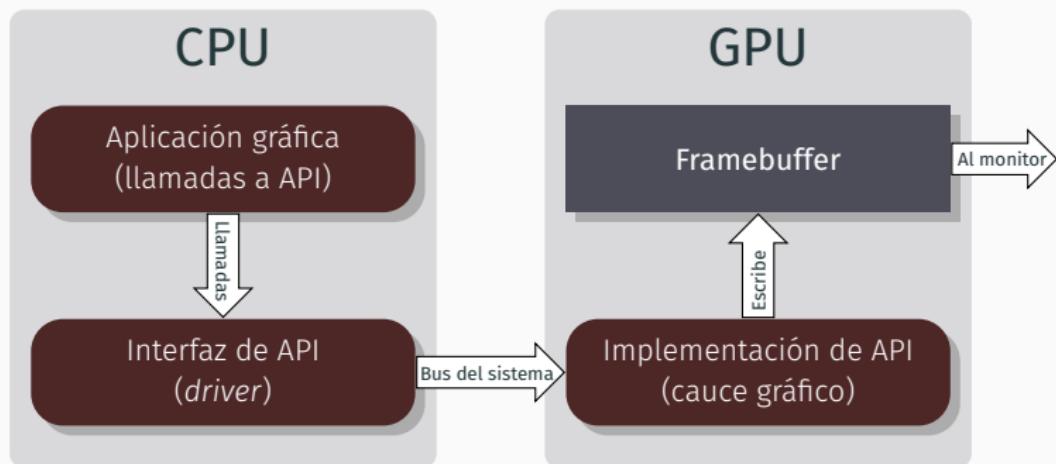
El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad** y **acceso simultáneo**



- ▶ 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 2021-22.

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.

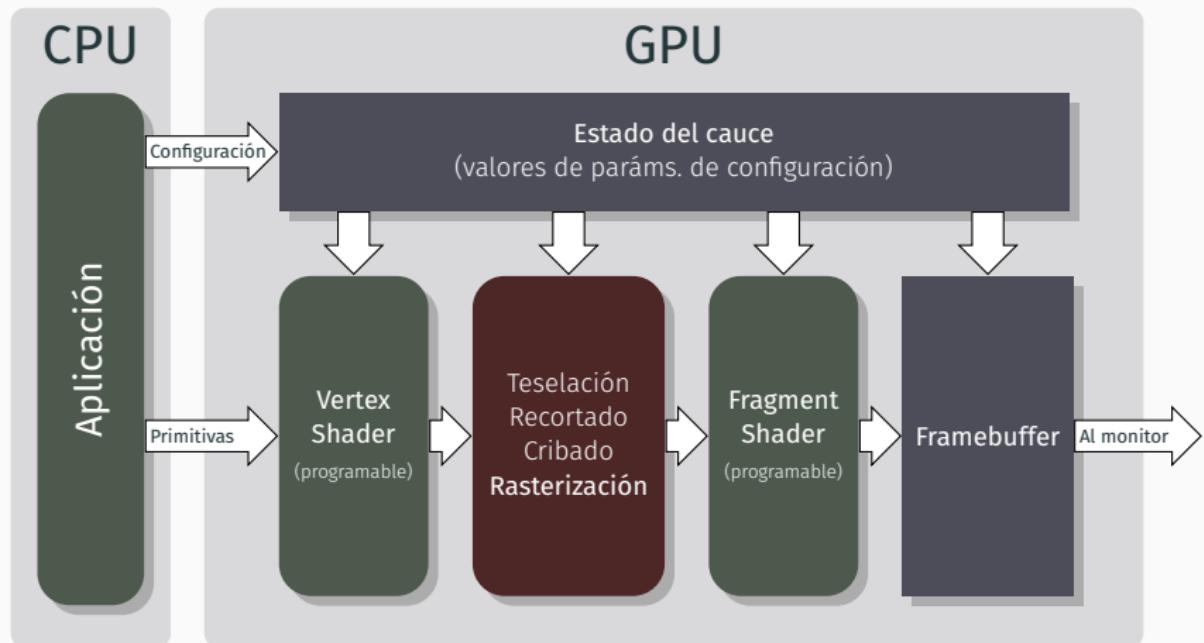
## Etapas del cauce gráfico (2/2)

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 2021-22.

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.

- ▶ 1980: Los programas gráficos se escribían para hardware específico.
- ▶ 1988: Silicon Graphics inc. era líder en estaciones gráficas. Sus sistemas usaban IRIS GL, que era propiedad de Silicon Graphics.
- ▶ 1990: Otras empresas empiezan a desarrollar hardware gráfico (SUN, IBM, HP).
- ▶ 1991: Silicon Graphics decide abrir su API para aumentar su influencia en el mercado, creando OpenGL.
- ▶ 1992: Silicon Graphics crea el OpenGL Architectural Review Board (ARB), en la que también participan Microsoft, IBM, DEC y Intel. Es el comité encargado de acordar las especificaciones de las distintas versiones de OpenGL.
- ▶ 1992: Se diseña y publica la primera versión de OpenGL.
- ▶ 2003: Se diseña y publica la primera versión de OpenGL ES.
- ▶ 2008: Se publica la versión 3.0 de OpenGL.
- ▶ 2018: Se publica la versión 3.2 de OpenGL ES y la versión 4.6 de OpenGL. (se tie

## Bibliotecas complementarias: GLU y GLFW

Las implementaciones de OpenGL se distribuyen junto con la de la biblioteca **GLU** (*OpenGL Utility Library*). Esta biblioteca contiene, entre otras

- ▶ Funciones para configuración de la cámara virtual.
- ▶ Dibujo de primitivas complejas (esferas, cilindros, discos).
- ▶ Funciones de dibujo de alto nivel (superficies, polígonos concavos).

**GLFW** es una librería auxiliar portable y *open source*, sirve para:

- ▶ Gestión de ventanas (usando el gestor de ventanas del S.O.)
- ▶ Gestión de eventos de entrada (leer de teclado y ratón).

Los nombres de las funciones de GLU comienzan con `glu` y las de GLFW 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 este repositorio público de github:

☞ [github.com/carlos-urena/opengl-minimo](https://github.com/carlos-urena/opengl-minimo)

- ▶ Se puede compilar en Linux, Windows o macOS.
- ▶ Se dan instrucciones y archivos de compilación en Linux y macOS (por ahora).
- ▶ 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 2021-22.

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 2021-22.

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 **polígonos**:

- ▶ 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).
- ▶ uno o varios segmentos de recta, en concreto:
  - ▶  $n/2$  segmentos independientes ( $n$  par).
  - ▶  $n - 1$  segmentos formando una **polilínea abierta** ( $n \geq 2$ ).
  - ▶  $n$  segmentos formando una **polilínea cerrada** ( $n \geq 3$ ).

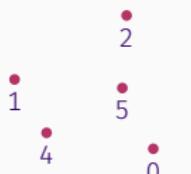
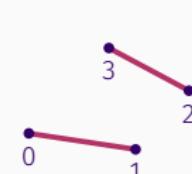
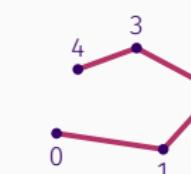
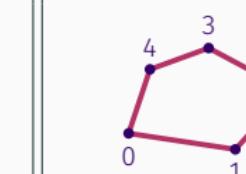
## Tipos de primitivas: polígonos

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

- ▶  $n/3$  triángulos ( $n$  múltiplo de 3).
- ▶  $n/4$  cuadriláteros ( $n$  múltiplo de 4).
- ▶ un **polígono** con  $n$  lados ( $n \geq 3$ )
- ▶  $n - 2$  triángulos compartiendo aristas (**tira de triángulos**), cada triángulo comparte dos vértices con el anterior ( $n \geq 3$ ).
- ▶  $n - 1$  triángulos compartiendo un vértice (**abanico de triángulos**) todos los triángulos comparten el primer vértice, y cada triángulo comparte dos vértices con el anterior ( $n \geq 3$ ).
- ▶  $(n - 2)/2$  cuadriláteros compartiendo aristas (**tira de cuadriláteros**), cada cuadrilátero comparte dos vértices con el anterior ( $n \geq 4$ ,  $n$  par).

# Primitivas de tipo puntos y segmentos.

Las coordenadas  $(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1})$  forman puntos o segmentos

Puntos <b>GL_POINTS</b>	Segmentos <b>GL_LINES</b>	Polilínea abierta <b>GL_LINE_STRIP</b>	Polilínea cerrada <b>GL_LINE_LOOP</b>
 A set of six red dots representing points. They are labeled with integers: 1 (top-left), 2 (top-right), 3 (middle-right), 4 (bottom-left), 5 (middle-left), and 0 (bottom-center).	 Two red line segments. The first segment connects point 0 at the bottom to point 1 below it. The second segment connects point 1 to point 2 at the top-right.	 A sequence of five red line segments forming an open polygon. The vertices are labeled 0, 1, 2, 3, and 4. The segments connect 0 to 1, 1 to 2, 2 to 3, 3 to 4, and 4 back to 0.	 A sequence of five red line segments forming a closed polygon. The vertices are labeled 0, 1, 2, 3, and 4. The segments connect 0 to 1, 1 to 2, 2 to 3, 3 to 4, and 4 to 0.

- ▶ En OpenGL se definen varias constantes de tipo **GLenum** (entero sin signo) para identificar los distintos tipos de primitivas
- ▶ En estos casos se usan las constantes **GL\_POINTS**, **GL\_LINES**, **GL\_LINE\_STRIP** o **GL\_LINE\_LOOP**)

## Polígonos delanteros y traseros. Cribado.

Cada primitiva de tipo polígono (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 polígonos.

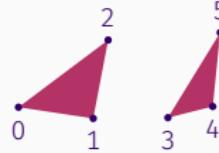
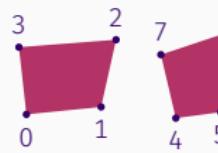
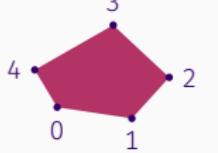
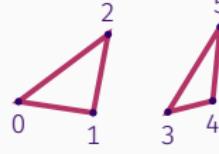
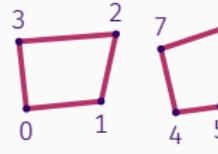
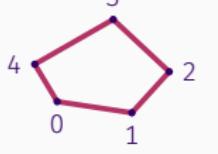
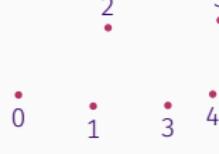
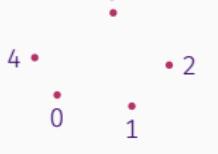
En el caso de las primitivas de polígonos, OpenGL puede visualizarlos de varias formas, según el valor de un parámetro de configuración en el estado de OpenGL, que se llama el **modo de visualización de polígonos**, y que permite seleccionar una de estas opciones:

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

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

# Primitivas tipo polígonos (no adyacentes)

Visualización de una secuencia de  $n$  vértices:  $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_{n-1}$

Primitivas/ modo de pol	Triángulos <b>GL_TRIANGLES</b>	Cuadriláteros <b>GL_QUADS</b>	Polígono <b>GL_POLYGON</b>
modo relleno <b>GL_FILL</b>			
modo líneas <b>GL_LINE</b>			
modo puntos <b>GL_POINT</b>			

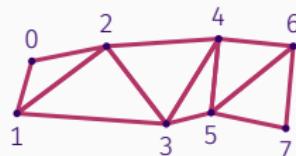
# Primitivas tipo polígonos (adyacentes)

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

## Tira de triángulos **GL\_TRIANGLE\_STRIP**

Polígonos:

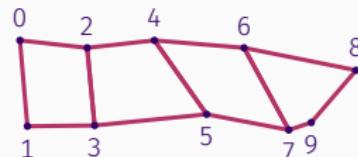
$(0,1,2), (2,1,3), (2,3,4),$   
 $(4,3,5), (4,5,6), (6,5,7), \dots$



## Tira de cuadriláteros **GL\_QUAD\_STRIP**

Polígonos:

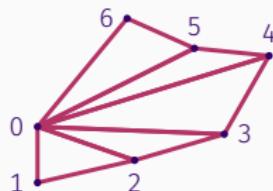
$(0,1,3,2), (2,3,5,4),$   
 $(4,5,7,6), (6,7,9,8), \dots$



## Abanico de triángulos **GL\_TRIANGLE\_FAN**

Polígonos:

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



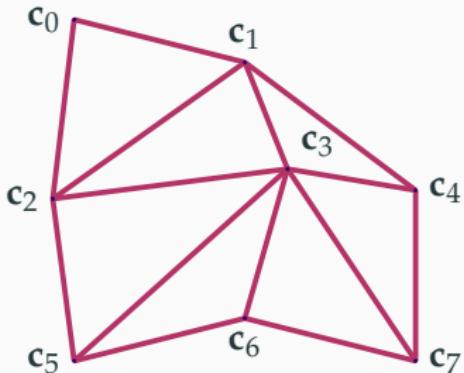
# Polígonos de más de tres vértices

Respecto de las primitivas de tipo polígono de más de 3 vértices y los cuadriláteros:

- ▶ Deben cumplir estos requisitos:
  - ▶ Deben tener todos sus vértices en el mismo plano
  - ▶ Las aristas no deben intersecarse entre ellas
  - ▶ Deben de ser convexos
- (si no cumplen alguno de ellos, no se visualizan correctamente).
- ▶ Internamente, se convierten en triángulos (las GPUs solo rasterizan triángulos). Se dice que los polígonos son *teselados*.
- ▶ En la versión 3.0 de OpenGL (2008), se declararon *obsoletas* este tipo de primitivas, y en posteriores se eliminaron (no existen las constantes **GL\_POLYGON**, **GL\_QUADS** ni **GL\_QUAD\_STRIP**).

# 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:

```
{   c0, c2, c1, c1, c2, c3,  
    c1, c3, c4, c2, c5, c3,  
    c3, c5, c6, c3, c6, c7,  
    c3, c7, c4 }
```

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  $c_3$  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:

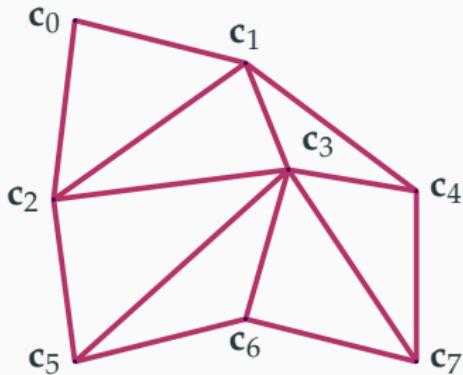
- ▶ Se parte de una secuencia  $V_n$  de  $n$  coordenadas arbitrarias de vértices  $V_n = \{\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}\}$ , que pueden ser únicos.
- ▶ Se usa una secuencia  $I_m$  de  $m$  **índices**  $I_m = \{i_0, i_1, \dots, i_{m-1}\}$  donde cada valor  $i_j$  es un entero 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 2021-22.

Teoría. Tema 1. Introducción.

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

Subsección 3.4.

Atributos de vértices.

# Atributos de vértices

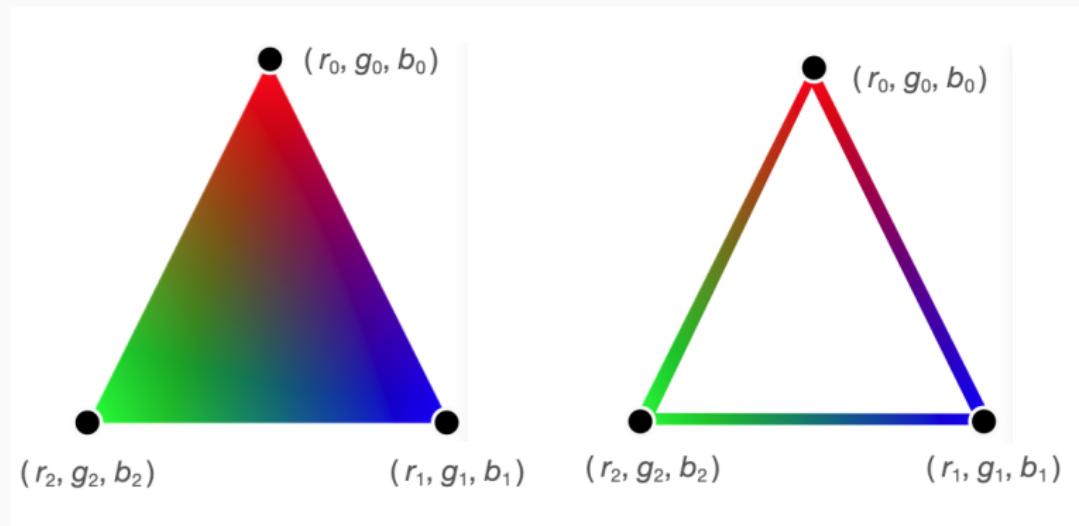
Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se pueden opcionalmente usar otros atributos, por ejemplo:

- ▶ 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 2.1.

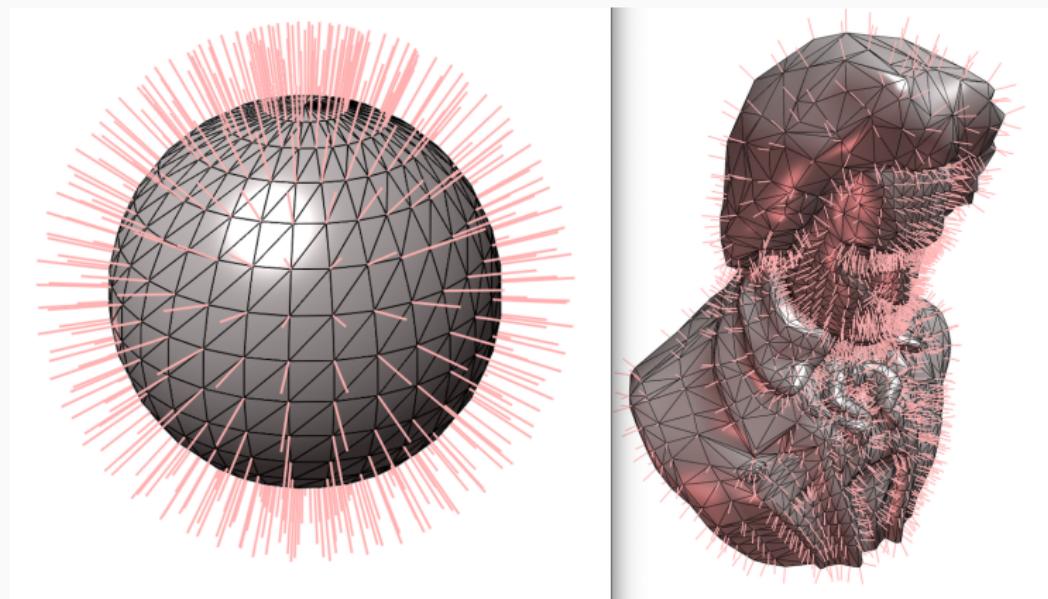
## Atributos: colores de vértices

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.



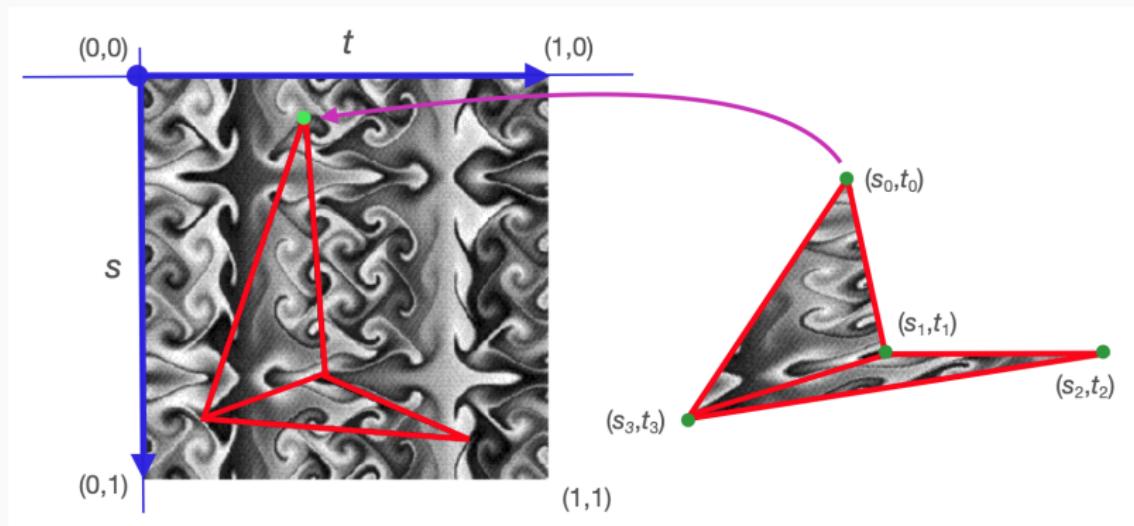
## Atributos: normales

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.
- ▶ 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 el mismo valor de un atributo para todos los vértices de una primitiva, o bien especificar un valor para cada uno.

El valor de cada atributo está definido en cada pixel donde se proyecta la primitiva (en cualquier modo de polígono). Estos valores se calculan durante la rasterización usando **interpolación** de los valores en los vértices.

Informática Gráfica, curso 2021-22.

Teoría. Tema 1. Introducción.

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

Subsección 3.5.

Modos de envío..

## Modos de envío

Cada vez que queramos visualizar las secuencias de vértices y atributos:

- ▶ Podemos enviar las coordenadas y atributos vértice a vértice (una llamada por vértice y atributo). No requiere tenerlos almacenados.
- ▶ Podemos enviar tablas (arrays) completos almacenados en la **memoria principal** (RAM) del proceso (una única llamada para visualizar una secuencia de vértices con todos sus atributos). Esto incluye tablas de coordenadas, atributos o índices.
- ▶ Podemos enviar las tablas con una única llamada, teniendo dichas tablas almacenadas en la **memoria de la GPU**, hay que transferirlas antes desde memoria principal a la GPU, una sola vez.

En los dos primeros casos se habla de **modo inmediato (*immediate mode*)**, y en el tercero de **modo diferido (*deferred mode*)**.

## Envío en *Modo Inmediato*

OpenGL hace posible varios modos de enviar las primitivas al cauce gráfico. En primer lugar veremos el **modo inmediato**:

- ▶ El programa **envía a OpenGL** la secuencia de coordenadas, en orden.
- ▶ La implementación de OpenGL procesa la secuencia de vértices y **visualiza las primitivas** correspondientes en el *framebuffer* activo durante el envío.
- ▶ OpenGL **no almacena las coordenadas** tras la visualización.
- ▶ Para visualizar una primitiva más de una vez, es necesario volver a enviar las mismas coordenadas de vértices cada vez.
- ▶ Cada vértice es una tupla de 3 coordenadas en el espacio euclídeo 3D.

## Envío en Modo Diferido

El modo inmediato es muy ineficiente en tiempo por requerir el envío de todos los vértices por el bus del sistema a la GPU en cada visualización, aunque no cambien. 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 indica a OpenGL que lea de la memoria de la GPU, en lugar de la memoria RAM.
- ▶ 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 2021-22.

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. = {  $\underbrace{x_0, y_0, z_0}_{\text{posición 0}}, \underbrace{r_0, g_0, b_0}_{\text{color 0}}, \underbrace{n_{x0}, n_{y0}, n_{z0}}_{\text{normal 0}}, \underbrace{s_0, t_0}_{\text{c.c. 0}}, \underbrace{x_1, y_1, z_1}_{\text{posición 1}}, \underbrace{r_1, g_1, b_1}_{\text{color 1}}, \dots, \underbrace{s_{n-1}, t_{n-1}}_{\text{c.c. } n-1} \}$  } 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  $\rightarrow \{ \underbrace{u_0, v_0}_{\text{c.c.t. 0}}, \underbrace{u_1, v_1}_{\text{c.c.t. 1}}, \dots, \underbrace{u_{n-1}, v_{n-1}}_{\text{c.c.t. } n-1} \}$

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

## *Vertex Buffer Objects* para modo diferido

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).

# Operaciones sobre *Vertex Buffer Objects*

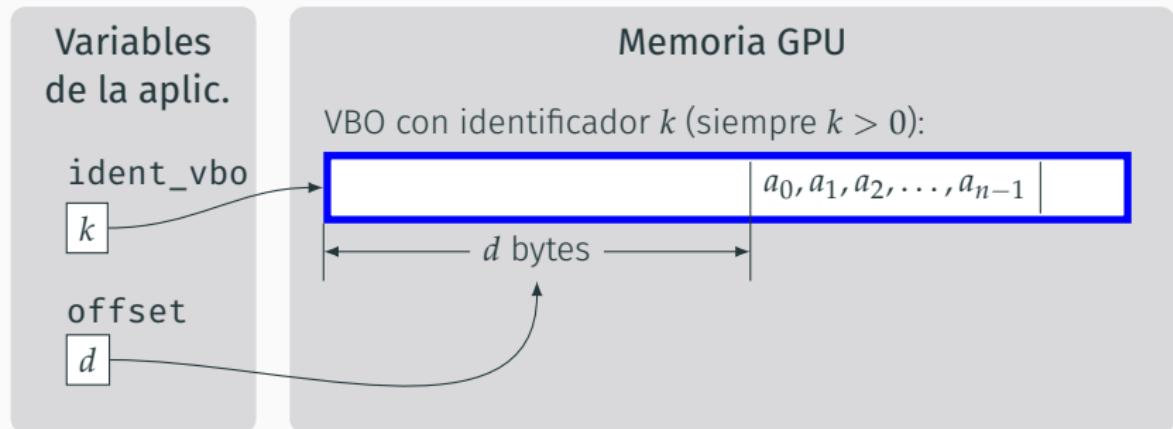
Las siguientes funciones permiten acceder a los VBOs:

- ▶ **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).

Si hay un VBO activado, podemos referirnos a bytes específicos del VBO activo, usando un **desplazamiento (offset)** (número de bytes desde el inicio del VBO hasta la dirección de memoria del byte).

# 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:



## Descriptores de tabla (1/3)

Una tabla de atributos (en SOA) o de índices se puede describir usando un conjunto de valores o metadatos relativos a la propia tabla. A ese conjunto lo llamamos **descriptor de tabla**. En OpenGL podemos usar estos valores enteros (o puntero):

1. **Tipo de tabla** (**tipo\_tabla**): un valor entero para indicar si es una tabla de índices (**GL\_ELEMENT\_ARRAY\_BUFFER**) o bien una tabla de coordenadas o atributos (**GL\_ARRAY\_BUFFER**)
2. **Atributo** (**atributo**): en el caso de tablas de coordenadas o atributos, se usa otro valor entero para discriminar de que tabla concreta se trata. Puede valer:
  - ▶ **GL\_VERTEX\_ARRAY** : coordenadas (posiciones)
  - ▶ **GL\_COLOR\_ARRAY** : colores.
  - ▶ **GL\_NORMAL\_ARRAY** : normales.
  - ▶ **GL\_TEXTURE\_COORD\_ARRAY**: coordenadas de textura.

## Descriptores de tabla (2/3)

3. Tipo de valores (**tipo\_valores**): valor que codifica el tipo de datos. Usaremos **GL\_UNSIGNED\_INT** para los índices, y podemos usar los valores **GL\_FLOAT** (para **float**) o **GL\_DOUBLE** (para **double**), tanto para las coordenadas como para los atributos.
4. Número de bytes por valor (**num\_bytes\_valor**), se puede calcular con **sizeof** exclusivamente en función de **tipo\_valores** ( $> 0$ ).
5. Número de valores por tupla (**num\_vals\_tupla**): número de valores reales por vértice, puede ser 2, 3 o 4 para las tablas de atributos, y debe ser 1 para las tablas de índices ( $> 0$ ).
6. Número de tuplas o índices (**num\_tuplas\_ind**): Para tablas de atributos es el número de vértices, para las de índices es el número de índices ( $> 0$ ).

## Descriptores de tabla (3/3)

7. Tamaño en bytes (**tamano\_en\_bytes**) tamaño de la tabla completa, se calcula como producto de **num\_bytes\_valor**, **num\_vals\_tupla** y **num\_tuplas\_ind**.
8. Puntero a datos (**datos**): puntero a la dirección de memoria del programa donde está el primer byte de la tabla (no puede ser nulo) (se usa para envío en modo inmediato).
9. Nombre del VBO (**nombre\_vbo**): vale 0 si la tabla solo está en la memoria del programa, y un valor positivo si se ha hecho una copia en la memoria de la GPU (se usa para envío en modo diferido)

# La clase DescTabla para descriptores de tablas

Los metadatos de una tabla (junto con un puntero a la misma) pueden encapsularse en instancias de la clase **DescrTabla**:

```
class DescrTabla
{
public: // constructor: inicializa y comprueba que los datos son correctos
DescrTabla(const GLenum p_tipo_tabla, const GLenum p_atributo,
           const GLenum p_tipo_valores, const GLint p_num_vals_tupla,
           const GLsizei p_num_tuplas_ind, const GLvoid* p_datos );
.....
private:
    GLenum      tipo_tabla      = 0; // tipo: atributos o índices
    GLenum      atributo        = 0; // si atributo, identifica cual
    GLenum      tipo_valores    = 0; // tipo de valores (float,double,int,..)
    GLsizeiptr  num_bytes_valor = 0; // número de bytes por cada valor
    GLint       num_vals_tupla  = 0; // número de valores por cada tupla
    GLsizeii   num_tuplas_ind   = 0; // número de tuplas
    GLsizeiptr  tamano_en_bytes = 0; // tamaño total en bytes
    GLuint      nombre_vbo      = 0; // si tabla en GPU: nombre del VBO
    const GLvoid * datos = nullptr; // los datos no se modifican desde aquí
};
```

# Datos sobre secuencias de vértices

Los datos que es necesario conocer para visualizar una secuencia de vértices son:

- ▶ Localización en memoria y formato de la tabla de coordenadas
- ▶ Para cada tabla de atributos:
  - ▶ Un valor lógico que indica si se usará o no dicha tabla.
  - ▶ Si se usa, la localización en memoria y el formato de la tabla.

La palabra *formato* aquí se refiere al tipo de datos, la longitud de las tuplas, el número de tuplas, etc... Toda esta información se debe de enviar a OpenGL para visualizar la secuencia:

- ▶ En modo inmediato hay que hacerlo para cada visualización
- ▶ En modo diferido se puede enviar una sola vez y luego reusar la información muchas veces para muchas visualizaciones.

# Vertex Array Objects (VAOs) de OpenGL

Un **Vertex Array Object** guarda datos de una secuencia de vértices:

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.

- ▶ 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.
- ▶ Para visualización en modo inmediato se usa 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.
- ▶ Para visualización en modo diferido, 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 del VAO por defecto

El VAO por defecto (con identificador 0) tiene esta estructura.

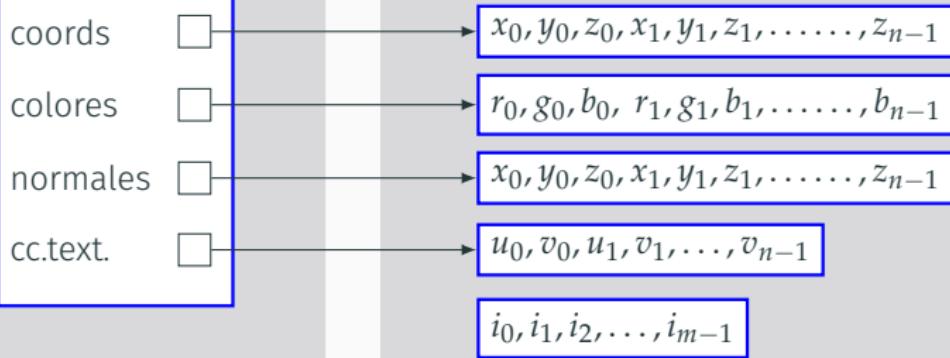
## Estado de OpenGL

VAO por defecto:

coords	<input type="checkbox"/>
colores	<input type="checkbox"/>
normales	<input type="checkbox"/>
cc.text.	<input type="checkbox"/>

## Memoria dinámica de la aplicación

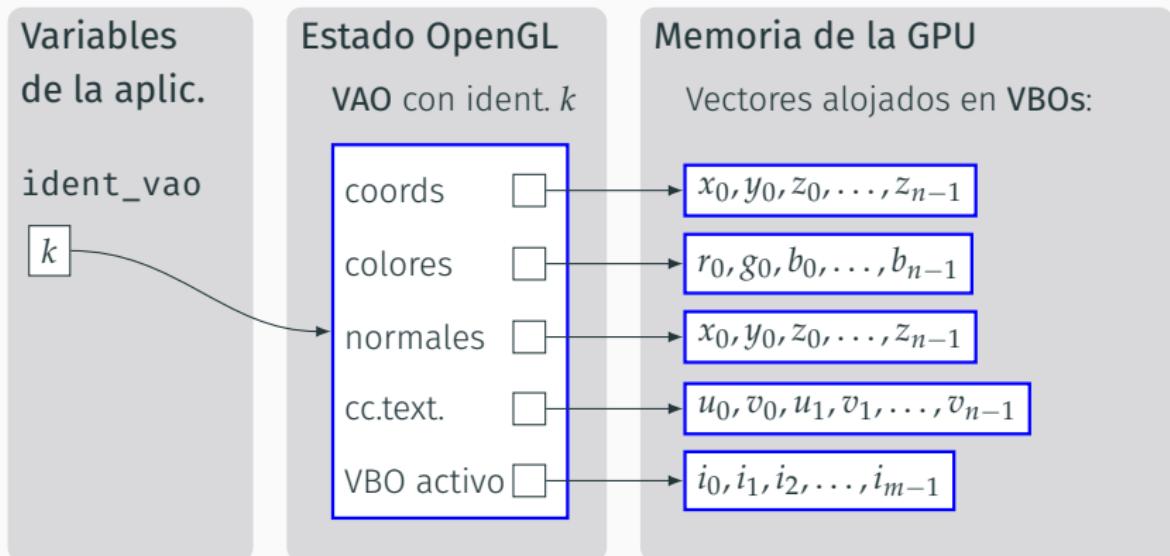
Vectores alojados por la aplicación:



Este VAO no guarda información sobre la tabla de índices (se especifica visualizar)

# Esquema de un VAO para modo diferido

Los VAOs para modo diferido son creados por la aplicación:



Si existe, la tabla de índices está en un VBO que queda activado en el VAO (al visualizar se leen los índices del VBO activo)

# 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
- ▶ **glEnableClientState**, **glDisableClientState**: habilita o deshabilita el uso de una tabla de atributos concreta en el VAO activo.
- ▶ **glVertexPointer**, **glColorPointer**, **glTexCoordPointer**, **glNormalPointer**: especifican la localización y el formato de cada tablas de atributos en el VAO activo (en OpenGL 2.1)

En cada VAO puede haber un VBO activo distinto.

## La clase ArrayVertices (1/2)

Esta clase sirve para encapsular en nuestra aplicación toda la información sobre una secuencia de vértice (facilita la visualización)

```
class ArrayVertices
{
public:
    // constructor (se indican las coordenadas)
    ArrayVertices( const GLenum tipo_valores, const GLint num_vals_tupla,
                  const GLsizei p_num_vertices, const GLvoid * datos );
    ~ArrayVertices();

    // Métodos para crear y añadir los descriptores de tablas
    void fijarColores ( GLenum tipo_valores, GLint num_vals_tupla,
                        const GLvoid *datos );
    void fijarCoordText( GLenum tipo_valores, GLint num_vals_tupla,
                        const GLvoid *datos );
    void fijarNormales ( GLenum tipo_valores, const GLvoid *datos );
    void fijarIndices ( GLenum tipo_valores, GLsizei p_num_indices,
                        const GLvoid * datos );
    .....
}
```

## La clase ArrayVertices (2/2)

Las variables de instancia son básicamente los punteros a los descriptores de las distintas tablas

```
class ArrayVertices
{
    .....
private:
    GLuint    nombre_vao    = 0; // se crea al visualizar en M.D. la primera vez
    unsigned  num_vertices  = 0, // núm. de vértices (>0)
              num_indices   = 0; // núm. de índices (0 si no hay índices)

    // Descriptores de tablas de datos (son propiedad de este 'ArrayVertices')
    DescrTabla
    {
        * coordenadas     = nullptr, // debe ser creado en el constructor
        * colores         = nullptr, // creado en 'fijarColores'
        * normales        = nullptr, // creado en 'fijarNormales'
        * coords_textura = nullptr, // creado en 'fijarCoordText'
        * indices         = nullptr; // creado en 'fijarIndices'
    };
}
```

# Representación de secuencias. Estructuras de datos en C++11.

La aplicación debe crear las tablas de atributos e índices (la clase **ArrayVertices** solo contiene punteros a dichas tablas). La creación se simplifica usando C++ moderno (versión 2011 o posteriores):

- ▶ 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 estándar 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>    coordenadas; // coordenadas 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 vért.  
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**.

# Creación e inicialización de objetos **ArrayVertices**

Una vez que la aplicación ha creado las tablas anteriores, en algún momento antes de la primera visualización se debe de crear e inicializar una instancia de **ArrayVertices**. A modo de ejemplo, para las tablas citadas antes:

```
// crear un objeto (av) que encapsula toda la información de las tablas, dar coords.  
// (el objeto 'av' no modifica ni destruye las tablas, solo las lee)  
ArrayVertices * av = new ArrayVertices( GL_FLOAT, 3,  
                                coordenadas.size(), coordenadas.data() );  
  
// opcionalmente: añadir a 'av' información sobre atributos o índices  
av->fijarColores ( GL_FLOAT, 3, colores.data() );  
av->fijarNormales ( GL_FLOAT, normales.data() );  
av->fijarCoordText( GL_FLOAT, 2, coord_text.data() );  
av->fijarIndices ( GL_UNSIGNED_INT, indices.size(), indices.data() );
```

Una vez hemos creado e inicializado el objeto **ArrayVertices**, usaremos los métodos para visualizarlo (según los modos de envío). Lo vemos en la siguiente sección.

# Problemas: generación de tablas (1/2)

## Problema 1.1.

Escribe el código que genera una tabla de coordenadas de vértices con las coordenadas de los vértices de un polígono regular de  $n$  lados o aristas, con centro en el origen y radio unidad.

El valor de  $n$  ( $> 2$ ) es un parámetro (un entero sin signo). La tabla sería la base para visualizar el polígono (las aristas, relleno, o ambas cosas), considerando la tabla de vértices como una **secuencia de vértices no indexada** (no escribas el código de visualización).

Escribe dos variantes del código: una variante (a) crea la tabla para ser visualizada usando el tipo de primitiva **GL\_LINE\_LOOP** (rellenos o solo aristas), y la otra (b) usando **GL\_LINES** (únicamente aristas).

## Problemas: generación de tablas (2/2)

### 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 que comparten un nuevo vértice en el centro del polígono.

Igual que el problema anterior (1.1) las tablas serían la base para visualizar los triángulos, ahora formando una **secuencia indexada** (no escribas el código de visualización).

Escribe dos variantes del código: una variante (a) crea la tabla de índices para ser visualizadas usando el tipo de primitiva **GL\_TRIANGLES** (rellenos o solo aristas), y la otra (b) usando **GL\_LINES** (únicamente aristas).

Informática Gráfica, curso 2021-22.

Teoría. Tema 1. Introducción.

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

Subsección 3.7.

Envío de vértices y atributos..

## Formas de visualización en modo *inmediato*

Las coordenadas se pueden enviar para visualizar de dos formas:

- ▶ Una llamada a **glVertex** por vértice (entre **glBegin/glEnd**).
  - ▶ El método es lento pues **requiere una llamada por vértice**.
  - ▶ Funcionalidad declarada **obsoleta** en OpenGL 3.0 y **eliminada** de OpenGL 3.1.
- ▶ Una única llamada a **glDrawArrays** (secuencias no indexadas) o a **glDrawElements** (secuencia indexada)
  - ▶ Requiere **almacenar las secuencias de coordenadas, atributos e índices** (si procede) en un array en la memoria RAM.
  - ▶ OpenGL recibe la dirección del array, lee todas las coordenadas y visualiza.
  - ▶ Por tanto, las implementaciones pueden hacerlo de forma más eficiente en tiempo que con **glBegin/glEnd**

## Visualización con `glBegin` / `glVertex` / `glEnd`

La primera forma de visualización es usando las funciones `glBegin`, seguida de `glVertex` (una por cada vértice) y `glEnd`:

- ▶ Usaremos `glVertex3f` (una variante del `glVertex`), que admite tres valores reales del tipo `GLfloat` (un tipo flotante definido por OpenGL, que suele ser igual a `float`, aunque no necesariamente).
- ▶ Los tres valores son las **coordenadas cartesianas** ( $x,y,z$ ) que definen la posición del vértice en el espacio.
- ▶ La secuencia de llamadas a `glVertex` se inicia con una llamada a `glBegin` y termina con una llamada a `glEnd`.
- ▶ En la llamada a `glBegin` se indica como parámetro el tipo de primitiva a usar para visualizar la secuencia.

# Envío con glBegin/glEnd

Una posibilidad es enviar y dibujar los vértices dando explicitamente sus coordenadas en cada llamada a **glVertex**

```
glBegin( tipo_prim );
    glVertex3f( x0, y0, z0 );           // envío del 1er vértice
    glVertex3f( x1, y1, z1 );           // envío del 2o vértice
    ...
    glVertex3f( xn-1, yn-1, zn-1 ); // envío del n-ésimo vértice
glEnd();
```

- ▶ Aquí **tipo\_prim** es una expresión de tipo **GLenum** (entero sin signo) que codifica el tipo de primitiva.
- ▶ Podríamos usar **GL\_TRIANGLES** para triángulos (o bien **GL\_POINTS**, **GL\_LINES**, **GL\_LINE\_STRIP**, **GL\_LINE\_LOOP**, etc...)

# Envío de atributos con glBegin/glEnd

Si queremos enviar atributos, usamos las funciones:

- ▶ **glColor3f** fija el color RGB actual (se pasan 3 valores **float**).
- ▶ **glNormal** fija la normal actual (se pasan 3 valores **float**).
- ▶ **glTexCoord2f**, fijas las coords. de textura actuales (2 **float**).

Cuando envíamos unas coordenadas con **glVertex**, al vértice se le asocian siempre los atributos actuales. P.ej., para enviar colores distintos para cada vértice:

```
glBegin( tipo_prim );
    glColor3f( r0, g0, b0 ); glVertex3f( x0, y0, z0 );
    glColor3f( r1, g1, b1 ); glVertex3f( x1, y1, z1 );
    ...
    glColor3f( rn-1, gn-1, bn-1 ); glVertex3f( xn-1, yn-1, zn-1 );
glEnd();
```

igual haríamos con las normales y las coordenadas de textura ....

# Envío con glBegin/glEnd: coords. en arrays

Lo más frecuente es tener las coordenadas de los vértices almacenados en un array en la memoria de la aplicación:

```
std::vector<float> coords ; // declaración de array de coordenadas  
....  
coords =      // inicialización explícita del array (en C++ 11 o posteriores)  
{   x0, y0, z0,           // coordenadas del 1er vértice  
    x1, y1, z1,           // coordenadas del 2o vértice  
    ...  
    xn-1, yn-1, zn-1 // coords. n-ésimo vértice  
} ;
```

En el ámbito de **vertices**, el envío se hace con **glVertex3f**:

```
glBegin( tipo_prim );  
for( int i = 0 ; i < coords.size()/3 ; i++ )  
    glVertex3f( coords[3*i+0], coords[3*i+1], coords[3*i+2] );  
glEnd();
```

# Variantes de `glVertex`

Existen otras funciones en la familia de `glVertex`

- ▶ `glVertex2f` permite especificar únicamente las coordenadas  $x$  e  $y$ , y hace  $z = 0$ . Es útil para visualización 2D (en el plano XY).
- ▶ `glVertex3d`: sus parámetros son de tipo de `GLdouble`, que normalmente es equivalente a `double` y tienen más precisión que `GLfloat`.
- ▶ `glVertex3fv`: las coordenadas se especifican usando un puntero a un array con 3 valores de tipo `GLfloat` consecutivos en memoria. A modo de ejemplo

```
GLfloat coords[3] = { 3.5, 6.7, 8.9 } ;  
glVertex3fv( coords ) ;
```

Estas variantes se pueden combinar, por ejemplo se puede usar `glVertex2dv` u otras. (en todos los casos, a la GPU llega un triple  $(x, y, z)$  en simple o doble precisión).

## Visu. con `glBegin/glEnd` de objetos `ArrayVertices` (1/2)

Podemos definir un método de la clase `ArrayVertices` para visualización usando `glBegin/glVertex/glEnd`

- ▶ Usa las variantes de `glVertex`, `glNormal`, `glTexCoord` y `glColor` que aceptan punteros a números flotantes de simple precisión (`float`).
- ▶ Si hay tabla de índices, recorre los vértices en el orden dado por los índices, si no hay, recorre los vértices en el orden en que aparacen en la tabla de coordenadas.
- ▶ Envía cada atributos de cada vértice si existe la correspondiente tabla de ese atributo.
- ▶ Según el tipo de datos y la longitud de las tuplas, habría que llamar a una versión distinta de las funciones citadas.
- ▶ A modo de ejemplo, se muestra una versión en la cual se asumen datos de tipo flotante y tuplas de 3 reales (excepto las coords. de textura con 2).

## Visu. con glBegin/glEnd de objetos ArrayVertices (2/2)

Método de ejemplo para visualizar en M.I con **glBegin/glEnd**:

```
void ArrayVertices::visualizarGL_MI_BVE( const GLenum tipo_primitiva )
{
    const int nv = (indices != nullptr) ? indices->num_tuplas_ind
                                         : num_vertices ;
    glBegin( tipo_primitiva );
    for( GLuint i = 0 ; i < nv ; i++ ) // recorre todos los vértices
    {
        // recuperar índice de vértice en la tabla de coordenadas (iv)
        const GLuint iv = (indices != nullptr) ?
                           ((const GLuint *) indices->datos)[i] : i ;
        // enviar atributos de vértice que corresponda
        if ( colores != nullptr )
            glColor3fv( (const GLfloat *) colores->datos + 3*iv );
        if ( normales != nullptr )
            glNormal3fv( (const GLfloat *) normales->datos + 3*iv );
        if ( coords_textura != nullptr )
            glTexCoord2fv( (const GLfloat *) coords_textura->datos + 2*iv );
        // enviar coordenadas
        glVertex3fv( (const GLfloat *) coordenadas->datos + 3*iv );
    }
    glEnd();
}
```

## Envío en modo inmediato con una única llamada

Es mucho más eficiente usar una única llamada para enviar la secuencia completa. Usamos estas funciones:

- ▶ Para una secuencia **no indexada**:

```
glDrawArrays( GLenum tipo_prim, GLint primero, GLsizei num_vertices );
```

(**primero** siempre será 0, permitiría visu. parte del array).

- ▶ Para una secuencia **indexada**:

```
glDrawElements( GLenum tipo_prim, GLsizei num_indices,
                GLenum tipo_datos_ind,
                const GLvoid * puntero_offset_ind );
```

En ambos casos es necesario especificar antes donde se encuentran la tabla de vértices y las de atributos que queramos usar (inicializar el VAO 0). La visualización se realiza **de forma asíncrona** (solo se inicia).

## Ejemplo en modo inmediato con glDrawArrays

Usamos el VAO 0 para visualizar una secuencia de vért. no indexada con colores. Los vértices tienen todos Z=0 (es visualización 2D).

```
constexpr unsigned num_verts = .... ; // número de vértices que se van a dibujar
GLfloat    coordenadas[ num_verts*2 ] ; // tabla de coordenadas (2 float por vér.)
GLfloat    colores      [ num_verts*3 ] ; // tabla de colores (3 float por vér.)
.....                                         // poblar tablas coords. y colores

glBindVertexArray( 0 ) ;                      // activar VAO por defecto
glDisableClientState( GL_TEXTURE_COORD_ARRAY ); // deshab. punt. a cc.text.
glDisableClientState( GL_NORMAL_ARRAY );       // deshab. punt. a normales

glVertexPointer( 2, GL_FLOAT, 0, coordenadas ); // puntero y formato coords.
glEnableClientState( GL_VERTEX_ARRAY );         // habilita array coords.
glColorPointer( 3, GL_FLOAT, 0, colores );       // puntero y formato colores
glEnableClientState( GL_COLOR_ARRAY );           // habilita array de colores

glDrawArrays( GL_POLYGON, 0, num_verts );        // inicia visualización

glDisableClientState( GL_VERTEX_ARRAY );          // deshab. array coords.
glDisableClientState( GL_COLOR_ARRAY );           // deshab. array colores
```

# Ejemplo en modo inmediato con glDrawElements

Ejemplo similar, con una secuencia indexada. La localización y formato de los índices se da en **glDrawElements**:

```
constexpr unsigned num_verts = ..., num_inds = ...; // núm. vért. e índices
GLfloat    coordenadas[ num_verts*2 ] ; // tabla de coordenadas (2 float por vért.)
GLfloat    colores     [ num_verts*3 ] ; // tabla de colores (3 float por vért.)
GLuint     indices     [ num_inds ] ;      // tabla de índices
.....                                         // poblar tablas coords, colores, índices

glBindVertexArray( 0 ) ;                      // activar VAO por defecto
glDisableClientState( GL_TEXTURE_COORD_ARRAY ); // deshab. punt. a cc.text.
glDisableClientState( GL_NORMAL_ARRAY );       // deshab. punt. a normales

glVertexPointer( 2, GL_FLOAT, 0, coordenadas ); // puntero y formato coords.
glEnableClientState( GL_VERTEX_ARRAY );         // habilita array coords.
glColorPointer( 3, GL_FLOAT, 0, colores );       // puntero y formato colores
glEnableClientState( GL_COLOR_ARRAY );           // habilita array de colores

glDrawElements( GL_POLYGON, num_inds, GL_UNSIGNED_INT, indices ); // visu.

glDisableClientState( GL_VERTEX_ARRAY );         // deshab. array coords.
glDisableClientState( GL_COLOR_ARRAY );           // deshab. array colores
```

# Problemas: visualización en modo inmediato (1/2)

## Problema 1.3.

Crea una copia del repositorio **opengl-minimo**, y modifica el código de ese repositorio para visualizar en **modo inmediato** (con una única llamada) un polígono regular de  $n$  lados, usando la tabla de coordenadas (no indexada) que codifica dicho polígono regular, según se describe en el enunciado del problema 1.1 (variante (a)).

El polígono se dibujará relleno de un color plano (distinto del color de fondo) y además con las aristas visualizadas de otro color. Intenta que tu solución sea lo más eficiente en tiempo y simple posible (menor número de llamadas a funciones OpenGL posibles).

Incluye todo el código en una función que acepte  $n$  como parámetro, y que únicamente cree la tabla en la primera llamada (para eso puedes usar alguna variables locales a la función, pero declaradas con **static**, de forma que conservan sus valores entre las llamadas).

## Problemas: visualización en modo inmediato (2/2)

### Problema 1.4.

Crea una copia del repositorio **opengl-minimo**, y modifica el código de ese repositorio para visualizar en **modo inmediato** (con una única llamada) los  $n$  triángulos que forman un polígono regular, usando las tablas de coordenadas e índices (es una secuencia indexada) que se describen en el enunciado del problema 1.2 (variante (a)).

Los triángulos se dibujarán rellenos de un color plano (distinto del color de fondo) y además con las aristas visualizadas de otro color. Intenta que tu solución sea lo más eficiente en tiempo y simple posible (menor número de llamadas a funciones OpenGL posibles).

Incluye todo el código en una función que acepte  $n$  como parámetro, y que únicamente cree las tablas en la primera llamada.

## Fijar puntero a tabla genérica, usando DescrTabla

En el caso genérico podemos usar la clase **DescrTabla**. Este método indica la localización (puntero a memoria u offset de VBO) y formato de una tabla de atributos:

```
void DescrTabla::fijarPuntero( const GLvoid * ptr_offset )
{
    switch( atributo )
    {
        case GL_VERTEX_ARRAY :
            glVertexPointer( num_vals_tupla, tipo_valores, 0, ptr_offset );
            break ;
        case GL_TEXTURE_COORD_ARRAY :
            glTexCoordPointer( num_vals_tupla, tipo_valores, 0, ptr_offset );
            break ;
        case GL_COLOR_ARRAY :
            glColorPointer( num_vals_tupla, tipo_valores, 0, ptr_offset );
            break ;
        case GL_NORMAL_ARRAY :
            glNormalPointer( tipo_valores, 0, ptr_offset );
            break ;
    }
}
```

## Activación de tablas genéricas en M.I. usando DescrTabla

Este otro método *activa* una tabla de atributos en M.I. Activar aquí quiere decir indicarle a OpenGL que la debe usar y especificar la dirección y formato de la tabla:

```
void DescrTabla::activar_mi()
{
    assert( tipo_tabla == GL_ARRAY_BUFFER ); // NO sirve para tabla de índices
    fijarPuntero( datos ); // especifica localización y formato
    glEnableClientState( atributo ); // habilita el uso de la tabla en OpenGL
}
```

Este método no se debe invocar para la tabla de índices (no es necesario en M.I., ya que los datos de la tabla de índices se especifican en la llamada a la función **glDrawElements**)

## Visu. en M.I. con llamada única de objetos **ArrayVertices** (1/2)

La clase **ArrayVertices** se puede usar para visualizar en el caso genérico (tp es el tipo de primitiva).

```
void ArrayVertices::visualizarGL_MI_DAE( const GLenum tp )
{
    glBindVertexArray( 0 );      // activa VAO por defecto
    deshabilitar_tablas();      // deshab. todas las tablas
    coordenadas->activar_mi(); // activar la tabla de coordenadas

    // activar las tablas de atributos que corresponda (en modo inmediato)
    if ( colores != nullptr ) colores->activar_mi() ;
    if ( normales != nullptr ) normales->activar_mi() ;
    if ( coords_textura != nullptr ) coords_textura->activar_mi() ;

    // invocar 'glDrawElements' o 'glDrawArrays' en función de si hay o no hay índices
    if ( indices != nullptr )
        glDrawElements( tp, num_indices, indices->tipo_valores, indices->datos );
    else
        glDrawArrays( tp, 0, num_vertices );

    // dejar tablas deshabilitadas para visualizar otros arrays de vértices
    deshabilitar_tablas();
}
```

## Visu. en M.I. con llamada única de objetos **ArrayVertices** (1/2)

El método **deshabilitar\_tablas** usa **glDisableClientState** para deshabilitar todas las tablas de atributos.

```
void ArrayVertices::deshabilitar_tablas()
{
    glDisableClientState( GL_VERTEX_ARRAY );
    glDisableClientState( GL_COLOR_ARRAY );
    glDisableClientState( GL_TEXTURE_COORD_ARRAY );
    glDisableClientState( GL_NORMAL_ARRAY );
}
```

Debe usarse al inicio y al final de **visualizarGL\_MI\_DAE** para desahabilitar las tablas anteriormente habilitadas (al inicio) y para que no se quede ninguna habilitada (al final).

## Funciones de envío 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 en la GPU,
  - ▶ Almacenar la información de la localización y formato de las tablas en un VAO,
- ▶ En cada visualización solo es necesario activar el VAO (con **glBindVertexArray**) y visualizar con una única llamada (con **glDrawArrays** o **glDrawElements**).

El modo diferido es **mucho más eficiente en tiempo** que el modo inmediato.

# Ejemplo en modo diferido con glDrawArrays (1/2)

Ejemplo de una secuencia con coordenadas y colores, no indexada.  
Declaraciones y visualización:

```
const unsigned num_verts = ..... ; // número de vértices
GLenum    id_vao      = 0 ; // identificador de VAO, (0 hasta 1a visu.)
GLfloat  coordenadas[ num_verts*2 ], colores[ num_verts*3 ] ; // tablas

if ( id_vao == 0 ) // si el VAO no está creado, crearlo (una sola vez)
{
    glGenVertexArrays( 1, &id_vao ); // crear VAO
    glBindVertexArray( id_vao ); // activa VAO
    glDisableClientState( GL_NORMAL_ARRAY ); // no usaremos normales
    glDisableClientState( GL_TEXTURE_COORD_ARRAY ); // no usamos cc.t.
    // crear y activar VBO de coordenadas dentro del VAO .....
    // crear y activar VBO de colores dentro del VAO .....
}
else
    glBindVertexArray( id_vao ); // VAO ya creado: activarlo

// dibujar y desactivar el VAO
glDrawArrays( GL_POLYGON, 0, num_verts );
glBindVertexArray( 0 );
```

## Ejemplo en modo diferido con glDrawArrays (2/2)

La creación de los VBOs se haría así para este ejemplo:

```
// crear y activar VBO de coordenadas dentro del VAO .....
GLenum id_vbo_coordenadas ;
glGenBuffers( 1, &id_vbo_coordenadas ); // crea VBO verts.
glBindBuffer( GL_ARRAY_BUFFER, id_vbo_coordenadas ); // activa VBO verts.
glBufferData( GL_ARRAY_BUFFER, 2*num_verts*sizeof(float), // copia:
              coordenadas, GL_STATIC_DRAW ); // RAM ->GPU
glVertexPointer( 2, GL_FLOAT, 0, 0 ); // indica puntero a array de coordenadas
glBindBuffer( GL_ARRAY_BUFFER, 0 );
glEnableClientState( GL_VERTEX_ARRAY ); // habilita uso de array de coordenadas

// crear y activar VBO de colores dentro del VAO ...
GLenum id_vbo_colores ;
glGenBuffers( 1, &id_vbo_colores ); // crea VBO colores
glBindBuffer( GL_ARRAY_BUFFER, id_vbo_colores ); // activa VBO colores
glBufferData( GL_ARRAY_BUFFER, 3*num_verts*sizeof(float), // copia:
              colores, GL_STATIC_DRAW ); // RAM ->GPU
glColorPointer( 3, GL_FLOAT, 0, 0 ); // indica puntero a array de colores
glBindBuffer( GL_ARRAY_BUFFER, 0 );
glEnableClientState( GL_COLOR_ARRAY ); // habilita uso de array de colores
```

# Ejemplo en modo diferido con glDrawElements (1/2)

Ejemplo de una secuencia con coordenadas y colores, indexada:

```
const unsigned num_verts = ..... , num_inds = ... ;
GLenum id_vao = 0 ; // identificador de VAO, (0 hasta 1a visu.)
GLfloat coordenadas[ num_verts*2 ], colores[ num_verts*3 ]; // tablas
GLuint indices[ num_inds ] ; // tabla de índices (unsigned int)

if ( id_vao == 0 ) // si el VAO no está creado, crearlo (una sola vez)
{
    glGenVertexArrays( 1, &id_vao ); // crear VAO
    glBindVertexArray( id_vao ); // activa VAO
    glDisableClientState( GL_NORMAL_ARRAY ); // no usaremos normales
    glDisableClientState( GL_TEXTURE_COORD_ARRAY ); // no usamos cc.t.
    // crear y activar VBO de coordenadas dentro del VAO ....
    // crear y activar VBO de colores dentro del VAO ....
    // crear y activar VBO de índices dentro del VAO ....
}
else
    glBindVertexArray( id_vao ); // VAO ya creado: activarlo

// dibujar y desactivar el VAO
glDrawElements( GL_POLYGON, num_inds, GL_UNSIGNED_INT, 0 ); // visu.
glBindVertexArray( 0 );
```

## Ejemplo en modo diferido con `glDrawElements` (2/2)

La creación de los VBOs de vértices y colores se haría igual que para la secuencia no indexada. Además, habría que activar la tabla de índices, justo después de las otras:

```
// crear y activar VBO de índices dentro del VAO:  
GLenum id_vbo_indices ;  
 glGenBuffers( 1, &id_vbo_indices ); // crea VBO indices  
 glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, id_vbo_indices ); // activa VBO inds.  
 glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, num_inds*sizeof(unsigned), // copia:  
               indices, GL_STATIC_DRAW ); // RAM ->GPU
```

Es importante que la última llamada a **glBindBuffer** haya sido la de los índices, de forma que el VBO de índices quede activado en el VAO antes de visualizar con **glDrawElements**

# Problemas: visualización en modo diferido

## Problema 1.5.

Crea una copia del repositorio **opengl-minimo**, y modifica el código de ese repositorio según se indica en el enunciado del problema 1.3, pero ahora define una función que use el **modo diferido**, en lugar del modo inmediato.

Minimiza las llamadas a OpenGL y transferencias de datos entre CPU y GPU. Ten en cuenta que puedes usar una variable local **static** también para el identificador del VAO, de forma que dicho VAO se cree una única vez en la primera llamada.

## Problema 1.6.

De forma similar al problema anterior, modifica ahora el código de **opengl-minimo** para hacer de forma eficiente la visualización descrita en el enunciado del problema 1.4, pero ahora usando el **modo diferido**.

# Activación en M.D. de objetos DescrTabla

En el caso genérico, activar una tabla es lo mismo que en modo inmediato, pero además, en la primera activación debemos de crear el VBO en la memoria de la GPU y transferir los datos:

```
void DescrTabla::activar_md()
{
    // si el VBO no está creado todavía en la GPU, crearlo y transferir los datos
    if ( nombre_vbo == 0 )
    {
        glGenBuffers( 1, &nombre_vbo );           // crear un nuevo nombre de VBO
        glBindBuffer( tipo_tabla, nombre_vbo ); // activar el nuevo VBO
        // reservar memoria en la GPU y transferir bytes de la tabla
        glBufferData( tipo_tabla, tamano_en_bytes, datos, GL_STATIC_DRAW );
    }
    else // el VBO ya está creado
        glBindBuffer( tipo_tabla, nombre_vbo ); // activar el VBO ya creado

    if ( tipo_tabla == GL_ARRAY_BUFFER ) // si es tabla de atributos:
    {
        fijarPuntero( 0 );                  // especificar loc. y formato
        glBindBuffer( tipo_tabla, 0 );       // desactiva el VBO
        glEnableClientState( atributo );   // habilitar uso de la tabla
    }
}
```

# Visu. en M.D. de objetos ArrayVertices

Es necesario crear el VAO la primera vez, después solo activarlo.

```
void ArrayVertices::visualizarGL_MD_VAO( const GLenum tp )
{
    if ( nombre_vao == 0 ) // si el VAO no estaba creado
    { glGenVertexArrays( 1, &nombre_vao ); // crea el nuevo VAO
        glBindVertexArray( nombre_vao ); // activa nuevo VAO
        // activar las tablas que proceda (incluyendo índices)
        coordenadas->activar_md();
        if ( colores != nullptr ) colores->activar_md();
        if ( normales != nullptr ) normales->activar_md();
        if ( coords_textura != nullptr ) coords_textura->activar_md();
        if ( indices != nullptr ) indices->activar_md();
    }
    else // el VAO ya estaba creado
        glBindVertexArray( nombre_vao ); // activarlo
    // invocar 'glDrawElements' o 'glDrawArrays' en función de si hay indices o no.
    if ( indices != nullptr )
        glDrawElements( tp, num_indices, indices->tipo_valores, 0 );
    else
        glDrawArrays( tp, 0, num_vertices );
    glBindVertexArray( 0 ); // deshabilitar el VAO,
}
```

Informática Gráfica, curso 2021-22.

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)

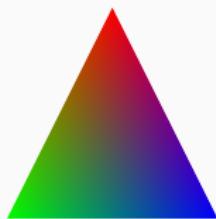
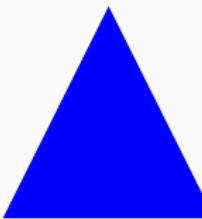
El modo de visualización de polígonos se cambia usando la llamada:

```
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

La función **glShadeModel** permite cambiar el modo actual de sombreado, usado para las siguientes primitivas de tipo línea o polígonos llenos:



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

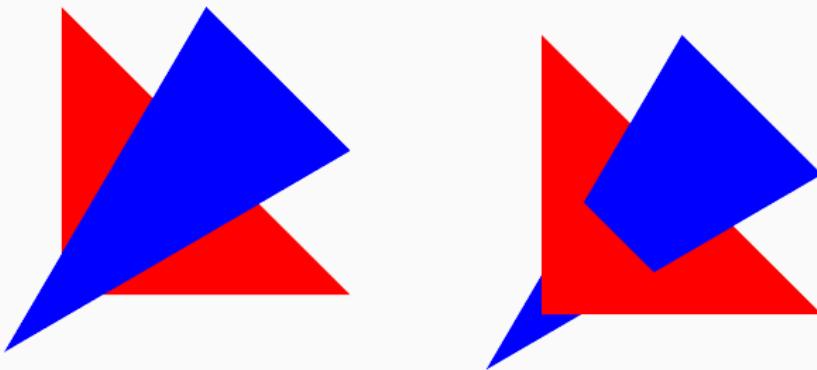
## Eliminación de partes ocultas (EPO) con Z-buffer

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).
  - ▶ **Ancho** (en pixels) de los puntos (real).
- ▶ Otros atributos:
  - ▶ **Color** que será usado cuando se limpie la ventana (antes de dibujar) (RGBA).

# 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)
    CError();
    // establecer color de fondo: (1,1,1) (blanco)
    glColor3f( 1.0, 1.0, 1.0, 1.0 );
    // establecer color inicial para todas las primitivas, hasta que se cambie
    glColor3f( 0.7, 0.2, 0.4 );
    // establecer ancho de líneas o segmentos (en pixels)
    glLineWidth( 2.0 );
    // establecer diámetro de los puntos (en pixels)
    glPointSize( 3.0 );
    // habilitar eliminación de partes ocultas usando el Z-buffer
    glEnable( GL_DEPTH_TEST );
    // comprobar si ha habido algún error en esta función
    CError();
}
```

## 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.

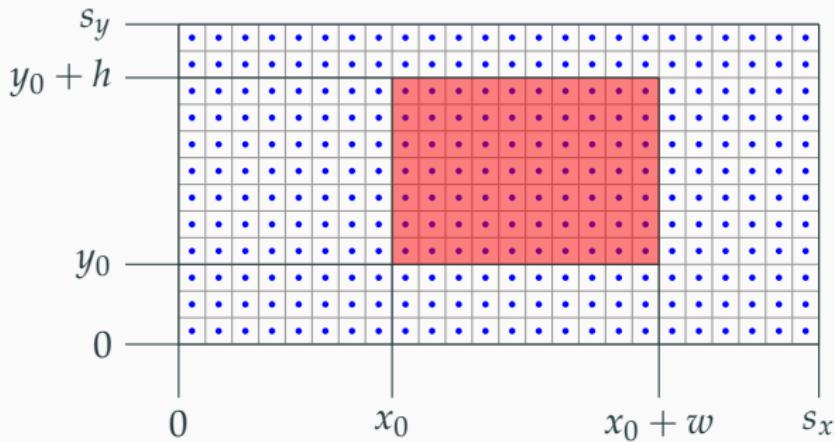
```
glViewport( izqui, abajo, ancho, alto ) ;
```

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

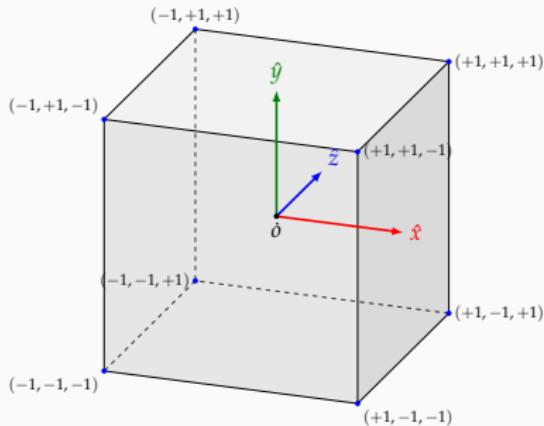
El evento de cambio de tamaño de la ventana se produce siempre una vez tras crear la ventana, y además siempre después de que se cambie su tamaño.

- ▶ 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

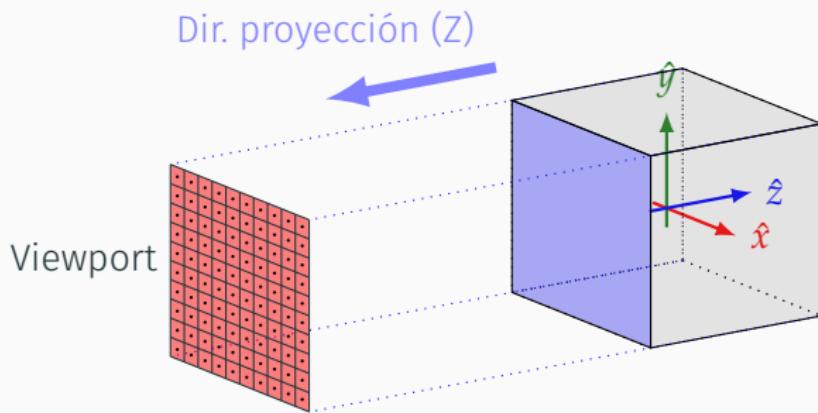
La **región visible** en pantalla es (initialmente) un cubo de lado 2 y centro en el origen (ocupa el intervalo  $[-1, 1]$  en los tres ejes):



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

# Región visible y proyección sobre el viewport

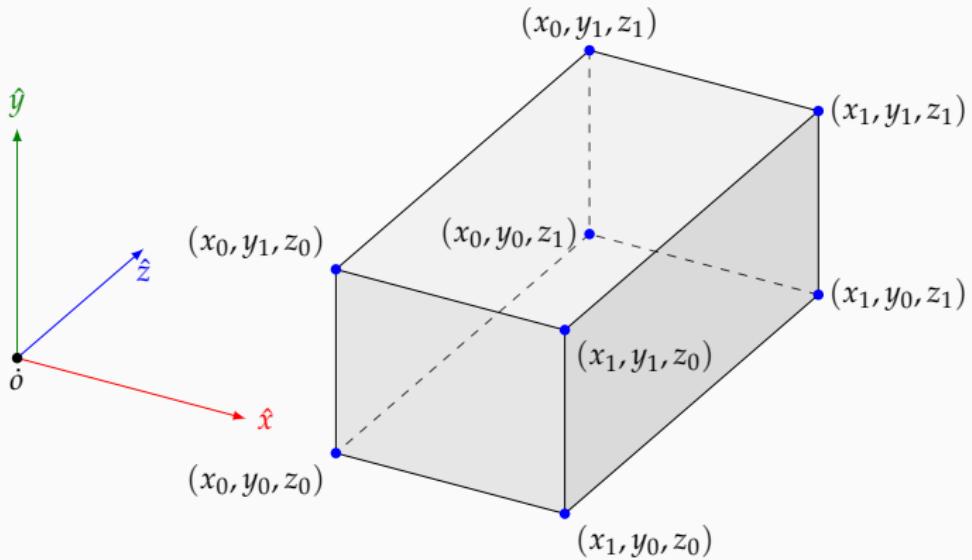
Inicialmente, OpenGL usa una proyección paralela al eje Z, hacia la rama negativa de dicho eje. Podemos imaginar los pixels de *viewport* sobre la *cara delantera* (en azul) del cubo:



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  $4 \times 4$  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  
};
```

En el cauce fijo se puede hacer así:

```
glMatrixMode( GL_PROJECTION );  
glLoadIdentity();  
glMultTransposeMatrixf( matriz_proyeccion );
```

En el cauce programable debemos usar una variable (`loc_proyeccion`) con un entero que identifica a la matriz de proyección:

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

# Problema: definición de la región visible sin deformaciones

## Problema 1.7.

Modifica el código del ejemplo **opengl-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)
    objeto1.visualizarGL_MD_VAO( tp1 );
    objeto2.visualizarGL_MD_VAO( tp2 );
    ....
    // 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

Para verificar los errores y obtener un mensaje descriptivo se puede invocar una macro como **CError()**. Se declara como:

```
#define CError() CompruebaErrorOpenGL(__FILE__,__LINE__)
void CompruebaErrorOpenGL( const char * nomArchivo, int linea ) ;
```

y se define así:

```
void CompruebaErrorOpenGL( const char * nomArchivo, int linea )
{ const GLint codigoError = glGetError() ;
  if ( codigoError != GL_NO_ERROR )
  { cout
    << endl
    << "Detectado error de OpenGL. Programa abortado." << endl
    << "  linea      : " << linea << endl
    << "  archivo    : " << nomArchivo << endl
    << "  descripcion : " << gluErrorString(codigoError) << endl
    << endl << flush ;
    exit(1);
}
```

# Documentación on-line sobre OpenGL y GLFW

- ▶ Páginas de referencia de OpenGL (y GLU)
  - ▶ Versión 2.1: ↗ [www.opengl.org/sdk/docs/man2](http://www.opengl.org/sdk/docs/man2)
  - ▶ Versión 3.3: ↗ [www.opengl.org/sdk/docs/man3](http://www.opengl.org/sdk/docs/man3)
  - ▶ Versión 4.0: ↗ [www.opengl.org/sdk/docs/man](http://www.opengl.org/sdk/docs/man)
- ▶ OpenGL Programming Guide (*the red book*)
  - ▶ OpenGL 1.1 (en html): ↗ [www.glprogramming.com/red/](http://www.glprogramming.com/red/)
- ▶ Registry (documentos de especificación oficiales de OpenGL):
  - ▶ Actuales (ver 4.6): ↗ [www.opengl.org/registry/#apispecs](http://www.opengl.org/registry/#apispecs)
  - ▶ Versiones anteriores: ↗ [www.opengl.org/registry/#oldspecs](http://www.opengl.org/registry/#oldspecs)
- ▶ Librería GLFW (documentación, código fuente, binarios)
  - ▶ Sitio web: ↗ [www.glfw.org](http://www.glfw.org)
  - ▶ Documentación: ↗ [www.glfw.org/documentation.html](http://www.glfw.org/documentation.html)
- ▶ Página de referencia de GLSL:
  - ▶ Todas las ver.: ↗ [www.opengl.org/sdk/docs/manglsl/](http://www.opengl.org/sdk/docs/manglsl/)

## 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 2021-22.

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 cálculo de OpenGL que (usualmente) se ejecutan en la GPU o la librería gráfica:

## 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 recorte de polígonos.

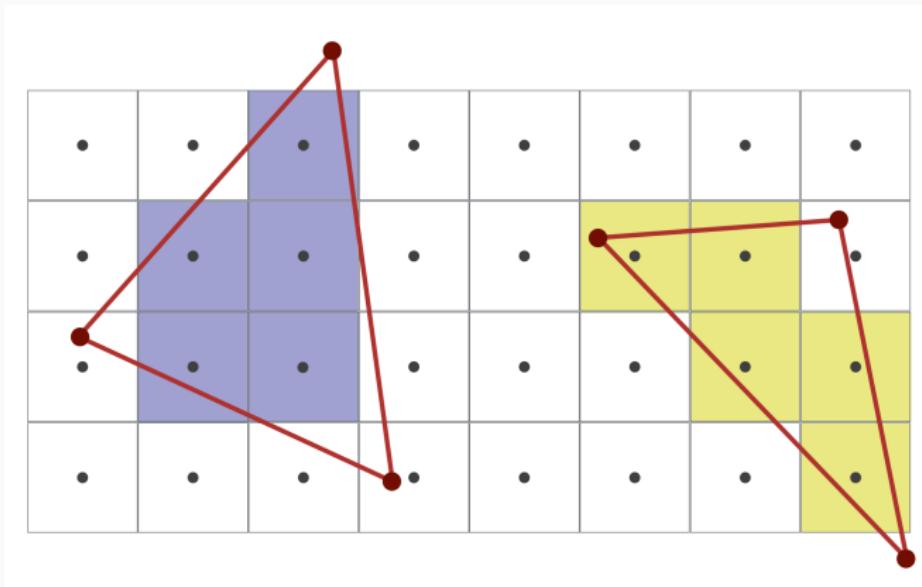
# Los shaders. Tipos.

Los *shaders* son programas que hacen transformación y sombreado.  
Hay dos tipos:

1. **Procesador de vértices (vertex shader):** subprograma encargado de la transformación de coordenadas.
  - ▶ Se ejecuta cada vez que se especifica una coordenada de un vértice nuevo (con **glVertex**, **glDrawArrays** u otras llamadas).
  - ▶ 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**.

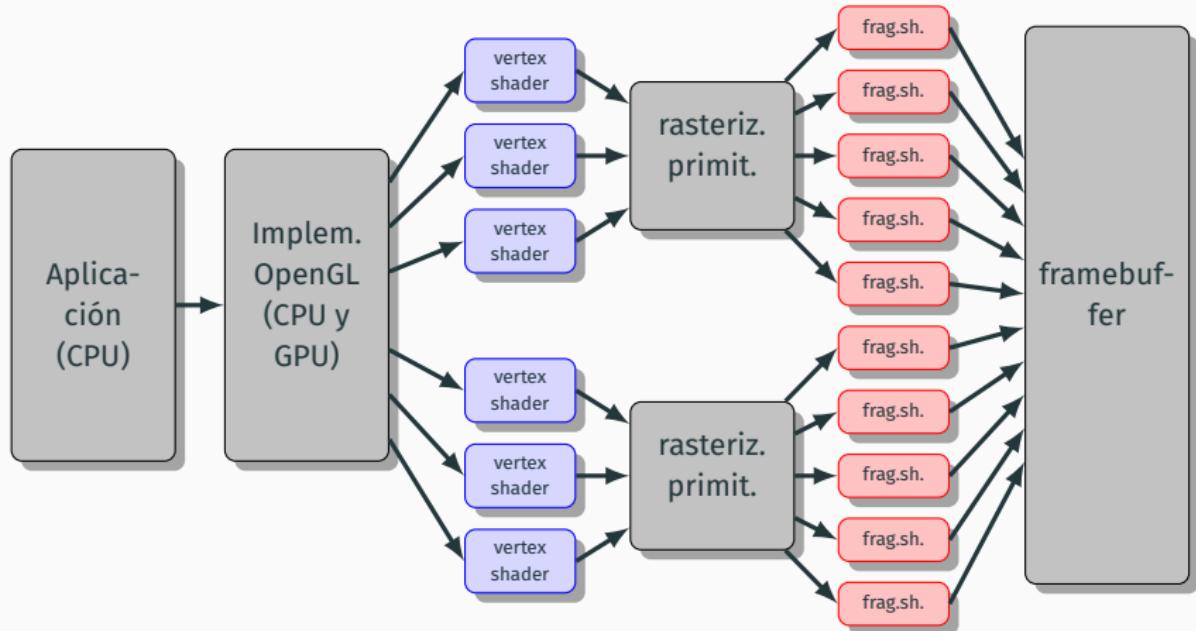
## Ejemplo: Visualización de 2 triángulos

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:



# 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.

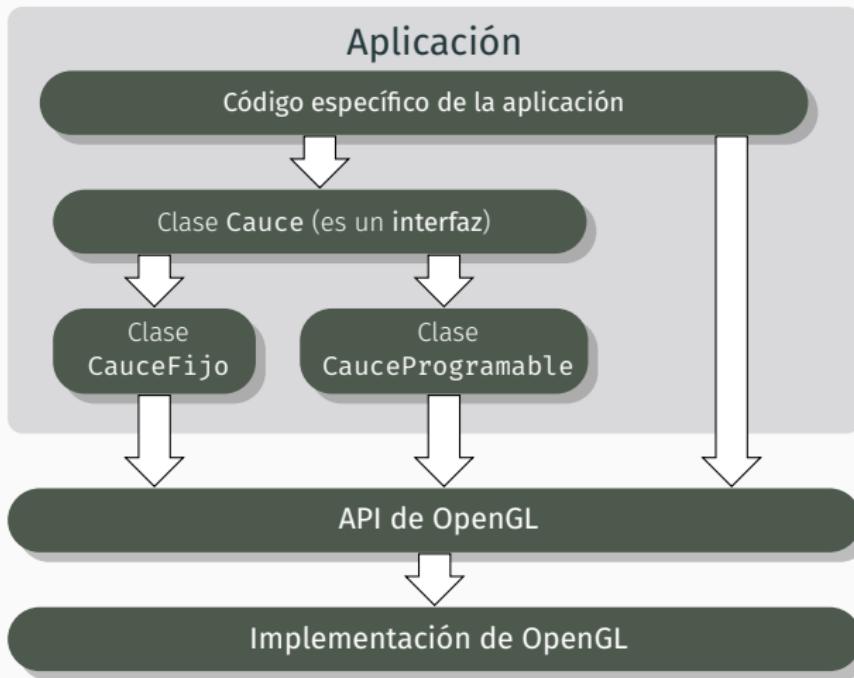
# Implementación de cauce fijo/programable

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 abstracta (**Cauce**) para realizar un acceso uniforme a ambos tipos de cauce. Esta clase incluye **métodos virtuales puros**.
- ▶ De esa clase se derivan dos clases: **CauceFijo** y **CauceProgramable**, cada una de ellas contiene implementaciones distintas de cada método virtual de la clase base **Cauce**.

# Clases para acceso al cauce

Para cada visualización de un frame, la aplicación de prácticas puede usar uno de los dos cauces (el que esté activo)



## Ejemplo: fijar evaluación de iluminación

A modo de ejemplo, vemos como se cambia una variable de estado que indica si se debe evaluar iluminación o no se debe.

En el cauce fijo se usan las funciones **glEnable** y **glDisable**:

```
void CauceFijo::fijarEvalMIL( const bool nue_eval_mil )  
{  
    if ( nue_eval_mil ) glEnable( GL_LIGHTING );  
    else                 glDisable( GL_LIGHTING );  
}
```

En el cauce programable se cambia un parámetro de los shaders, usando **glUniform**:

```
void CauceProgramable::fijarEvalMIL( const bool nue_eval_mil )  
{  
    glUseProgram( id_prog );  
    glUniform1i( loc_eval_mil, nue_eval_mil ? 1 : 0 );  
}
```

Informática Gráfica, curso 2021-22.

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. En GLSL moderno se llaman variables **out** (en el vertex shader) o variables **in** (en el fragment shader).
- ▶ Atributos de vértices: no son siempre necesarias: podemos usar declaraciones implícitas.
- ▶ 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)

- ▶ Entrada: **parámetros uniform** y la posición y atributos de cada vértice enviados por la aplicación en las variables predefinidas: **gl\_Vertex**, **gl\_Color**, **gl\_Normal** y **gl\_MultiTexCoord0**.
- ▶ Salida: **variables varying** (**u out**).

### Fragment Shaders (se ejecutan una vez por fragmento o pixel)

- ▶ Entrada: **parámetros uniform** y **variables varying** (o **in**) ya interpoladas en cada vértice).
- ▶ Salida: variable predefinida **gl\_FragColor** (color del pixel).

## Ejemplo de *shaders* mínimos: declaraciones

En el repositorio de github ([opengl-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: código GLSL

Vertex shader:

```
#version 120          // indica que se trata de código GLSL versión 1.2
uniform mat4 u_proyeccion; // matriz 4x4 'modelview' (transf. de coordenadas).
uniform mat4 u_modelview; // matriz 4x4 de proyección
varying vec4 v_color ;    // var. de salida: color del vértice

void main()
{
    // usa variables de entrada: gl_Color, gl_Vertex, enviadas por la aplic.
    v_color      = gl_Color ;    // copia color de entrada en color de salida
    gl_Position = u_proyeccion * u_modelview * gl_Vertex; // transf. coords.
}
```

Fragment shader:

```
#version 120          // indica que se trata de código GLSL versión 1.2
varying vec4 v_color ; // variable de entrada: color interpolado en el pixel.

void main()
{
    gl_FragColor = v_color ; // copia color interpolado en el color del pixel
}
```

## 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 (**matrizP**) que tiene en cuenta la perspectiva.
- ▶ Además de la matriz *modelview* para coordenadas (**matrizMV**), hay una versión de esa matriz para las normales (**matrizMV\_nor**).
- ▶ Incorporación de texturas: usa coordenadas de textura y consulta de la imagen de textura (función **texture2D**).
- ▶ Simulación de iluminación en 3D: usa las normales, se debe evaluar un *modelo de iluminación local*, en la función **EvaluarMIL**, no especificada.

## Ejemplo de *Vertex Shader* más complejo

```
// parámetros de entrada uniform (iguales en todos los vértices de cada primitiva)
uniform mat4 matrizMV ;           // matriz 4x4 de transf. de coord. de vértices
uniform mat4 matrizMV_nor;        // matriz 4x4 de transf. de normales
uniform mat4 matrizP ;           // matriz 4x4 de proyección (produce coord.pantalla)

// variables de salida varying (atributos de vértice: serán interpolados a pixels)
varying vec4 var_posic_ec;       // posición (en coords de cámara)
varying vec3 var_normal_ec;      // normal (en coords. de camara)
varying vec4 var_color;          // color
varying vec2 var_coord_text;     // coordenadas de textura

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord0

void main() // escribe variables 'varying', más 'gl_Position'
{
    var_posic_ec = matrizMV * gl_Vertex;           // transf. coord. recibida
    var_normal_ec = matrizMV_nor * gl_Normal;        // transf. normal recibida
    var_color = gl_Color ;                         // usar color enviado
    var_coord_text = gl_MultiTexCoord0.st ;         // usa cc.t. enviadas
    gl_Position = matrizP * var_posic_ec;           // proyecta a pantalla
}
```

# Ejemplo de *Fragment Shader* más complejo

```
// parámetros de entrada uniform (iguales en todos los pixels de cada primitiva)
uniform int eval_mil;      // evaluar el MIL si (1) o no (0)
uniform int sombr_plano;   // 0 -> usar smooth shading, 1 -> usar flat shading
uniform int eval_text;     // 0 -> no texturas, 1 -> evaluar textura
uniform sampler2D textu;   // objeto asociado a textura activa
.....
// variables de entrada varying (interpolados en este pixel)
varying vec4 var_posic_ec; // posición (en coords de cámara)
varying vec3 var_normal_ec; // normal (en coords. de camara)
varying vec4 var_color;    // color
varying vec2 var_coord_text; // coordenadas de textura

// función que evalua la iluminación (usa variables 'varying')
vec4 EvaluarMIL() { .... }

void main() // escribe variable 'gl_FragColor' (color final del pixel)
{
    if ( eval_mil == 1 )           // si activada iluminación
        gl_FragColor = EvaluarMIL(); //      calcular iluminación
    else if ( eval_text == 1 )      // si activadas texturas
        gl_FragColor = texture2D( textu, var_coord_text ); // consultar textura
    else                          // si no activadas ilum. ni texturas
        gl_FragColor = var_color ; //      usar color interpolado
}
```

Informática Gráfica, curso 2021-22.

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 **CauceProgramable** crea un programa y almacena el identificador en **id\_prog**:

```
CauceProgramable::CauceProgramable()
{
    // inicializar los nombres de los archivos fuente:
    frag_fn = "../recursos/shaders/cauce21-frag.glsl" ;
    vert_fn = "../recursos/shaders/cauce21-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 cauces programable y fijo
    Cauce * cauce_prog = new CauceProgramable(),
           * cauce_fijo = new CauceFijo(),
           * cauce      = ..... ; // usar uno de los dos.
}
```

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 las clases **CauceFijo** y **CauceProgramable**, ya citadas.

# Uso de la clase Cauce y derivadas

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

```
void CauceFijo::activar() { glUseProgram( 0 ); }
void CauceProgramable::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 2021-22.

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 (en macOS la función no hace nada, es innecesario).

```
void Inicializar_GLEW()
{
    // 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);
    }
    // comprobar si OpenGL ver 2.0 + está soportado
    if ( ! GLEW_VERSION_2_0 )
    {   cout << "OpenGL 2.0 no soportado." << endl << flush ;
        exit(1);
    }
}
```

## 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
- 5.8. Representación y operaciones sobre matrices.

Informática Gráfica, curso 2021-22.

Teoría. Tema 1. Introducción.

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

Subsección 5.1.

Puntos y vectores.

## Puntos y vectores

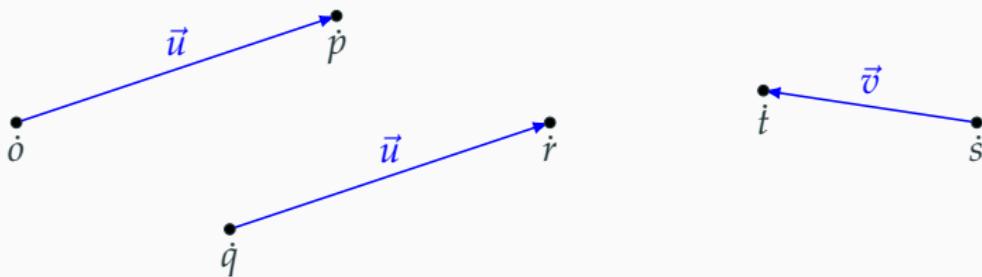
Los modelos 3D y 2D de objetos y figuras que vamos a representar se pueden construir cada uno de ellos en base a una conjunto abstracto (con estructura de **espacio afín**), cuyos elementos son puntos de un determinado espacio donde imaginamos el modelo. Cada uno de estos **puntos o localizaciones**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



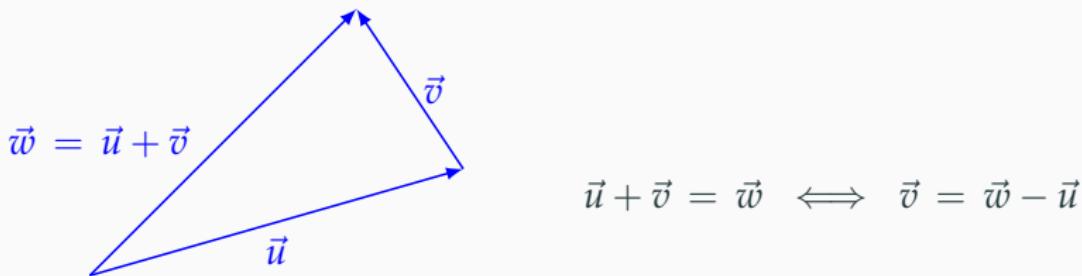
- ▶ Cada vector va asociado a la distancia y la dirección entre un punto y otro. El vector de  $\dot{p}$  a  $\dot{q}$  se escribe como  $\dot{q} - \dot{p}$ .
- ▶ Dos pares distintos de puntos pueden tener asociado el mismo vector (los pares  $\dot{o}, \dot{p}$  y  $\dot{q}, \dot{r}$  tienen ambos asociado 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:  $\dot{p} + \vec{w} = (\dot{p} + \vec{u}) + \vec{v}$ .



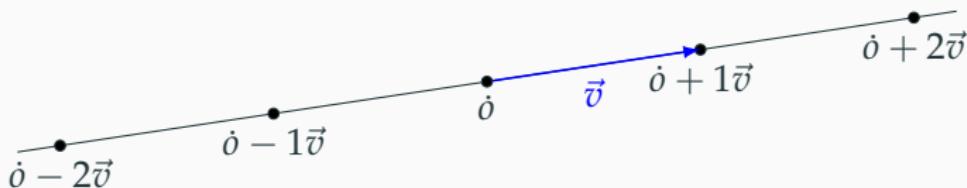
El **vector nulo** lo notamos como  $\vec{0}$ , y se define como  $\vec{0} = \dot{p} - \dot{p}$  (para cualquier punto  $\dot{p}$ ).

# 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 2021-22.

Teoría. Tema 1. Introducción.

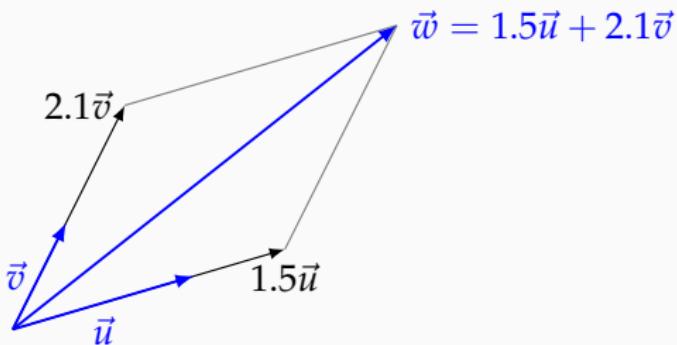
Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Subsección 5.2.

Marcos de referencia y coordenadas.

## Bases de vectores

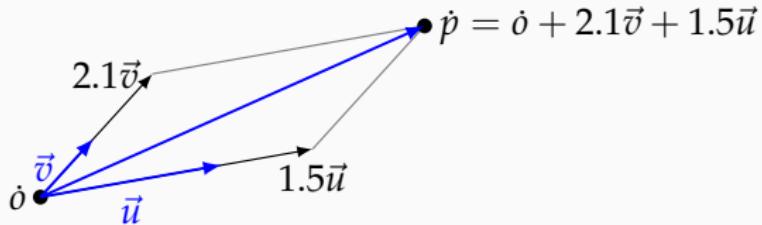
Usando dos vectores cualesquiera  $\vec{u}$  y  $\vec{v}$  del plano (no paralelos ni nulos), podemos escribir cualquier otro vector  $\vec{w}$  como una combinación lineal de ellos:



- ▶ 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

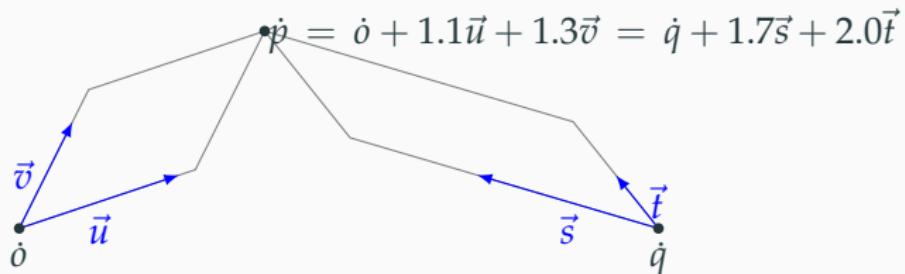
Si fijamos un punto  $\dot{o}$  (origen) y una base  $\{\vec{u}, \vec{v}\}$ , cualquier punto  $\dot{p}$  del plano se puede escribir como  $\dot{p} = \dot{o} + a\vec{u} + b\vec{v}$ :



- ▶ 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 2021-22.

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 2021-22.

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:

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

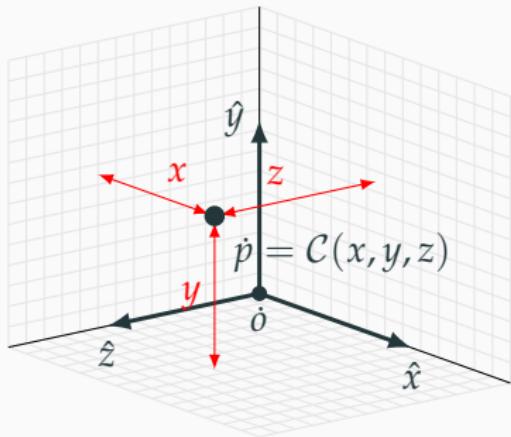
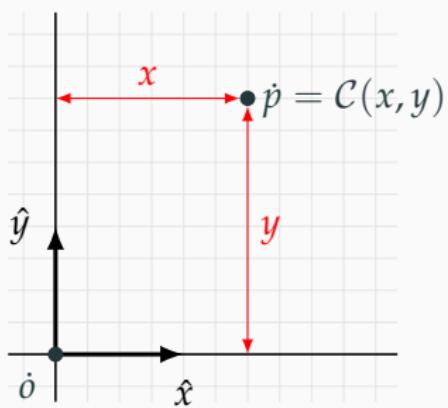
- ▶ La **orientación** es semejante a la de  $\mathcal{E}$ , es decir:

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

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:

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

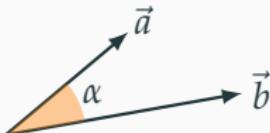
- ▶ El módulo de un vector coincide con su **longitud** en el espacio (ya que de los versores de  $\mathcal{E}$  dijimos que tenían longitud unidad por definición). El módulo, calculado así, es siempre el mismo en cualquier marco cartesiano.

# Ángulo entre vectores

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

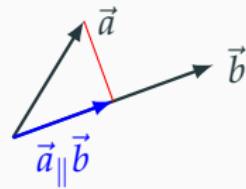
- El **ángulo  $\alpha$**  entre  $\vec{a}$  y  $\vec{b}$  se define como el arco cuyo coseno es el producto escalar de  $\vec{a}/\|\vec{a}\|$  y  $\vec{b}/\|\vec{b}\|$ . Se cumple:

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



- 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  $\alpha$  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:

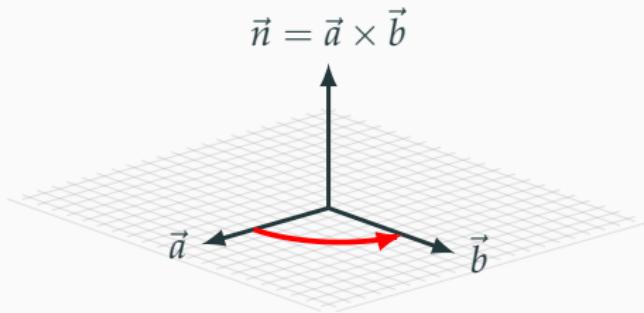
$$\mathcal{C} \begin{pmatrix} a_x \\ a_y \\ a_z \\ 0 \end{pmatrix} \times \mathcal{C} \begin{pmatrix} b_x \\ b_y \\ b_z \\ 0 \end{pmatrix} = \mathcal{C} \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \\ 0 \end{pmatrix}$$

- ▶ 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 y vectorial

## 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.

## 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

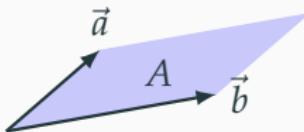
La longitud del vector obtenido como producto vectorial de otros dos está relacionada con el área entre esos otros dos vectores:

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

Informática Gráfica, curso 2021-22.

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:

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

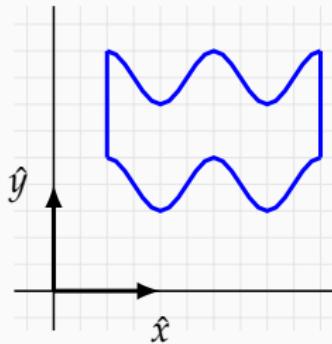
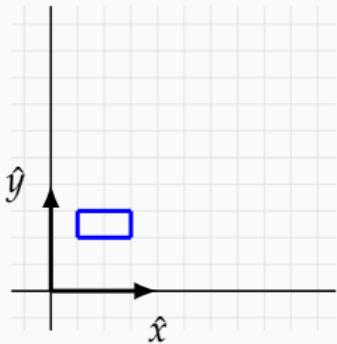
Lógicamente, para una única transformación  $T$ , las funciones  $f_0$ ,  $f_1$  y  $f_2$  dependen del sistema de referencia  $\mathcal{R}$  en uso.

## 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:

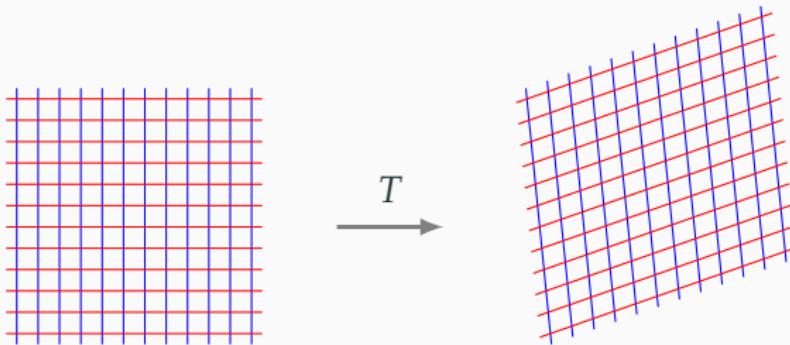
$$f_0(x, y) = 4x - 1 \quad f_1(x, y) = 2y + \frac{2 + \cos((8x - 2)\pi)}{4}$$

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 2021-22.

Teoría. Tema 1. Introducción.

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

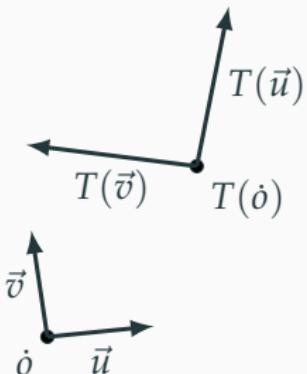
Subsección 5.6.  
Matrices de transformación.

# 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  $4 \times 4$ , mientras que en 2D será una matriz  $3 \times 3$ .
- ▶ 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

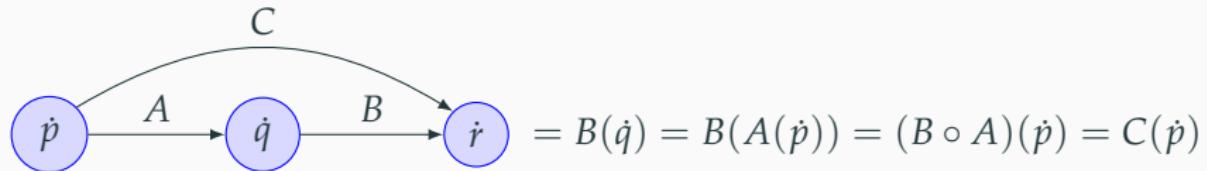
El uso de coordenadas homogéneas permite unificar la matriz  $R$  y la matriz  $D$  en una única matriz 4x4

- ▶ 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

$$C = B \circ A:$$



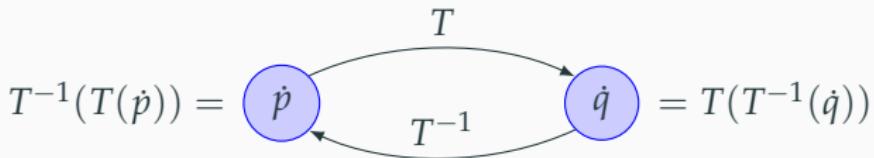
La composición es, en general, **no conmutativa**. Se puede extender a 3 o más transformaciones:

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

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

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

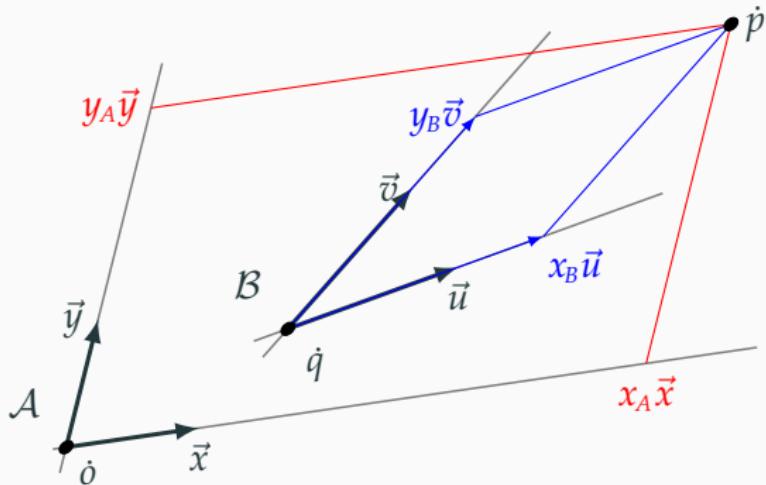
La matriz  $M_U$  asociada a  $U$  en un marco  $\mathcal{R}$  será el producto de las matrices  $M_i$  asociadas a  $T_i$  en ese marco: asociadas a cada una de las  $T_i$ :

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

(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:

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

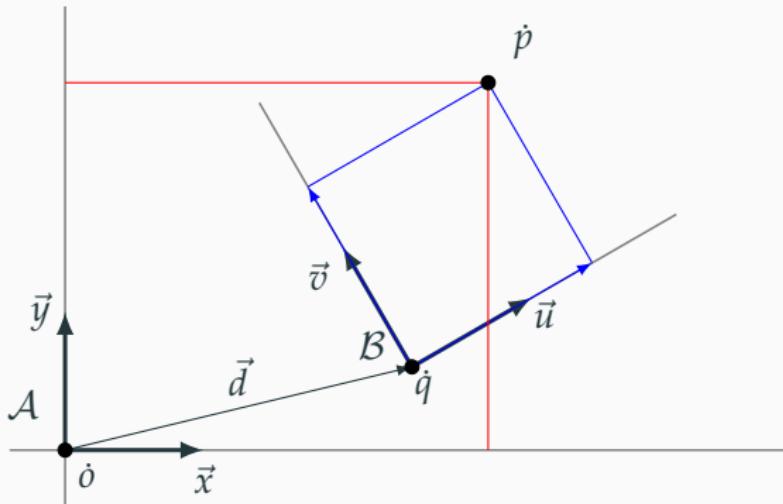
la matriz  $D_{\mathcal{A}, \mathcal{B}}^{-1}$  es un desplazamiento que lleva  $\dot{q}$  a  $\dot{o}$ , es decir, un desplazamiento por el vector  $\dot{o} - \dot{q} = \mathcal{A}(-d_x, -d_y, -d_z, 0)^t$ .

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:



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 2021-22.

Teoría. Tema 1. Introducción.

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

Subsección 5.7.

Representación y operaciones con tuplas.

# Representación en memoria de coordenadas.

Para representar en memoria las tuplas de coordenadas (como tipos-valor), podemos usar una *plantilla de clase*, como las del archivo **tuplag.hpp**. A partir de la plantilla, se declaran (entre otras) estas clases (tipos)

```
// 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 2021-22.

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.



UNIVERSIDAD  
DE GRANADA

# Informática Gráfica: Teoría. Tema 2. Modelos de objetos.

---

Carlos Ureña

2021-22

**Grado en Informática y Matemáticas**  
Dpt. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

## Teoría. Tema 2. Modelos de objetos.

### Índice.

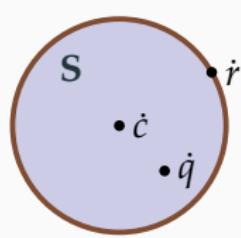
1. Modelos geométricos. Introducción.
2. Modelos de fronteras: mallas de polígonos.
3. Representación en memoria de modelos de fronteras.
4. Transformaciones geométricas
5. Modelos jerárquicos. Representación y visualización.

Sección 1.  
Modelos geométricos. Introducción..

# Modelos geométricos formales

Un **modelo geométrico** es un modelo matemático que sirve para representar un objeto geométrico que existe en un espacio afín  $\mathbf{E}$  (2D o 3D).

Los modelos geométricos matemáticos más generales posibles son los **subconjuntos de puntos** de  $\mathbf{E}$ , por ejemplo, una esfera:



$$\begin{aligned} & \bullet s \\ & \bullet c \\ & \bullet r \\ & \bullet q \\ \mathbf{S} = & \{ p \in \mathbf{E} \text{ t.q. } \|p - c\| \leq 1 \} \\ p \in & \mathbf{S} \\ r \in & \mathbf{S} \quad r \in \partial \mathbf{S} \\ s \notin & \mathbf{S} \end{aligned}$$

Cada subconjunto o **región  $S$** :

- ▶ Es **cerrado** (incluye a su propia **superficie** o **frontera**,  $\partial S$ ),
- ▶ Es **acotado** (no tiene extensión infinita),
- ▶ Su superficie es **diferenciable** (**plana** a escala muy pequeña)

# Representaciones computacionales

El modelo basado en subconjuntos de puntos del espacio:

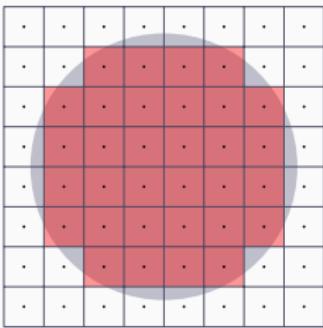
- ▶ permite representar matemáticamente cualquier objeto (es el modelo **más general** posible)
- ▶ pero **no se puede representar en la memoria** (finita, discreta) de un ordenador

Ante esto hay representaciones aproximadas pero que usan una cantidad finita de memoria (**modelos geométricos computacionales**), se usan básicamente dos opciones :

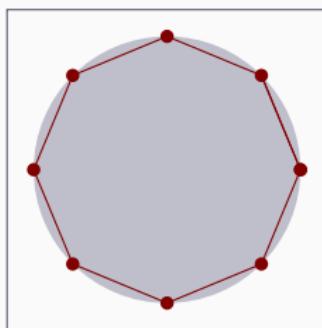
- ▶ **Enumeración espacial:** se partitiona el espacio en celdas (llamados **voxels**), y cada una se clasifica como interior o exterior al objeto.
- ▶ **Modelos de fronteras:** se representa la frontera (la superficie) en lugar de todo el interior, para ello se usan conjuntos finitos de polígonos planos adyacentes entre ellos (**caras**)

## Ejemplo 2D de los modelos aproximados

Las dos formas computacionales de representar objetos son aproximadamente iguales al modelo ideal basado (un subconjunto), con un error que disminuye al aumentar la cantidad de memoria usada (la precisión o resolución).



Enumeración espacial

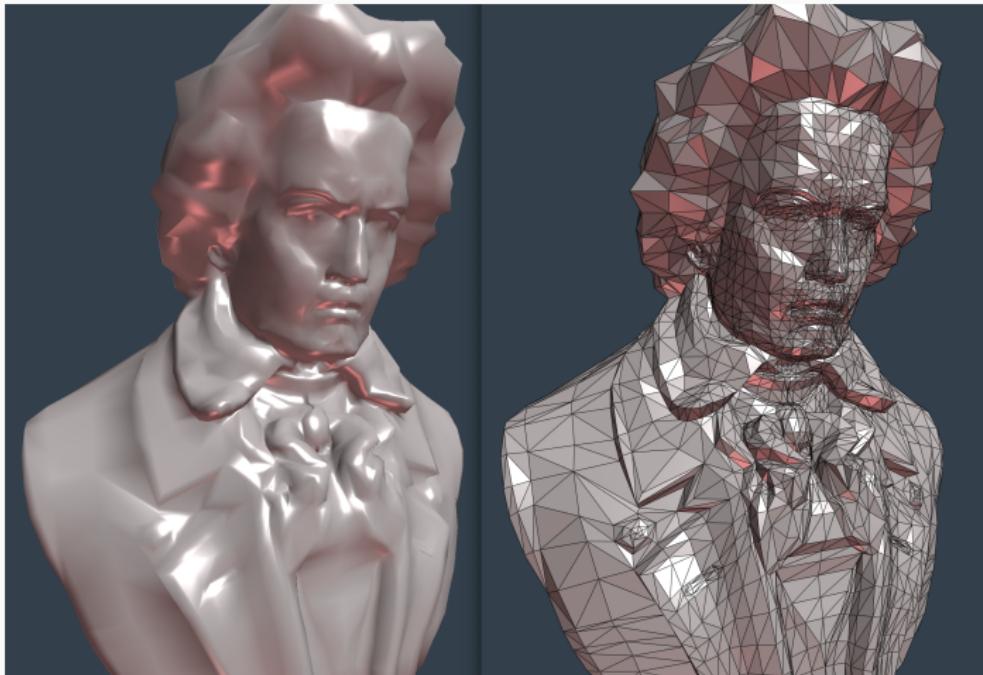


Modelos de fronteras

- ▶ Modelos de fronteras: usados en la mayoría de las aplicaciones.
- ▶ Enumeración espacial: muy útiles en aplicaciones específicas

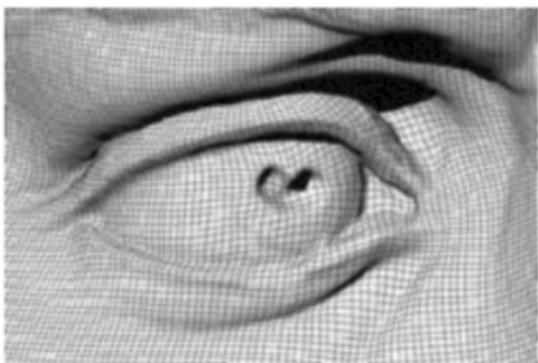
## Ejemplos de modelos de fronteras 3D (1/2)

Ejemplo de una **mallas de polígonos** (de las prácticas). A la izquierda el modelo con iluminación, a la derecha vemos las caras que forman el modelo:



## Ejemplos de modelos de fronteras 3D (2/2)

Los modelos de fronteras (a muy alta resolución) permiten representar fielmente casi cualquier objeto real:



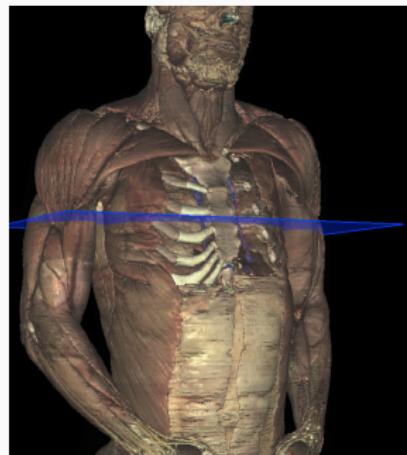
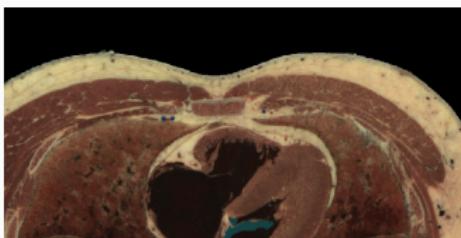
Escaneo 3D del *David* de Miguel Angel en Florencia en 1999.

Marc Levoy et al. The Digital Michelangelo Project:

☞ <https://accademia.stanford.edu/mich/>

# Enumeración espacial 3D

Las **modelos volumétricos** se usan en aplicaciones donde interesa todo el volumen del objeto (p.ej. en la **Tomografía Axial Computerizada**, TAC, para Medicina y Arqueología, o en Geología y Climatología)



Captura de pantalla del software *VH Dissector* de Toltech:

☞ <http://www.toltech.net/anatomy-software/solutions/vh-dissector-for-medical-education>

Sección 2.  
Modelos de fronteras: mallas de polígonos..

- 2.1. Elementos y adyacencia.
- 2.2. Atributos de vértices.

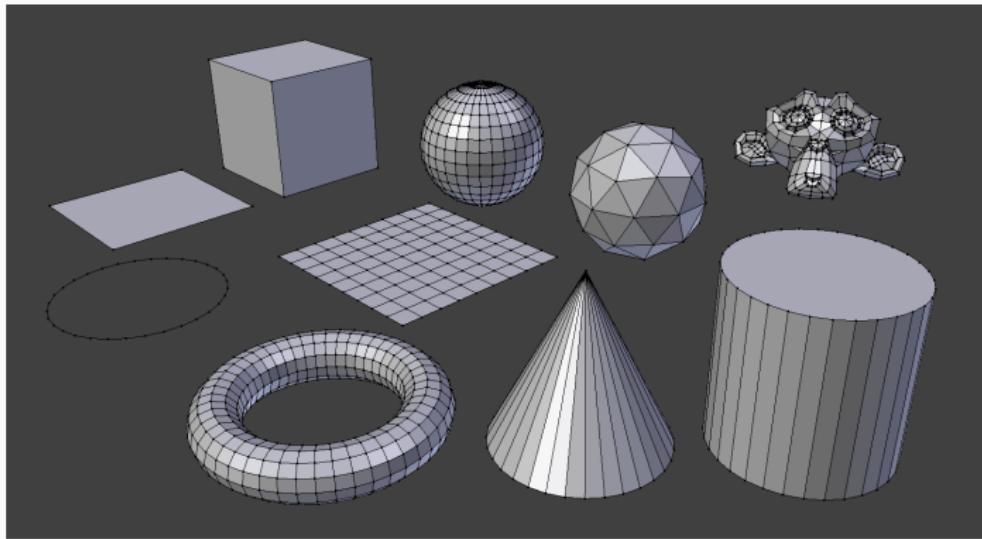
# Mallas de polígonos.

Una **malla de polígonos** (*polygon Mesh*) es un conjunto de puntos de un espacio afín que forman **caras** (*faces*) planas, usualmente adyacentes entre ellas, y que aproxima la frontera de un objeto en el espacio 3D

- ▶ El término *objeto* designa un conjunto de puntos como los descritos antes (de extensión finita, continuo). Esos puntos pertenecen todos a un mismo espacio afín.
  - ▶ Una cara es un conjunto de puntos en un plano de dicho espacio afín, delimitados por un polígono.
  - ▶ Las mallas aproxima una superficie, la cual
    - ▶ encierra completamente una región del espacio (el objeto tiene volumen), o bien
    - ▶ constituye en si misma el objeto, que tiene volumen nulo.
- Las primeras son mallas *cerradas* y las segundas *abiertas*.

# Ejemplos de mallas

Aquí vemos varios ejemplos de mallas de polígonos (excepto la circunferencia que no lo es). Algunas son cerradas y otras abiertas:

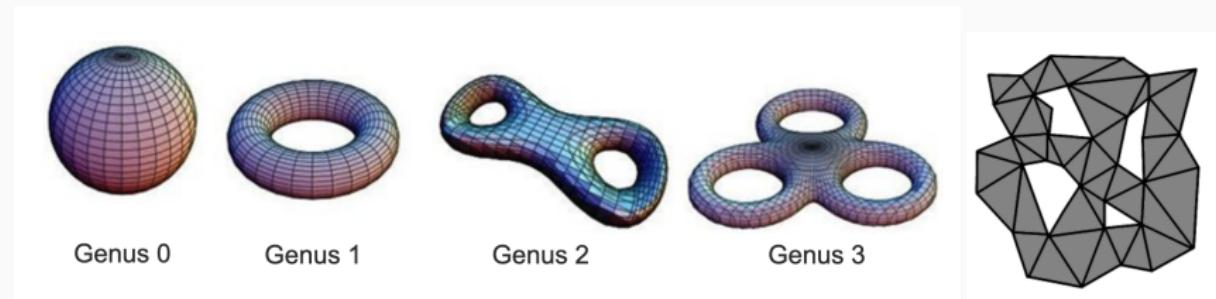


Catálogo de objetos predefinidos de la aplicación *Blender* para modelado 3D:

☞ <https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html>

# Características de las mallas

- ▶ Las mallas cerradas pueden tener cualquier *género topológico* (*genus*), aquí vemos mallas de género 0, 1, 2 y 3.
- ▶ Las mallas abiertas pueden tener huecos entre los polígonos:



Genus 0

Genus 1

Genus 2

Genus 3

Izquierda: Rudiger Westermann (Univ. Munich) - Computer Graphics Course slides  
↗ <https://slideplayer.com/slide/4642205/>

Derecha: ↗ Stackoverflow

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 2. Modelos de fronteras: mallas de polígonos.

Subsección 2.1.

Elementos y adyacencia..

## Elementos de las mallas: vértices

Un **vértice (vertex)** es un par formado por un **punto** del espacio afín (en el extremo de alguna una arista), y un **valor entero único** (entre 0 y  $n - 1$ , donde  $n$  es el número de vértices de la malla).

- ▶ Al punto lo llamamos la **posición** del vértice.
- ▶ Al entero lo llamamos **índice** del vértice.
- ▶ Dos vértices distintos (con distinto índice) pueden tener la misma posición.
- ▶ Usar estos índices:
  - ▶ permiten expresar la *topología* de una malla independientemente de su *geometría*.
  - ▶ facilita construir representaciones computacionales de las mallas con ciertas propiedades.

## Elementos de las mallas: caras

Una **cara** (*face*) contiene un conjunto de puntos coplanares que están delimitados por un único polígono plano. Se determina por una secuencia ordenada de índices de vértices que forman dicho polígono.

- ▶ En la secuencia de índices, cada vértice comparte una arista con el siguiente (y el último con el primero). En esta secuencia
  - ▶ es indiferente cual índice es el primero.
  - ▶ en principio, es indiferente en que sentido se recorren los vértices (solo hay dos posibilidades).
- ▶ Dos caras distintas no pueden tener asociado el mismo conjunto de índices, ni siquiera con distinto orden o empezando en distintos vértices.

## Elementos de las mallas: aristas. Representación de mallas.

Una **arista** (*edge*) contiene el conjunto de puntos en un lado del polígono que delimita una cara, puntos que forman un segmento de recta. Se determina por un par único de índices de vértices.

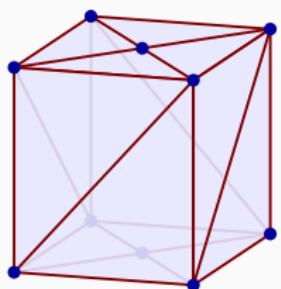
- ▶ Los dos índices de vértice de una arista no pueden coincidir.
- ▶ El orden en el que aparecen los índices en el par es irrelevante (las aristas no están orientadas).
- ▶ Dos aristas distintas no pueden tener el mismo par de índices de vértice, ni siquiera en distinto orden.

Esto implica que una malla viene determinada por:

- ▶ la secuencia  $\{\dot{p}_0, \dot{p}_1, \dots, \dot{p}_{n-1}\}$  de posiciones de sus  $n$  vértices.
- ▶ la secuencia de caras, cada una de ellas representada como una secuencia de  $k$  índices de vértice:  $\{i_0, \dots, i_{k-1}\}$  ( $k$  puede ser distinto en cada cara).

# Vértices, caras y aristas

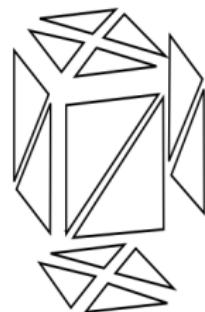
Elementos de una malla que forma la frontera de un paralelepípedo:



vertices



edges



faces



[Wikipedia: Polygon Mesh.](#)

## Adyacencia entre elementos de una malla

En una malla existen relaciones binarias de adyacencia entre estos elementos:

- ▶ Un vértice en el extremo de una arista es adyacente a la arista (por tanto, toda arista es adyacente a exactamente dos vértices).
- ▶ Una arista y una cara son adyacentes si la arista forma parte del polígono que delimita la cara.
- ▶ Dos vértices son adyacentes si hay una arista adyacente a ambos.
- ▶ Dos caras son adyacentes si hay una arista adyacente a ambas.
- ▶ Un vértice y una cara son adyacentes si hay una arista adyacente a ambos.

Puesto que los vértices están numerados, las relaciones de adyacencia se pueden expresar en términos de los índices de los vértices.

# Geometría y topología de las mallas

Una malla tiene una geometría y una topología:

- ▶ **Geometría:** conjunto de puntos que están en alguna cara (eso incluye los puntos que están en alguna arista y las posiciones de los vértices).
- ▶ **Topología:** conjunto de relaciones de adyacencia entre vértices, aristas y caras (sin tener en cuenta la geometría, es decir, considerando únicamente los índices de los vértices).

Esta definición permite que dos mallas distintas

- ▶ tengan la misma topología pero distinta geometría (p.ej., partimos de una malla y cambiamos las posiciones de sus vértices, pero mantenemos las adyacencias).
- ▶ tengan la misma geometría pero distinta topología (p.ej., partimos de una malla con caras de cuatro aristas y dividimos cada cara en dos caras triángulares coplanares).

## Características de las 2-variedades

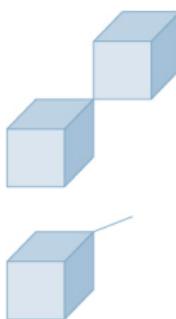
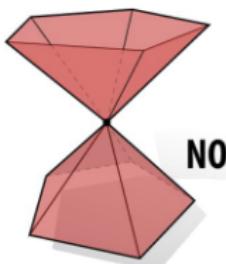
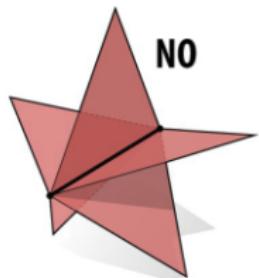
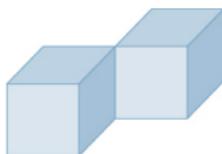
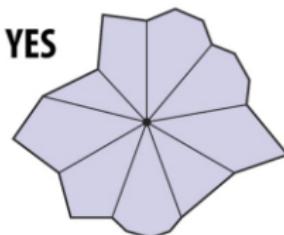
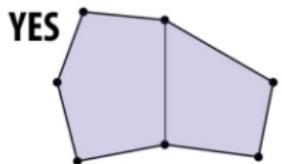
Vamos a usar exclusivamente mallas que son una **2-variedad** (**2-manifold**), esto implica que:

- ▶ Un vértice siempre es adyacente a dos aristas como mínimo (no hay vértices aislados).
- ▶ Una arista siempre es adyacente a una o a dos caras (no hay aristas aisladas, ni aristas adyacentes a 3 o más caras).
- ▶ Todas las caras adyacentes a un vértice se pueden ordenar en una secuencia en la cual cada cara es adyacente a la siguiente.

Estas propiedades aseguran, entre otras cosas, que se pueden asignar ciertos atributos a cada vértice de forma única (por ejemplo, normales y coordenadas de textura), ya que el entorno de un punto de la superficie siempre es *equivalente* a un plano.

# Ejemplos de mallas que no son 2-variedades

Solo son 2-variedades las mallas etiquetadas con yes:



Izquierda: Carnegie Mellon Computer Graphics Course: slides (fall 2018):

☞ <http://15462.courses.cs.cmu.edu/spring2018/lecture/meshes>

Derecha: J.F. Hughes at al.: Computer Graphics: Principles and Practice (3rd ed.)

## Conversión en 2-variedad

La topología de una malla que no es una 2-variedad puede modificarse para que lo sea, manteniendo la geometría:

- ▶ Se puede conseguir replicando vértices, es decir, añadiendo nuevos vértices con índices distintos en la misma posición de otros vértices ya existentes (p.ej.: dos conos unidos por el ápice se separan al insertar dos ápices en la misma posición, un ápice por cono)
- ▶ Esto puede implicar a veces también replicar aristas (p.ej.: dos cubos unidos con una arista en común se separan duplicando los dos vértices de dicha arista y la propia arista).

## Aristas y vértices de frontera

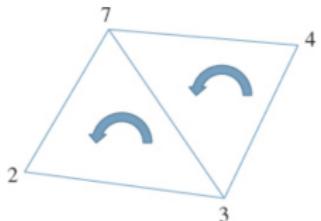
- ▶ Una arista es una **arista de frontera** (o de **borde**) (*boundary edge* o *border edge*) si es adyacente a una única cara.
- ▶ Un vértice es un vértice de frontera si es adyacente a alguna arista de frontera.
- ▶ Una malla es cerrada si y solo si no tiene aristas de frontera (todas las aristas son adyacentes a exactamente dos caras).
- ▶ Las mallas abiertas tienen al menos una cara de frontera.

# Orientación coherente de vértices en cada cara

El orden de enumeración de los vértices en las caras debe ser coherente:

- ▶ Dos caras adyacentes tienen los vértices siempre enumerados en el mismo sentido (horario o antihorario).
- ▶ Por convenio, en una malla cerrada, una cara vista desde el exterior presenta sus vértices en sentido anti-horario.
- ▶ En las mallas abiertas, esto define dos *lados* de la superficie (desde uno se ven las caras en sentido anti-horario y desde el otro en sentido horario).

A modo de ejemplo, para enumerar los vértices de las dos caras



- ▶ es correcto:  $(2, 3, 7)$  y  $(4, 7, 3)$ .
- ▶ es correcto:  $(3, 2, 7)$  y  $(7, 4, 3)$ .
- ▶ es incorrecto:  $(2, 3, 7)$  y  $(4, 3, 7)$ .

## Conversión en 2-variedad

La topología de una malla que no es una 2-variedad puede modificarse para que lo sea, manteniendo la geometría:

- ▶ Se puede conseguir replicando vértices, es decir, añadiendo nuevos vértices con índices distintos en la misma posición de otros vértices ya existentes (p.ej.: dos conos unidos por el ápice se separan al insertar dos ápices en la misma posición, un ápice por cono)
- ▶ Esto puede implicar a veces también replicar aristas (p.ej.: dos cubos unidos con una arista en común se separan duplicando los dos vértices de dicha arista y la propia arista).

## Marco de referencia de la malla.

La posición de cada vértice se representa en el ordenador por sus coordenadas respecto de un marco de referencia cartesiano  $\mathcal{R}$  único

- ▶ A dicho marco de referencia se le denomina **marco de referencia local de la malla**
- ▶ El  $i$ -ésimo vértice (en el punto  $\vec{p}_i$ ) tiene coordenadas  $\mathbf{c}_i = (x_i, y_i, z_i, 1)$  en el marco  $\mathcal{R}$ , es decir:

$$\vec{p}_i = \mathcal{R} \vec{c}_i = \mathcal{R} (x_i, y_i, z_i, 1)^T$$

- ▶ A la tupla  $\mathbf{c}_i$  se le denomina las **coordenadas locales** del vértice  $i$ -ésimo.
- ▶ No se suele almacenar en memoria la componente  $w$  ya que siempre es 1, aunque a veces se podría hacer para acortar algo el tiempo de procesamiento (a costa de usar más memoria).

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 2. Modelos de fronteras: mallas de polígonos.

Subsección 2.2.

Atributos de vértices..

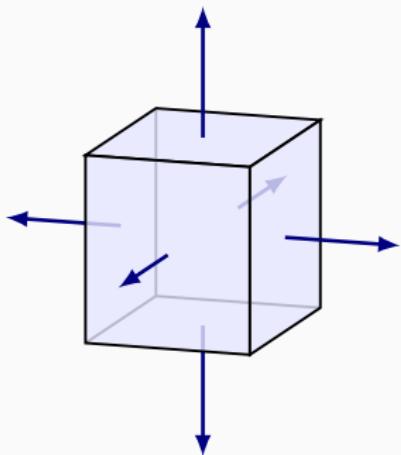
# Atributos de las mallas

Como modelos de objetos reales, las mallas suelen incluir más información geométrica o del aspecto del objeto:

- ▶ **Normales (normals):** vectores de longitud unidad
  - ▶ **normales de caras:** vector unitario perpendicular a cada cara, de longitud unidad, apuntando al exterior de la malla si es una malla cerrada. Precalculado a partir del polígono.
  - ▶ **normales de vértices:** vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.
- ▶ **Colores:** ternas (usualmente RGB) con tres valores entre 0 y 1.
  - ▶ **colores de caras:** útil cuando cada cara representa un trozo de superficie de color homogéneo.
  - ▶ **colores de vértices:** color de la superficie en cada vértice (la superficie varía de color de forma continua entre vértices).
- ▶ Otros atributos: coords. de textura, vectores tangente y bitangente, etc...

## Normales de caras

Pueden ser útiles cuando el objeto que se modela con la malla está realmente compuesto de caras planas (p.ej., un cubo), o bien cuando se quiere hacer *sombreado plano*:



Para un polígono (con dos aristas  $\vec{a}, \vec{b}$ , vectores distintos, no nulos), su normal  $\vec{n}$  se define como:

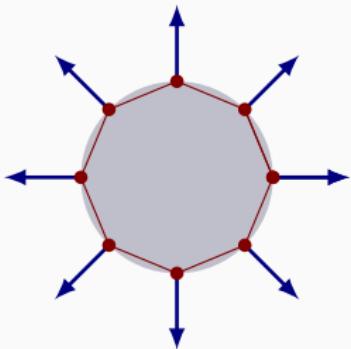
$$\vec{n} = \frac{\vec{m}}{\|\vec{m}\|} \quad \text{donde } \vec{m} = \vec{a} \times \vec{b}$$

En estos casos la normal se puede precalcular y almacenar en la malla para lograr eficiencia en tiempo de visualización.

# Normales de vértices

Tienen sentido cuando la malla aproxima una superficie curvada:

- ▶ A veces la superficie original es conocida, y las normales se definen fácilmente (p.ej. una esfera).
- ▶ Si la superficie original es desconocida, las normales se pueden definir exclusivamente usando la malla



Para un vértice (con  $k$  caras adyacentes) su normal  $\vec{n}$  se define como:

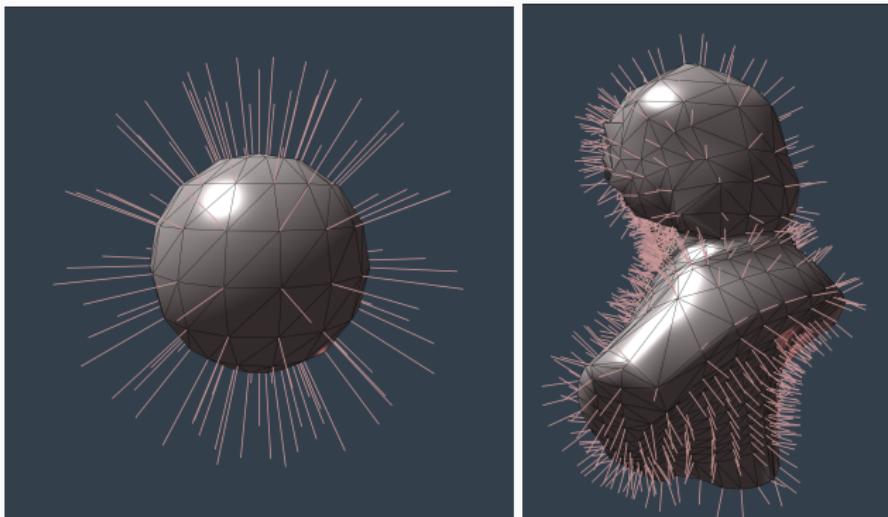
$$\vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \quad \text{donde } \vec{s} = \sum_{i=0}^{k-1} \vec{m}_i$$

donde  $\vec{m}_0, \vec{m}_1, \dots, \vec{m}_{k-1}$  son las normales de las caras adyacentes al vértice.

Las coordenadas de estas normales también se pueden precalcular y almacenar.

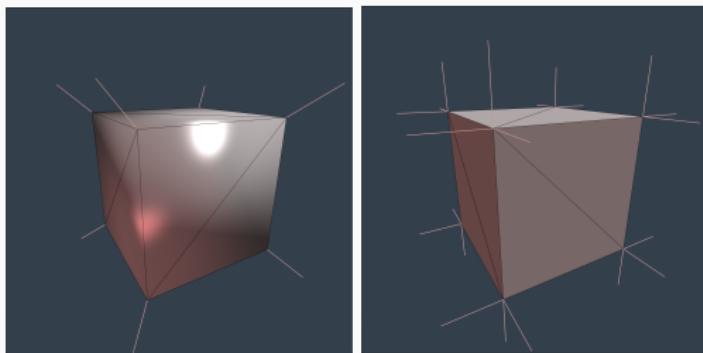
## Ejemplos de normales de vértices calculadas

Aquí vemos visualizadas las normales de una esfera de baja resolución (calculadas analíticamente), y de una malla arbitraria (calculadas promediando normales de caras):



# Discontinuidades de la normal

Algunos objetos reales presentan aristas o vértices donde la normal es discontinua (p.ej. un cubo), en ese caso promediar normales es mala idea, y es necesario replicar vértices y aristas (y después promediar):



El cubo de la izquierda tiene 8 vértices y el de la derecha 24 vértices. La iluminación es correcta a la derecha. El mismo problema puede aparecer con las coordenadas de textura, o los colores.

## Colores de vértices

En algunos casos, es conveniente asignar colores RGB a los vértices. La utilidad más frecuente de esto es hacer interpolación de color en las caras durante la visualización:

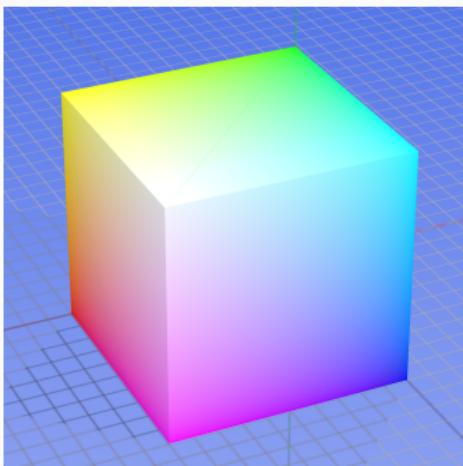


Imagen: [http://en.wikipedia.org/wiki/File:RGB\\_color\\_solid\\_cube.png](http://en.wikipedia.org/wiki/File:RGB_color_solid_cube.png)  
(Wikimedia Commons)

## Sección 3. Representación en memoria de modelos de fronteras..

- 3.1. Triángulos aislados (TA).
- 3.2. Tiras de triángulos (TT).
- 3.3. Mallas indexadas.
- 3.4. Representación con estructura de aristas aladas

# Representación en memoria

En esta sección veremos distintas formas de representar las mallas en la memoria de un ordenador:

- ▶ **Triángulos aislados, tiras de triángulos:** no representan explícitamente la topología.
- ▶ **Mallas indexadas:** lo más común, representan explícitamente la topología.
- ▶ **Aristas aladas:** extensión de las mallas indexadas para eficiencia en tiempo.

OpenGL está diseñado para visualizar directamente los triángulos aislados, las tiras de triángulos y las mallas indexadas.

Por simplicidad, nos restringimos a **caras triangulares**, que es lo más común en la inmensa mayoría de las aplicaciones.

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.1.

Triángulos aislados (TA)..

# Tabla de triángulos aislados.

La más simple es usar una lista o tabla de **triángulos aislados**. La malla se representa como un vector o lista con tres entradas (tres variables de tipo **Tupla3f**) para cada triángulo:

Malla TA ( $n$ triángulos)		
0	$x_0$	$y_0$
1	$x_1$	$y_1$
2	$x_2$	$y_2$
3	$x_3$	$y_3$
4	$x_4$	$y_4$
5	$x_5$	$y_5$
:	:	
$3n - 3$	$x_{3n-3}$	$y_{3n-3}$
$3n - 2$	$x_{3n-2}$	$y_{3n-2}$
$3n - 1$	$x_{3n-1}$	$y_{3n-1}$

- ▶ Para cada triángulo se almacenan las coordenadas locales de cada uno de sus tres vértices (9 valores flotantes en total).
- ▶ La tabla se puede almacenar en memoria con todas las coordenadas continuas.
- ▶ En total, incluye  $9n$  valores flotantes.

## Triángulos aislados: valoración.

Esta representación es poco eficiente en tiempo y memoria:

- ▶ Si un vértice es adyacente a  $k$  triángulos, sus coordenadas aparecen repetidas  $k$  veces en la tabla y se procesan  $k$  veces al visualizar.
- ▶ En una malla típica que representa una rejilla de triángulos, los vértices internos (la mayoría) son adyacentes a 6 triángulos, es decir, cada tupla aparece casi 6 veces como media.

Además, no hay información explícita sobre la **topología** de la malla:

- ▶ La topología se puede calcular comparando coordenadas de vértices, pero tendría una complejidad en tiempo cuadrática con el número de vértices y es poco robusto.

En algunos casos muy particulares, podría ser útil (objetos realmente compuestos de muchos triángulos realmente aislados, objetos muy sencillos).

# Implementación de mallas: clase base abstracta.

Cualquier objeto que se pueda visualizar se implementará usando una clase concreta derivada de la clase **Objeto3D**:

```
class Objeto3D
{ public:
    virtual void visualizarGL( ContextoVis & cv ) = 0 ;
    // ..... (otros métodos)
} ;
```

- ▶ Cualquier tipo de objeto que se pueda dibujar con OpenGL llevará asociada una clase concreta que implementará una versión concreta del método **visualizarGL**
- ▶ **ContextoVis** es una clase con información de contexto sobre la visualización.

```
class ContextoVis
{ public:
    unsigned modo_vis ; // modo de visualización (alambre, sólido,...)
    // .....
} ;
```

# Implementación de mallas de triángulos aislados

Como ejemplo, podemos usar una clase (**MallaTA**), con un vector con las coordenadas (contiguas) en memoria. En total, para  $n_t$  triángulos, se guardan  $9n_t$  valores reales:

```
// Malla de triángulos aislados:  
class MallaTA : public Objeto3D  
{  
protected:  
    std::vector<Tupla3f> vertices ; // tabla de vértices ( $3n_t$  tuplas)  
    // .... (otros métodos)  
public:  
    virtual void visualizarGL( ContextoVis & cv ) ;  
} ;
```

La visualización puede hacerse en modo inmediato o diferido, considerando la secuencia de vértices como una **secuencia no indexada** (no hay índices), y usando **GL\_TRIANGLES** como tipo de primitiva.

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.2.

Tiras de triángulos (TT)..

## Tiras de triángulos: motivación y características

Las representación en memoria usando **tiras de triángulos** pretende reducir la memoria y el tiempo que necesitan la representación de triángulos aislados:

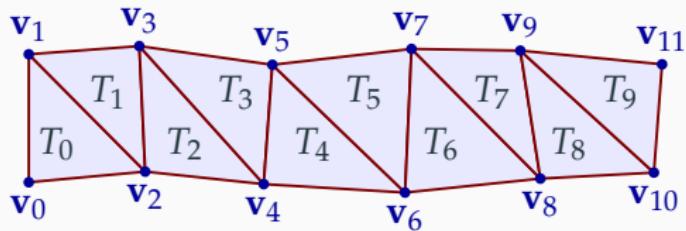
- ▶ Para conseguir esto, esta representación reduce el número de veces que aparecen replicadas unas coordenadas en memoria.
- ▶ Como consecuencia, se reduce el tiempo de procesamiento.

Sin embargo, esta representación

- ▶ No evita totalmente las redundancias.
- ▶ Tampoco incluye información explícita sobre la topología de la malla.

## Tiras de triángulos en una malla.

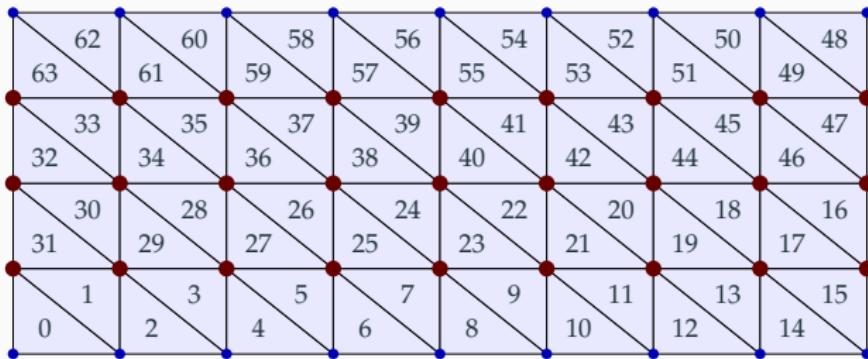
Podemos identificar una parte de una malla como una **tira de triángulos**: cada triángulo  $T_{i+1}$  en la secuencia es adyacente al anterior  $T_i$ , con lo cual  $T_{i+1}$  comparte con  $T_i$  una arista y dos vértices, vértices cuyas coordenadas no tienen que ser repetidas en memoria:



- ▶ Cada tira de  $n$  triángulos necesita  $n + 2$  tuplas de coordenadas de vértices (tres para el primer triángulo y después una más por cada triángulo adicional)
- ▶ Se almacena una tabla que en la  $i$ -ésima entrada almacena las coordenadas del  $i$ -ésimo vértice.

# Tiras de triángulos para mallas no simples

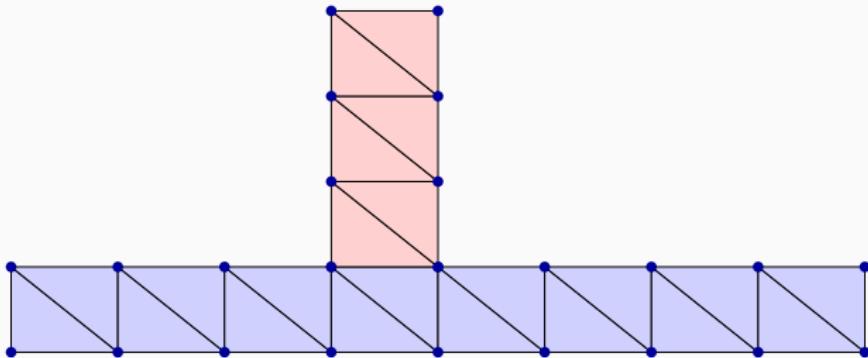
En la mayoría de los casos, las tiras obligan a repetir algunas coordenadas de vértices (aunque en mucho menor grado que los triángulos aislados). Ejemplo de una tira en zig-zag:



- ▶ las coords. de los vértices en rojo (grandes) se repiten dos veces.
- ▶ las de los vértices en azul (pequeños) aparecen una sola vez.

## Mallas con varias tiras

En algunos casos, es inevitable tener que recurrir a más de una tira para una única malla:



Por este motivo la implementación de una malla con tiras debe prever más de una tira en la misma malla.

# Representación en memoria

Una malla es una estructura con varias tiras. La tira número  $i$  (con  $n_i$  triángulos) es un array con  $n_i + 2$  celdas, en cada una están las coordenadas maestras de un vértice.

Tira 0 ( $n_0$  triángulos)

$x_0$	$y_0$	$z_0$
$x_1$	$y_1$	$z_1$
$x_2$	$y_2$	$z_2$
$x_3$	$y_3$	$z_3$
$x_4$	$y_4$	$z_4$
:	:	:
$x_{n_0}$	$y_{n_0}$	$z_{n_0}$
$x_{n_0+1}$	$y_{n_0+1}$	$z_{n_0+1}$
$x_{n_0+2}$	$y_{n_0+2}$	$z_{n_0+2}$

Tira 1 ( $n_1$  triángulos)

$x_0$	$y_0$	$z_0$
$x_1$	$y_1$	$z_1$
$x_2$	$y_2$	$z_2$
$x_3$	$y_3$	$z_3$
$x_4$	$y_4$	$z_4$
:	:	:
$x_{n_1}$	$y_{n_1}$	$z_{n_1}$
$x_{n_1+1}$	$y_{n_1+1}$	$z_{n_1+1}$
$x_{n_1+2}$	$y_{n_1+2}$	$z_{n_1+2}$

Tira 2 ( $n_2$  triángulos)

$x_0$	$y_0$	$z_0$
:	:	:
$x_{n_2+2}$	$y_{n_2+2}$	$z_{n_2+2}$

## Tiras de triángulos: valoración

La mejora de las tiras frente a los triángulos aislados es que usan menos memoria, sin embargo, tiene estos inconvenientes:

- ▶ Al requerir probablemente más de una tira de triángulos, la representación es algo más compleja.
- ▶ Se necesitan algoritmos (complejos) para calcular las tiras a partir de una malla representada de alguna otra forma. Se intenta optimizar de forma que el número de coordenadas a almacenar sea el menor posible.
- ▶ El numero promedio de veces que se repite cada coordenada en memoria es prácticamente siempre superior a la 1, y cercano a 2.
- ▶ Esta representación tampoco incorpora información explícita sobre la conectividad.

# Implementación

Una malla hecha de tiras de triángulos se puede representar con una estructura similar a **MallaTT**, para cada tira hay una instancia de **TiraTriangulos**

```
// Tira de triángulos (coords. de vértices)
struct TiraTri // datos de una única tira
{
    unsigned long      num_tri; // número de triángulos en esta tira
    std::vector<Tupla3f> ver;   // coords. de vert. (num_tri+2 entradas)
} ;
// Malla compuesta de tiras de triángulos
class MallaTT : public Objeto3D
{
protected:
    std::vector<TiraTri> tiras; // vector de tiras
    ....
public:
    virtual void visualizarGL( ContextoVis & cv );
} ;
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.3.

Mallas indexadas..

## Mallas Indexadas.

Para solucionar los problemas de uso de memoria y tiempo de procesamiento de las soluciones anteriores, se puede usar una estructura con dos tablas:

- ▶ **Tabla de vértices**: tiene una entrada por cada vértice, incluye sus coordenadas
- ▶ **Tabla de triángulos**: tiene una entrada por triángulo, incluye los índices de sus tres vértices en la tabla anterior.

En esta solución:

- ▶ No se repiten coordenadas de vértices: se ahorra memoria y se puede visualizar sin repetir cálculos (se repiten índices enteros).
- ▶ Hay información explícita de la topología (conectividad): se almacenan explícitamente los vértice adyacentes a un triángulo y se pueden calcular fácilmente el resto de adyacencias.

# Estructura de datos

La tabla de triángulos (para  $n$  triángulos), almacena un total de  $3n$  índices de vértices (enteros sin signo), y la de vértices  $3m$  valores reales:

Tabla Triángulos ( $n$  tri.)

$i_{0,0}$	$i_{0,1}$	$i_{0,2}$
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$
$\vdots$	$\vdots$	$\vdots$
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$

$$0 \leq i_{jk} < m$$

$$i_{1,2} = 3$$

Tabla Vértices ( $m$  verts.)

0	$x_0$	$y_0$	$z_0$
1	$x_1$	$y_1$	$z_1$
2	$x_2$	$y_2$	$z_2$
3	$x_3$	$y_3$	$z_3$
4	$x_4$	$y_4$	$z_4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$m-2$	$x_{m-2}$	$y_{m-2}$	$z_{m-2}$
$m-1$	$x_{m-1}$	$y_{m-1}$	$z_{m-1}$

# Implementación de las mallas indexadas

Usaremos una clase de nombre **MallaInd**:

```
class MallaInd : public Objeto3D
{
protected:
    std::vector<Tupla3f> vertices ; // tabla de vértices
    std::vector<Tupla3i> triangulos; // tabla de triángulos (índices)
    .....
public:
    virtual void visualizarGL( ContextoVis & cv ) ;
} ;
```

Incluye un total de  $3n_v$  valores reales contiguos en memoria (coordenadas de vértices), y  $3n_t$  valores naturales (índices de vértices), también contiguos.

# Archivos con mallas indexadas: el formato PLY

El formato PLY fue diseñado por Greg Turk y otros en la Univ. de Stanford a mediados de los 90. Codifica una malla indexada en un archivo ASCII o binario (usaremos la versión ASCII). Tiene tres partes:

- ▶ Cabecera: describe los atributos presentes y su formato, se indica el número de vértices y caras, ocupa varias líneas.
- ▶ Tabla de vértices: un vértice por línea, se indican sus coordenadas X,Y y Z (flotantes) en ASCII, separadas por espacios
- ▶ Tabla de caras: una cara por línea, se indica el número de vértices de la cara, y después los índices de los vértices de la cara (comenzando en cero para el primer vértice de la tabla de vértices).
- ▶ El formato es extensible de forma que un archivo puede incluir otros atributos (p.ej., colores de vértices).

Su simplicidad hace fácil usarlo.

# Ejemplo de archivo PLY

```
ply
format ascii 1.0
comment Archivo de ejemplo del formato PLY (8 vertices y 6 caras)
element vertex 8
property float x
property float y
property float z
element face 6
property list uchar int vertex_index
end_header
0.0 0.0 0.0
0.0 0.0 2.0
0.0 1.3 1.0
0.0 1.4 0.0
1.1 0.0 0.0
1.0 0.0 2.0
1.0 0.8 1.5
0.5 1.0 0.0
4 0 1 2 3
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

## Tabla de vértices y tiras de triángulos

Es posible representar una malla como una tabla de vértices y varias tiras de triángulos. Cada tira almacena índices de vértices en lugar de coordenadas de vértices.

- ▶ Las coordenadas no se repiten en memoria.
- ▶ Se repiten en memoria los índices de vértices, pero menos veces que con tabla de vértices y triángulos.
- ▶ El modelado usando tiras es más complejo.

Se pueden visualizar usando **glDrawElements** con **GL\_TRIANGLE\_STRIP** como tipo de primitivas:

- ▶ Las coordenadas de vértice se envían y se procesan una sola vez
- ▶ Los índices de vértices se envían repetidos, pero solo un par de veces de media aprox.

## Tabla de aristas

En una malla indexada podría ser conveniente (para ganar tiempo de procesamiento en ciertas aplicaciones) almacenar explicitamente las aristas (ahora esa info. está implícita en la tabla de triángulos). Se puede hacer usando un **tabla de aristas**:

- ▶ Contiene una entrada por cada arista.
- ▶ En cada entrada hay dos índices (naturales) de vértices (los dos vértices en los extremos de la arista).
- ▶ El orden de los vértices en cada entrada es irrelevante, pero las aristas no deben estar duplicadas.

La disponibilidad de esta tabla permite, por ejemplo, dibujar en modo alambre con begin/end sin repetir dos veces el dibujo de aristas adyacentes a dos triángulos.

# Problema: comparación de eficiencia en memoria (1/2)

## Problema 2.1.

Supongamos que queremos codificar una esfera de radio  $1/2$  y centro en el origen de dos formas:

- ▶ Por enumeración espacial, dividiendo el cubo que engloba a la esfera en celdas, de forma que haya  $k$  celdas por lado del cubo, todas ellas son cubos de  $1/k$  de ancho. Cada celda ocupa un bit de memoria (si su centro está en la esfera, se guarda un 1, en otro caso un 0).
- ▶ Usando un modelo de fronteras (una malla indexada de triángulos), en el cual se usa una rejilla de triángulos y aristas que siguen los meridianos y paralelos, habiendo en cada meridiano y en cada paralelo un total de  $k$  vértices (se guarda únicamente la tabla de vértices y la de triángulos).

(continua en la siguiente transparencia)

## Problema: comparación de eficiencia en memoria (2/2)

### Problema 2.1. (continuación)

Asumiendo que un `float` y un `int` ocupan 4 bytes cada uno, contesta a estas cuestiones:

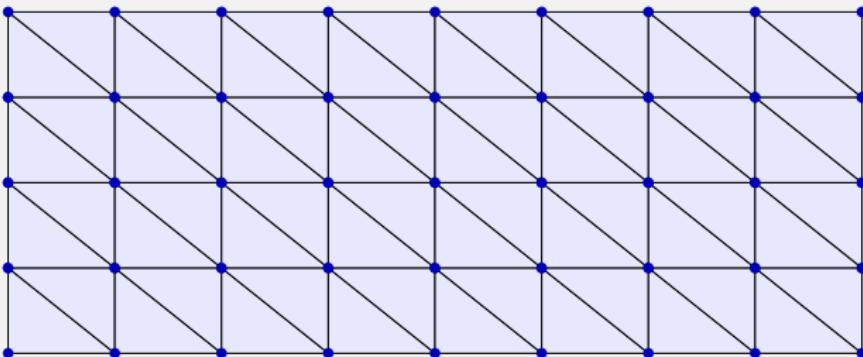
- ▶ Expresa el tamaño de ambas representaciones en bytes como una función de  $k$ .
- ▶ Suponiendo que  $k = 16$  calcula cuantos KB de memoria ocupa cada estructura.
- ▶ Haz lo mismo asumiendo ahora que  $k = 1024$  (expresa los resultados en MB)

Compara los tamaños de ambas representaciones en ambos casos ( $k = 16$  y  $k = 1024$ ).

# Problema: uso de memoria en mallas indexadas (1/2)

## Problema 2.2.

Considera una malla indexada (tabla de vértices y caras, esta última con índices de vértices) con topología de rejilla como la de la figura, en la cual hay  $n$  columnas de pares de triángulos y  $m$  filas (es decir, hay  $n + 1$  filas de vértices y  $m + 1$  columnas de vértices, con  $n, m > 0$ , en el ejemplo concreto de la figura,  $n = 8$  y  $m = 4$ ).



(continua en la siguiente transparencia)

## Problema: uso de memoria en mallas indexadas (2/2)

### Problema 2.2. (continuación)

En relación a este tipo de mallas, responde a estas dos cuestiones:

- (a) Supongamos que un **float** ocupa 4 bytes (igual a un **int**) ; que tamaño en memoria ocupa la malla completa, en bytes ? (tener en cuenta únicamente el tamaño de la tabla de vértices y triángulos, suponiendo que se almacenan usando los tipos **float** e **int**, respectivamente). Expresa el tamaño como una función de  $m$  y  $n$ .
- (b) Escribe el tamaño en KB suponiendo que  $m = n = 128$ .
- (c) Supongamos que  $m$  y  $n$  son ambos grandes (es decir, asumimos que  $1/n$  y  $1/m$  son prácticamente 0). deduce que relación hay entre el número de caras  $n_C$  y el número de vértices  $n_V$  en este tipo de mallas.

# Problema: uso de memoria en tiras y mallas indexadas (1/2)

## Problema 2.3.

Imagina de nuevo una malla como la del problema anterior, supongamos que usamos una representación como tiras de triángulos, de forma que cada fila de triángulos (con  $2n$  triángulos) se almacena en una tira, habiendo un total de  $m$  tiras.

La tabla de punteros a tiras tiene un entero (el número de tiras) y  $m$  punteros, cada puntero suponemos que tiene 8 bytes de tamaño. De nuevo, asume que las coordenadas son de tipo **float** (4 bytes).

Responde a estas cuestiones:

(continua en la siguiente transparencia)

# Problema: uso de memoria en tiras y mallas indexadas (2/2)

## Problema 2.3. (continuación)

- (a) Indica que cantidad de memoria ocupa esta representación, en estos dos casos:
  - (a.1) Como función de  $n$  y  $m$ , en bytes.
  - (a.2) Suponiendo  $m = n = 128$ , en KB.
- (b) Para  $m$  y  $n$  grandes (es decir, cuando  $1/n$  y  $1/m$  son casi nulos), describe que relación hay entre el tamaño en memoria de la malla indexada del problema anterior y el tamaño de la malla almacenada como tiras de triángulos.
- (c) Si suponemos que la transformación de cada vértice se hace en un tiempo constante igual a la unidad, describe que relación hay entre los tiempos de procesamiento de vértices para esta malla cuando se representa como una malla indexada y como tiras de triángulos.

# Problema: número de vértices, aristas y caras

## Problema 2.4.

Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un *poliedro* simplemente conexo de caras triangulares). Considera el número de vértices  $n_V$ , el número de aristas  $n_A$  y el número de caras  $n_C$  en este tipo de mallas.

Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

$$n_A = 3(n_V - 2)$$

$$n_C = 2(n_V - 2)$$

(nótese que, al igual que en el problema anterior, sigue siendo cierto que el número de caras es aproximadamente el doble que el de vértices).

# Problema: creación de la tabla de aristas

## Problema 2.5.

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector `ari`, que en cada entrada tendrá una tupla de tipo `Tupla2i` (contiene dos `int`) con los índices en la tabla de vértices de los dos vértices en los extremos de la arista. El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

Escribe el código de una función C++ para crear y calcular la tabla de aristas a partir de la tabla de triángulos. Intenta encontrar una solución con la mínima complejidad en tiempo y memoria posible. Suponer que el número de vértices adyacentes a uno cualquiera de ellos es como mucho un valor constante  $k > 0$ , valor que no depende del número total de vértices, que llamamos  $n$ .

(continua en la transparencia siguiente)

# Problema: creación de la tabla de aristas

## Problema 2.5. (continuación)

Considerar dos casos:

1. Los triángulos se dan con orientación *no coherente*: esto quiere decir que si un triángulo está formado por los vértices  $i, j, k$ , estos tres índices pueden aparecer en cualquier orden en la correspondiente entrada de la tabla de triángulos (puede aparecer como  $i, j, k$  o como  $i, k, j$ , o como  $k, j, i$ , etc....)
2. Los triángulos se dan con orientación *coherente*: esto quiere decir que si dos triángulos comparten una arista entre los vértices  $i$  y  $j$ , entonces en uno de los triángulos la arista aparece como  $(i, j)$  y en el otro aparece como  $(j, i)$  (decimos que en el triángulo  $a, b, c$  aparecen las tres aristas  $(a, b)$ ,  $(b, c)$  y  $(c, a)$ ). Además, asumimos que la malla es *cerrada*, es decir, que cada arista es compartida por exactamente dos triángulos.

# Problema: cálculo del área de una malla indexada

## Problema 2.6.

Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función que acepta un puntero a una **MallaInd** y devuelve un número real (asumir que se dispone del tipo **Tupla3f** y de los operadores usuales de tuplas o vectores, es decir suma **+**, resta **-**, producto escalar **\***, producto vectorial **×**, módulo **||**, etc ...).

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.4.

Representación con estructura de aristas aladas.

## Motivación

La estructura de malla indexada permite, por ejemplo, consultar con tiempo en  $O(1)$  si un vértice es adyacente a un triángulo, pero:

- ▶ Para consultar si dos vértices son adyacentes, hay que buscar en la tabla de triángulos si los vértices aparecen contiguos en alguno: esto requiere un tiempo en  $O(n_t)$ .
- ▶ No se guarda información de las aristas. Las consultas relativas a aristas se resuelven también en  $O(n_t)$ .
- ▶ En general, las consultas sobre adyacencia son costosas en tiempo.

Para poder reducir los tiempos de cálculo de adyacencia a  $O(1)$ , se puede usar más memoria de la estrictamente necesaria para la malla indexada. Veremos la estructura de **aristas aladas** (para mallas que encierran un volumen, es decir: siempre hay **dos caras adyacentes a una arista**, no necesariamente triangulares)

## Estructura de aristas aladas: tabla de aristas.

Una malla se puede codificar usando una tabla de vértices (**tver**) (similar a la de las mallas indexadas), mas una tabla de aristas (**tari**). Esta última:

- ▶ Tiene una entrada por cada arista, con dos índices:
  - ▶ **vi** = índice de vértice inicial
  - ▶ **vf** = índice de vértice final(es indiferente cual se selecciona como inicial y cual como final).
- ▶ Tambien tiene (cada arista es adyacente a dos triángulos):
  - ▶ **ti** = índice del triángulo a la izquierda
  - ▶ **td** = índice del triángulo a la derecha(izquierda y derecha entendidos según se recorre la arista en el sentido que va desde vértice inicial al final)
- ▶ Esto permite consultas en O(1) sobre adyacencia arista-vértice y arista-triángulo.

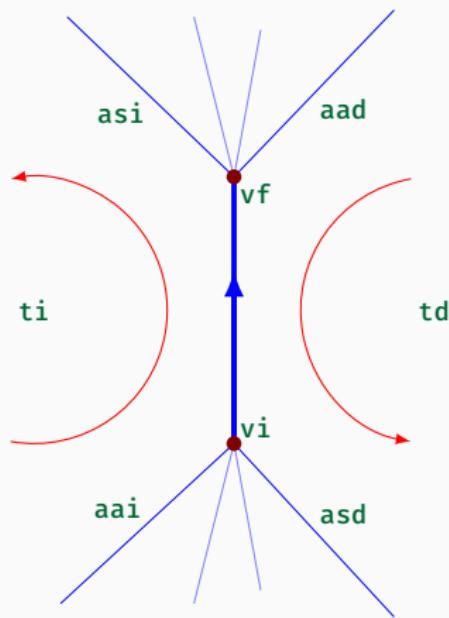
## Aristas siguiente y anterior

Además de los datos anteriores, se guarda, para cada arista, los índices de otras cuatro adyacentes a ella.

- ▶ Si se recorren las aristas del triángulo de la izquierda aparecerá la arista en cuestión entre otras dos, cuyos índices se guardan en la tabla:
  - ▶ **aai** = índice de la **arista anterior**
  - ▶ **asi** = índice de la **arista siguiente**
- (el recorrido de las aristas se hace en sentido anti-horario cuando se observa el triángulo desde el exterior de la malla).
- ▶ Igualmente, se guardan los dos índices de arista anterior y siguiente relativas al recorrido anti-horario del triángulo de la derecha:
  - ▶ **aad** = índice de la **arista anterior (derecha)**
  - ▶ **asd** = índice de la **arista siguiente (derecha)**

# Índices asociados a una arista

Índices (valores naturales) en la entrada correspondiente a la arista vertical en una malla (no necesariamente de triángulos):



## Uso de las aristas siguiente y anterior.

El hecho de almacenar las aristas siguiente y anterior permite hacer recorridos por las entradas de la tabla de aristas siguiendo esos índices:

- ▶ Dada una arista y un triángulo adyacente (el izquierdo o el derecho), se pueden obtener estas listas:
  - ▶ aristas adyacentes al triángulo.
  - ▶ vértices adyacentes al triángulo
- ▶ Dada una arista y un vértice adyacente (el inicial o el final), se pueden obtener estas listas:
  - ▶ aristas que inciden en el vértice (esto permite resolver fácilmente adyacencias **arista-arista**)
  - ▶ triángulos adyacentes al vértice.

Con toda esta información (los 8 valores), se puede resolver directamente cualquier adyacencia que involucre una arista al menos (consultando su entrada). Aun no podemos resolver el resto.

## Tablas adicionales. Uso.

Para hacer todas las consultas en  $O(1)$ , añadimos **taver** y **tatri**:

- ▶ **taver** = tabla de **aristas de vértice**: para cada vértice, almacenamos el índice de una arista adyacente cualquiera:
  - ▶ dado un vértice, permite recuperar todas las aristas, vértices y triángulos adyacentes.
  - ▶ por tanto, permite consultas de adyacencia **vértice-vértice** y **vértice-tríangulo**.
- ▶ **tatri** = tabla de **aristas de triángulo**: para cada triángulo, se almacena el índice de una arista cualquiera adyacente:
  - ▶ dado un triángulo, permite recuperar todas las aristas, vértices y triángulos adyacentes,
  - ▶ por tanto, permite consultas de adyacencia **triángulo-tríangulo** y **vértice-tríangulo** (esta última se puede hacer de dos formas).

## Sección 4. Transformaciones geométricas.

- 4.1. Concepto de transformación geométrica (TG).
- 4.2. Transformaciones usuales en IG.
- 4.3. Representación y operaciones sobre matrices.
- 4.4. Transformaciones en OpenGL con el cauce fijo
- 4.5. Gestión de la matriz *modelview* en el cauce programable.

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.1.

Concepto de transformación geométrica (TG)..

## Coordenadas del mundo e instanciación de objetos.

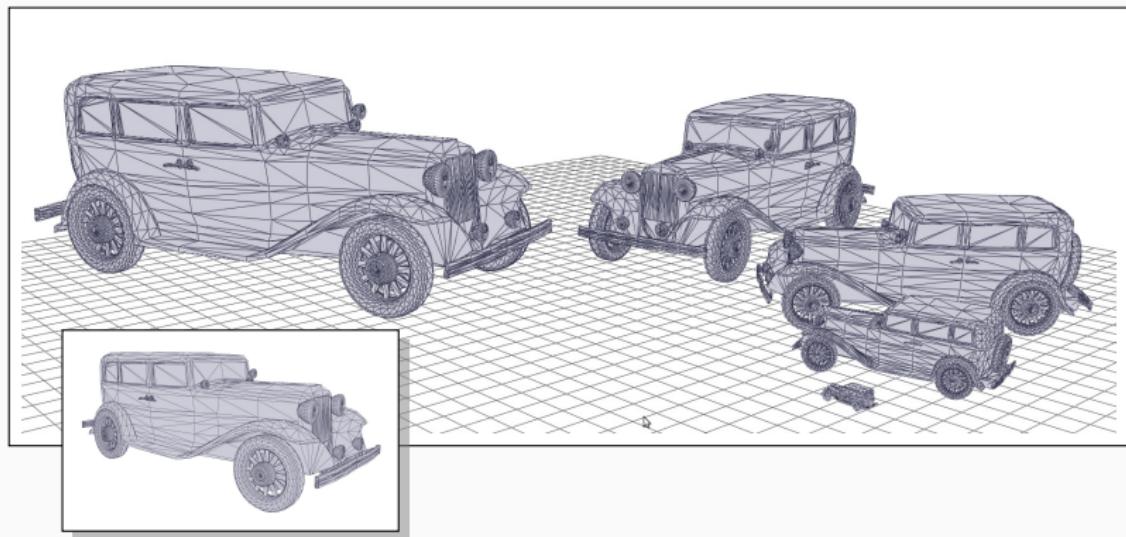
Las mallas que hemos visto en la sección anterior tienen las coordenadas de sus vértices definidas respecto de un sistema de referencia local (coordenadas maestras o locales) , pero :

- ▶ En una escena con varias mallas (o, en general, varios objetos) todos los vértices deben aparecer referidos a un único sistema de referencia común
- ▶ Dicho sistema es el llamado **marco de coordenadas de la escena**, o **marco de coordenadas del mundo**, (*world coordinate system*), y es un marco cartesiano.
- ▶ Las coordenadas de los vértices, respecto de dicho sistema de referencia, se llaman **coordenadas del mundo** (*world coordinates*)
- ▶ Esto permite separar la definición de los objetos (en coordenadas maestra), de su uso en una escena concreta, lo cual es usual en la industria de la infografía 3D actualmente.

# Instanciación de una malla en una escena

Un objeto se define una vez pero se puede **instanciar** muchas veces en una o distintas escenas.

A modo de ejemplo, una malla indexada se podría instanciar varias veces en distintas posiciones, orientaciones y tamaños:



# Transformación geométrica

Para lograr la instanciación, hay que modificar la posición de los vértices de cada objeto, o, lo que es lo mismo, calcular sus coordenadas del mundo a partir de las coordenadas locales o maestras. Para esto se usan transformaciones geométricas

- ▶ Lo más frecuente es que esas transformaciones sean transformaciones afines
- ▶ En este tema veremos las transformaciones afines más comunes:
  - ▶ Rotaciones
  - ▶ Traslaciones
  - ▶ Escalado (uniformes y no uniformes)
  - ▶ Cizallas

Las transformaciones geométricas se usan para instanciar objetos, o, en general, modificar su posición, orientación y tamaño.

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.2.

Transformaciones usuales en IG..

# Transformaciones geométricas usuales en IG

Existen varias transformaciones geométricas simples que son muy útiles en Informática Gráfica para la definición de escenas y animaciones, y que constituyen la base de otras transformaciones:

- ▶ **Traslación:** desplazar todos los puntos del espacio de igual forma, es decir, en la misma dirección y la misma distancia.
- ▶ **Escalado:** estrechar o alargar las figuras en una o varias direcciones.
- ▶ **Rotación:** rotar los puntos un ángulo en torno a un eje
- ▶ **Cizalla:** se puede ver como un desplazamiento de todos los puntos en la misma dirección, pero con distancias distintas.

En lo que sigue veremos con algo de más detalle estas transformaciones básicas, junto con algunas formas útiles de componerlas.

## Transformaciones afines: propiedades.

Es fácil verificar formalmente que una transformación afín tiene las siguientes propiedades:

- ▶ Transforma líneas rectas en líneas rectas (y planos en planos)
- ▶ Transforma dos líneas paralelas en otras dos líneas paralelas.
- ▶ Conserva los ratios entre las longitudes de dos segmentos en dos líneas paralelas.
- ▶ No conserva el área o volumen de los objetos.
- ▶ No conserva las distancias.
- ▶ No conserva los ángulos entre líneas o planos.

# Transformación de traslación en 3D

Si  $\vec{t}$  es un vector de un espacio afín, la transformación de **traslación** por  $\vec{t}$  en dicho espacio es la transformación afín que desplaza cada punto según el vector  $\vec{t}$ , pero no afecta a los vectores, es decir, para cualquier punto  $\dot{p}$  y vector  $\vec{v}$ :

$$T(\dot{p}) \equiv \dot{p} + \vec{t} \qquad T(\vec{v}) \equiv \vec{v}$$

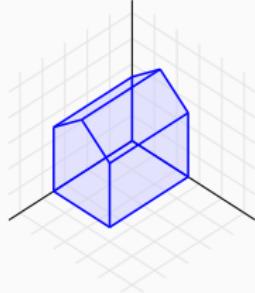
En un marco  $\mathcal{R}$  3D las funciones y la matriz de transformación son:

$$\left. \begin{array}{rcl} x' & = & x + t_x w \\ y' & = & y + t_y w \\ z' & = & z + t_z w \\ w' & = & w \end{array} \right\} \quad \text{Tra}[d_x, d_y, d_z] \equiv \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

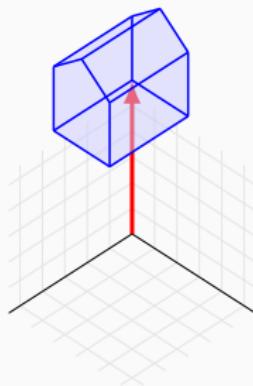
donde  $(t_x, t_y, t_z, 0)^t$  son las coordenadas del  $\vec{t}$  en  $\mathcal{R}$ .

# Ejemplos de traslaciones

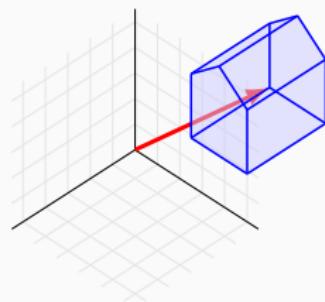
Aquí vemos a modo de ejemplo varias matrices de traslación aplicadas a los puntos de una figura sencilla, en un marco cartesiano



$$\text{Tra}[0, 0, 0]$$



$$\text{Tra}[0, 1.2, 0]$$



$$\text{Tra}[0.7, 0.6, -0.5]$$

# Transformación de escalado en 3D

Un **escalado** es una transformación afín  $S$  que escala o multiplica las componentes de un vector paralelas a cada uno de los ejes de un marco arbitrario  $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$ . El punto  $\dot{o}$  no varía. Los factores de escala son  $(s_x, s_y, s_z)$ . Por tanto, se define:

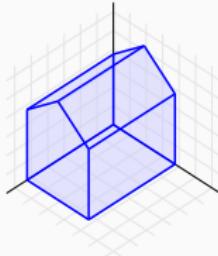
- ▶ Para puntos:  $S(\vec{p}) \equiv \dot{o} + S(\vec{p} - \dot{o})$
- ▶ Para vectores:  $S(c_x \vec{x} + c_y \vec{y} + c_z \vec{z}) \equiv s_x c_x \vec{x} + s_y c_y \vec{y} + s_z c_z \vec{z}$

Por tanto, las funciones y la matriz (expresadas en coordenadas de  $\mathcal{R}$ ) son:

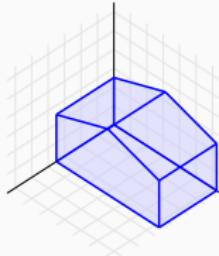
$$\left. \begin{array}{rcl} x' & = & e_x x \\ y' & = & e_y y \\ z' & = & e_z z \\ w' & = & w \end{array} \right\} \quad \text{Esc}[s_x, s_y, s_z] \equiv \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Ejemplos de transformaciones de escalado

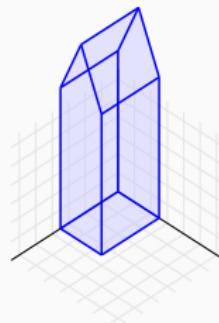
Aquí vemos varios ejemplos de la transformación de escalado.



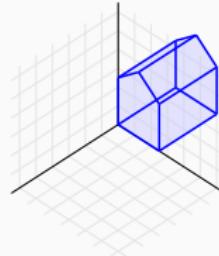
Esc[1.5, 1.5, 1.5]  
uniforme  
 $e_x = e_y = e_z$



Esc[2.5, 1, 1]  
no uniforme  
 $e_x \neq e_y = e_z$



Esc[1, 3, 1]  
no uniforme  
 $e_y \neq e_x = e_z$



Esc[1, 1, -1]  
espejo  
 $e_z < 0$

Las transformaciones de escalado no uniforme no conservan los ángulos, pero los escalados uniformes sí lo hacen.

# Transformaciones de cizalla

Una transformación de cizalla (*shear*)  $C$  es como una translación en la dirección de un eje de un marco  $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$ , pero usando una distancia proporcional (según un factor  $a$ ) a la componente paralela a otro eje. El punto  $\dot{o}$  no varía. A modo de ejemplo, definimos la cizalla que desplaza X en proporción a Y:

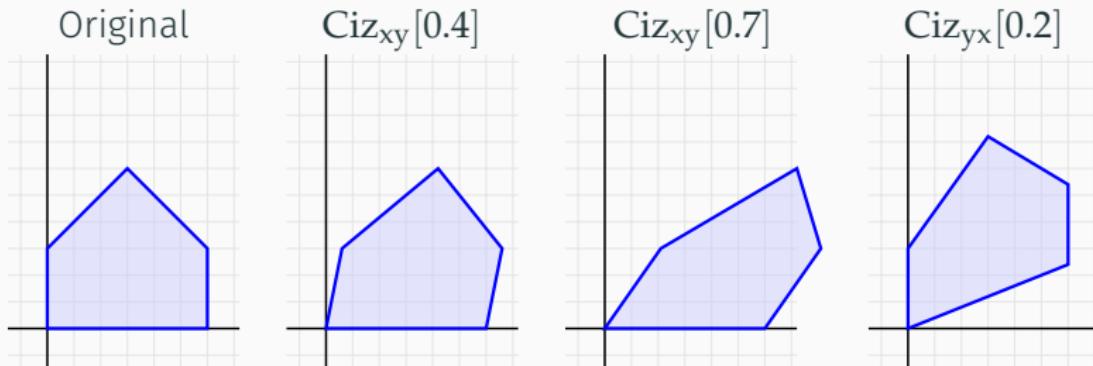
- ▶ Para puntos:  $C(\vec{p}) \equiv \dot{o} + C(\vec{p} - \dot{o})$
- ▶ Para vectores:  $C(c_x \vec{x} + c_y \vec{y} + c_z \vec{z}) \equiv (c_x + ac_y) \vec{x} + c_y \vec{y} + c_z \vec{z}$

Hay 6 posibles cizallas en 3D como esta, aquí vemos las funciones y matriz correspondiente a la definición anterior en coords. de  $\mathcal{R}$ :

$$\left. \begin{array}{rcl} x' & = & x + ay \\ y' & = & y \\ z' & = & z \\ w' & = & w \end{array} \right\} \quad \text{Ciz}_{xy}[a] \equiv \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Ejemplos de transformaciones de cizalla.

En el caso 2D, hay dos posibles cizallas. Aquí las vemos actuando sobre una figura simple:

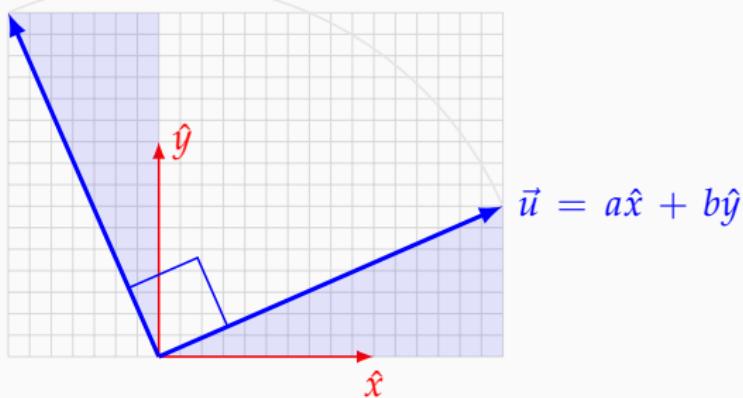


Las transformaciones de cizalla no conservan los ángulos ni las longitudes.

## Rotaciones de $90^\circ$ a izquierdas en 2D

En 2D definimos una transformación afín  $P$  tal que, aplicada a un vector  $\vec{u} = \mathcal{C}(a, b)$ , produce otro vector  $P(\vec{u}) = C(-b, a)$  perpendicular a  $\vec{u}$  (girado  $90^\circ$  a izquierdas).

$$P(\vec{u}) = -b\hat{x} + a\hat{y}$$



aquí  $\mathcal{C} = [\hat{x}, \hat{y}, \delta]$  es un marco cartesiano cualquiera. Si se aplica  $P$  a un punto, se produce una rotación de  $90^\circ$  entorno al origen de  $\mathcal{C}$ .

# Problemas: transformación afín $P$ en 2D

## Problema 2.7.

Demuestra que  $\vec{u}$  y  $P(\vec{u})$  son siempre perpendiculares según la definición anterior (es decir, siempre  $\vec{u} \cdot P(\vec{u}) = 0$ ).

## Problema 2.8.

Describe como se podría definir una rotación hacia la derecha (en el sentido de las agujas del reloj) en lugar de a izquierdas.

## Problema 2.9.

Demuestra que la transformación afín  $P$  (cuando se aplica a vectores, no a puntos) no depende del marco cartesiano  $\mathcal{C}$  con respecto al cual expresamos las coordenadas  $(a, b)$  (en el caso de aplicarla a puntos, la rotación de  $90^\circ$  es entorno al punto origen  $\mathbf{o}$  de  $\mathcal{C}$ ).

## Rotaciones de $90^\circ$ en 3D

En 3D definimos una transformación afín similar, pero ahora hay infinitos vectores perpendiculares a  $\vec{u}$  (todos los que están en el plano perpendicular a  $\vec{u}$ ).

- ▶ Para poder determinar bien la transformación afín usamos un vector  $\vec{e}$  no nulo cualquiera (para cada  $\vec{e}$  se define una transformación distinta).
- ▶ Nombramos las transformación como  $P_{\vec{e}}$  (en lugar de simplemente  $P$ ) y la definimos usando el producto vectorial, así:

$$P_{\vec{e}}(\vec{u}) \equiv \vec{e} \times \vec{u}$$

- ▶ Produce un vector perpendicular a  $\vec{u}$  siempre (el único que también será perpendicular a  $\vec{e}$ ), por las propiedades del producto vectorial.

# Matrices para la obtención un vector perpendicular

Las matrices correspondientes a  $P$  y  $P_{\vec{e}}$ , definidas respecto a un marco cartesiano  $\mathcal{C}$  cualquiera, son:

- En 2D:

$$P \equiv \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- En 3D:

$$P_{\mathbf{e}} \equiv \begin{pmatrix} 0 & -e_z & e_y & 0 \\ e_z & 0 & -e_x & 0 \\ -e_y & e_x & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde  $\mathbf{e} = (e_x, e_y, e_z, 0)^t$  son las coordenadas de  $\vec{e}$  respecto de  $\mathcal{C}$

- Estas matrices, si se aplican a puntos, suponen rotaciones de  $90^\circ$  entorno al origen de  $\mathcal{C}$

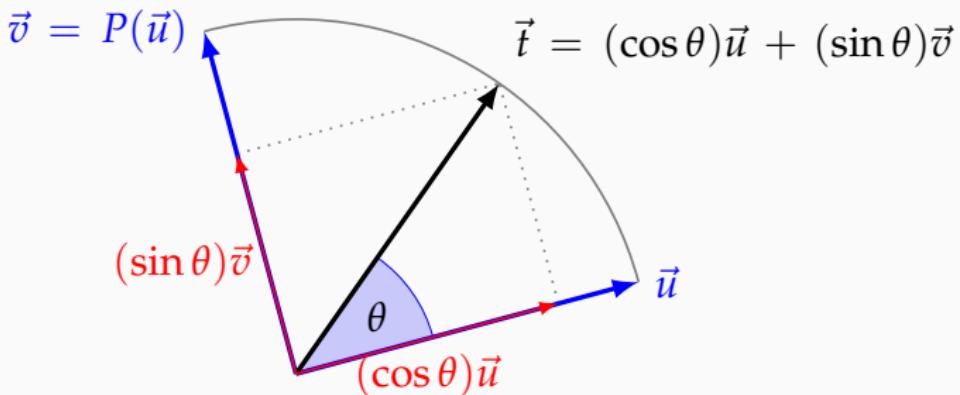
# Rotaciones arbitrarias en 2D

Una transformación de rotación  $R_\theta$  en 2D por un ángulo  $\theta$  entorno a un punto  $\dot{o}$  transforma un vector  $\vec{u}$  o un punto  $\dot{p}$  de esta forma:

$$R_\theta(\vec{u}) = (\cos \theta)\vec{u} + (\sin \theta)P(\vec{u})$$

$$R_\theta(\dot{p}) = \dot{o} + R_\theta(\dot{p} - \dot{o})$$

El vector rotado  $\vec{t} = R_\theta(\vec{u})$  es una combinación de  $\vec{u}$  y  $\vec{v} = P(\vec{u})$ :



## Matriz de rotación arbitraria en 2D

Para obtener la matriz de rotación  $\text{Rot}[\theta]$ , relativa a un marco cartesiano cualquiera  $\mathcal{C}$ , usamos la matriz identidad  $I$  y la matriz  $P$ . Si  $\mathbf{u}$  son las coordenadas homogéneas de un vector en  $\mathcal{C}$ , se cumple:

$$\begin{aligned}\text{Rot}[\theta] \mathbf{u} &= (\cos \theta) \mathbf{u} + (\sin \theta) P \mathbf{u} = (\cos \theta) I \mathbf{u} + (\sin \theta) P \mathbf{u} \\ &= [(\cos \theta) I + (\sin \theta) P] \mathbf{u}\end{aligned}$$

Por tanto, deducimos como es la matriz de rotación:

$$\text{Rot}[\theta] \equiv (\cos \theta) I + (\sin \theta) P \equiv \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

y las expresiones:

$$\begin{aligned}x' &= (\cos \theta)x - (\sin \theta)y \\ y' &= (\sin \theta)x + (\cos \theta)y\end{aligned}$$

## Ejemplo de rotación arbitraria en 2D

Aquí vemos el ejemplo de una rotación por un ángulo  $\theta$  (mayor que cero)



Las rotaciones son **transformaciones rígidas**: conservan los ángulos y las distancias.

## Problema: invarianza ante rotaciones 2D

### Problema 2.10.

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo  $\theta$  y vectores  $\vec{a}$  y  $\vec{b}$  se cumple:

$$R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) = \vec{a} \cdot \vec{b}$$

(usa las coordenadas de  $\vec{a}$  y  $\vec{b}$  en un marco cartesiano cualquiera)

### Problema 2.11.

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo  $\theta$  y vector  $\vec{v}$ , se cumple:

$$\| R_\theta(\vec{v}) \| = \| \vec{v} \|$$

# Transformaciones de rotación elementales en 3D.

En un espacio afín 3D consideramos un marco cartesiano

$\mathcal{C} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$  y una transformación de rotación de  $\theta$  radianes, con eje en un vector  $\vec{e}$  no nulo.

- ▶ La rotación es en sentido anti-horario cuando  $\theta > 0$  (según se mira hacia  $\dot{o}$  desde la punta del vector  $\vec{e}$ ).
- ▶ Llamamos rotaciones **elementales** a las que tienen como eje los versores de  $\mathcal{C}$ . Hay 3 rotaciones elementales (una por eje). A las correspondientes matrices las llamamos:

$$\text{Rot}_x[\theta] \equiv \text{Rot}[\theta, (1, 0, 0)]$$

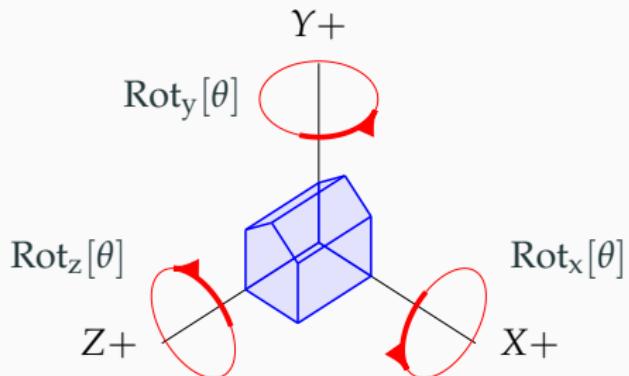
$$\text{Rot}_y[\theta] \equiv \text{Rot}[\theta, (0, 1, 0)]$$

$$\text{Rot}_z[\theta] \equiv \text{Rot}[\theta, (0, 0, 1)]$$

Las rotaciones elementales en 3D se pueden definir como las rotaciones arbitrarias en 2D, afectando a dos coordenadas y dejando invariantes la otra.

# Transformaciones de rotación elementales en 3D.

En la figura, los círculos simbolizan indican como se rotan los puntos bajo cada rotación elemental:



- ▶ Son rotaciones **siempre en sentido anti-horario** (para  $\theta > 0$ ) cuando se mira hacia el origen desde la rama positiva del eje de giro).

# Expresiones de las rotaciones elementales

Definiendo  $c \equiv \cos \theta$  y  $s \equiv \sin \theta$ , las expresiones son:

$$\text{Rot}_x[\alpha]$$

$$\begin{aligned} x' &= x \\ y' &= cy - sz \\ z' &= sy + cz \end{aligned}$$

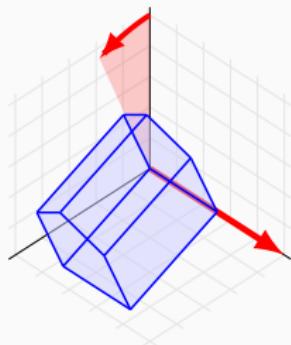
$$\text{Rot}_y[\alpha]$$

$$\begin{aligned} x' &= cx + sz \\ y' &= y \\ z' &= -sx + cz \end{aligned}$$

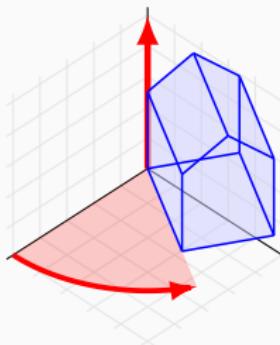
$$\text{Rot}_z[\alpha]$$

$$\begin{aligned} x' &= cx - sy \\ y' &= sx + cy \\ z' &= z \end{aligned}$$

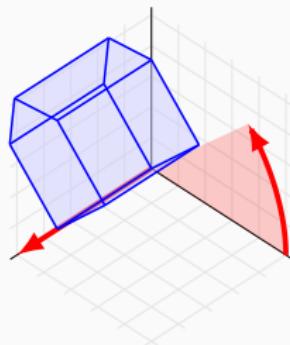
$$\text{Rot}_x[23^\circ]$$



$$\text{Rot}_y[60^\circ]$$



$$\text{Rot}_z[45^\circ]$$



# Problemas: invarianza ante rotaciones elementales 3D

## Problema 2.12.

Demuestra que el producto escalar de vectores en 3D es invariante por rotaciones elementales (usa tu solución al problema 10)

## Problema 2.13.

Demuestra que las rotaciones elementales en 3D no modifican la longitud de un vector (usa tu solución al problema 11)

## Problema 2.14.

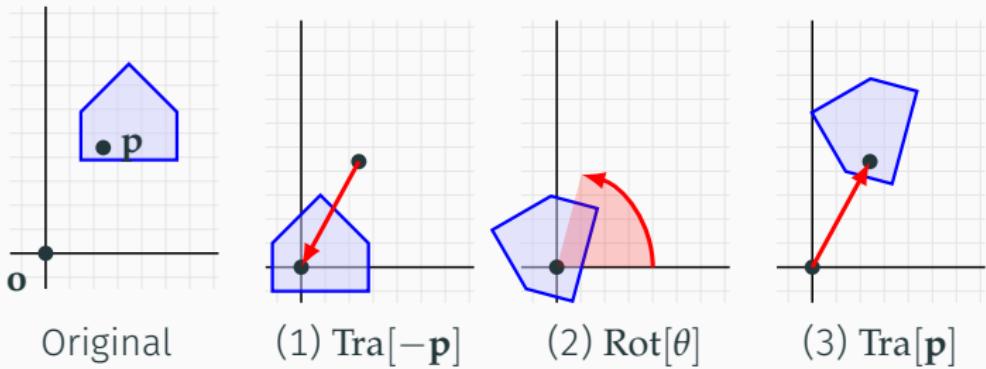
Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualquiera dos vectores  $\vec{a}$  y  $\vec{b}$  y un ángulo  $\theta$ , se cumple:

$$R_\theta(\vec{a} \times \vec{b}) = R_\theta(\vec{a}) \times R_\theta(\vec{b})$$

# Rotaciones sobre puntos arbitrarios en 2D y 3D

En un marco cartesiano  $\mathcal{C}$ , la matriz de rotación entorno a un punto  $\dot{p}$  arbitrario, distinto del origen  $\dot{o}$  del marco, se consigue componiendo (1) traslación al origen de  $\mathcal{C}$  (por el vector  $\dot{o} - \dot{p}$ ), (2) rotación por  $\theta$  (en torno al origen  $\dot{o}$ ) y (3) traslación inversa (por  $\dot{p} - \dot{o}$ ). Por tanto:

$$\text{Rot}[\theta, \mathbf{p}] = \text{Tra}[\mathbf{p}] \cdot \text{Rot}[\theta] \cdot \text{Tra}[-\mathbf{p}]$$



(donde  $\mathbf{p}$  son las coords. de  $\dot{p}$  en  $\mathcal{C}$ )

## Rotaciones de eje arbitrario en 3D (1/2)

En 3D una rotación  $R_\theta$  tiene como **eje de rotación** una línea que pasa por  $\dot{o}$ , paralela a un vector  $\hat{e}$  (lo suponemos de longitud unidad).

- ▶ Para un punto  $\dot{p}$  se define en términos de la rotación de vectores:

$$R_\theta(\dot{p}) = \dot{o} + R_\theta(\dot{p} - \dot{o})$$

- ▶ Si  $\vec{u}$  es un vector perpendicular a  $\hat{e}$ , entonces  $\vec{u}$  rota igual que en 2D, pero en el plano perpendicular a  $\hat{e}$ . Este plano es generado por  $\vec{u}$  y otro vector  $\vec{v}$ , perpendicular a  $\hat{e}$  y  $\vec{u}$ , por tanto:

$$R_\theta(\vec{u}) = (\cos \theta)\vec{u} + (\sin \theta)\vec{v}$$

donde  $\vec{v}$  se obtiene mediante la transformación  $P_{\hat{e}}$  (prod. vect.):

$$\vec{v} \equiv \hat{e} \times \vec{u} = P_{\hat{e}}(\vec{u})$$

## Rotaciones de eje arbitrario en 3D (2/2)

Si  $\vec{s}$  es un vector cualquiera (no necesariamente perpendicular a  $\hat{e}$ ) descomponemos  $\vec{s}$  en dos componentes: una ( $\vec{w}$ ) es paralela a  $\hat{e}$  y otra ( $\vec{u}$ ) es perpendicular a  $\hat{e}$ . El vector  $\vec{w}$  no rota, mientras que  $\vec{u}$  lo hace como antes. Por tanto:

$$R_\theta(\vec{s}) = \vec{w} + (\cos \theta)\vec{u} + (\sin \theta)\vec{v}$$

$$\begin{aligned}\vec{w} &= \vec{s} - \vec{u} \\&= \vec{s} + P_{\hat{e}}^2(\vec{s}) \\&= (\vec{s} \cdot \hat{e})\hat{e} \\&\quad \text{Diagrama: Un sistema de coordenadas 3D con el origen } O. Un vector } \vec{s} \text{ se descompone en } \vec{w} \text{ (paralelo a } \hat{e}) \text{ y } \vec{u} \text{ (perpendicular a } \hat{e}). \text{ El resultado de la rotación } R_\theta(\vec{s}) \text{ es } \vec{v} = \hat{e} \times \vec{s} = P_{\hat{e}}(\vec{s}), \text{ que es perpendicular a } \hat{e}. \\&\quad \vec{u} = -\hat{e} \times \vec{v} \\&= -P_{\hat{e}}^2(\vec{s}) \\&= \vec{s} - (\vec{s} \cdot \hat{e})\hat{e}\end{aligned}$$

Los vectores  $\vec{u}, \vec{v}$  y  $\vec{w}$  forman un marco de referencia ortonormal.

## Fórmula de Rodrigues y expresión matricial.

En un marco cartesiano  $\mathcal{C}$  cualquiera podemos escribir las coordenadas  $\mathbf{s}'$  del vector rotado, deduciendo la **fórmula de Rodrigues de la rotación**:

$$\begin{aligned}\mathbf{s}' &= \mathbf{w} + (\cos \theta)\mathbf{u} + (\sin \theta)\mathbf{v} \\&= \mathbf{w} + (\cos \theta)(\mathbf{s} - \mathbf{w}) + (\sin \theta)\mathbf{e} \times \mathbf{s} \\&= (\cos \theta)\mathbf{s} + (1 - \cos \theta)\mathbf{w} + (\sin \theta)\mathbf{e} \times \mathbf{s} \\&= (\cos \theta)\mathbf{s} + (1 - \cos \theta)(\mathbf{s} \cdot \mathbf{e})\mathbf{e} + (\sin \theta)\mathbf{e} \times \mathbf{s}\end{aligned}$$

Donde  $\mathbf{e}, \mathbf{s}, \mathbf{w}, \mathbf{u}$  y  $\mathbf{v}$  son las coordenadas en  $\mathcal{C}$  de los vectores  $\vec{e}, \vec{s}, \vec{w}, \vec{u}$  y  $\vec{v}$ . Usando las matrices  $I$  y  $P_{\mathbf{e}}$  podemos escribir:

$$\begin{aligned}\mathbf{s}' &= \mathbf{w} + (\cos \theta)\mathbf{u} + (\sin \theta)\mathbf{v} \\&= \left( I\mathbf{s} + P_{\mathbf{e}}^2 \mathbf{s} \right) - (\cos \theta)P_{\mathbf{e}}^2 \mathbf{s} + (\sin \theta)P_{\mathbf{e}} \mathbf{s} \\&= \left[ I + (\sin \theta)P_{\mathbf{e}} + (1 - \cos \theta)P_{\mathbf{e}}^2 \right] \mathbf{s}\end{aligned}$$

# Matriz de rotación de eje arbitrario en 3D

De la última igualdad se deduce cual es la matriz de rotación en 3D

$$\text{Rot}[\theta, \mathbf{e}] = I + (\sin \theta) P_{\mathbf{e}} + (1 - \cos \theta) P_{\mathbf{e}}^2$$

Haciendo la composición de las matrices, obtenemos la definición explícita:

$$\text{Rot}[\theta, \mathbf{e}] \equiv \begin{pmatrix} a_{00} + c & a_{01} - s e_2 & a_{02} + s e_1 & 0 \\ a_{10} + s e_2 & a_{11} + c & a_{12} - s e_0 & 0 \\ a_{20} - s e_1 & a_{21} + s e_0 & a_{22} + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde  $\mathbf{e} \equiv (e_0, e_1, e_2, 0)^t$  son las coordenadas en  $\mathcal{C}$  de  $\vec{e}$ , y

$$s \equiv \sin \theta$$

$$c \equiv \cos \theta$$

$$a_{ij} \equiv (1 - c)e_i e_j$$

## Resumen: matrices de transformación 3D usuales

$$\text{Esc}[s_x, s_y, s_z] = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Tra}[d_x, d_y, d_z] = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Ciz}_{xy}[a] = \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rot}_x[\alpha] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Rot}_y[\alpha] = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Rot}_z[\alpha] = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde:  $c \equiv \cos(\alpha)$  y  $s \equiv \sin(\alpha)$ , y  $\alpha$  es el ángulo de rotación, en radianes

## Transformación de normales (1/2)

Si transformamos las posiciones de todos los vértices de una malla usando una matriz  $4 \times 4 A$  (matriz de modelado), también querremos transformar los vectores normales de dichos vértices, en caso de que la malla tenga normales:

- (1) Representamos las normales como vectores columna ( $1 \times 3$ ), igual que los vectores (siempre  $w = 0$ )
- (2) Al igual que los vectores, las normales no se ven afectadas por traslaciones, es decir, podemos considerar  $A$  como una matriz  $3 \times 3$  sin términos de traslaciones.
- (3) Supongamos que un vértice (en la malla original) tiene una normal  $\mathbf{n}$ , y que el vector tangente a la superficie en el vértice es  $\mathbf{t}$ , puesto que  $\mathbf{t}$  y  $\mathbf{n}$  son perpendiculares, se cumple

$$\mathbf{t} \cdot \mathbf{n} = 0 \iff \mathbf{t}^T \mathbf{n} = 0$$

## Transformación de normales (2/2)

- (4) Al transformar la superficie por  $A$  el vector tangente transformado es  $A\mathbf{t}$  (los vectores tangentes son vectores libres y se transforman como cualquier otro vector).
- (5) El vector normal se transforma por una matriz  $3 \times 3$   $U$  (desconocida en principio), si queremos que la transformación preserve la perpendicularidad se debe cumplir

$$(A\mathbf{t}) \cdot (U\mathbf{n}) = 0 \iff (A\mathbf{t})^T(U\mathbf{n}) = (\mathbf{t}^T A^T)(U\mathbf{n}) = 0$$

De (5) deducimos que  $\mathbf{t}^T(A^T U)\mathbf{n} = 0$ , luego por (3) deducimos que  $A^T U = I$ , y así sabemos que  $U$  es la **inversa de la traspuesta** de  $A$ :

$$U = (A^T)^{-1} = (A^{-1})^T$$

(aplicar  $U$  a una normal de longitud 1 puede producir una normal de long.  $\neq 1$ ).

Informática Gráfica, curso 2021-22.

**Teoría. Tema 2. Modelos de objetos.**

Sección 4. Transformaciones geométricas

Subsección 4.3.

Representación y operaciones sobre matrices..

# El tipo Matriz4f

Cada variable de tipo **Matriz4f** almacena los 16 valores (**float**) de forma contigua, proporciona operaciones para acceder a una matriz y multiplicarla por otra matriz o por una tupla de flotantes (incluir **matrizg.h**)

```
// 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 c4a, c4b ={1.0,2.0,3.5,1.0}; Tupla3f c3a, c3b = {1.0,1.6,2.8};
c4a = m2*c4b ; c3a = m2 * c3b ;
// conversión a puntero a 16 flotantes (float *) (formato compatible con OpenGL)
float * pm = m ; const float * pcm = m ;
// escritura en la salida estándar (varias líneas)
cout << m << endl ;
```

# Matrices más usuales

También podemos usar funciones C++ para calcular las matrices más usuales (incluir `matrices-tr.h`)

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

// devuelve matrices de traslación, escalado y rotación (eje arbitrario) en 3D
Matriz4f MAT_Traslacion( const float d[3] );
Matriz4f MAT_Traslacion( const float dx, const float dy, const float dz );
Matriz4f MAT_Escalado ( const float sx, const float sy, const float sz ) ;
Matriz4f MAT_Rotacion ( const float ang_gra, const Tupla3f & eje ) ;
Matriz4f MAT_Rotacion ( const float ang_gra,
                        const float ex, const float ey, const float ez );

// funciones auxiliares (construir por filas o por columnas, obtener inversa o transpuesta)
Matriz4f MAT_Columnas      ( const Tupla3f colum[3] );
Matriz4f MAT_Filas         ( const Tupla3f fila[3] );
Matriz4f MAT_Filas         ( const Tupla3f & fila0, const Tupla3f & fila1,
                            const Tupla3f & fila2 );
Matriz4f MAT_Inversa       ( const Matriz4f & m );
Matriz4f MAT_Transpuesta3x3( const Matriz4f & org ) ;
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.4.

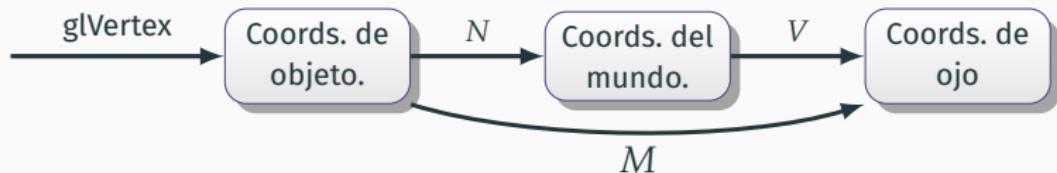
Transformaciones en OpenGL con el cauce fijo.

# La matriz *Modelview* en OpenGL.

En el cauce fijo, OpenGL almacena, como parte de su estado, una matriz  $4 \times 4 M$  que codifica una transformación geométrica, y que se llama *modelview matrix* (**matriz de modelado y vista**). Dicha matriz se puede ver como la composición de otras dos,  $V$  y  $N$ :

- ▶  $N \equiv$  **matriz de modelado** (*modeling matrix*): posiciona los puntos en su lugar en coordenadas del mundo.
- ▶  $V \equiv$  **matriz de vista** (*view matrix*): posiciona los puntos en su lugar en coordenadas relativas a la cámara

La matriz *modelview*  $M$  se aplica a todos los puntos generados con **glVertex**:



# Especificación de la matriz de modelado en el cauce fijo

La matriz *modelview* se puede especificar en OpenGL mediante estos pasos:

1. Hacer la llamada **glMatrixMode(GL\_MODELVIEW)**, para indicar que las siguientes operaciones actúan sobre la matriz *modelview*  $M$ .
2. usar **glLoadIdentity()** para hacer  $M$  igual a la matriz identidad ( $M := \text{Ide}$ ).
3. usar **gluLookAt** u otras para componer una matriz de vista  $V$  en  $M$  ( $M := M \cdot V$ )
4. usar una (o varias) llamadas para componer una matriz de modelado con  $N$  (llamadas a las funciones **glMultMatrix**, **glScale**, **glRotate**, o **glTranslate**) ( $M := M \cdot N$ )

Al final, se tiene  $M = V \cdot N$ . OpenGL construye  $M$  componiendo las matrices que se le proporcionan en los pasos 3 y 4.

# Funciones para la *modelview* en el cauce fijo.

OpenGL incorpora estas llamadas (también con d en lugar de f)

```
// hace  $M := M \cdot \text{Rot}[\alpha, e_x, e_y, e_z]$  ( $\alpha$  en grados)
glRotatef( GLfloat  $\alpha$ , GLfloat  $e_x$ , GLfloat  $e_y$ , GLfloat  $e_z$  )

// hace  $M := M \cdot \text{Tra}[d_x, d_y, d_z]$ .
glTranslatef( GLfloat  $d_x$ , GLfloat  $d_y$ , GLfloat  $d_z$  )

// hace  $M := M \cdot \text{Esc}[s_x, s_y, s_z]$ .
glScalef( GLfloat  $s_x$ , GLfloat  $s_y$ , GLfloat  $s_z$  )

// hace  $M := M A$ , donde  $A = (a_{ij})$  es una matriz (por columnas)
GLfloat  $A[16] = \{ a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, // \text{columna 0}$ 
                     $a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1}, // \text{columna 1}$ 
                     $a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2}, // \text{columna 2}$ 
                     $a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3} // \text{columna 3}$ 
                } ;
glMultMatrixf( GLfloat *  $A$  ) ;
```

## Generación de *modelview* por composición.

En general, el código permite ir componiendo matrices con la matriz *modelview*:

```
glMatrixMode(GL_MODELVIEW); // indicamos que, a partir de aquí,  
                           // se modifica la matriz modelview, M.  
glLoadIdentity() ; // hacemos  $M := \text{Id}_e$  (matriz identidad)  
glMultMatrix( T3 ); // hacemos  $M := M \cdot T_3$   
glMultMatrix( T2 ); // hacemos  $M := M \cdot T_2$   
glMultMatrix( T1 ); // hacemos  $M := M \cdot T_1$ 
```

Al final de ejecutar estas instrucciones, la matriz *modelview* será:

$$M = T_3 \cdot T_2 \cdot T_1$$

es decir, el efecto de  $M$  es igual al efecto de  $T_1$  seguido de  $T_2$  seguido de  $T_3$

# Ejemplo de un programa usando el cauce fijo.

La llamada **obj->visualizarGL(cv)** envía los vértices del objeto **obj** en coordenadas maestras. El siguiente trozo de código manipula la matriz *modelview* de forma que dicho objeto se viese

- (1) rotado alrededor del eje X un ángulo igual a 45 grados, y
- (2) desplazado según el vector (5,6,7)

```
// Definir matriz 'modelview' en el Estado de OpenGL (OpenGL 'clásico')
glMatrixMode( GL_MODELVIEW );      // vamos a modificar 'modelview' (M)
glLoadIdentity();                  // M = Ide
gluLookAt( ox,oy,oz, ax,ay,az,   // M := M · V
           ux,uy,uz );           //      (V ≡ matriz de vista, tipo look-at)
glTranslatef( 5.0,6.0,7.0 );       // M := M · Tra[5,6,7]
glRotatef( 45.0, 1.0,0.0,0.0 );  // M := M · Rot[45º,x]

// Visualizar con M ≡ V · Tra[5,6,7] · Rot[45º,x]
obj->visualizarGL( cv ) ;
```

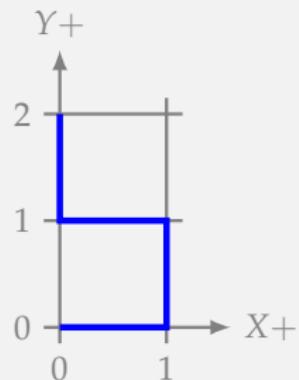
# Problema: motivo básico para transformaciones

## Problema 2.15.

Escribe una función llamada **gancho** para dibujar con OpenGL en modo diferido la polilínea de la figura (cada segmento recto tiene longitud unidad, y el extremo inferior está en el origen).

La función debe ser neutra respecto de la matriz *modelview*, el color o el grosor de la línea, es decir, usará la matriz *modelview*, el color y grosor del estado de OpenGL en el momento de la llamada (y no los cambia).

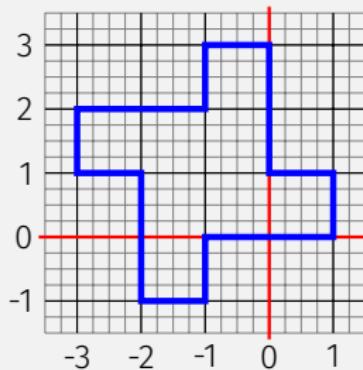
Usa la plantilla en el repositorio **opengl-minimo** para esto.



# Problema: múltiples instancias del motivo básico

## Problema 2.16.

Usando (exclusivamente) la función **gancho** del problema anterior, construye otra función (**gancho\_x4**) para dibujar con OpenGL, usando el **cauce fijo**, el polígono que aparece en la figura:



Usa exclusivamente **glTranslatef** y **glRotatef** (no **glPushMatrix** ni **glPopMatrix**).

# Problema: cálculo directo de *modelview*

## Problema 2.17.

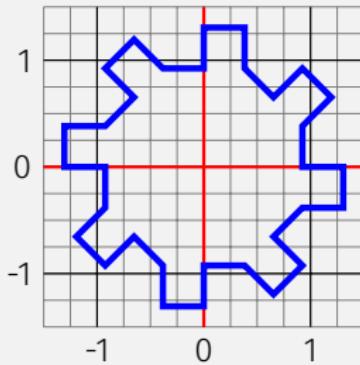
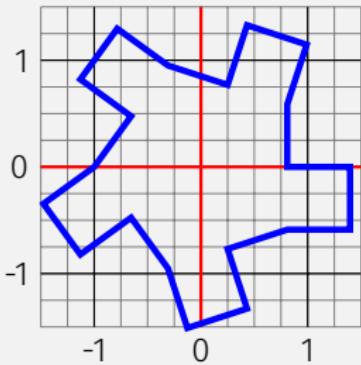
Escribe el pseudocódigo OpenGL otra función (**gancho\_2p**) para dibujar esa misma figura, pero escalada y rotada de forma que sus extremos coincidan con dos puntos arbitrarios disintos  $\vec{p}_0$  y  $\vec{p}_1$ , puntos cuyas coordenadas de mundo son  $\mathbf{p}_0 = (x_0, y_0, 1)^t$  y  $\mathbf{p}_1 = (x_1, y_1, 1)^t$ . Estas coordenadas se pasan como parámetro a dicha función (como **Tupla3f**)

Escribe una solución (a) acumulando matrices de rotación, traslación y escalado en la matriz *modelview* de OpenGL. Escribe otra solución (b) en la cual la matriz *modelview* se calcula directamente sin necesidad de usar funciones trigonométricas (como lo son el arcotangente, el seno, coseno, arcoseno o arcocoseno).

# Problema: figura compleja

## Problema 2.18.

Uso la función del problema anterior para construir estas dos nuevas figuras, en las cuales hay un número variable de instancias de la figura original, dispuestas en círculo (vemos los ejemplos para 5 y 8 instancias, respectivamente).



# Métodos en Cauce para manipulación de *modelview*

La clase base **Cauce** incluye métodos para manipulación de la matriz *modelview* ( $M$ ) Las clases derivadas de **Cauce** tienen distintas implementaciones. Los métodos son:

- ▶ **fijarMatrizVista( $V$ )**

fijar  $V$  como matriz *modelview* (se asume que  $V$  tiene una matriz de vista, de tipo **Matriz4f**), es decir, hace:

$$M := V$$

- ▶ **compMM( $N$ ):**

componer en la matriz *modelview* actual  $M$  una matriz de modelado  $N$  por la derecha (tambien **Matriz4f**), es decir, hace:

$$M := MN$$

## Uso del interfaz de la clase Cauce.

El uso del interfaz público de la clase permite que el código pueda funcionar tanto con el cauce fijo como con otros tipos de cauces.

El ejemplo anterior lo podemos implementar equivalentemente así:

```
// Definir matriz 'modelview' en el Estado de OpenGL (usando clase Cauce)
cauce->fijarMatrizVista( MAT_LookAt( o, a, u )); // M := V
cauce->compMM( MAT_Traslacion( 5.0,6.0,7.0 )); // M := M · Tra[5,6,7]
cauce->compMM( MAT_Rotacion( 45.0, {1.0,0.0,0.0} )); // M := M · Rot[45°,x]

// Visualizar con M ≡ V · Tra[5,6,7] · Rot[45°,x]
obj->visualizarGL( cv );
```

Este código funciona igual tanto si **cauce** apunta al cauce fijo como al cauce programable.

# Implementación en el cauce fijo.

Los métodos **fijarMatrizVista** y **compMM** se implementan en el cauce fijo usando la funcionalidad de OpenGL para manipular la matriz modelview ( $M$ ) de su estado interno:

```
void CauceFijo::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glMultMatrixf( nue_mat_vista );
}
```

```
void CauceFijo::compMM( const Matriz4f & mat_comp )
{
    glMatrixMode( GL_MODELVIEW );
    glMultMatrixf( mat_comp );
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.5.

Gestión de la matriz *modelview* en el cauce programable..

# Gestión de matrices en el cauce programable

La funcionalidad de OpenGL relacionada con la gestión de matrices está declarada **obsoleta** en OpenGL versión 3.1 y **eliminada** de OpenGL 3.3 y posteriores (igualmente, ha sido eliminada de OpenGL ES 2.0).

- ▶ Es necesario escribir el código para la gestión de las matrices (o usar alguna librería a tal efecto).
- ▶ En nuestro caso, podemos usar el tipo **Matriz4f**, con las operaciones ya vistas.

Vamos a ver ejemplos para gestionar desde la aplicación la matriz de modelado, sin usar la funcionalidad descrita para el cauce fijo:

- ▶ Esto requiere pasar la matrices de modelado y vista al vertex shader como parámetros uniform.
- ▶ Veremos como hacerlo en un ejemplo de vertex shader.

## Código del vertex shader

En este ejemplo, el uniform **matrizMV** tiene la matriz *modelview*:

```
// parámetros de entrada uniform (iguales en todos los vértices de cada primitiva)
uniform mat4 matrizMV ;          // matriz 4x4 de transf. de coord. de vértices
uniform mat4 matrizMV_nor;        // matriz 4x4 de transf. de normales
uniform mat4 matrizP ;           // matriz 4x4 de proyección (produce coord.pantalla)

// variables de salida varying (atributos de vértice: serán interpolados a pixels)
varying vec4 var_posic_ec;       // posición (en coords de cámara)
varying vec3 var_normal_ec;      // normal (en coords. de cámara)
varying vec4 var_color;          // color
varying vec2 var_coord_text;     // coordenadas de textura

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord0

void main() // escribe variables 'varying', más 'gl_Position'
{
    var_posic_ec = matrizMV * gl_Vertex;          // transf. coord. recibida
    var_normal_ec = matrizMV_nor * gl_Normal;       // transf. normal recibida
    var_color = gl_Color;                          // usar color enviado
    var_coord_text = gl_MultiTexCoord0.st;          // usa cc.t. enviadas
    gl_Position = matrizP * var_posic_ec;          // proyecta a pantalla
}
```

# Fijar valores de los parámetros uniform

El vertex shader tiene como parámetros de entrada:

- ▶ la matriz *modelview* (uniform **matrizMV**)
- ▶ la matriz *modelview* para normales (uniform **matrizMV\_nor**).

La aplicación debe de:

- ▶ Obtener la localización de ambos uniform al inicio (usando **glGetUniformLocation**)
- ▶ Fijar los valores de ambos parámetros uniform antes de visualizar una escena o conjunto de objetos (con **glUniformMatrix4fv**, usando la localización)

# Fijar la matriz de vista en el cauce programable

La clase **CauceProgramable** contiene (como var. de instancia):

- ▶ matriz de vista (**mat\_vista**),
- ▶ matriz de modelado (**mat\_modelado**)
- ▶ matriz de modelado para normales (**mat\_modelado\_nor**)

La implementación de **fijarMatrizVista** es esta:

```
void CauceProgramable::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    // inicializar matrices (variables de instancia)
    mat_vista      = nue_mat_vista; // guardar copia de V
    mat_modelado   = MAT_Ident();   // 'reset' de la matr. de modelado
    mat_modelado_nor = MAT_Ident(); // 'reset' de la matr. de modelado de norm.

    // inicializar pila
    .....

    // fijar uniforms matrizMV y matrizMV_nor
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_matrizMV,      1, GL_FALSE, mat_vista );
    glUniformMatrix4fv( loc_matrizMV_nor, 1, GL_FALSE, mat_vista );
}
```

# Componer una matriz de modelado en el cauce programable

En este caso se deben actualizar la matriz de modelado y la matriz de modelado de normales (usando la transpuesta de la inversa de la matriz a componer), y después actualizar los uniforms a su nuevo valor:

```
void CauceProgramable::compMM( const Matriz4f & mat_comp )
{
    // actualizar matrices de modelado
    const Matriz4f
        mat_comp_nor = MAT_Transpuesta3x3( MAT_Inversa( mat_comp ) );

    mat_modelado      = mat_modelado*mat_comp;
    mat_modelado_nor = mat_modelado_nor*mat_comp_nor ;

    // fijar uniforms matrizMV y matrizMV_nor
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_matrizMV,      1,GL_FALSE,mat_vista*mat_modelado);
    glUniformMatrix4fv( loc_matrizMV_nor,1,GL_FALSE,mat_vista*mat_modelado_nor);
}
```

## Sección 5. Modelos jerárquicos. Representación y visualización..

- 5.1. Modelos Jerárquicos y grafo de escena.
- 5.2. Grafos tipo PHIGS. Ejemplo.
- 5.3. Representación de grafos.
- 5.4. Visualización de grafos en OpenGL.
- 5.5. Grafos parametrizables.

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.1.

Modelos Jerárquicos y grafo de escena..

## La escena como vector de objetos.

En una escena típica actual se incluyen muchas instancias distintas de muchas mallas u **objetos geométricos** (una escena  $S$  es una serie de  $n$  objetos  $S = \{O_1, O_2, \dots, O_n\}$ )

- ▶ Cada objeto  $O_i$  se incluye con una transformación  $T_i$ .
- ▶ Un mismo objeto puede instanciarse más de una vez ( $O_i = O_j$ ), pero con distintas transformaciones ( $T_i \neq T_j$ ).
- ▶ Cada transformación sirve para situar al objeto (definido en su propio marco de referencia  $\mathcal{R}_i$ ) en su lugar en relación al marco de referencia de la escena (es el **marco de referencia del mundo**, lo llamamos  $\mathcal{W}$ ).
- ▶ Podemos ver la matriz  $T_i$  como algo que sirve para convertir desde coordenadas relativas a  $\mathcal{R}_i$  hacia coordenadas relativas a  $\mathcal{W}$

## Jerarquías: objetos simples y compuestos. DAGs

A pesar de ser versátil, el esquema anterior es complejo de usar para escenarios muy complejos. En estos casos se usan **modelos jerárquicos**.

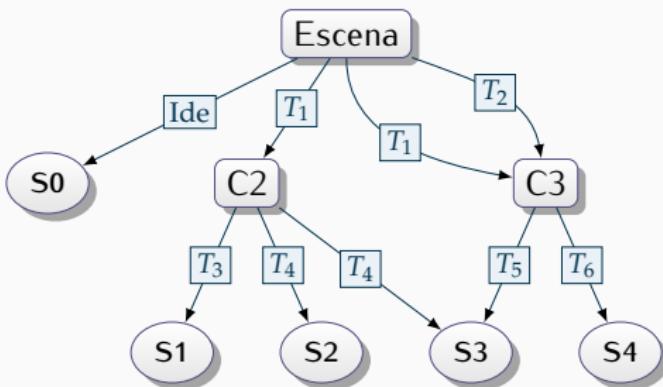
Cada objeto  $O_i$  del esquema anterior puede ser de **dos tipos** de objetos geométricos:

- ▶ **Objeto simple:** el objeto  $O_i$  es una malla u otros objetos que no están compuestos de otros objetos más simples.
- ▶ **Objeto compuesto:** el objeto  $O_i$  es una sub-escena, es decir, está compuesto de varios objetos que se instancian mediante diferentes transformaciones.
- ▶ Una escena ahora es un único objeto, que puede ser simple o compuesto (esto último es lo más frecuente).

Una escena es, por tanto, una estructura de tipo **Grafo Dirigido Acíclico (Directed Acyclic Graph)** o **DAG**.

# Grafo de escena

La estructura que representa una de estas escenas se denomina **Grafo de Escena (Scene Graph)**.



- ▶ cada objeto compuesto se identifica con un **nodo no terminal**
- ▶ cada objeto simple se identifica con un **nodo terminal**
- ▶ cada arco se etiqueta con una **transformación geométrica**.

# Instanciaciones en el grafo

En el grafo anterior:

- ▶ Algunas transformaciones pueden ser la matriz identidad (**Ide**)
- ▶ Algunos pares de hermanos aparecen instanciados con la misma transformación.
- ▶ Algunos nodos (simples o compuestos) aparecen instanciados más de una vez (en el mismo o distinto parente)
- ▶ La transformaciones por las que se instancia a un nodo en la escena (marco  $\mathcal{W}$ ), se obtienen **siguiendo todos los caminos posibles desde la raíz al nodo**. A modo de ejemplo: **S3** aparece instanciado 3 veces, con estas transformaciones:
  - ▶  $T_1 \cdot T_4$
  - ▶  $T_1 \cdot T_5$
  - ▶  $T_2 \cdot T_5$

(nótese que el efecto de la transformaciones debe leerse desde el nodo hacia la raíz).

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

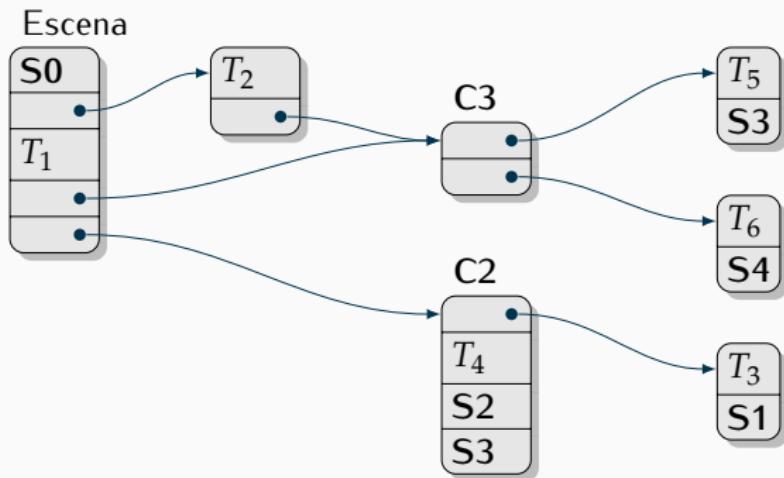
Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.2.

Grafos tipo PHIGS. Ejemplo..

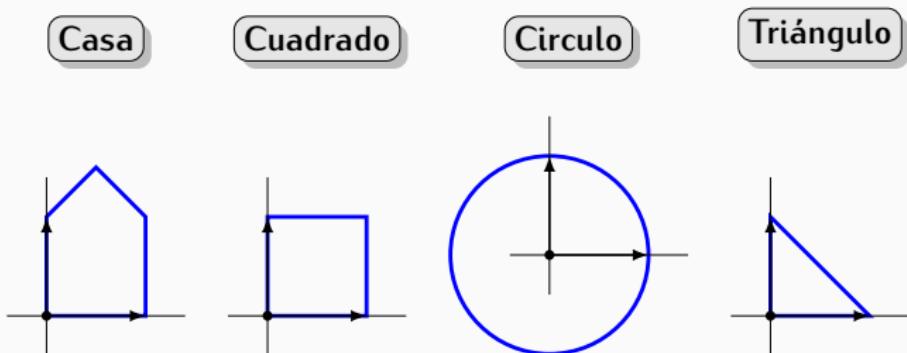
# Notación inspirada en PHIGS para grafos de escena

Por su mayor simplicidad y proximidad a la implementación OpenGL, usaremos una notación inspirada en el antiguo estándar PHIGS. El grafo anterior se puede expresar de forma equivalente así:



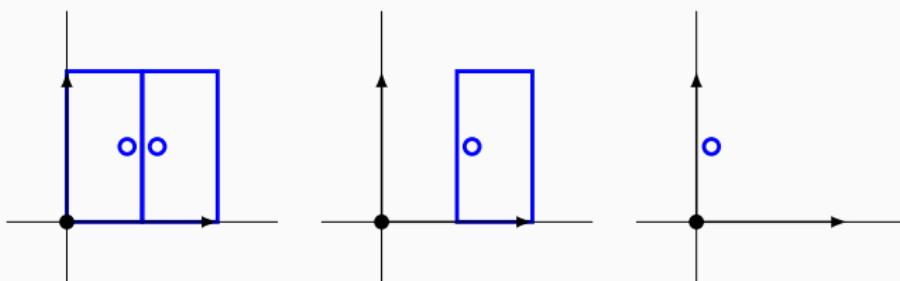
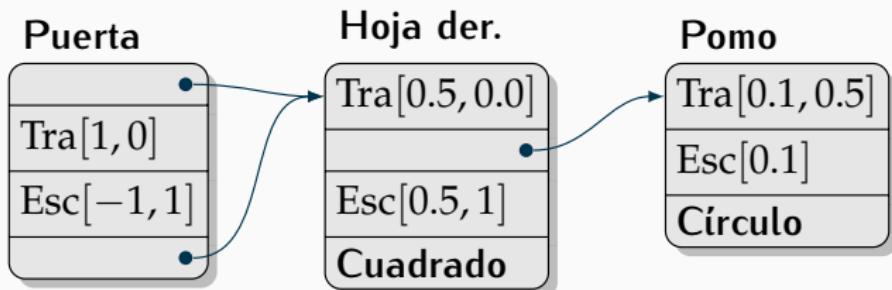
## Ejemplo en 2D: objetos simples

Una escena puede estar formada por un único objeto simple, en ese caso el grafo tiene un único nodo con una sola entrada (vemos tres ejemplos en 2D)



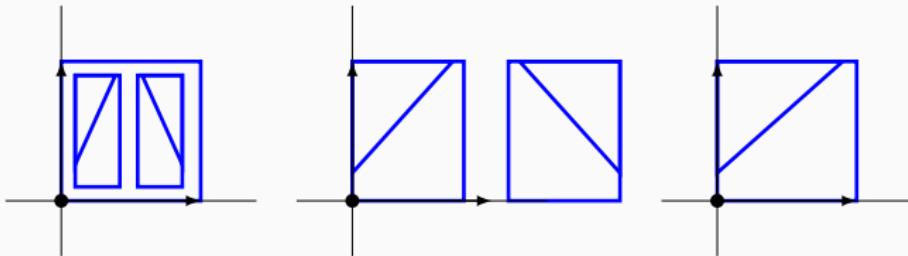
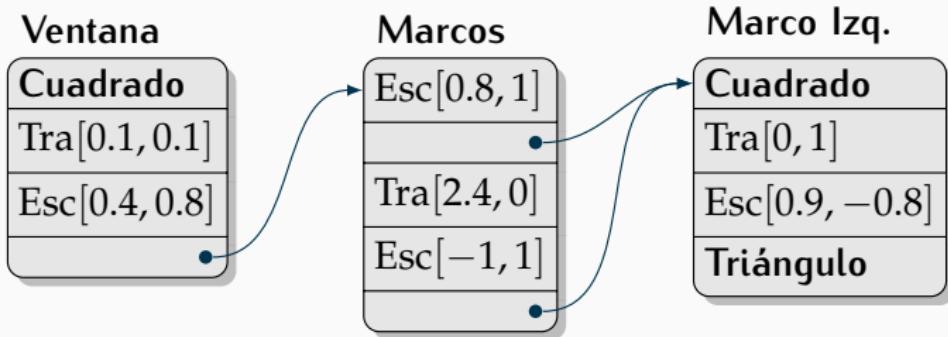
# Objeto compuesto: Puerta

Este grafo define una puerta a partir de cuadrados y círculos:



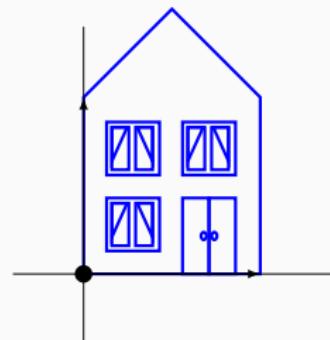
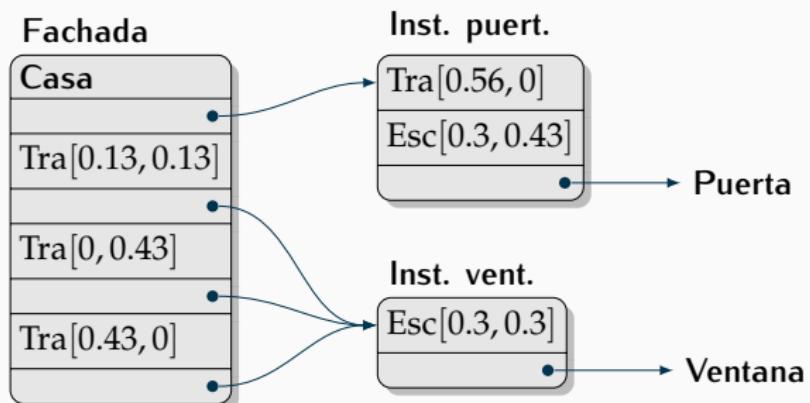
# Objeto compuesto: Ventana

Una ventana hecha de cuadrados y triángulos:



# Objeto compuesto: Fachada

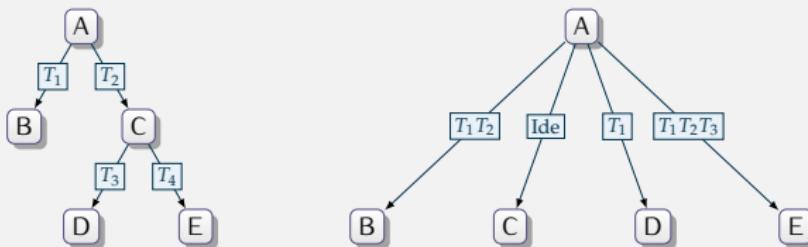
Los dos objetos compuestos creados se pueden reutilizar en un objeto de más alto nivel:



# Problema: grafos de escena: arcos etiquetados versus tipo PHIGS

## Problema 2.19.

Dados los dos siguientes grafos de escena sencillos:



Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones:  $T_1, T_2$  y  $T_3$ .

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.3.

Representación de grafos..

# Representación de grafos 3D

Cada nodo del grafo de escena es un tipo especial de **Objeto3D** con una lista o vector de entradas. Cada entrada puede ser de dos tipos:

- ▶ Un objeto 3D: se almacena un puntero a un **Objeto3D** (puede ser otro nodo, una malla o cualquier otro tipo de objeto).
- ▶ Una transformación: se usa un puntero a una matriz (**Matriz4f**).

Una escena se puede representar usando un puntero al nodo raíz. Un nodo puede verse como un objeto 3D compuesto de otros objetos y transformaciones.

# Entradas de los nodos

Cada entrada del nodo puede ser una instancia de esta clase:

```
// tipo enumerado con los tipos de entradas de los nodos del grafo:  
enum class TipoEntNGE { objeto, transformacion, .... } ;  
  
// Entrada del nodo del Grafo de Escena  
struct EntradaNGE  
{  
    TipoEntNGE tipoE ; // tipo de entrada (enumerado)  
  
    union  
    {  Objeto3D * objeto ; // ptr. a un objeto (propietario)  
       Matriz4f * matriz ; // ptr. a matriz 4x4 transf. (prop.)  
       ....  
    } ;  
    // constructores (uno por tipo)  
    EntradaNGE( Objeto3D * pObjeto ) ; // (copia solo puntero)  
    EntradaNGE( const Matriz4f & pMatriz ) ; // (crea copia)  
    ....  
};
```

# Los objetos 3D tipo nodo del grafo

Los nodos del grafo son básicamente vectores de entradas.

```
class NodoGrafoEscena : public Objeto3D
{
protected:
    std::vector<EntradaNGE> entradas ; // vector de entradas
public:
    // visualiza usando OpenGL
    virtual void visualizarGL( ContextoVis & cv ) ;

    // añadir una entrada (al final). Devuelve índice entrada.
    unsigned agregar( EntradaNGE * entrada ); // genérica
    // construir una entrada y añadirla (al final)
    unsigned agregar( Objeto3D * pObjeto ); // objeto (copia puntero)
    unsigned agregar( const Matriz4f & pMatriz ); // matriz (crea copia)
};
```

# Creación de estructuras

Para crear la estructura se pueden crear clases concretas derivadas de **NodoGrafoEscena**:

- ▶ Los constructores de dichas clases se encargan de crear las entradas.
- ▶ Cada constructor crea los sub-objetos de forma recursiva, así como las transformaciones necesarias.
- ▶ Si la estructura es de árbol, la liberación de la memoria puede hacerse recursivamente, si es un grafo acíclico, dicha liberación puede ser más complicada.

# Ejemplo de creación

Suponiendo el mismo ejemplo de la casa:

- ▶ Las clases **Cuadrado** y **Triangulo** son clases derivadas de **Objeto3D**, son la primitivas de las que partimos (suponemos que incluyen un método para visualizar un cuadrado y un triángulo, respectivamente).
- ▶ A modo de ejemplo, vamos a ver como construir las clases **Ventana**, **Marcos** y **MarcoIzq**.
- ▶ La clase **Fachada** se podría construir de forma similar.

Para cada clase se define un constructor que crea la estructura.

- ▶ El constructor añade las entradas (mediante **agregar**) en el orden adecuado.
- ▶ Llama recursivamente los constructores de los sub-objetos.

# Ejemplo de creación

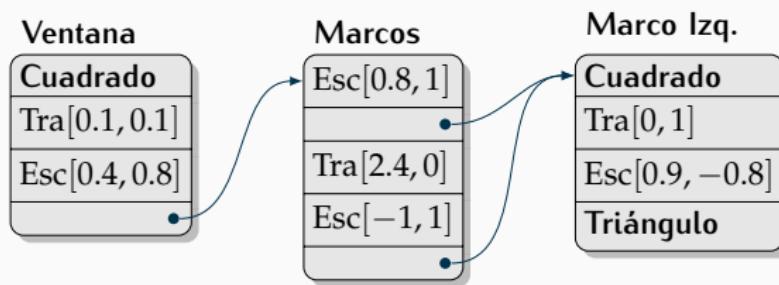
La declaración de las clases se puede hacer así:

```
// clase para el nodo del grafo etiquetado como Ventana
class Ventana : public NodoGrafoEscena
{ public:
    Ventana() ; // constructor
} ;
// clase para el nodo del grafo etiquetado como Marcos
class Marcos : public NodoGrafoEscena
{ public:
    Marcos() ; // constructor
} ;
// clase para el nodo del grafo etiquetado como Marco Izq.
class MarcoIzq : public NodoGrafoEscena
{ public:
    MarcoIzq() ; // constructor
} ;
```

# Implementación de constructores (1)

La implementación de los constructores se podría hacer así:

```
Ventana::Ventana()
{
    agregar( new Cuadrado ); // Cuadrado
    agregar( MAT_Traslacion( 0.1,1.0,0.0 ) ); // Tra[0.1,0.1]
    agregar( MAT_Escalado( 0.4,0.8,1.0 ) ); // Esc[0.4,0.8]
    agregar( new Marcos ); // Marcos
}
```



# Implementación de constructores (2)

```
MarcoIzq::MarcoIzq()
{
    agregar( new Cuadrado );                                // Cuadrado
    agregar( MAT_Traslacion( 0.0,1.0,0.0 ) ); // Tra[0,1]
    agregar( MAT_Escalado( 0.9,-0.8,1.0 ) ); // Esc[0.9, -0.8]
    agregar( new Triangulo );                                // Triangulo
}
Marcos::Marcos()
{
    MarcoIzq * marco_izq = new MarcoIzq ;
    agregar( MAT_Escalado( 0.8,0.1,1.0 ) ); // Esc[0.8,0.1]
    agregar( marco_izq );                           // Marco Izq.
    agregar( MAT_Traslacion( 2.4,0.0,0.0 ) ); // Tra[2.4,0]
    agregar( MAT_Escalado( -1.0,1.0,1.0 ) ); // Esc[-1,1]
    agregar( marco_izq );                           // Marco Izq.
}
```

## Creación directa

También es posible crear nodos directamente, sin definir una sub-clases, el código anterior puede hacerse también así:

```
Marcos::Marcos()
{
    // crear un nuevo nodo (marco izquierdo) cmo instancia de
    NodoGrafoEscena * marco_izq = new NodoGrafoEscena() ;
    marco_izq->agregar( new Cuadrado );
    marco_izq->agregar( MAT_Traslacion( 0.0,1.0,0.0 ) );
    marco_izq->agregar( MAT_Escalado( 0.9,-0.8,1.0 ) );
    marco_izq->agregar( new Triangulo );

    // construir el objeto 'Marcos'
    agregar( MAT_Escalado( 0.8,0.1,1.0 ) );      // Esc[0.8,0.1]
    agregar( marco_izq );                          // Marco Izq.
    agregar( MAT_Traslacion( 2.4,0.0,0.0 ) ); // Tra[2.4,0]
    agregar( MAT_Escalado( -1.0,1.0,1.0 ) ); // Esc[-1,1]
    agregar( marco_izq );                          // Marco Izq.
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.4.

Visualización de grafos en OpenGL..

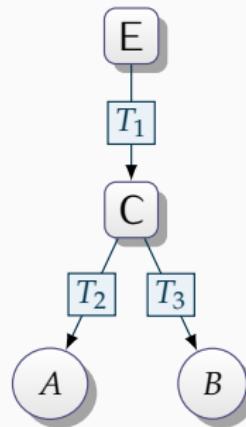
# Visualización de varios objetos

La visualización de grafos en OpenGL se basa en operaciones que permiten guardar y recuperar la matriz modelview.

Supongamos que queremos visualizar dos objetos  $A$  y  $B$ :

- ▶ Para el objeto  $A$  usamos la matriz de modelado  $N_A = T_1 \cdot T_2$
- ▶ Para el objeto  $B$  usamos la matriz de modelado  $N_B = T_1 \cdot T_3$
- ▶ Para ambos queremos usar una matriz de vista  $V$ .

Esta situación es típica si  $A$  y  $B$  están en distintas ramas de un sub-árbol en un grafo de escena.



# Visualización con OpenGL (replicando sentencias)

Para hacer la visualización con OpenGL clásico (usando directamente el cauce de funcionalidad fija) se debe usar este código:

```
glMatrixMode( GL_MODELVIEW );  
  
glLoadIdentity(); // M := Ide  
glMultMatrixf( V ); // M := M · V  
glMultMatrixf( T1 ); // M := M · T1  
glMultMatrixf( T2 ); // M := M · T2  
VisualizarObjeto( A ); // visualizar A con M == V · T1 · T2  
  
glLoadIdentity(); // M := Ide  
glMultMatrixf( V ); // M := M · V  
glMultMatrixf( T1 ); // M := M · T1  
glMultMatrixf( T3 ); // M := M · T3  
VisualizarObjeto( B ); // visualizar B con M == V · T1 · T3
```

Es decir: es necesario replicar llamadas para acumular en  $M$  las matrices  $V$  y  $T_1$ . Esto puede ser muy complejo en escenas complejas.

## Guardar y recuperar la matriz Modelview

Para evitar esas repeticiones (es lento), OpenGL clásico dispone de dos instrucciones que permiten en guardar la matriz de OpenGL modelview  $M$  y restaurarla después:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity() ;      //  $M := \text{Id}_e$ 
glMultMatrixf( V ) ;  //  $M := M \cdot V$ 
glMultMatrixf( T1 ) ; //  $M := M \cdot T_1$ 

glPushMatrix() ;        //  $C := M$  (guarda una copia de  $M$  en  $C$ )
glMultMatrix( T2 ) ;  //  $M := M \cdot T_2$ 
VisualizarObjeto( A ) ; // visualizar  $A$  con  $M == V \cdot T_1 \cdot T_2$ 
glPopMatrix() ;         //  $M := C$  (restaura copia de  $M$ )

glPushMatrix() ;        //  $C := M$  (guarda una copia de  $M$  en  $C$ )
glMultMatrix( T2 ) ;  //  $M := M \cdot T_3$ 
VisualizarObjeto( B ) ; // visualizar  $B$  con  $M == V \cdot T_1 \cdot T_3$ 
glPopMatrix() ;         //  $M := C$  (restaura copia de  $M$ )
```

## Anidamiento de *push* y *pop*

El código entre *push* y *pop* es **neutro** en cuanto a la matriz  $M$  (es decir: no la modifica)

- ▶ es cierto incluso cuando *push* y *pop* se anidan
- ▶ para lo cual OpenGL **necesita tener en su estado una pila LIFO  $P$  de matrices** de transformación modelview (inicialmente vacía), en lugar de una sola matriz  $C$ :

```
// al inicio tope == 0
glMatrixMode( GL_MODELVIEW ); // push y pop actuarán sobre M
...
glPushMatrix() ; // P[tope]=M ; tope=tope+1
...
glPopMatrix() ; // tope=tope-1 ; M=P[tope]
...
```

(lógicamente, los *push* y *pop* deben estar balanceados)

## Visualización de grafos de escena con el cauce fijo

Un programa que siempre visualiza el mismo grafo de escena (tipo PHIGS) puede implementarse traduciendo dicho grafo a código:

- ▶ Cada nodo se implementa con una secuencia de llamadas, entre operaciones *push* y *pop* (no modifica  $M$ )
- ▶ Una entrada correspondiente a un nodo simple (una malla), supone una llamada al procedimiento para visualizarla
- ▶ Las entradas correspondientes a transformaciones suponen acumular la correspondiente matriz en modelview
- ▶ Una entradas correspondiente a una referencia a otro nodo  $B$  puede traducirse en:
  - ▶ Una secuencia de llamadas correspondientes a  $B$  entre *push* y *pop*
  - ▶ Una llamada a un procedimiento específico del nodo  $B$  (esto mejor si el nodo  $B$  está referenciado desde más de un sitio, para no repetir código).

## Ejemplo: el nodo Fachada

```
void Fachada()
{
    glPushMatrix();
    Casa();
    glPushMatrix(); // inst.puerta
    glTranslatef(0.56,0.0,0.0);
    glScalef(0.3,43,1.0);
    Puerta();
    glPopMatrix();
    glTranslatef(0.13,0.13,0.0);
    InstVentana() ;
    glTranslatef(0.0,0.43,0.0);
    InstVentana() ;
    glTranslatef(0.43,0.0,0.0);
    InstVentana() ;
    glPopMatrix();
}
```

```
void InstVentana()
{
    glPushMatrix();
    glScalef(0.3,0.3,1.0);
    Ventana();
    glPopMatrix();
}
```

# Problema: uso de `glPushMatrix`/`glPopMatrix` (1/2)

## Problema 2.20.

Escribe una función llamada `FiguraSimple` que dibuje con OpenGL en modo diferido la figura que aparece aquí (un cuadrado de lado unidad, relleno de color, con la esquina inferior izquierda en el origen, con un triángulo inscrito, relleno del color de fondo).

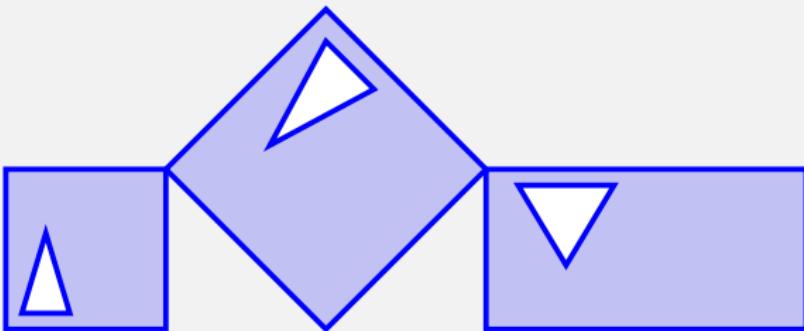


(usa el repositorio `opengl-minimo`)

## Problema: uso de `glPushMatrix`/`glPopMatrix` (2/2)

### Problema 2.21.

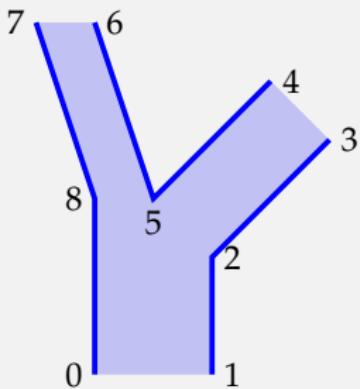
Usando exclusivamente llamadas a la función del problema 21, construye otra función llamada **FiguraCompleja** que dibuja la figura de aquí. Para lograrlo puedes usar manipulación de la pila de la matriz modelview (**glPushMatrix** y **glPopMatrix**), junto con **glTranslatef** y **glScalef**.



# Problema: tronco de figura recursiv

## Problema 2.22.

Escribe el código OpenGL de una función (llamada **Tronco**) que dibuje la figura que aparece a aquí. El código dibujará el polígono relleno de color azul claro, y las aristas que aparecen de color azul oscuro (ten en cuenta que no todas las aristas del polígono relleno aparecen).

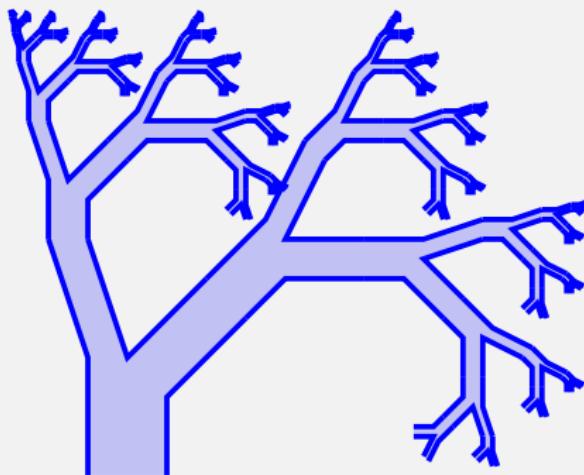


Índice	Coordenadas
0	(+0.0, +0.0)
1	(+1.0, +0.0)
2	(+1.0, +1.0)
3	(+2.0, +2.0)
4	(+1.5, +2.5)
5	(+0.5, +1.5)
6	(+0.0, +3.0)
7	(-0.5, +3.0)
8	(+0.0, +1.5)

# Problema: figura recursiva.

## Problema 2.23.

Escribe una función **Arbol** la cual, mediante múltiples llamadas a **Tronco** del problema 2.22, dibuje el árbol que aparece en la figura de abajo. Diseña el código usando recursividad, de forma que el número de niveles sea un parámetro modificable en dicho código (en la figura es 6)



# Uso del interfaz de la clase Cauce

La clase base **Cauce** incluye dos métodos para manipular la pila de matrices modelview que debe guardar cualquier tipo de cauce, los métodos son:

- ▶ **pushMM**: inserta una copia de la matriz *modelview M* en el tope de la pila de matrices de modelado
- ▶ **popMM**: (asumiendo que la pila de matrices de modelado no está vacía), copia el elemento en el tope sobre la matriz *modelview M* actual, y elimina el tope de la pila.

Se usan junto a **compMM** (ya visto), por ejemplo:

```
void InstVentana()
{
    cauce->pushMM();
    cauce->compMM( MAT_Escalado( {0.3,0.3,1.0} ) );
    Ventana();
    cauce->popMM();
}
```

# Implementación de la pila en el cauce fijo

La implementación de **pushMM** y **popMM** en el cauce fijo usa directamente **glPushMatrix** y **glPopMatrix**

```
void CauceFijo::pushMM()
{
    glMatrixMode( GL_MODELVIEW );
    glPushMatrix();
}
```

```
void CauceFijo::popMM()
{
    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();
    CError(); // detecta error OpenGL por intento de pop sobre pila vacía
}
```

# Implementación de la pila en el cauce programable

La clase **CauceProgramable** incluye, como variables de instancia, una pila de matrices de modelado y otra de matrices de modelado de normales (cada una es un **vector<Matriz4f>**):

```
void CauceProgramable::pushMM()
{
    // hacer push en cada pila de las correspondiente matriz
    pila_mat_modelado.push_back( mat_modelado );
    pila_mat_modelado_nor.push_back( mat_modelado_nor );
}

void CauceProgramable::popMM()
{
    // copiar tope de cada pila en la correspondiente matriz
    const unsigned n = pila_mat_modelado.size() ; assert(0<n);
    mat_modelado      = pila_mat_modelado[n-1] ;
    mat_modelado_nor = pila_mat_modelado_nor[n-1] ;
    // eliminar tope de cada pila
    pila_mat_modelado.pop_back();
    pila_mat_modelado_nor.pop_back();
    // fijar uniforms matrizMV y matrizMV_nor
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_matrizMV,      1,GL_FALSE,mat_vista*mat_modelado);
    glUniformMatrix4fv( loc_matrizMV_nor, 1,GL_FALSE,mat_vista*mat_modelado_nor);
}
```

# Inicialización de las pilas en el cauce programable

```
void CauceProgramable::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    // inicializar matrices (variables de instancia)
    ...

    // inicializar pilas:
    pila_mat_modelado.clear();
    pila_mat_modelado_nor.clear();

    // fijar uniforms matrizMV y matrizMV_nor
    ...
}
```

# Visualización de grafos

El método de visualización anterior supone escribir código que visualiza un único modelo jerárquico (el grafo de escena está implícito en el código)

- ▶ Ventajas: es sencillo para escenas muy sencillas o partes simples de una escena, no requiere ninguna estructura de datos en memoria.
- ▶ Inconvenientes: distintas escenas requieren distinto código, no es posible cargar un grafo de escena almacenado en un archivo en disco.

A continuación veremos como visualizar con OpenGL los grafos de escena que se han introducido en esta sección. Tiene estas ventajas:

- ▶ Un único código sirve para visualizar cualquier grafo de escena.
- ▶ Permite visualizar grafos almacenados en archivos, creados con herramientas de diseño asistido.

# El método *visualizar* en grafos.

El método **visualizarGL** dibuja recursivamente estructuras completas. Para ello usa el cauce que esté guardado en **cv**:

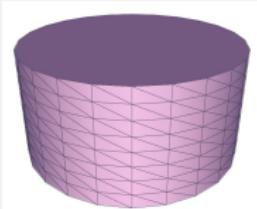
```
void NodoGrafoEscena::visualizarGL( ContextoVis & cv )
{
    // guarda modelview actual
    cv.cauce->pushMM();

    // recorrer todas las entradas del array que hay en el nodo:
    for( unsigned i = 0 ; i < entradas.size() ; i++ )
        switch( entradas[i].tipoE )
        {
            case TipoEntNGE::objeto :                      // entrada objeto:
                entradas[i].objeto->visualizarGL( cv ); // visualizar objeto
                break ;
            case TipoEntNGE::transformacion :               // entrada transf.:
                cv.cauce->compMM( *(entradas[i].matriz)); // componer matriz
                break ;
            case ....
            ....
        }
    // restaura modelview guardada
    cv.cauce->popMM() ;
}
```

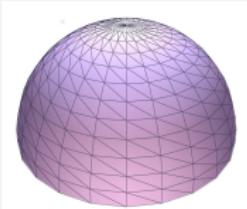
# Problema: grafo PHIGS en 3D, e implementación (1/2)

## Problema 2.24.

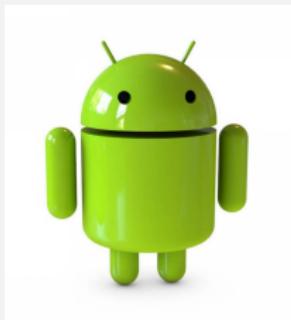
Supón que dispones de dos funciones para dibujar dos mallas u objetos simples: **Semiesfera** y **Cilindro**. La semiesfera (en coordenadas maestras) tiene radio unidad, centro en el origen y el eje vertical en el eje Y. Igualmente el cilindro tiene radio y altura unidad, el centro de la base está en el origen, y su eje es el eje Y.



**Cilindro**



**Semiesfera**



**Android**

(continua en la siguiente transparencia)

# Problema: grafo PHIGS en 3D, e implementación (2/2)

## Problema 2.24. (continuación)

Con estas dos primitivas queremos escribir el código que visualiza la figura Android, usando la plantilla de código de prácticas. Para ello:

- ▶ Diseña el grafo de escena (tipo PHIGS) correspondiente, ten en cuenta que hay objetos compuestos que se pueden instanciar más de una vez (cada brazo o pierna se puede construir con un objeto compuesto de dos semiesferas en los extremos de un cilindro).
- ▶ Escribe el código OpenGL para visualizarlo, usando una clase llamada **Android**, derivada de **NodoGrafoEscena**.

Informática Gráfica, curso 2021-22.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.5.

Grafos parametrizables..

# Modelos parametrizables

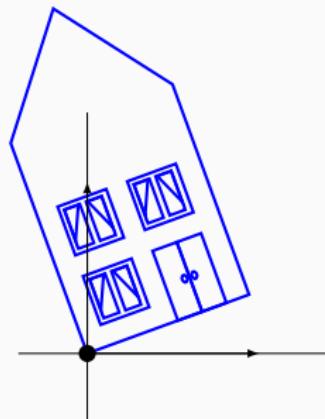
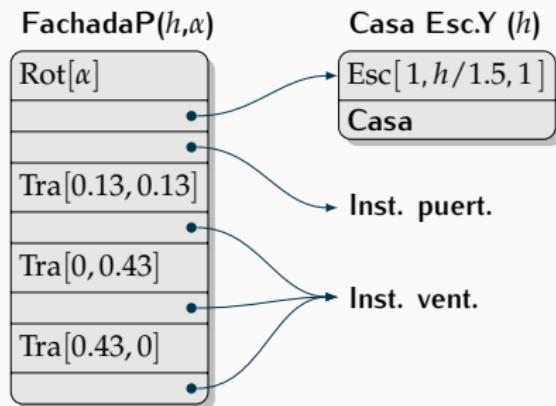
Un grafo de escena puede estar parametrizado respecto de ciertos valores reales:

- ▶ Dichos valores pueden controlar, por ejemplo, un transformación
  - ▶ Ángulo de rotación.
  - ▶ Factor de escala en una dimensión.
  - ▶ Distancia de traslación en una dirección dada.
- ▶ En otros casos pueden definir las dimensiones de un objeto.
- ▶ U otros valores, como por ejemplo la posición de puntos de control en objetos deformables.

Un mismo grafo de escena parametrizado se traduce en objetos con distinta geometría para distintos valores concretos de los parámetros.

# Grafos parametrizados

Un grafo de escena puede venir definido por uno más parámetros (**grados de libertad**), usualmente valores reales. P.ej.: el objeto compuesto **FachadaP** admite dos parámetros: altura de **Casa** ( $h$ ) y ángulo de rotación de ( $\alpha$ ):



# Visualización de FachadaP

Usando exclusivamente código, se le añaden parámetros a las funciones para representar los grados de libertad:

```
void FachadaP( float h, float alpha )
{
    glPushMatrix();
    glRotatef(alpha,0,0,1);
    CasaEscY(h);
    InstPuerta();
    glTranslatef(0.13,0.13,0.0);
    InstVentana();
    glTranslatef(0.0,0.43,0.0);
    InstVentana();
    glTranslatef(0.43,0.0,0.0);
    InstVentana();
    glPopMatrix();
}
```

```
void CasaEscY( float h )
{
    glPushMatrix();
    glScalef(1,h/1.5,1);
    Casa();
    glPopMatrix();
}
```

# Representacion de grafos parametrizables

Una transformación parametrizable en un grafo puede representarse con una clase derivada de **NodoGrafoEscena**, en la cual hay:

- ▶ Un método para cambiar el valor de cada parámetro.
- ▶ Un puntero a la matriz o matrices afectadas por el parámetro.

A modo de ejemplo, la clase **FachadaP** podría quedar así:

```
class FachadaP : public NodoGrafoEscena
{
protected: // punteros a matrices
    Matriz4f * pm_rot_alpha = nullptr,
              * pm_escy_h   = nullptr ;
public:
    FachadaP( const float h_inicial, const float alpha_inicial );
    void fijarH( const float h_nuevo ) ;
    void fijarAlpha( const float alpha_nuevo );
};
```

# Métodos para cambiar un parámetro

Estos dos métodos recalcularan las matrices correspondientes del grafo, en función del nuevo valor de un parámetro:

```
void FachadaP::fijarAlpha( const float alpha_nuevo )
{
    *pm_rot_alpha = MAT_Rotacion( alpha_nuevo, 0,0,1 );
}
void FachadaP::fijarH( const float h_nuevo )
{
    *pm_escy_h = MAT_Escalado( 1.0, h_nuevo/1.5, 1.0 );
}
```

# Constructor de un nodo parametrizado

El constructor puede quedar así:

```
FachadaP::FachadaP( const float h_inicial, const float alpha_inicial )
{
    // crear sub-nodo tipo Casa escalada en Y (casa_ey)
    NodoGrafoEscena * casa_ey = new NodoGrafoEscena() ;
    unsigned ind1 = casa_ey->agregar(MAT_Escalado(1.0,h_inicial/1.5,1.0));
    casa_ey->agregar( new Casa() );

    // crear inst. de ventana y añadir entradas de FachadaP
    Objeto3D * inst_ventana = new InstVentana();
    unsigned ind2 = agregar( MAT_Rotacion( alpha_inicial, {0,0,1} ) );
    agregar( casa_ey );
    agregar( new InstPuerta() );
    agregar( MAT_Traslacion({0.13,0.13,0.13})); agregar( inst_ventana );
    agregar( MAT_Traslacion({0.0,0.43,0.0}));   agregar( inst_ventana );
    agregar( MAT_Traslacion({0.43,0.0,0.0}));   agregar( inst_ventana );

    // guardar los punteros a las matrices que dependen de los parámetros:
    pm_escy_h      = casa_ey->leerPtrMatriz( ind1 ) ;
    pm_rot_alpha = leerPtrMatriz( ind2 ) ;
}
```

# Lectura de punteros a matrices

El método **leerPtrMatriz** de la clase **NodoGrafoEscena** devuelve el puntero a una matriz en una de sus entradas, dado el índice de la entrada

```
Matriz4f * NodoGrafoEscena::leerPtrMatriz( const unsigned indice )
{
    assert( indice < entradas.size() );
    assert( entradas[indice].tipo == TipoEntNGE::transformacion );
    assert( entradas[indice].matriz != nullptr );

    return entradas[indice].matriz ;
}
```

- ▶ Se comprueba que el índice corresponde a una entrada que contiene un puntero no nulo a una matriz.
- ▶ El índice se obtiene como valor devuelto por **agregar** (normalmente se ignora).

# Problema: grafo 3D parametrizado e implementación

## Problema 2.25.

Escribe una segunda versión del grafo de escena del problema 2.24, de forma que las transformaciones estén parametrizadas por dos valores reales ( $\alpha$  y  $\beta$ ) que expresan el ángulo de rotación del brazo izquierdo y derecho (respectivamente), en torno al eje que pasa por los centros de las dos semiesferas superiores de los brazos.

Asimismo, habrá otro parámetro ( $\phi$ ) que es el ángulo de rotación de la cabeza (completa: con los ojos y antenas) entorno al eje vertical que pasa por su centro (cuando estos ángulo valen 0, el androide está en reposo y tiene exactamente la forma de la figura del problema anterior).

Escribe el código de una nueva clase (**AndroidParam**, derivada de **NodoGrafoEscena**) para visualizar el androide parametrizado de esta forma.

Fin de la presentación.



UNIVERSIDAD  
DE GRANADA

# Informática Gráfica: Teoría. Tema 3. Visualización.

---

Carlos Ureña

2021-22

**Grado en Informática y Matemáticas**  
Dpt. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

## Teoría. Tema 3. Visualización.

### Índice.

1. Cauce gráfico y definición de la cámara.
2. Modelos de Iluminación, texturas y sombreado.
3. Iluminación y texturas con el cauce fijo
4. Iluminación y texturas en el cauce programable
5. Representación de materiales, texturas y fuentes.

## Sección 1. Cauce gráfico y definición de la cámara..

- 1.1. El cauce gráfico del algoritmo Z-buffer.
- 1.2. Transformación de vista
- 1.3. Transformación de proyección
- 1.4. Recortado y división por  $W$
- 1.5. Transformación de viewport
- 1.6. Representación de cámaras

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.1.

El cauce gráfico del algoritmo Z-buffer..

# Introducción.

El término **cauce gráfico** (*graphics pipeline*) se suele usar para referirnos al conjunto de pasos de cálculo que se realizan para visualizar polígonos en el contexto del **algoritmo de Z-buffer**

- ▶ El algoritmo de Z-buffer se usa para presentar polígonos incluyendo **eliminación de partes ocultas** (EPO) en 3D (es decir: lograr presentar únicamente las partes visibles de los polígonos que se dibujan).
- ▶ OpenGL, DirectX y otras librerías 3D usan Z-buffer.
- ▶ Estos pasos **se implementan en hardware** en las tarjetas gráficas modernas (GPUs: *Graphics Processing Units*).
- ▶ Estos pasos **no se aplican en otros algoritmos** de visualización y EPO en 3D, como por ejemplo en Ray-tracing.

# Pasos del cauce gráfico.

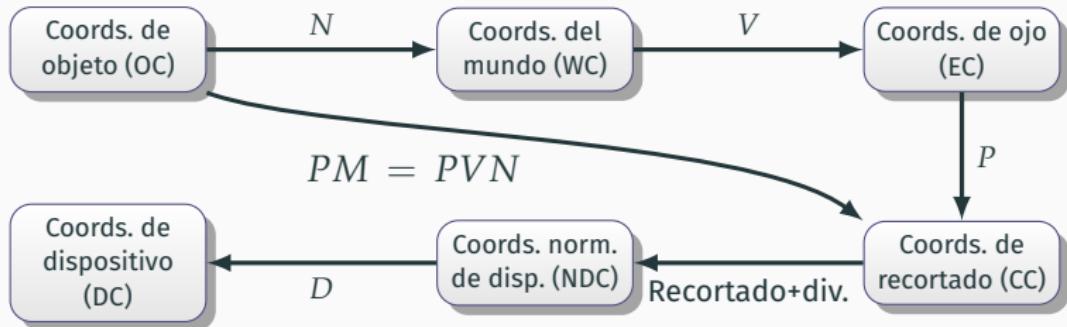
Los pasos del cauce gráfico suelen implementarse en secuencia, cada paso obtiene datos del anterior, los transforma de alguna manera y los entrega al siguiente paso. Los pasos (muy resumidos) son:

1. **Transformación** de coordenadas de vértices: cálculo de donde se proyecta en pantalla cada vértice.
2. **Recortado**: eliminación de partes de polígonos fuera de la zona visible.
3. **Rasterización y EPO**: cálculo de los píxeles donde se proyecta un polígono.
4. **Iluminación y texturización**: cálculo del color de cada pixel donde se proyecta un polígono.

la transformación y el recortado se pueden mezclar de diversas formas, ambos pasos son necesariamente previos a los otros dos, que también se pueden combinar de varias formas entre ellos

# Esquema de la transformación y recortado

En estas etapas del cauce gráfico, esencialmente los datos que se transforman son coordenadas de vértices y conectividad entre ellos. El esquema es el siguiente:



este esquema corresponde al recortado en CC (hay otras posibilidades, esta es la mejor).

# Sistemas de coordenadas

El cauce gráfico de OpenGL contempla los siguientes:

- ▶ **Coordenadas de objeto o maestras:** son distancias relativas a un sistema de referencia específico o distinto de cada objeto, que se crea en este espacio.
- ▶ **Coordenadas del mundo:** son distancias relativas a un sistema de referencia común para todos los objetos de una escena
- ▶ **Coordenadas de cámara:** son distancias relativas a un sistema de referencia posicionado y alineado con la cámara virtual en uso.
- ▶ **Coordenadas de recortado:** son distancias normalizadas, con  $w \neq 1$ , relativas a un sistema asociado al rectángulo que forma la imagen en pantalla.
- ▶ **Coordenadas normalizadas de dispositivo (NDC):** similares a CC, pero con  $w = 1$ , y dentro de la zona visible.
- ▶ **Coordenadas de dispositivo:** similares a NDC, pero en unidades de pixels.

# Matrices de transformación

Las matrices de transformación ( $4 \times 4$ ) involucradas permiten convertir coordenadas en un sistema de coordenadas a coordenadas en otro:

- ▶ **La matriz de modelado y vista (modelview)  $M$** , compuesta de:
  - ▶ **Matriz de modelado  $N$** : convierte de OC a WC
  - ▶ **Matriz de vista  $V$** : convierte de WC a EC
- ▶ **La matriz de proyección  $P$** : convierte de EC a CC. (recibe coordenadas con  $w = 1$ , pero produce coordenadas en general con  $w \neq 1$ )
- ▶ **La matriz del viewport  $D$** : convierte de NDC a DC (depende de la resolución de la imagen en pantalla y de la zona de esta donde se visualiza).

Las coordenadas de dispositivo (con  $w = 1$  y en unidades de pixels) se usan como entrada para las siguientes etapas del cauce gráfico (rasterización, EPO, iluminación y texturas)

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.2.

Transformación de vista.

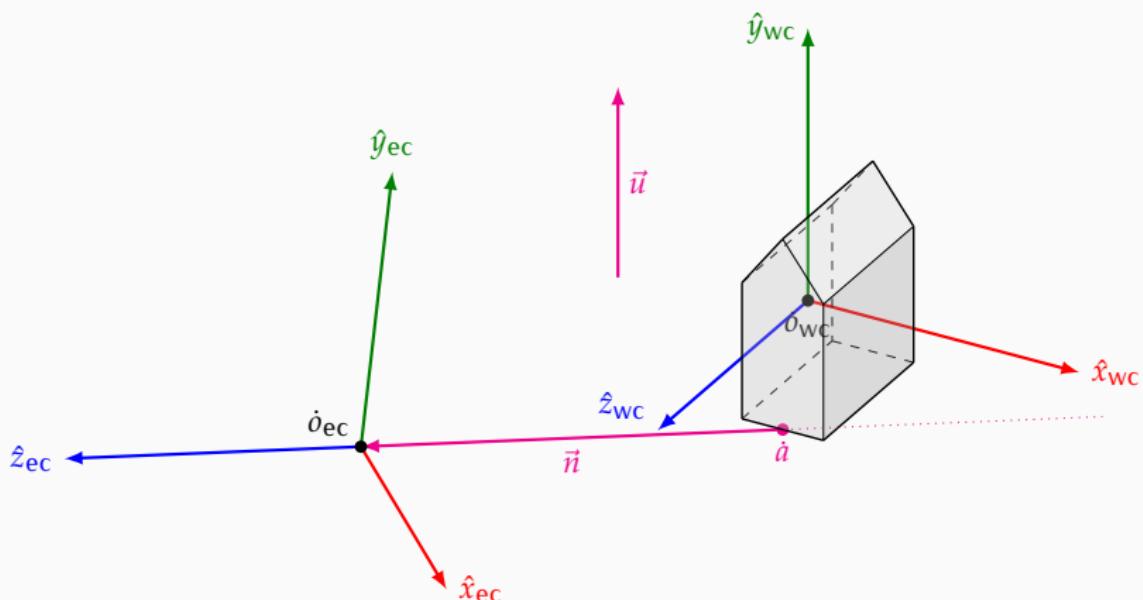
## La transformación de vista.

La transformación de vista es el cálculo que permite convertir coordenadas de mundo (*world coordinates*, WCC) en coordenadas de ojo (o de cámara) (*eye or camera coordinates*, ECC).

- ▶ Se usa un marco de referencia cartesiano  $\mathcal{V} = [\hat{x}_{\text{ec}}, \hat{y}_{\text{ec}}, \hat{z}_{\text{ec}}, \dot{o}_{\text{ec}}]$ , llamado **marco de cámara (o de vista)**, que está posicionado y alineado con la cámara virtual. Las **coordenadas de cámara (o de vista)** de un punto son las coordenadas de ese punto en el marco  $\mathcal{V}$ .
- ▶ Para hacer la conversión de coordenadas se debe usar la **matriz de vista**, la llamamos  $V$ .
- ▶ Puesto que el marco de coordenadas de mundo  $\mathcal{W}$  es cartesiano y  $\mathcal{V}$  también, la matriz  $V$  puede construirse fácilmente como la composición de una matriz de traslación por  $\dot{o}_{\text{wc}} - \dot{o}_{\text{ec}}$  seguida de una matriz (ortonormal) de rotación, que tiene las coordenadas de mundo de  $\hat{x}_{\text{ec}}$ ,  $\hat{y}_{\text{ec}}$  y  $\hat{z}_{\text{ec}}$  en sus filas.

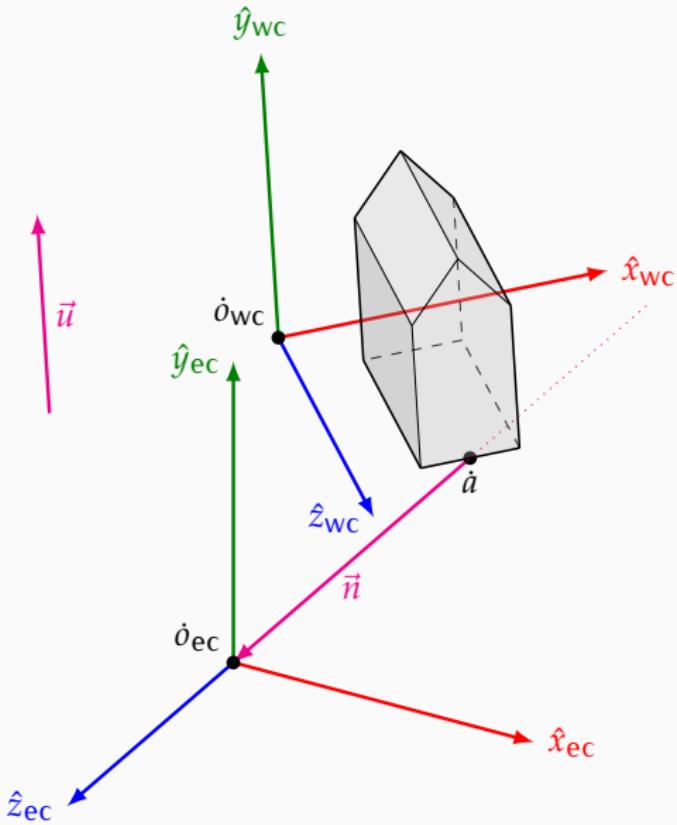
# El marco de coordenadas de vista o de cámara.

El marco (cartesiano) de coordenadas de vista se construye usando el punto  $\dot{a}$ , el punto  $\dot{o}_{ec}$ , el vector  $\vec{u}$  y el vector  $\vec{n}$ . El observador estaría situado en  $\dot{o}_{ec}$  mirando en la dirección de  $-\hat{z}_{ec}$ .



# Escena posterior a transformación de vista.

Aquí vemos la escena anterior una vez transformada por la matriz  $V$



## Cálculo del marco de vista.

El marco de referencia de vista  $\mathcal{V}$ , se define a partir de los siguientes parámetros

$\dot{o}_{ec}$  = es el punto del espacio foco de la proyección, donde estaría situado el observador ficticio que contempla la escena (*projection reference point, PRP*)

$\vec{n}$  = vector libre perpendicular al *plano de visión* (plano ficticio donde se proyecta la imagen perpendicular al *eje óptico* de la cámara virtual). (*view plane normal, VPN*).

$\dot{a}$  = punto en el eje óptico, también llamado *punto de atención* o *look-at point*.

$\vec{u}$  = es un vector libre que indica una dirección que el observador ve proyectada en vertical en la imagen (apuntando hacia arriba) (*view-up vector, VUP*)

De los tres parámetros  $\dot{o}_{ec}$ ,  $\vec{n}$  y  $\dot{a}$  solo hay que especificar dos, ya que no son independientes (se cumple  $\dot{o}_{ec} = \dot{a} + \vec{n}$ ).

## Cálculo del marco de vista.

A partir de esos parámetros se obtiene se calculan los versores del marco de vista:

$$\hat{z}_{\text{ec}} = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{eje Z paralelo a VPN, normalizado})$$

$$\hat{x}_{\text{ec}} = \frac{\vec{u} \times \vec{n}}{\|\vec{u} \times \vec{n}\|} \quad (\text{eje X perpendicular a VPN y VUP, normalizado})$$

$$\hat{y}_{\text{ec}} = \hat{z}_{\text{ec}} \times \hat{x}_{\text{ec}} \quad (\text{eje Y perpendicular a los otros dos})$$

Para que este cálculo pueda hacerse, los vectores  $\vec{u}$  y  $\vec{n}$  no pueden ser nulos ni paralelos, de forma que siempre  $\|\vec{u} \times \vec{n}\| > 0$ .

## Coordenadas del mundo del marco de vista $\mathcal{C}$

El marco de referencia de vista se suele representar en memoria usando las coordenadas del mundo de los vectores y el punto (coordenadas relativas a  $\mathcal{W}$ ), es decir:

$$\begin{aligned}\hat{x}_{\text{ec}} &= \mathcal{W}(a_x, a_y, a_z, 0)^t = \mathcal{W}\mathbf{x}_{\text{ec}} \\ \hat{y}_{\text{ec}} &= \mathcal{W}(b_x, b_y, b_z, 0)^t = \mathcal{W}\mathbf{y}_{\text{ec}} \\ \hat{z}_{\text{ec}} &= \mathcal{W}(c_x, c_y, c_z, 0)^t = \mathcal{W}\mathbf{z}_{\text{ec}} \\ \dot{o}_{\text{ec}} &= \mathcal{W}(o_x, o_y, o_z, 1)^t = \mathcal{W}\mathbf{o}_{\text{ec}}\end{aligned}$$

Estas coordenadas se calculan a partir de las coordenadas de mundo (en el marco  $\mathcal{W}$ ) de los vectores  $\vec{u}, \vec{n}$  y el punto  $\dot{o}$  como hemos visto. Esas coordenadas son **u**, **n** y **o**, respectivamente.

La matriz  $V$  se puede construir directamente a partir de ellas.

# Cálculo de la matriz de vista

La matriz de vista  $V$  es la matriz que convierte desde coordenadas en  $\mathcal{W}$  hacia coordenadas en  $\mathcal{V}$ . Se obtiene como la composición de una matriz de traslación (por  $-\mathbf{o}_{\text{ec}}$ ) seguida de una matriz de rotación (de eje arbitrario)  $R$ :

$$V \equiv R \cdot \text{Tra}[-\mathbf{o}_{\text{ec}}]$$

Donde  $R$  es la matriz (ortonormal) que tiene a  $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$  y  $\mathbf{z}_{\text{ec}}$  en sus filas:

$$V = \begin{pmatrix} a_x & a_y & a_z & d_x \\ b_x & b_y & b_z & d_y \\ c_x & c_y & c_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \underbrace{\begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_R \underbrace{\begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Tra}[-\mathbf{o}_{\text{ec}}]}$$

(donde  $d_x \equiv -\mathbf{x}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$ ,  $d_y \equiv -\mathbf{y}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$ ,  $d_z \equiv -\mathbf{z}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$ ).

# Fijar la matriz de vista en OpenGL con el cauce fijo

Por tanto, para fijar la matriz de vista, el código OpenGL puede ser del estilo de este:

```
const GLfloat V[4][4] = // matriz V asociada al marco  $\mathcal{V}$  (por filas)
{{ ax, ay, az, dx }, // coords. de mundo de  $\mathbf{x}_{\text{ec}}$ , y dx = - $\mathbf{o}_{\text{ec}} \cdot \mathbf{x}_{\text{ec}}$ 
{ bx, by, bz, dy }, // coords. de mundo de  $\mathbf{y}_{\text{ec}}$ , y dy = - $\mathbf{o}_{\text{ec}} \cdot \mathbf{y}_{\text{ec}}$ 
{ cx, cy, cz, dz }, // coords. de mundo de  $\mathbf{z}_{\text{ec}}$ , y dz = - $\mathbf{o}_{\text{ec}} \cdot \mathbf{z}_{\text{ec}}$ 
{ 0, 0, 0, 1 } // origen de  $\mathcal{V}$ 
};
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glMultTransposeMatrixf( V );
```

- ▶ Estas instrucciones ponen la matriz de modelado igual a la identidad.
- ▶ Puesto que la matriz está en memoria por filas, usamos **glMultTransposeMatrixf**
- ▶ Si la matriz estuviese en memoria por columnas, usaríamos **glMultMatrixf**.

# Fijando la matriz de vista usando GLU

La función **gluLookAt** (de la librería GLU) permite componer una matriz de vista de forma más cómoda, ya que acepta directamente como parámetros las coordenadas de mundo **o<sub>ec</sub>**, **a** y **u** de  $\vec{o}_{ec}$ ,  $\vec{a}$  y  $\vec{u}$ . Está declarada como sigue:

```
void gluLookAt
( GLdouble o_x, GLdouble o_y, GLdouble o_z,
  GLdouble a_x, GLdouble a_y, GLdouble a_z,
  GLdouble u_x, GLdouble u_y, GLdouble u_z
) ;
```

donde:

$$\begin{aligned}\mathbf{o}_{ec} &= (o_x, o_y, o_z) \\ \mathbf{a} &= (a_x, a_y, a_z) \\ \mathbf{u} &= (u_x, u_y, u_z)\end{aligned}$$

# Construcción explícita de la matriz de vista

Se puede construir explicitamente en la aplicación la matriz de vista, usando la librería de generación de matrices (es especialmente útil para el cauce programable):

```
// construye la misma matriz que glutLookAt:  
Matriz4f MAT_LookAt( const float origen[3],  
                      const float centro[3],  
                      const float vup[3] );
```

O bien, si se conocen las tuplas  $x_{ec}$ ,  $y_{ec}$ ,  $z_{ec}$  y  $\mathbf{o}_{ec}$  que definen el marco de cámara, se puede componer la matriz explicitamente usando **MAT\_Filas** y **MAT\_Traslacion**:

```
// construye V usando la matriz de traslación seguida con la de alineamiento  
Matriz4f V = MAT_Filas( {x_{ec},y_{ec},z_{ec}} )*  
                  MAT_Traslacion( -o_{ec} ) ;
```

# Matriz de vista en el cauce programable

En un cauce programable, la aplicación es necesario:

- ▶ Construir la matriz de vista  $V$  en la aplicación, usando las coordenadas de los vectores y el punto que definen el marco de coordenadas de vista.
- ▶ Construir la matriz *modelview*, como composición de la matriz de modelado  $N$  y matriz de vista  $V$ . Esto puede hacerse bien el shader, bien en la aplicación.
- ▶ Declarar un parámetro uniform en el vertex shader que contiene la matriz *modelview* (o bien la matriz de vista únicamente, de forma separada a la matriz de modelado).
- ▶ Escribir el código del vertex shader de forma que la posición de los vértices sea transformada usando la matriz *modelview*.
- ▶ Fijar el valor del uniform usando **`glUniformMatrix4fv`** y la localización.

# Transformación de vista en el *vertex shader*

En este ejemplo, el uniform **matrizMV** tiene la matriz *modelview*:

```
// parámetros de entrada uniform (iguales en todos los vértices de cada primitiva)
uniform mat4 matrizMV ;           // matriz 4x4 de transf. de coord. de vértices
uniform mat4 matrizMV_nor;        // matriz 4x4 de transf. de normales
uniform mat4 matrizP ;           // matriz 4x4 de proyección (produce coord.pantalla)

// variables de salida varying (atributos de vértice: serán interpolados a pixels)
varying vec4 var_posic_ec;       // posición (en coords de cámara)
varying vec3 var_normal_ec;      // normal (en coords. de camara)
varying vec4 var_color;          // color
varying vec2 var_coord_text;     // coordenadas de textura

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord0

void main() // escribe variables 'varying', más 'gl_Position'
{
    var_posic_ec  = matrizMV * gl_Vertex;           // transf. coord. recibida
    var_normal_ec = matrizMV_nor * gl_{Normal}; // transf. normal recibida
    var_color     = gl_Color ;                      // usar color enviado
    var_coord_text= gl_MultiTexCoord0.st ;          // usa cc.t. enviadas
    gl_Position    = matrizP * var_posic_ec;        // proyecta a pantalla
}
```

## Fijar $V$ usando la clase **Cauce**

Para facilitar la definición de  $V$  en el cauce fijo y el programable, se puede usar la función **fijarMatrizVista(v)** de la clase **Cauce**, usando una **Matriz4f** de vista V:

- ▶ Copia la matriz V sobre la matriz de vista o modelview actual, que pierde el valor que tuviese antes.
- ▶ Fija la matriz de modelado  $N$  como igual a la matriz identidad.
- ▶ Fija la matriz de modelado de normales como igual a la matriz identidad.
- ▶ Reinicializa la pila de matrices de modelado.
- ▶ Reinicializa la pila de matrices de modelado de normales

En la siguiente transparencia vemos el código de este método en el cauce programable y en el fijo.

# Implementaciones de fijarMatrizVista

```
void CauceFijo::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glMultMatrixf( nue_mat_vista );
}

void CauceProgramable::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    // registrar matriz de vista y matriz de modelado en la instancia
    mat_vista      = nue_mat_vista ;
    mat_modelado   = MAT_Ident();
    mat_modelado_nor = MAT_Ident();

    // cambiar matriz de vista y matriz de modelado en los shaders
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_vista, 1, GL_FALSE, mat_vista );
    glUniformMatrix4fv( loc_mat_modelado, 1, GL_FALSE, mat_modelado );
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor );

    // vaciar pila de matrices de modelado
    pila_mat_modelado.clear();
    pila_mat_modelado_nor.clear();
}
```

# Problemas: parámetros para una vista concreta (1/3)

## Problema 3.1.

Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja, verde y azul), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

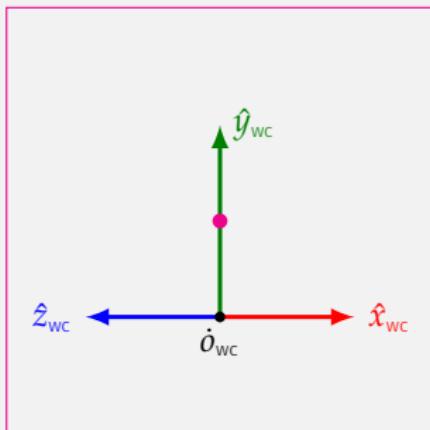
1. El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2. El punto de coordenadas  $(0,0.5,0)$  (aparece como un disco de color morado en la figura) debe aparecer en el centro del viewport
3. El observador (foco de la proyección) estará a 3 unidades de distancia del punto  $(0,0.5,0)$

(continua en la siguiente transparencia).

## Problemas: parámetros para una vista concreta (2/3)

### Problema 3.1. (continuación)

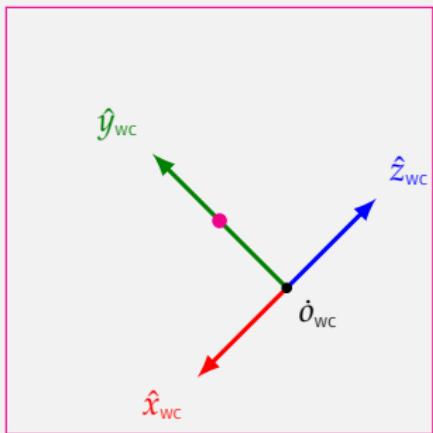
Escribe unos valores que podríamos usar para  $\mathbf{a}$ ,  $\mathbf{u}$  y  $\mathbf{n}$  de forma que se cumplan estos requisitos. En la figura se observa una vista esquemática de como quedaría la figura en un viewport cuadrado, no necesariamente a escala.



# Problema: parámetros para una vista concreta (3/3)

## Problema 3.2.

Repite el problema anterior, pero ahora para esta vista:



# Problema: construcción de la matriz de vista

## Problema 3.3.

Escribe el código para calcular los vectores de coordenadas  $\mathbf{x}_{ec}$ ,  $\mathbf{y}_{ec}$ ,  $\mathbf{z}_{ec}$  y  $\mathbf{o}_{ec}$  que definen el marco de vista a partir de los vectores de coordenadas  $\mathbf{a}$ ,  $\mathbf{u}$  y  $\mathbf{n}$  (todos estos vectores de coordenadas son de tipo **Tupla3f**).

## Problema 3.4.

Partiendo de los vectores de coordenadas  $\mathbf{x}_{ec}$ ,  $\mathbf{y}_{ec}$ ,  $\mathbf{z}_{ec}$  y  $\mathbf{o}_{ec}$  que se calculan en el problema anterior, escribe el código que calcula explícitamente las 16 entradas de la matriz de vista (crea una **Matriz4f** llamada  $\mathbf{V}$  y luego asigna valor a  $\mathbf{V}(i, j)$  para cada fila  $i$  y columna  $j$ , ambas entre 0 y 3).

Informática Gráfica, curso 2021-22.

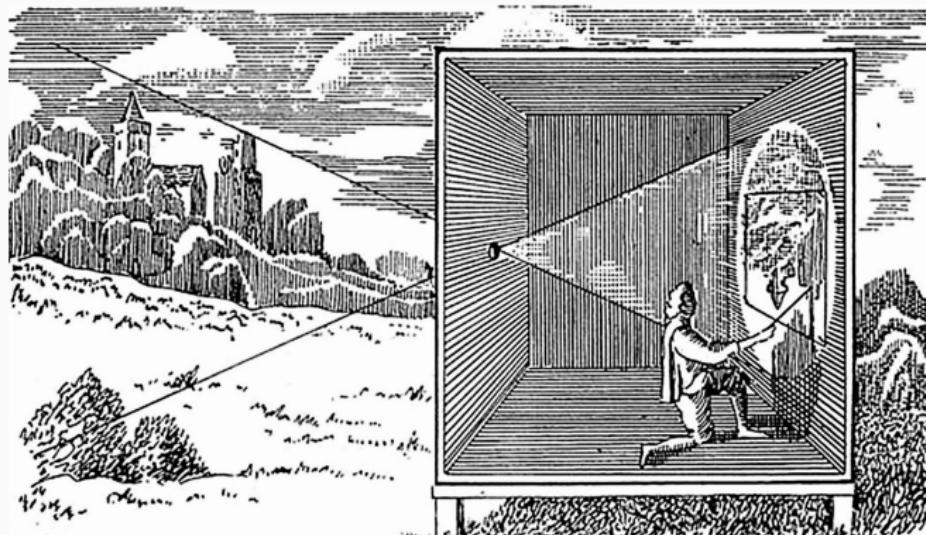
Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.3.  
Transformación de proyección.

# Introducción

La transformación de proyección emula la proyección que ocurre idealmente en una cámara oscura, sobre la pared opuesta a la apertura. Es similar a lo que ocurre en una cámara de fotografía, al proyectarse la escena sobre el sensor.



# El plano de visión. Tipos de proyección

Los vértices se *proyectan* sobre un plano alineado con el sistema de referencia de la cámara:

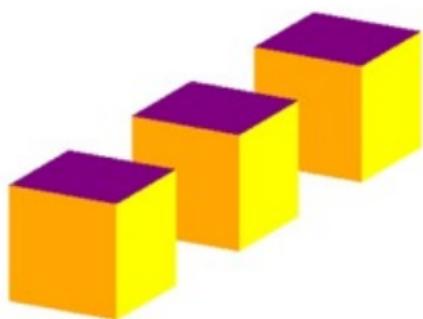
- ▶ Dicho plano se denomina **plano de visión (viewplane)**, es siempre perpendicular al eje Z del marco de vista.

La proyección puede ser de dos tipos

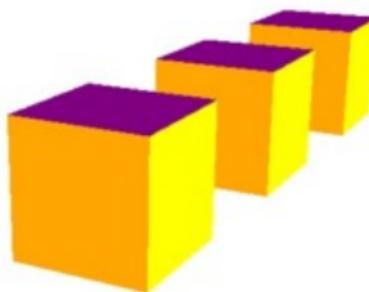
- ▶ **Proyección perspectiva:** los vértices se proyectan sobre el plano de visión usando líneas que van desde cada punto al origen del marco de coordenadas de la cámara (a esas líneas se les llama **proyectores**, el origen actua como **foco** de la proyección). La coordenada Z del plano de visión debe ser estrictamente positiva.
- ▶ **Proyección ortográfica (o paralela):** los proyectores son todos paralelos al eje Z. El plano de visión puede estar situado en cualquier valor de Z. Es un caso límite de la perspectiva, con el foco infinitamente alejado de la escena.

# Comparación de proyecciones

Aunque ninguna de las dos formas de proyección es igual al comportamiento del sistema visual humano, la proyección perspectiva nos parece más natural (la ortográfica es poco realista):



Orthographic Projection



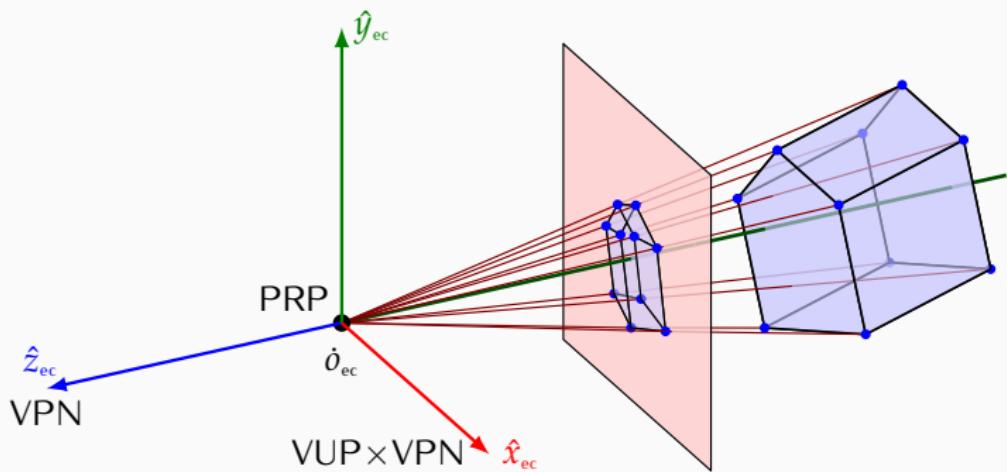
Perspective Projection

A la izquierda, se interpreta que el cubo más lejano es más grande que los otros, aunque en la imagen son los tres del mismo tamaño

Microsoft WPF documentation: 3D Graphics Overview

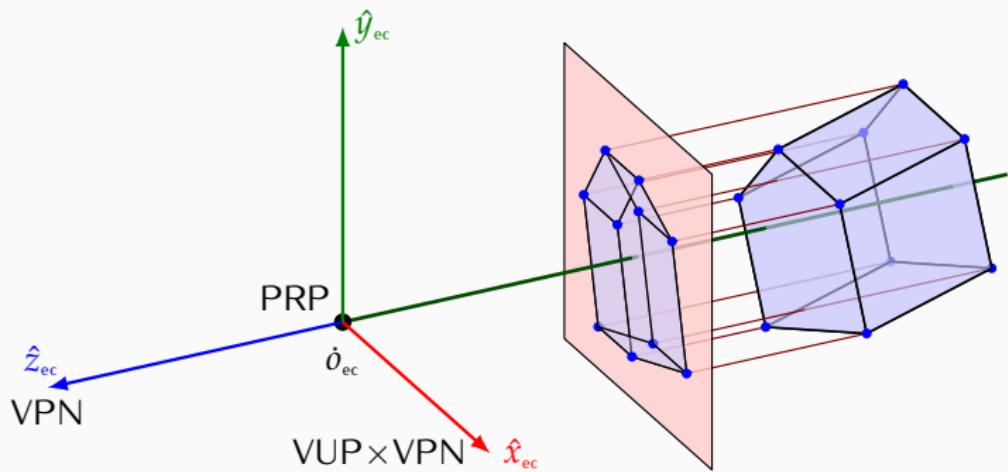
# Proyección perspectiva

Cambia el tamaño de los objetos, usando un factor de escala  $s$  que crece de forma inv. proporcional a la distancia ( $d_z$ ) en Z desde el objeto al foco ( $s$  es de la forma  $1/(ad_z + b)$ )



# Proyección paralela

En este caso, no hay transformación de escala, y la proyección se puede ver como una transformación afín:



# El *view-frustum*

El *view-frustum* designa la región del espacio de la escena que es visible en el viewport. Su forma depende del tipo de proyección:

- ▶ Perspectiva: es un tronco de pirámide rectangular (izq.).
- ▶ Ortográfica: es un paralelepípedo ortogonal u ortoedro (der.).

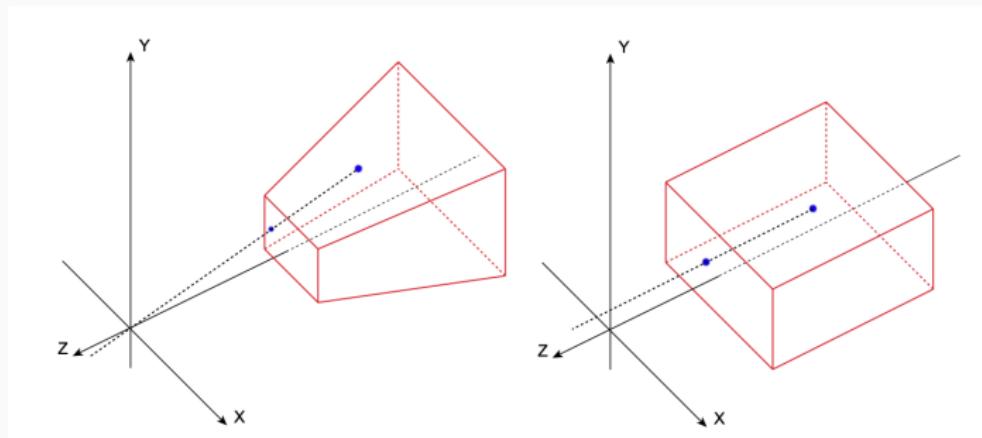


Imagen obtenida de: SGI® OpenGL Multipipe™ SDK User's Guide

## Transformación del view-frustum en un cubo

El view-frustum está determinado por los 6 planos que contienen a las 6 caras que lo delimitan.

- ▶ Estos planos se determinan por sus coordenadas en el marco de coordenadas de vista.
- ▶ La transformación de proyección transforma el view-frustum (en coordenadas de vista) en un cubo de lado 2 centrado en el origen, entre -1 y 1 en los tres ejes (en coordenadas de recortado, normalizadas).
- ▶ La proyección ortográfica es una transformación afín: una traslación seguida de escalado no necesariamente uniforme.
- ▶ La proyección perspectiva no es una transformación afín, aunque se puede expresar usando una transformación afín en coordenadas homogéneas 4D, seguida de una proyección de 4D a 3D.

## Parámetros del view-frustum. Extension en Z

Los 6 valores  $l, r, t, b, n$  y  $f$  (los **parámetros** de frustum) determinan la transformación de la tupla  $(x_{ec}, y_{ec}, z_{ec})$ , que está en coordenadas de vista en la tupla  $(x_{ndc}, y_{ndc}, z_{ndc})$  en NDCC (*coordenadas normalizadas de dispositivo*, entre -1 y 1):

- ▶ Los valores  $n$  (**near**) y  $f$  (**far**) son los límites en Z del view-frustum, pero cambiados de signo (se cumple  $n \neq f$ ).
  - ▶ El plano  $z_{ec} = -n$  en EC se transforma en el plano  $z_{ndc} = -1$  en NDC.
  - ▶ El plano  $z_{ec} = -f$  en EC se transforma en el plano  $z_{ndc} = +1$  en NDC.
- ▶ En la proyección perspectiva, se exige además  $0 < n$  y  $0 < f$ .
- ▶ Aunque no se exige así, lo usual es que seleccione  $n < f$ , es decir, el view-frustum se extiende en Z en el intervalo  $[-f, -n]$ .

En adelante supondremos  $n < f$ , de forma que:

- ▶ El plano  $z_{ec} = -n$  se llama **plano de recorte delantero**
- ▶ El plano  $z_{ec} = -f$  se llama **plano de recorte trasero**

## Parámetros del view-frustum. Extension en X e Y.

Respecto de los otros cuatro valores ( $l, r, b$  y  $t$ ), determinan la extensión en X y en Y:

- ▶  $l$  (**left**) y  $r$  (**right**) son los límites en X del view-frustum ( $l \neq r$ ).
- ▶  $b$  (**bottom**) y  $t$  (**top**) son los límites en Y ( $b \neq t$ ).
- ▶ En proy. ortográfica:
  - ▶ El plano  $x_{ec} = l$  en EC se transforma en el plano  $x_{ndc} = -1$  en NDC.
  - ▶ El plano  $x_{ec} = r$  en EC se transforma en el plano  $x_{ndc} = +1$  en NDC.
  - ▶ El plano  $y_{ec} = b$  en EC se transforma en el plano  $y_{ndc} = -1$  en NDC.
  - ▶ El plano  $y_{ec} = t$  en EC se transforma en el plano  $y_{ndc} = +1$  en NDC.
- ▶ En proy. perspectiva:
  - ▶ El plano  $-nx_{ec} = lz_{ec}$  (EC) se transf. en el plano  $x_{ndc} = -1$  en NDC.
  - ▶ El plano  $-nx_{ec} = rz_{ec}$  (EC) se transf. en el plano  $x_{ndc} = +1$  en NDC.
  - ▶ El plano  $-ny_{ec} = bz_{ec}$  (EC) se transf. en el plano  $y_{ndc} = -1$  en NDC.
  - ▶ El plano  $-ny_{ec} = tz_{ec}$  (EC) se transf. en el plano  $y_{ndc} = +1$  en NDC.

# Propiedades de la extensión en X e Y

Hay que tener en cuenta que:

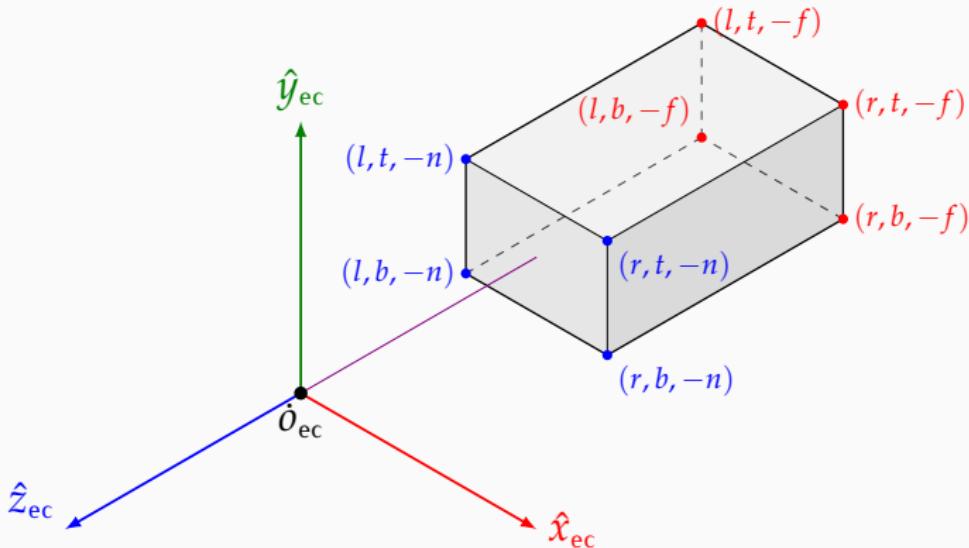
- ▶ Aunque esto no es requerido estrictamente, usualmente se seleccionan los parámetros de forma que  $l < r$  y  $b < t$ .
- ▶ Cuando se cumple  $l = -r$  y  $b = -t$ , decimos que el **view-frustum está centrado** (el eje Z pasa por el centro de las caras delantera y trasera). Esto es lo más usual, y se corresponde con lo que ocurre en una cámara.
- ▶ El valor  $(r - l)/(t - b)$  suele coincidir con la relación de aspecto del viewport (ancho/alto, o bien `núm.columnas/núm.filas`). Si esto no ocurre los objetos aparecerán deformados en la imagen.

En adelante supondremos que siempre seleccionamos  $l < r$  y  $b < t$ .

# Parámetros en la proyección ortográfica.

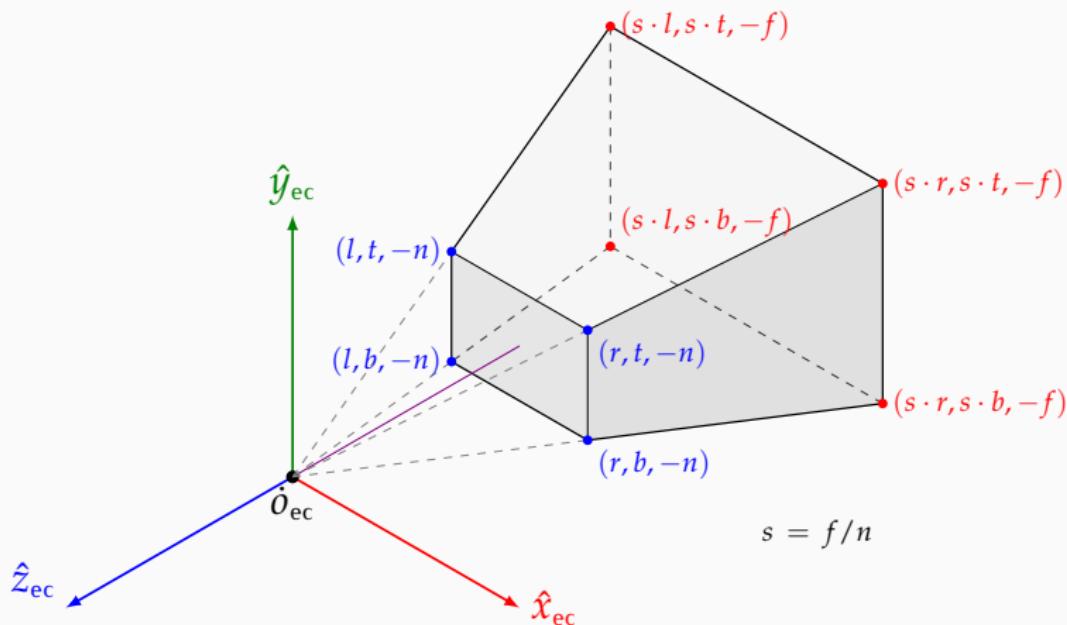
En pr. ortográfica el view-frustum es un **ortoedro**. Contiene los puntos cuyas coordenadas de cámara  $(x_{ec}, y_{ec}, z_{ec})$  cumplen:

$$l \leq x_{ec} \leq r \quad b \leq y_{ec} \leq t \quad -f \leq z_{ec} \leq -n$$



# Parámetros en la proyección perspectiva (1/2)

En perspectiva, el view-frustum es una pirámide rectangular truncada:



## Parámetros en la proyección perspectiva (2/2)

Los puntos dentro del view-frustum son aquellos cuyas coordenadas de cámara  $(x_{ec}, y_{ec}, z_{ec})$  cumplen:

En el eje X:

$$l \leq x_{ec} \left( \frac{n}{-z_{ec}} \right) \leq r$$

En el eje Y:

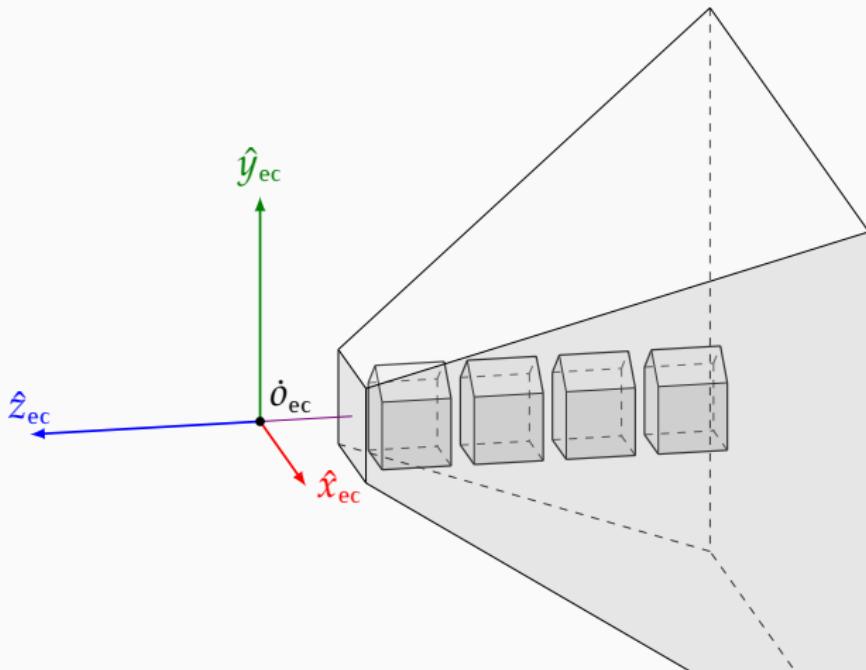
$$b \leq y_{ec} \left( \frac{n}{-z_{ec}} \right) \leq t$$

En el eje Z:

$$-f \leq z_{ec} \leq -n$$

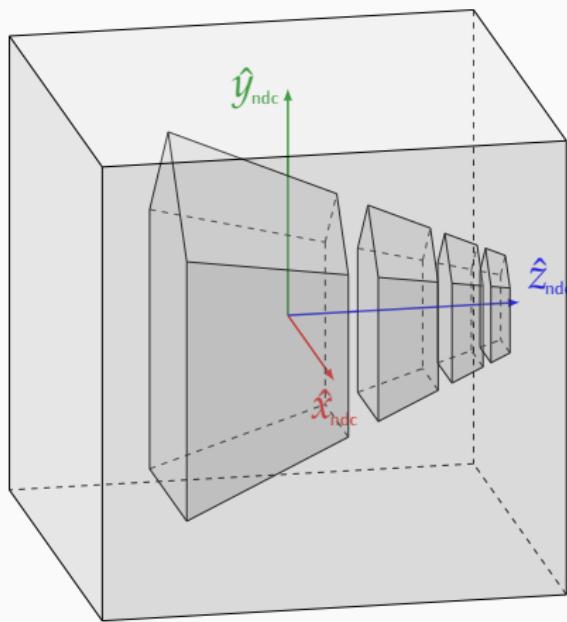
# Escena de ejemplo para transformación perspectiva

Suponemos que partimos de una escena que vamos a proyectar usando perspectiva. En el espacio de coordenadas de cámara, la escena es esta:



# Escena proyectada por transformación perspectiva

El efecto de la transformación de proyección perspectiva será hacer más pequeños los objetos más alejados del observador, y situar la escena en un cubo de lado 2:



## Proyección perspectiva sobre el plano delantero

Podemos suponer que los puntos se proyectan sobre el plano frontal del view-frustum (coord. Z igual a  $-n$ ) con foco en  $\mathbf{o}_{ec}$ :

- ▶ Dado un punto  $\mathbf{p} = \mathcal{V}(x_{ec}, y_{ec}, z_{ec}, w_{ec})$  queremos calcular las coordenadas de su proyección  $(x', y', z', w')$  (en principio, con  $w' = w_{ec} = 1$ ).
- ▶ Si se asume  $z_{ec} < 0$  (el punto está en la rama negativa del eje Z), podemos hacer:

$$x' \equiv \frac{x_{ec}n}{-z_{ec}} \quad y' \equiv \frac{y_{ec}n}{-z_{ec}} \quad z' \equiv \frac{z_{ec}n}{-z_{ec}} = -n$$

esta transformación tiene dos problemas:

- ▶ Las coordenadas resultado no están entre  $-1$  y  $1$ .
- ▶ Colapsa o *aplana* todas las coordenadas Z (son todas  $-n$ ).

## Normalización de coordenadas X e Y

Si el punto original está en el view-frustum, entonces:

- ▶  $x'$  está en el intervalo  $[l, r]$
- ▶  $y'$  está en el intervalo  $[b, t]$ .
- ▶ queremos dejar ambas coordenadas en el intervalo  $[-1, 1]$
- ▶ podemos usar un escalado y traslación adicionales en X e Y:

$$x'' \equiv 2 \left( \frac{x' - l}{r - l} \right) - 1 = \frac{a_0 x_{ec}}{-z_{ec}} - a_1 = \frac{a_0 x_{ec} + a_1 z_{ec}}{-z_{ec}}$$

$$y'' \equiv 2 \left( \frac{y' - b}{t - b} \right) - 1 = \frac{b_0 y_{ec}}{-z_{ec}} - b_1 = \frac{b_0 y_{ec} + b_1 z_{ec}}{-z_{ec}}$$

donde hemos usado estas cuatro constantes:

$$a_0 \equiv \frac{2n}{r - l} \quad a_1 \equiv \frac{r + l}{r - l} \quad b_0 \equiv \frac{2n}{t - b} \quad b_1 \equiv \frac{t + b}{t - b}$$

# Información de profundidad y normalización en Z

El problema está de hacer  $z' = -n$  está en que **se pierde información de profundidad en Z**, que es necesaria para EPO). Para evitarlo, se usa una función lineal de  $z$  con dos constantes  $c_0$  y  $c_1$ :

$$z'' \equiv \frac{c_0 z_{ec} + c_1}{-z_{ec}} \quad \text{donde:} \quad c_0 \equiv \frac{n+f}{n-f} \quad c_1 \equiv \frac{2fn}{n-f}$$

- ▶ los dos valores  $c_2$  y  $c_3$  se eligen de forma que, para  $z_{ec} = -n$ , se hace  $z'' = -1$ , y para  $z_{ec} = -f$ , se hace  $z'' = 1$ .
- ▶ es decir: el rango  $[-f, -n]$  se lleva al rango  $[-1, 1]$  (invirtiendo el orden).
- ▶ esta transformación conserva el orden (invertido) de las coordenadas Z (*no aplana*)
- ▶ ahora, valores menores de Z implican más cercanos al observador, y valores mayores, más lejanos.

# Coordenadas cartesianas del punto proyectado

En resumen, tenemos estas tres igualdades:

$$x'' \equiv \frac{a_0 x_{ec} + a_1 z_{ec}}{-z_{ec}}$$

$$y'' \equiv \frac{b_0 x_{ec} + b_1 z_{ec}}{-z_{ec}}$$

$$z'' \equiv \frac{c_0 z_{ec} + c_1}{-z_{ec}} = \frac{c_0 z_{ec} + c_1 w_{ec}}{-z_{ec}}$$

esta transformación incluye una división, y por tanto

- ▶ no se puede implementar con una matriz como hacíamos con las anteriores (no es lineal)
- ▶ aunque sí transforma líneas rectas en líneas rectas

## Obtención de las coordenadas de recortado

Para solventar el problema anterior (para poder usar una matriz), se definen las **coordenadas de recortado (*clip coordinates*)**, a partir de las coordenadas de cámara del original:

$$x_{cc} \equiv a_0 x_{ec} + a_1 z_{ec}$$

$$y_{cc} \equiv b_0 y_{ec} + b_1 z_{ec}$$

$$z_{cc} \equiv c_0 z_{ec} + c_1 w_{ec}$$

$$w_{cc} \equiv -z_{ec}$$

Esta transformación ya **sí se puede hacer con una matriz 4x4**:

- ▶ se ha eliminado la división por  $-z_{ec}$ , el resto es igual
- ▶ esta división se hace más adelante en el cauce gráfico
- ▶ para ello, el denominador de la división ( $-z_{ec}$ ) queda guardado en  $w_{cc}$  (que ya no es 1).

## La matriz de proyección perspectiva $Q$

Con todo lo dicho, la proyección perspectiva se puede realizar usando una matriz  $Q$ , que se aplica a coordenadas de cámara (con  $w_{ec} = 1$ ) y produce coordenadas de recortado (con  $w_{cc} \neq 1$ ):

$$\begin{pmatrix} x_{cc} \\ y_{cc} \\ z_{cc} \\ w_{cc} \end{pmatrix} = \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_{ec} \\ y_{ec} \\ z_{ec} \\ 1 \end{pmatrix}$$

evidentemente, podemos definir entonces la matriz  $Q$  de esta forma:

$$Q \equiv \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

# La proyección ortográfica.

En el caso de la **proyección ortográfica** (*orthographic projection*), se hace proyección en una dirección paralela al eje Z:

- ▶ esta transformación **solo requiere la normalización de los rangos de valores en los tres ejes** (se usa traslación más escalado)
- ▶ (1) traslación  $T$  (lleva el centro del paralelepípedo al origen)

$$T \equiv \text{Tra} \left[ -\frac{l+r}{2}, -\frac{t+b}{2}, -\frac{f+n}{2} \right]$$

- ▶ (2) escalado  $S$  (deja los valores en  $[-1, 1]$  en los tres ejes):

$$S \equiv \text{Esc} \left[ \frac{2}{r-l}, \frac{2}{t-b}, \frac{-2}{f-n} \right]$$

(en Z se cambia de signo para *invertir* el eje Z)

# La matriz de proyección ortográfica

La matriz de proyección ortográfica  $O$  se obtiene por tanto como composición de  $T$  seguido de  $S$ , es decir  $O = S \cdot T$ , o lo que es lo mismo:

$$O = \begin{pmatrix} a'_0 & 0 & 0 & a'_1 \\ 0 & b'_0 & 0 & b'_1 \\ 0 & 0 & c'_0 & c'_1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ donde: } \begin{cases} a'_0 \equiv \frac{2}{r-l} & a'_1 \equiv -\frac{r+l}{r-l} \\ b'_0 \equiv \frac{2}{t-b} & b'_1 \equiv -\frac{t+b}{t-b} \\ c'_0 \equiv \frac{-2}{f-n} & c'_1 \equiv -\frac{f+n}{f-n} \end{cases}$$

de forma que ahora hacemos:

$$(x_{cc}, y_{cc}, z_{cc}, w_{cc})^t = O(x_{ec}, y_{ec}, z_{ec}, w_{ec})^t$$

donde  $w_{cc}$  sí vale 1 con seguridad.

## Matriz de proyección en OpenGL (cauce fijo)

En el estado de OpenGL, cuando se usa el cauce de funcionalidad prefijada, hay una matriz de proyección (*projection matrix*) que llamaremos  $P$  y que puede ser manipulada mediante varias llamadas:

- ▶ La función **glMatrixMode** se puede usar para poner OpenGL en *modo matriz de proyección*, usando la constante **GL\_MATRIXMODE**, de forma que posteriores operaciones se realicen sobre la matriz  $P$ .
- ▶ A modo de ejemplo, estas llamadas hacen  $P$  igual a la matriz identidad:

```
glMatrixMode( GL_PROJECTION ); // entrar en modo 'matriz proyeccion'  
glLoadIdentity(); // hace P := Ide
```

## Definición de la matriz de proyección (cauce fijo)

Para componer la matriz  $Q$  con  $P$  se puede invocar a **glFrustum**, una función declarada como se indica aquí:

```
glFrustum( GLdouble l, GLdouble r,  
           GLdouble b, GLdouble t,  
           GLdouble n, GLdouble f ); // hace  $P := P \cdot Q$ 
```

Si queremos una proyección ortográfica (matriz  $O$ ), podemos usar **glOrtho** en lugar de **glFrustum**, con los mismos parámetros:

```
glOrtho( GLdouble l, GLdouble r,  
         GLdouble b, GLdouble t,  
         GLdouble n, GLdouble f ); // hace  $P := P \cdot O$ 
```

## Transf. de proyección con gluPerspective

En la librería GLU (funciona sobre OpenGL) se puede usar la llamada a la función **gluPerspective**, para componer una proyección perspectiva de forma más intuitiva:

```
gluPerspective( GLdouble β, GLdouble a, // calcula Q a partir de β,a,n,f  
GLdouble n, GLdouble f ) ; // y luego hace P := P · Q
```

esta función equivale a **glFrustum** centrado con:

$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

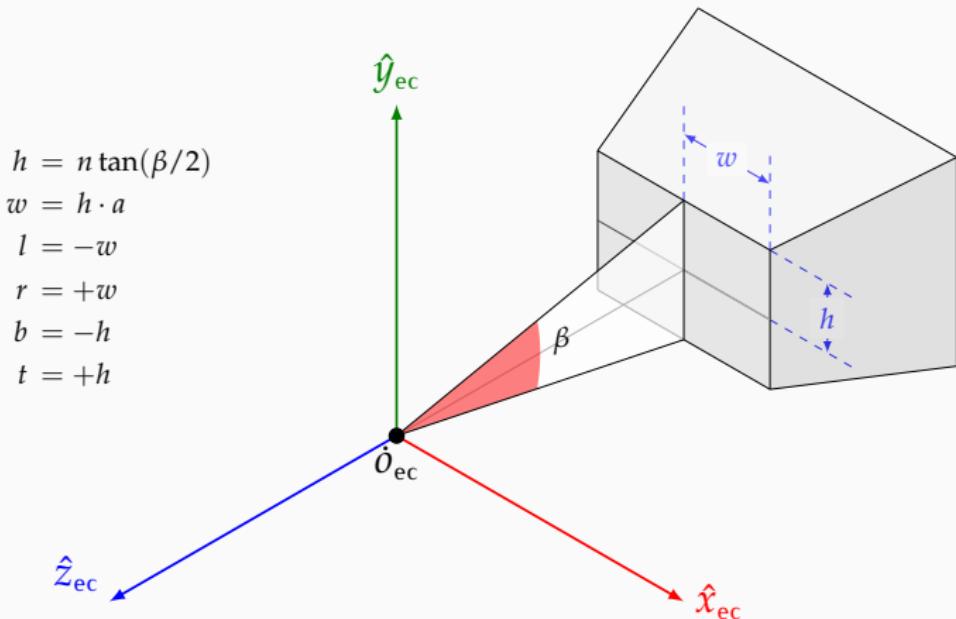
donde:

$\beta$   $\equiv$  es la **apertura vertical del campo de visión** (*fovy*), es el ángulo en grados (entre 0 y 180.0) que hay entre la cara superior y la inferior del frustum

$a$   $\equiv$  **relación de aspecto** (*aspect ratio*) de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).

# Parámetros de gluPerspective

El significado de los parámetros se aprecia en esta figura:



Esta perspectiva es *centrada*, ya que  $r = -l$  y  $t = -b$

# Construcción explícita de la matriz de proyección

Para construir explicitamente una matriz de proyección (para el cauce programable) se pueden usar funciones que devuelven **Matriz4f**. En el código de prácticas se encuentran disponibles estas:

```
#include <matrices-tr.h> // include de librería de generación de matrices

// construye matriz O (misma matriz que glOrtho)
Matriz4f MAT_Ortografica( const float l, const float r,
                           const float b, const float t,
                           const float n, const float f );

// construye matriz Q (misma matriz que glFrustum)
Matriz4f MAT_Frustum      ( const float l, const float r,
                            const float b, const float t,
                            const float n, const float f );

// construye matriz Q (misma matriz que gluPerspective)
Matriz4f MAT_Perspectiva( const float fovy, const float a,
                           const float n, const float f );
```

# Matriz de proyección en el cauce programable

Si se usa un cauce programable, es necesario:

- ▶ Declarar un parámetro *uniform* en el *vertex shader* de tipo **mat4**, que contendrá la matriz de proyección.
- ▶ Incluir código en el *vertex shader* de forma que la última etapa de transformación de la posición de un vértice sea multiplicar por esa matriz.
- ▶ Como parte de la inicialización del cauce, obtener y guardar la localización de dicho parámetro (con **glGetUniformLocation**).
- ▶ Al inicio de la aplicación (o al inicio de la visualización de un frame), se debe construir la matriz de proyección  $P$  (de tipo **Matriz4f**) usando **MAT\_Frustum**, **MAT\_Ortografica** u otras funciones parecidas.
- ▶ Finalmente, usando la localización y  $P$ , se debe fijar el valor del parámetro *uniform* con **glUniformMatrix4fv**.

## Código del vertex shader

En este ejemplo, el uniform **matrizP** tiene la matriz de proyección:

```
// parámetros de entrada uniform (iguales en todos los vértices de cada primitiva)
uniform mat4 matrizMV ;           // matriz 4x4 de transf. de coord. de vértices
uniform mat4 matrizMV_nor;        // matriz 4x4 de transf. de normales
uniform mat4 matrizP ;           // matriz 4x4 de proyección (produce coord.pantalla)

// variables de salida varying (atributos de vértice: serán interpolados a pixels)
varying vec4 var_posic_ec;       // posición (en coords de cámara)
varying vec3 var_normal_ec;      // normal (en coords. de camara)
varying vec4 var_color;          // color
varying vec2 var_coord_text;     // coordenadas de textura

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_color, gl_MultiTexCoord0

void main() // escribe variables 'varying', más 'gl_Position'
{
    var_posic_ec  = matrizMV * gl_Vertex;           // transf. coord. recibida
    var_normal_ec = matrizMV_nor * gl_Normal;        // transf. normal recibida
    var_color     = gl_color ;                      // usar color enviado
    var_coord_text= gl_MultiTexCoord0.st ;           // usa cc.t. enviadas
    gl_Position    = matrizP * var_posic_ec;         // proyecta a pantalla
}
```

## Uso de la clase cauce

Para facilitar la gestión del cauce fijo y el cauce programable se puede usar el método **fijarMatrizProyeccion** del la clase **Cauce**. Las respectivas implementaciones son básicamente así:

```
void CauceFijo::fijarMatrizProyeccion( const Matriz4f & nue_mat_pro )
{
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glMultMatrixf( nue_mat_pro );
}
```

```
void CauceProgramable::fijarMatrizProyeccion( const Matriz4f & nue_mat_pro )
{
    mat_proyeccion = nue_mat_pro; // no es estrictamente necesario
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_proyeccion, 1, GL_FALSE, mat_proyeccion );
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.4.

Recortado y división por  $W$ .

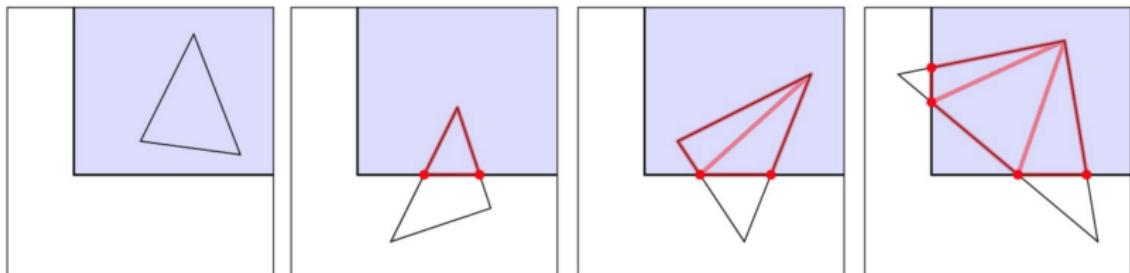
## Recortado

Una vez se tienen las coordenadas de recortado de los vértices, se comprueban que primitivas estan dentro o fuera del viewfrustum (que en CC es un cubo de lado dos unidades centrado en el origen):

- ▶ Las primitivas completamente dentro de la zona visible se mantienen
- ▶ Las primitivas completamente fuera de la zona visible se descartan
- ▶ Las primitivas parcialmente dentro se dividen en partes unas completamente dentro (se vis) y otras completamente fuera (que se descartan). Esto causa la inserción de nuevas primitivas con algunos vértices nuevos justo en los planos que delimitan el view-frustum.

# Inserción de nuevos vértices y triángulos

Varios ejemplos de recortado de triángulos:



- ▶ Las coordenadas y otros atributos de los nuevos vértices se interpolan a partir de los vértices en los dos extremos de la arista donde se inserta el nuevo.
- ▶ Se hace recortado independiente por cada uno de los 6 planos de recorte.

Figura obtenida de *CMU Computer Graphics Course (fall 2019)*:

☞ <http://15462.courses.cs.cmu.edu/fall2019/>

## División por W. Coordenadas normalizadas de dispositivo.

Los vértices (en el view-frustum) resultado del recorte tienen coordenadas de recorte con  $w_{cc} \neq 0$  (si  $P = Q$ , entonces además  $w_{cc} \neq 1$ ). El siguiente paso es hacer la división por  $w_{cc}$  de las tres componentes. Se obtienen las **coordenadas normalizadas de dispositivo**, con componente W de nuevo a 1:

$$(x_{ndc}, y_{ndc}, z_{ndc}, 1) = \frac{1}{w_{cc}} (x_{cc}, y_{cc}, z_{cc}, w_{cc})$$

los valores  $x_{ndc}$ ,  $y_{ndc}$  y  $z_{ndc}$  están los tres en el intervalo  $[-1, 1]$  (los vértices ya han pasado el recortado).

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.5.  
Transformación de viewport.

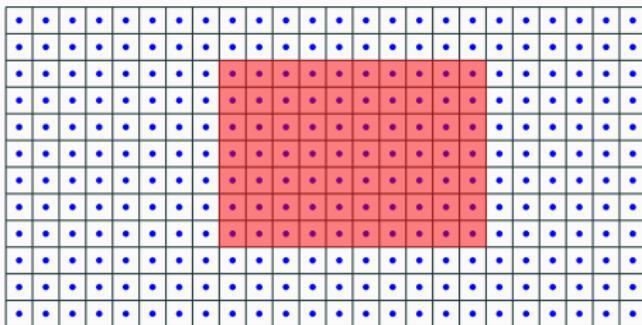
# Transformación de Viewport

El siguiente paso consiste en calcular en qué posiciones de la imagen se proyecta cada vértice:

- ▶ Este paso se puede modelar como una transformación lineal que llamaremos **transformación de viewport**. El término **viewport** hace referencia a la zona rectangular de la ventana donde se proyectarán los polígonos que están en el cubo visible (un bloque rectangular de pixels)
- ▶ Esta transformación produce **coordenadas de dispositivo o de ventana** (DC: *device coordinates*, o también llamadas *screen coordinates*, o *window coordinates*). Las coordenadas X e Y en DC se expresan en unidades de pixels.
- ▶ La transformación de viewport es lineal y consta simplemente de escalados y traslaciones.
- ▶ La coordenada Z se transforma y se conserva para poder hacer después *eliminación de partes ocultas*.

# Coordenadas de dispositivo y pixels del viewport

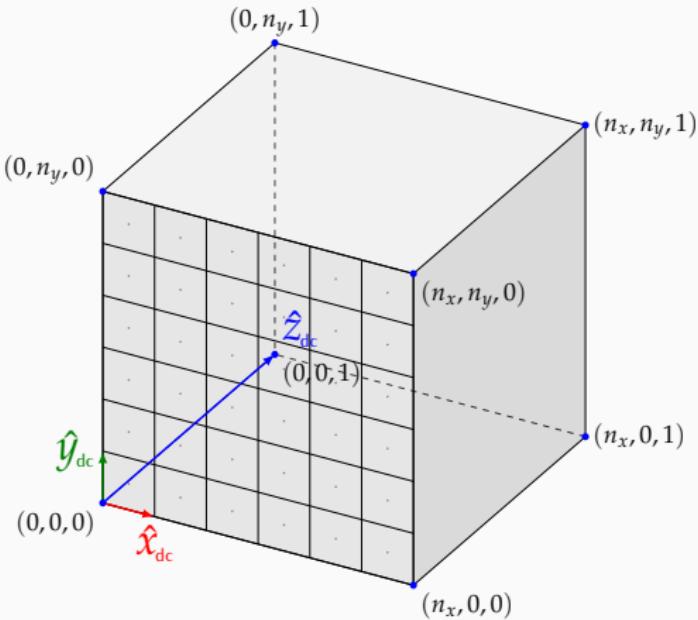
En coordenadas de dispositivo, podemos asociar una región cuadrada (de lado unidad) a cada pixel en el plano de la ventana. El viewport (en rojo) es un bloque rectangular de pixels, contenido en el bloque rectangular correspondiente a la ventana o imagen completa:



los centros de los pixels (puntos azules) tienen coordenadas de dispositivo con parte fraccionaria igual a  $1/2$ . Los bordes entre pixels tienen coordenadas sin parte fraccionaria (enteras).

# El espacio de coordenadas de dispositivo

En 3D el espacio de coordenadas de dispositivo es un ortoedro. Se puede visualizar como aparece aquí, incluyendo el marco de coordenadas de dispositivo:



## Matriz del viewport y parámetros en OpenGL.

OpenGL tiene en su estado una matriz  $4 \times 4$  que llamaremos  $V$ , y que depende de estos parámetros (ver fig.)

$x_l, y_b$  número de columna y fila (enteros no negativos) del pixel que ocupa, en la ventana, la esquina inferior izquierda del viewport.

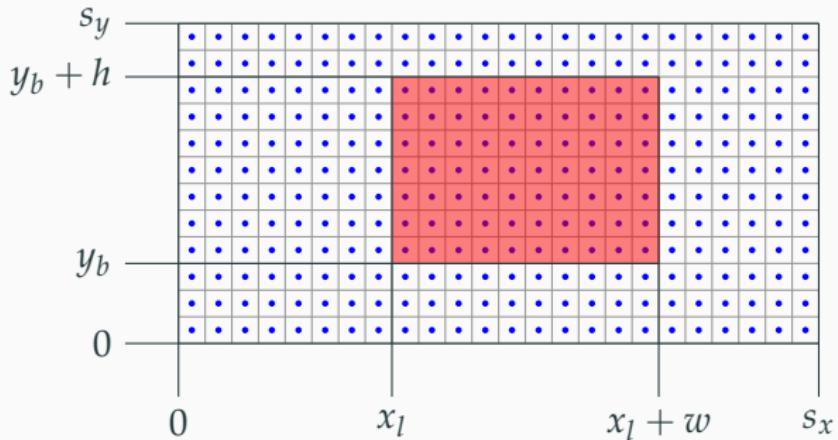
$w, h$  (*width* y *height*) número total (entero no negativo) de columnas y de filas de pixels (respectivamente) que ocupa el viewport.

$n_d, f_d$  rango de valores de salida en Z en DC. El valor  $n_d$  es la profundidad más cercana posible al observador, y  $f_d$  la más lejana. Por defecto  $n_d = 0$  y  $f_d = 1$ .

aunque los cuatro parámetros relevantes ( $x_l, y_b, w$  y  $h$ ) son enteros, las coordenadas de dispositivos son valores reales, ya que las posiciones de los vértices en DC son en general no enteras (no coinciden necesariamente con los centros o bordes de los pixels).

# Parámetros del viewport

Suponemos que la ventana tiene  $s_x$  columnas y  $s_y$  filas, y que el gestor de ventanas acepta coordenadas de pixels enteras no negativas:



se deben cumplir estas desigualdades:  $\begin{cases} 0 \leq x_l < x_l + w \leq s_x \\ 0 \leq y_b < y_b + h \leq s_y \end{cases}$

# La transformación de viewport

En NDC las coordenadas están en  $[-1, 1]$ , luego hay que hacer:

1. traslación de la esquina  $(-1, -1, -1)$  al origen.
2. escalado uniforme (por  $1/2$ ) y por  $(w, h, f_d - n_d)$
3. traslación del origen a  $(x_l, y_b, n_d)$ .

con lo cual la transformación  $D$  queda como:

$$D = \text{Tra}[x_l, y_b, n_d] \cdot \text{Esc}[w, h, f_d - n_d] \cdot \text{Esc}[1/2] \cdot \text{Tra}[1, 1, 1]$$

por tanto, las **coordenadas de dispositivo**  $(x_{dc}, y_{dc}, z_{dc}, 1)$  se definen a partir de las normalizadas  $(x_{ndc}, y_{ndc}, z_{ndc}, 1)$  de esta forma:

$$\begin{aligned}x_{dc} &= (x_{ndc} + 1)w/2 + x_l \\y_{dc} &= (y_{ndc} + 1)h/2 + y_b \\z_{dc} &= (z_{ndc} + 1)(f_d - n_d)/2 + n_d\end{aligned}$$

## La matriz de viewport $D$

Por tanto, la matriz de viewport  $D$  debe definirse así:

$$D = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x_l + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y_b + \frac{h}{2} \\ 0 & 0 & \frac{z_f - z_{ndc}}{2} & \frac{z_f + z_{ndc}}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

de forma que:

$$(x_{dc}, y_{dc}, z_{dc}, 1)^t = D (x_{ndc}, y_{ndc}, z_{ndc}, 1)^t$$

# Fijar la matriz de viewport en OpenGL

En cualquier momento (independientemente del *matrix mode* activo en dicho momento) es posible cambiar la matriz  $D$  que OpenGL almacena como parte de su estado.

Para ello llamamos a la función **glViewport**, declarada como sigue:

```
glViewport(GLint xl, GLint yb, GLsizei w, GLsizei h);
```

- ▶ Los rangos de valores permitidos para estos parámetros dependen de la implementación, del hardware subyacente y del gestor o librería de ventanas en uso.
- ▶ Si  $w$  y/o  $h$  son demasiado grandes, no se produce error, pero se truncan.
- ▶ Por defecto, OpenGL fija el viewport ocupando todos los pixels de la ventana.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.6.  
Representación de cámaras.

## Representación de cámaras

La clase **Camara** encapsula todos los parámetros relacionados con la matriz de vista y proyección.

- ▶ Es una clase base con funcionalidad mínima. Se derivan clases con funcionalidad más avanzada.
- ▶ Incluye una matriz  $4 \times 4$  de vista  $V$  y otra de proyección  $P$ .
- ▶ El método **activar(c)** permite activar una cámara en un cauce **c** (referencia a una instancia de una clase derivada de **Cauce**). Este método simplemente fija las matrices en el cauce usando las que hay en la instancia.
- ▶ El método **actualizarMatrices** es un método **virtual**, que se encarga de calcular  $V$  y  $P$  a partir de los parámetros específicos de cada tipo de cámara.
- ▶ Por defecto, esta clase base define una cámara ortográfica que visualiza un cubo de lado 2 unidades en X y centro en el origen (en coords. de  $\mathcal{W}$ ).

# Clase Camara: declaración

```
class Camara
{
public:
    // fija las matrices model-view y projection en el cauce
    void activar( Cauce & cauce ) ;
    // cambio el valor de 'ratio_vp' (alto/ancho del viewport)
    void fijarRatioViewport( const float nuevo_{cc}ratio ) ;
    // lee la descripción de la cámara (y probablemente su estado)
    virtual std::string descripcion() ;

protected:
    bool      matrices_actualizadas = false; // true si matrices actualizadas
    Matriz4f  matriz_vista        = MAT_Ident(),    // matriz de vista
              matriz_proye       = MAT_Ident();    // matriz de proyección
    float     ratio_vp            = 1.0 ;           // ratio viewport (alto/ancho)

    // actualiza matriz_vista y matriz_proye a partir de los parámetros
    // específicos de cada tipo de cámara
    virtual void actualizarMatrices() ;
};

}
```

El ratio  $Y/X$  se almacena siempre para evitar deformaciones.

# Activación y actualización de una cámara

Cualquier tipo de cámara se activa fijando las matrices en el cauce a partir de las que se guardan en la instancia. Antes de eso se actualizan las matrices (recalcula **matriz\_vista** y **matriz\_proyección** si no estaban actualizadas)

```
void Camara::activar( Cauce & cauce )
{
    actualizarMatrices(); // recalcula si no están actualizadas
    cauce.fijarMatrizVista( matriz_vista );
    cauce.fijarMatrizProyeccion( matriz_proye );
}
```

El método **fijarRatioViewport** permite cambiar **ratio\_vp** para adaptarlo a las proporciones del viewport en uso:

```
void Camara::fijarRatioViewport( const float nuevo_ratio )
{
    ratio_vp = nuevo_ratio ;           // registrar nuevo ratio
    matrices_actualizadas = false; // matrices deben actualizarse antes de activar
}
```

# Matrices de la clase base Camara

La cámara básica define un view-frustum de lado 2 en X y en Z, y de lado  $2r$  en Y (donde  $r$  es el ratio del viewport, **ratio\_vp**). Está centrado en el origen de  $\mathcal{W}$ .

Por tanto, el método **actualizarMatrices** queda así:

```
void Camara::actualizarMatrices() // método virtual: redefinido en derivadas.  
{  
    if ( matrices_actualizadas )  
        return ;  
    matriz_vista = MAT_Ident();  
    matriz_proye = MAT_Escalado( 1.0f, 1.0f/ratio_vp, 1.0f );  
    matrices_actualizadas = true ;  
}
```

# Cámara orbital simple

En prácticas usamos una instancia de **CamaraOrbitalSimple** (derivada indirectamente de **Camara**), que define una cámara centrada en el origen con tres parámetros: dos ángulos (**a** y **b**) y una distancia al origen (**d**):

```
void CamaraOrbitalSimple::actualizarMatrices()
{
    // matriz de vista:
    matriz_vista = MAT_Traslacion( 0.0, 0.0, -d ) * // (3) despl. en Z por d
                    MAT_Rotacion( b, 1.0,0.0,0.0 ) * // (2) rotación eje X por b
                    MAT_Rotacion( -a, 0.0,1.0,0.0 ) ; // (1) rotacion eje Y por a
    // matriz de proyección:
    constexpr float          // parámetros de la matriz perspectiva (fijos)
        fovy_grad = 60.0,    // apertura vertical de campo, en grados
        near      = 0.05,    // distancia al plano de recorte delantero
        far       = near+1000.0 ; // dist. al plano de recorte trasero
    matriz_proye = MAT_Perspectiva( fovy_grad, ratio_vp, near, far );

    matrices_actualizadas = true; // registra que las matrices están ya actualizadas
}
```

## Problema 3.5.

Queremos visualizar una escena con mallas indexadas de la cual sabemos que tiene todos los vértices dentro de un cubo de lado  $s$  unidades cuyo centro es el punto de coordenadas del mundo  $\mathbf{c} = (c_x, c_y, c_z)$ .

Para construir la matriz de vista, se situa el observador en el punto  $\mathbf{o}_{\text{ec}} = (c_x, c_y, c_z + s + 2)$ , el punto de atención  $\mathbf{a}$  se hace igual a  $\mathbf{c}$  (el centro del cubo se ve en el centro de la imagen), y el vector  $\mathbf{u}$  es  $(0, 1, 0)$ . Se visualizará en un viewport cuadrado.

(continua en la siguiente página)

## Problemas: parámetros de matriz de proyección (2)

### Problema 3.5. (continuación)

Queremos construir la matriz de proyección perspectiva  $Q$  de forma que se cumplan estos requerimientos:

1. No se recorta ningún triángulo.
2. El tamaño aparente de los objetos (proyectados en pantalla) es el mayor posible.
3. El valor del parámetro  $n$  es el mayor posible.
4. El valor del parámetro  $f$  es el menor posible.
5. Los objetos no aparecen deformados.

Con estos requerimientos, indica como calcular los valores  $l, r, t, b, n$  y  $f$  (para obtener la matriz  $Q$  de proyección), en función de  $s$  y  $(c_x, c_y, c_z)$ .

# Problemas: parámetros de matriz de proyección (3)

## Problema 3.6.

Repite el problema anterior 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora la escena, en lugar de estar contenida en un cubo de lado  $s$  unidades, está contenida en una esfera de radio  $r$  unidades (con centro igualmente en  $\mathbf{c}$ ).

## Problema 3.7.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora, en lugar de suponer que el *viewport* es cuadrado, sabemos que tiene  $w$  columnas de pixels y  $h$  filas de pixels, y no podemos suponer que  $w = h$ .

## Problema 3.8.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo  $\beta$  en grados que debe ser la apertura de campo vertical de la proyección perspectiva. Para ello, ahora tenemos libertad para situar al observador en la línea paralela al eje Z que pasa por  $\mathbf{c}$ , de forma que la apertura de campo vertical sea exactamente  $\beta$ .

Indica como calcular la coordenada Z que debemos usar ahora para  $\mathbf{o}_{ec}$  (la X y la Y son las mismas que antes), de forma que se cumpla lo dicho, también indica como debemos de calcular ahora los valores de  $l, r, t, b, n$  y  $f$  (todo ello en función de  $\beta, s$  y  $\mathbf{c} = (c_x, c_y, c_z)$ ).

## Sección 2. Modelos de Iluminación, texturas y sombreado..

- 2.1. Radiación visible: características, percepción y reproducción.
- 2.2. Emisión y reflexión de la radiación.
- 2.3. Modelos computacionales simplificados.
- 2.4. Texturas
- 2.5. Métodos de sombreado para rasterización.

# Introducción

En este capítulo se hará una introducción a las últimas etapas del cauce gráfico de OpenGL, las encargadas de calcular un color en cada pixel:

- ▶ Dicho cálculo se puede hacer emulando la iluminación real que ocurre en los objetos de la naturaleza.
- ▶ Para ello es necesario diseñar un **modelo de iluminación**, un modelo formal que incluya las características relevantes del color de los polígonos.
- ▶ OpenGL incorpora un modelo sencillo y computacionalmente eficiente para esto.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.1.

Radiación visible: características, percepción y reproducción..

# La luz como radiación electromagnética

La luz que observamos es radiación electromagnética (variaciones periódicas del campo eléctrico y magnético) de naturaleza similar a las ondas que se usan para los móviles, wifi, radio y televisión:

- ▶ El sistema visual humano ha evolucionado para percibir esa radiación solo cuando su longitud de onda  $\lambda$  está aprox. entre 390 y 750 nanómetros ( $\equiv$  *espectro visible*).
- ▶ La emisión e interacción de las ondas en los átomos nos permite percibir el entorno.
- ▶ Físicamente, la radiación se describen como algo que tiene características de onda y de corpúsculo a la vez (modelos complementarios).
- ▶ En Informática Gráfica se usa más frecuentemente el *modelo de partículas* (óptica geométrica) en lugar del *modelo de ondas* (óptica física).

# El modelo de partículas. La radiancia.

Bajo este modelo, la radiación se puede describir de forma idealizada como un flujo en el espacio de partículas puntuales llamadas **fotones**, con trayectorias rectilíneas.

- ▶ Cada uno tiene una *energía radiante* que depende únicamente de su longitud de onda (es inv. prop.)
- ▶ En un entorno de punto **p** del espacio (típicamente en la superficie de un objeto) podemos medir la densidad de energía radiante por unid. de tiempo de los fotones de una longitud de onda  $\lambda$  que pasan por **p** en una determinada dirección **v** (un vector libre)

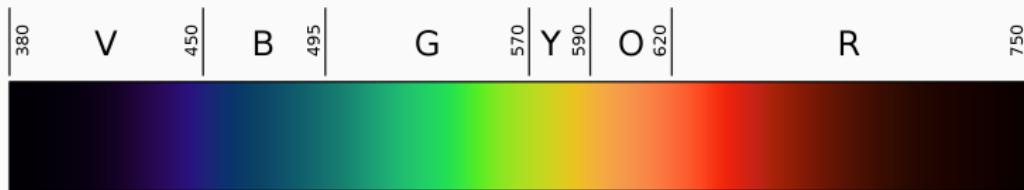
Esa energía se denomina **radiancia** y se nota como  $L(\lambda, \mathbf{p}, \mathbf{v})$ .

La radiancia determina el tono de color y el brillo con el que observamos el punto **p** cuando lo vemos desde la dirección **v**.

# Brillo y color de la radiancia

Desde un punto **p** en una dirección **v** pueden emitirse o reflejarse una gran cantidad de fotones con longitudes de onda distintas.

- ▶ La intensidad o brillo de la luz depende de la cantidad de fotones (es decir, de la radiancia total sumada en todas las longitudes de onda)
- ▶ El color con el que percibimos la luz depende de las distribución de las longitudes de onda de los fotones en el espectro visible.



(figura obtenida de: [http://en.wikipedia.org/wiki/Visible\\_spectrum](http://en.wikipedia.org/wiki/Visible_spectrum))

# Percepción de radiación visible

El ojo es la parte del *sistema visual humano* (SVH) capaz de enviar señales eléctricas al cerebro que dependen de las características de la luz que incide sobre las neuronas de su cara interna (la retina)

- ▶ En cada neurona de la retina, y para cada longitud de onda  $\lambda$ , se recibe una radiancia  $L(\lambda)$  distinta.
- ▶ El ojo funciona de forma tal que *simplifica* esa gran cantidad de información y la reduce (en cada neurona) a tres valores reales positivos que forman una tupla  $(s, m, l)$  que depende de  $L$ , es decir, el ojo tiene asociada una función  $f$  tal que:

$$f(L) = (s, m, l)$$

- ▶ Esta simplificación es aprox. lineal, es decir si  $f(L) = (s, m, l)$  y  $f(L') = (s', m', l')$ , entonces:

$$f(aL + bL') = a(s, m, l) + b(s', m', l')$$

donde  $a, b$  son valores reales arbitrarios no negativos.

# Los primarios RGB.

Si  $x$  es un valor real ( $x > 0$ ), entonces:

- ▶ la señal  $(x, 0, 0)$  enviada desde el ojo se interpreta o percibe en el cerebro (SVH) como de color rojo.
- ▶ la señal  $(0, x, 0)$  se percibe de color verde.
- ▶ la señal  $(0, 0, x)$  se percibe de color azul.

Como consecuencia, supongamos que tenemos tres distribuciones de radiancia  $L_r, L_g$  y  $L_b$  tales que:

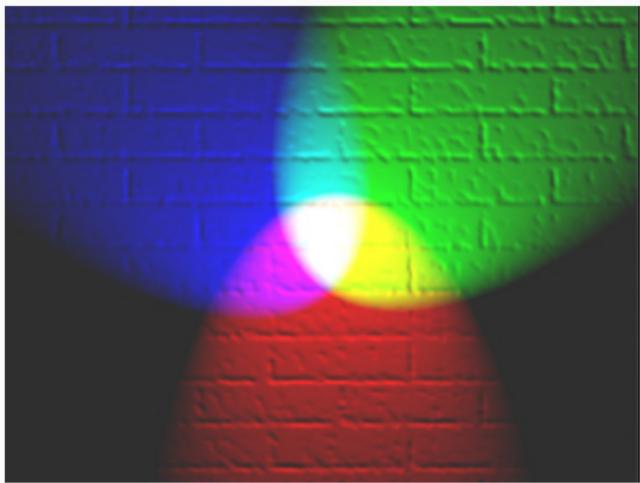
$$f(L_r) \approx (1, 0, 0) \quad f(L_g) \approx (0, 1, 0) \quad f(L_b) \approx (0, 0, 1) \quad (1)$$

A una terna de distribuciones  $L_r, L_g$  y  $L_b$  que cumplen lo anterior se le denomina una terna de **primarios RGB**, ya que son percibidos como rojo, verde y azul, respectivamente.

## Mezcla aditiva de primarios

Podemos usar una *mezcla aditiva* (suma ponderada) de tres primarios RGB para producir una señal arbitraria  $(r, g, b)$  en el ojo, ya que se cumple:

$$f(rL_r + gL_g + bL_b) \approx (r, g, b)$$



(imagen obtenida de: [http://en.wikipedia.org/wiki/RGB\\_color\\_model](http://en.wikipedia.org/wiki/RGB_color_model))

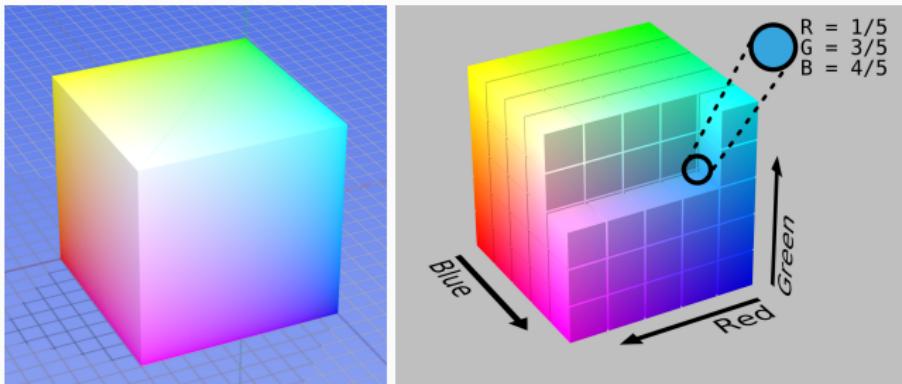
## Reproducción de ternas RGB

Un dispositivo de salida de color (monitor, impresora, proyector) tiene asociados tres primarios RGB (las distribuciones obtenidas cuando se muestra el rojo, verde y azul a máxima potencia en el dispositivo)

- ▶ Como consecuencia, cualquier color reproducible en un dispositivo se puede representar por una terna  $(r, g, b)$ , con  $0 \leq r, g, b \leq 1$ .
- ▶ El valor 0 indica que el correspondiente primario no aparece.
- ▶ El valor 1 representa la máxima potencia del dispositivo para cada primario.
- ▶ Una misma terna  $(r, g, b)$  produce tonos de color ligeramente distintos en dispositivos distintos.
- ▶ Una misma terna  $(r, g, b)$  niveles de brillo que pueden variar mucho entre dispositivos.

# El espacio RGB

Al conjunto de todas las ternas RGB con componentes entre 0 y 1 se le llama **espacio de color RGB**, y se puede visualizar como un cubo 3D con colores asociados a cada punto del mismo.



(obtenidas de: [http://en.wikipedia.org/wiki/RGB\\_color\\_model](http://en.wikipedia.org/wiki/RGB_color_model))

El espacio RGB no es el único esquema para representar computacionalmente los colores, pero sí el más usado hoy en día.

El color que se obtiene con una terna RGB en un dispositivo de salida depende de los primarios RGB que se usen en dicho dispositivo y del brillo máximo que pueda alcanzar:



(imagen obtenida de: sitio web de CBC news)

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.2.

Emisión y reflexión de la radiación..

## Fuentes de luz y reflectores:

La radiación electromagnética visible se genera en las **fuentes de luz**, por procesos físicos diversos que convierten otras formas de energía en energía radiante. Hay de dos tipos:

- ▶ Fuentes naturales: Sol o estrellas, fuego, objetos incandescentes, órganos de algunos animales, etc...
- ▶ Fuentes artificiales (luminarias): filamentos incandescentes, tubos fluorescentes, LEDs, etc...

Los fotones creados en las luminarias interactúan con los átomos de la materia, que absorben su energía y después pueden radiar de nuevo una parte de ella, proceso conocido como **reflexión**:

- ▶ parte de la energía recibida se convierte en calor
- ▶ parte de la energía recibida se convierte en radiación reflejada
- ▶ la radiación reflejada puede reflejarse de nuevo varias veces

## Modelo de la reflexión local en un punto

La radiancia  $L(\lambda, \mathbf{p}, \mathbf{v})$  se puede escribir como suma de:

- ▶ la **radiancia emitida** desde  $\mathbf{p}$  en la dirección  $\mathbf{v}$  (0 si  $\mathbf{p}$  no está en una fuente de luz), que llamamos  $L_{em}(\lambda, \mathbf{p}, \mathbf{v})$
- ▶ la **radiancia reflejada**, suma, para cada dirección  $\mathbf{u}_i$  del producto de:

$L_{in}(\lambda, \mathbf{p}, \mathbf{u}_i) \equiv$  radiancia incidente sobre  $\mathbf{p}$  desde  $\mathbf{u}_i$

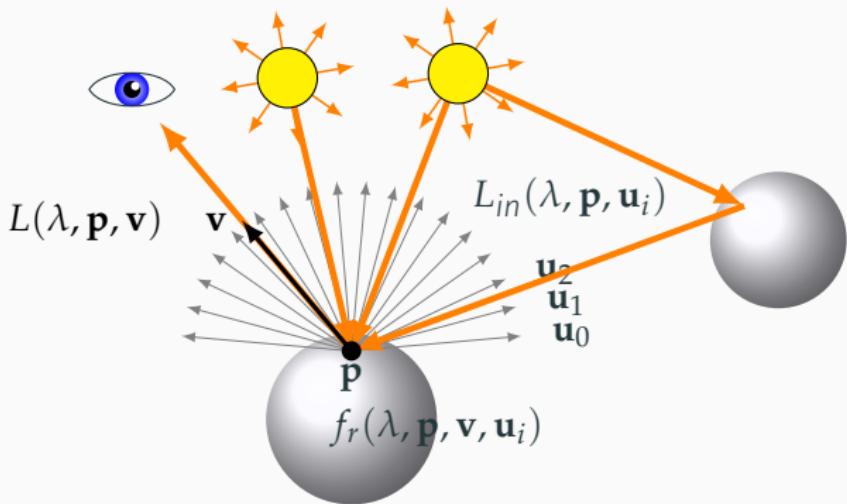
$f_r(\lambda, \mathbf{p}, \mathbf{v}, \mathbf{u}_i) \equiv$  fracción de radiancia que se refleja desde  $\mathbf{p}$  en la dirección  $\mathbf{v}$ , respecto del total incidente sobre  $\mathbf{p}$  proveniente de la dirección  $\mathbf{u}_i$  (con l.o.  $\lambda$ )

es decir:

$$L(\lambda, \mathbf{p}, \mathbf{v}) = L_{em}(\lambda, \mathbf{p}, \mathbf{v}) + \sum_i L_{in}(\lambda, \mathbf{p}, \mathbf{u}_i) f_r(\lambda, \mathbf{p}, \mathbf{v}, \mathbf{u}_i)$$

# Reflexión local en un punto

Hay muchas trayectorias de fotones que no acaban siendo detectadas por el observador (la mayoría), además las que sí llegan pueden hacerlo por muchos caminos distintos:



Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.3.

Modelos computacionales simplificados..

# Simplificaciones para modelos básicos

La ecuación anterior es complicada (larga) de calcular. Por tanto en OpenGL básico se hacen varias simplificaciones:

1. Las fuentes de luz son puntuales o unidireccionales, no extensas, y hay un número finito de ellas.
2. No se considera la luz incidente que no provenga directamente de las fuentes de luz (se usa una radiancia *ambiente* constante para suplir la iluminación indirecta).
3. Los objetos o polígonos son totalmente opacos (no hay transparencias ni mat. translúcidos).
4. No se consideran sombras arrojadas (las fuentes son visibles desde cualquier cara delantera respecto de ellas).
5. El espacio entre los objetos no dispersa la luz (la radiancia se conserva en el espacio entre los objetos).
6. En lugar de considerar todas las longitudes de onda  $\lambda$  posibles, usamos el modelo RGB.

# Efecto de las simplificaciones.

Aquí se observa una escena con iluminación compleja (izquierda) y simplificada (derecha)

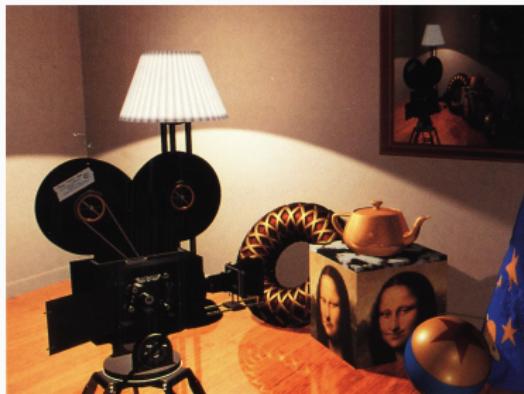


Imagen obtenida de: Computer Graphics: Principles and Practice in C (2nd Edition) Foley, van Dam, Feiner, Hughes.

# Modelo simplificado

El modelo que hemos visto antes se simplifica:

- ▶ La iluminación indirecta se reduce a un término ambiente  $L_{am}$  que no depende de  $\mathbf{v}$ .
- ▶ De todas las direcciones  $\mathbf{u}_i$ , solo es necesario considerar las que apuntan hacia una fuente de luz.
- ▶ Todas las fuentes de luz son visibles desde un punto.
- ▶ Los valores de radiancia ( $L, L_{em}, L_{in}, L_{am}$ ) son tuplas  $(r, g, b)$  (no acotadas)
- ▶ Los valores de reflectividad ( $f_r$ ) son tuplas  $(r, g, b)$  (entre 0 y 1)

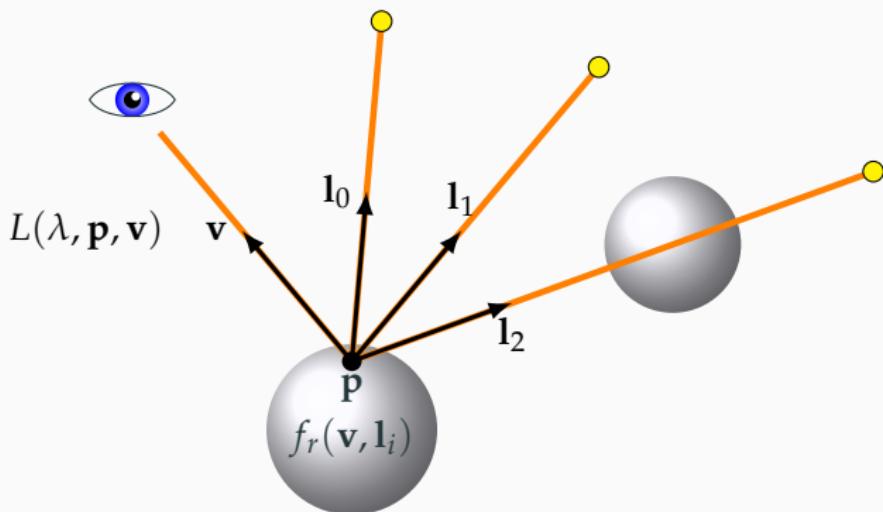
Por tanto:

$$L(\mathbf{p}, \mathbf{v}) = L_{em}(\mathbf{p}) + L_{am}(\mathbf{p}) + \sum_{i=0}^{n-1} L_{in}(\mathbf{p}, \mathbf{l}_i) f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) \quad (2)$$

donde:  $n \equiv$  número de fuentes de luz,  $\mathbf{l}_i \equiv$  vector que apunta desde  $\mathbf{p}$  en la dirección de la  $i$ -ésima fuente de luz.

## Modelo simplificado (figura)

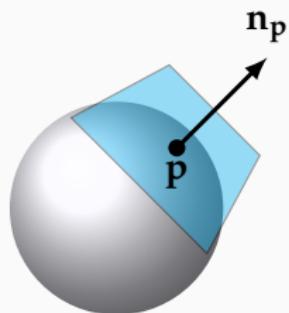
Ahora solo consideramos trayectorias desde las luminarias hacia  $\mathbf{p}$ , las luminarias se cuentan aunque la trayectoria esté bloqueada (no hay sombras arrojadas)



# El vector normal

La iluminación (la función  $f_r$  en la eq.2) depende la orientación de la superficie en el punto  $\mathbf{p}$ . Esta orientación esta caracterizada por el **vector normal  $\mathbf{n}_p$**  asociado a dicho punto:

- ▶  $\mathbf{n}_p$  es un vector, de longitud unidad, que depende de  $\mathbf{p}$ .
- ▶ idealmente es perpendicular al plano tangente a la superficie en el punto  $\mathbf{p}$  (en azul en la fig.)
- ▶ en modelos de fronteras, puede calcularse de varias formas (depende del *método de sombreado*, que veremos más adelante).
- ▶ constituye un parámetro de  $f_r$



# Tipos y atributos de las fuentes de luz

En el modelo de escena se puede incluir un conjunto de  $n$  fuentes de luz (numeradas de 0 a  $n - 1$ ), cada una de ellas puede ser de dos tipos:

- ▶ Fuentes de luz **posicionales**: ocupan un punto del espacio  $\mathbf{q}_i$ . Dado un punto  $\mathbf{p}$ , el vector unitario que apunta hacia la fuente de luz desde  $\mathbf{p}$  se calcula como:

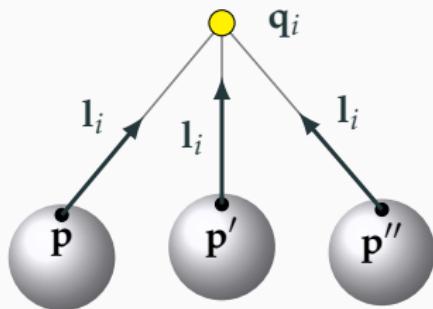
$$\mathbf{l}_i = \frac{\mathbf{q}_i - \mathbf{p}}{\|\mathbf{q}_i - \mathbf{p}\|}$$

- ▶ Fuentes de luz **direccionales**: están en un punto a distancia infinita, por tanto hay un vector  $\mathbf{l}_i$  que apunta a la fuente y que es el mismo para cualquier punto  $\mathbf{p}$  donde se quiera evaluar el MIL

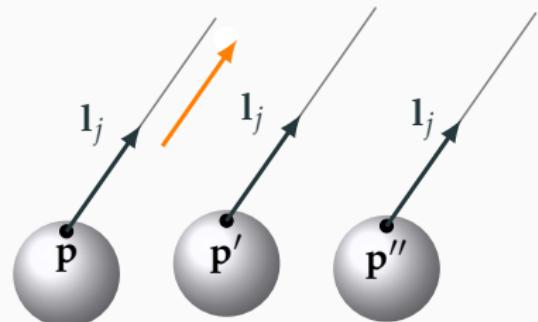
Además de esto, cada fuente de luz emite una radiancia  $S_i = (r, g, b)$  (en general no acotada).

# Posición o dirección de las luminarias

Fuente posicional ( $i$ )



Fuente direccional ( $j$ )



- ▶ Posicional: la dirección  $\mathbf{l}_i$  es distinta para cada punto  $p$  considerado. Es necesario recalcularla cada vez que se evalua el MIL.
- ▶ Direccional: La dirección  $\mathbf{l}_j$  es igual para todos los puntos  $p$  considerados. Es una constante.

## Radiancia incidente y tipos de reflexión.

En la ecuación 2 los términos que aparecen pueden reescribirse en términos de los atributos de las fuentes de luz y el material

- ▶ El término  $L_{in}(\mathbf{p}, \mathbf{l}_i)$  se hace igual a  $S_i$  (no tenemos en cuenta la distancia a la que está la fuente de luz)
- ▶ El término  $f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$  se descompone en tres sumandos o componentes
  - ▶ Luz indirecta reflejada, o término **ambiental**:  $f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$ .
  - ▶ Luz reflejada de forma **difusa**:  $f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$ .
  - ▶ Luz reflejada de forma **pseudo-especular**:  $f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$ .

La ecuación 2 queda como sigue:

$$L(\mathbf{p}, \mathbf{v}) = L_{em}(\mathbf{p}) + L_{am}(\mathbf{p}) + \sum_{i=0}^{n-1} S_i (f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)) \quad (3)$$

# Emisión y componente ambiental global

Los dos primeros sumandos de la ecuación 3 son:

- ▶ Radiancia emitida  $L_{em}(\mathbf{p})$ , que se puede hacer igual a una tupla RGB  $M_E(\mathbf{p})$  que depende del punto  $\mathbf{p}$  (es decir, del material, y que puede variar en función del polígono u objeto al que pertenece  $\mathbf{p}$ ) y que llamamos **emisividad del material**.
- ▶ El término ambiente  $L_{am}(\mathbf{p})$  se hace igual a una térrna RGB (que notamos como  $A_G(\mathbf{p})$ ) y que llamamos **luz ambiente global**.

la ecuación 3 se reescribe por tanto como:

$$L(\mathbf{p}, \mathbf{v}) = M_E(\mathbf{p}) + A_G(\mathbf{p}) + \sum_{i=0}^{n-1} S_i [f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)] \quad (4)$$

## Componente ambiental específica de cada punto

Cada objeto puede reflejar más o menos cantidad de iluminación indirecta proveniente de la  $i$ -ésima fuente de luz.

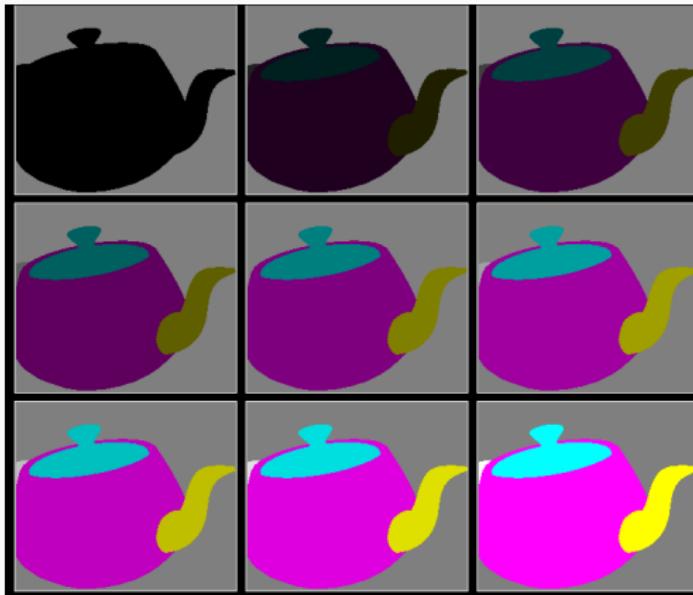
Para poder hacer esto, se asocia a cada polígono o punto una reflectividad difusa del material, una terna RGB que notamos como  $M_A(\mathbf{p})$  (con valores entre 0 y 1 pues se trata de una reflectividad), y hacemos:

$$f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_A(\mathbf{p}) \quad (5)$$

nótese que este valor no depende de la posición, distancia u orientación de la fuente de luz, ni del vector  $\mathbf{v}$ , y por tanto da lugar a colores planos en los objetos.

## Reflectividad ambiental del objeto:

En este caso, la reflectividad ambiental  $M_A(\mathbf{p})$  depende de en que parte de la tetera este el punto  $\mathbf{p}$ :



## Componente difusa: expresión.

La **componente difusa** modela como se refleja la luz en los objetos mate o difusos:

- ▶ La componente **sí depende** de la posición u orientación de la fuente de luz (es distinta según como esté orientada la fuente respecto de la superficie en  $\mathbf{p}$ , es decir, depende de  $\mathbf{n_p}$  y  $\mathbf{l}_i$ ),
- ▶ **no depende** de la dirección  $\mathbf{v}$  en la que miramos  $\mathbf{p}$  (el punto  $\mathbf{p}$  se ve de un color igual desde cualquier dirección que lo veamos).
- ▶ La fracción de luz reflejada es igual a una terna de reflectividades  $M_D(\mathbf{p})$  que puede hacerse depender de  $\mathbf{p}$

La expresión concreta de  $f_{rd}$  es esta:

$$f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_D(\mathbf{p}) \max(0, \mathbf{n_p} \cdot \mathbf{l}_i) \quad (6)$$

## Orientación de la superficie:

La orientación de la superficie respecto de la fuente de luz viene determinada por el valor  $\alpha$ , que es el ángulo que hay entre los vectores  $\mathbf{n}_p$  y  $\mathbf{l}_i$  (el valor  $\mathbf{n}_p \cdot \mathbf{l}_i$  es igual al coseno de  $\alpha$ ). Se pueden distinguir dos casos:

- ▶ Si  $\alpha > 90^\circ$ , entonces:
  - ▶  $\cos(\alpha)$  es negativo.
  - ▶ la superficie, en  $\mathbf{p}$ , está orientada de espaldas a la fuente de luz.
  - ▶ la contribución de esa fuente debe ser 0.
- ▶ Si  $0^\circ \leq \alpha \leq 90^\circ$ , entonces:
  - ▶ la superficie, en  $\mathbf{p}$ , está orientada de cara a la fuente de luz.
  - ▶  $\cos(\alpha)$  estará entre 0 y 1 (entre  $\cos(90^\circ)$  y  $\cos(0^\circ)$ ).
  - ▶ se puede demostrar que el valor  $\cos(\alpha)$  es proporcional a la densidad de fotones por unidad de área que inciden en el entorno de  $\mathbf{p}$ , provenientes de la  $i$ -ésima fuente de luz.

## Orientación de la superficie (2)

Aquí se ilustran tres posibles casos:

$$90^\circ < \alpha$$

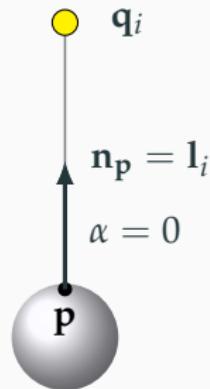
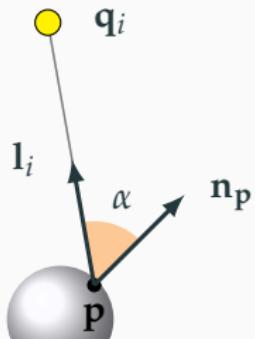
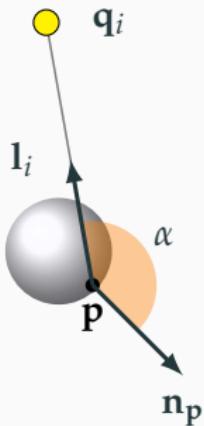
$$0 > \cos(\alpha)$$

$$0^\circ < \alpha < 90^\circ$$

$$1 > \cos(\alpha) > 0$$

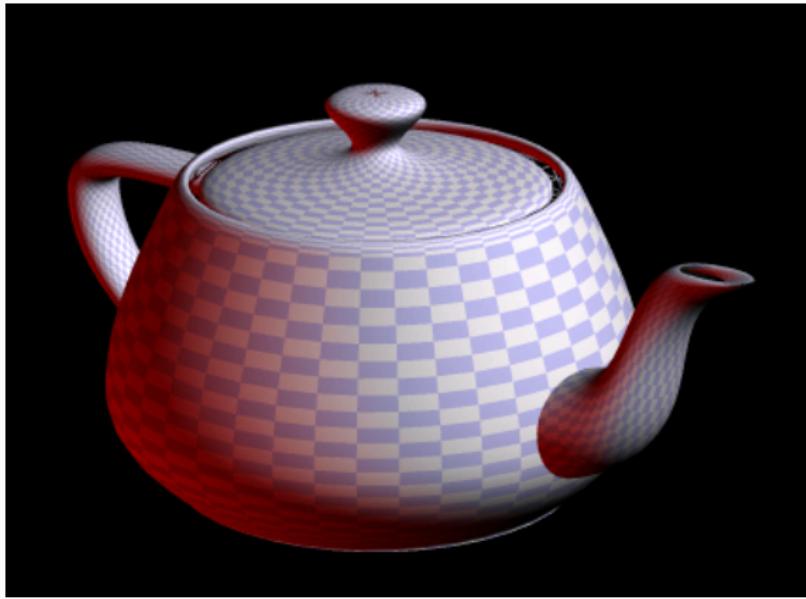
$$\alpha = 0^\circ$$

$$\cos(\alpha) = 1$$



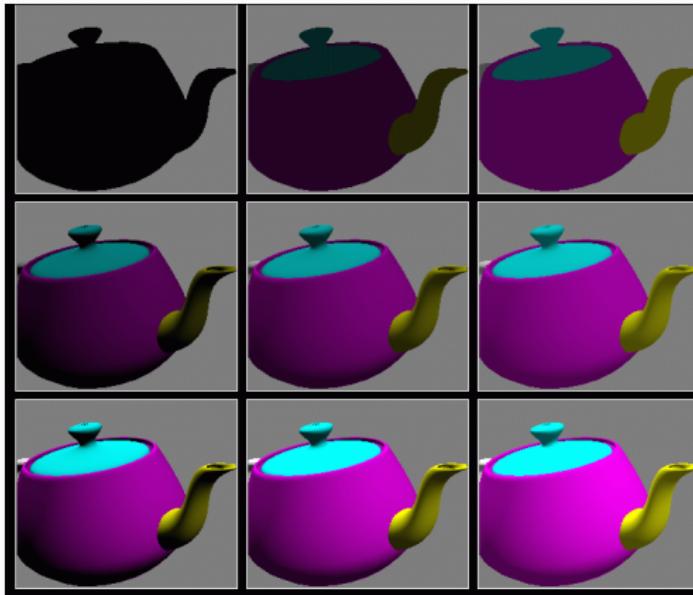
# Material difuso

Ejemplo con dos fuentes de luz direccionales,  
 $M_A(\mathbf{p}) = M_S(\mathbf{p}) = (0, 0, 0)$  (solo hay componente difusa)



# Material difuso+ambiental

Aquí  $M_A$  crece de izquierda a derecha, y  $M_D$  de arriba abajo,  $M_S = 0$ :



## Comp. pseudo-especular: modelo de Phong

La componente **pseudo-especular** modela como se refleja la luz en los objetos brillantes, en los cuales dichas zonas brillantes dependen de la posición del observador:

- ▶ La componente **sí depende** de la posición u orientación de la fuente de luz (es distinta según como esté orientada la fuente respecto de la superficie en  $\mathbf{p}$ ),
- ▶ **también depende** de la dirección en la que miramos  $\mathbf{p}$  (el punto  $\mathbf{p}$  se ve de un color diferente según la dirección en la que lo veamos).
- ▶ La fracción de luz reflejada es proporcional a una terna de reflectividades  $M_S(\mathbf{p})$  que puede hacerse depender de  $\mathbf{p}$

La expresión ideada por *Bui Tuong Phong*, y conocida como **modelo de Phong** es esta:

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_S(\mathbf{p}) d_i [\max(0, \mathbf{r}_i \cdot \mathbf{v})]^e \quad (7)$$

# Parámetros del modelo de Phong

En la expresión anterior:

$\mathbf{r}_i$   $\equiv$  **vector reflejado**, depende tanto de  $\mathbf{l}_i$  como de  $\mathbf{n_p}$ , y está en el plano formado por ambos, con  $\mathbf{n_p}$  como bisectriz de ellos, se obtiene como:

$$\mathbf{r}_i = 2(\mathbf{l}_i \cdot \mathbf{n_p})\mathbf{n_p} - \mathbf{l}_i$$

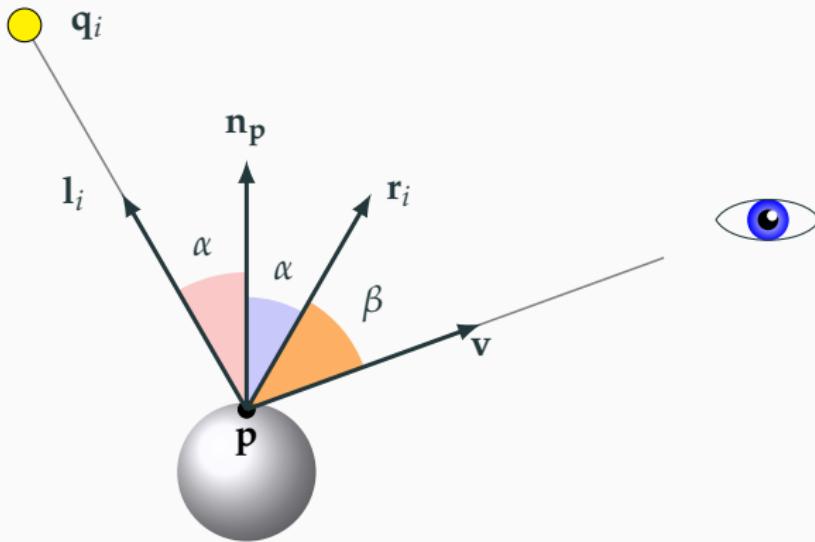
el vector  $\mathbf{r}_i$  indica la dirección desde  $\mathbf{p}$  en la cual la  $i$ -ésima fuente de luz produce el máximo brillo.

$e$   $\equiv$  **exponente de brillo**, un valor real positivo que permite variar el tamaño de las zonas brillantes (a mayor valor, menor tamaño y más pulida o especular).

$d_i$   $\equiv$  vale 1 si  $\mathbf{n_p} \cdot \mathbf{l}_i > 0$  (fuente de cara a la superficie), y 0 en otro caso (de espaldas)

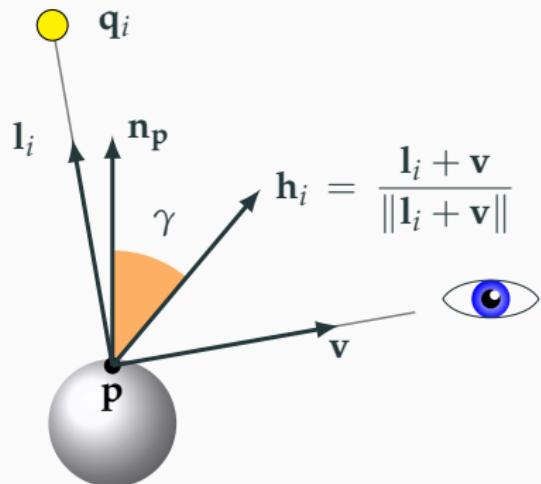
## Vectores del modelo de Phong

El valor  $\mathbf{r}_i \cdot \mathbf{v}$  es el coseno del ángulo  $\beta$  que hay entre la dirección de máximo brillo  $\mathbf{r}_i$  y la dirección  $\mathbf{v}$  hacia el observador. Cuando  $\mathbf{r}_i = \mathbf{v}$  entonces  $\beta = 0^\circ$ ,  $\cos(\beta) = 1$ , y el brillo es máximo:



## Comp. pseudo-especular: modelo de Blinn-Phong

Una alternativa al modelo anterior consiste en usar el vector *halfway*  $\mathbf{h}_i$  (bisectriz de  $\mathbf{l}_i$  y  $\mathbf{v}$ , normalizado). Ahora el brillo es proporcional al coseno del ángulo  $\gamma$  entre  $\mathbf{h}_i$  y  $\mathbf{n}_p$  (máximo cuando coinciden)



La expresión del **Modelo de Blinn-Phong** es la siguiente:

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_S(\mathbf{p}) d_i [ \mathbf{n}_p \cdot \mathbf{h}_i ]^e$$

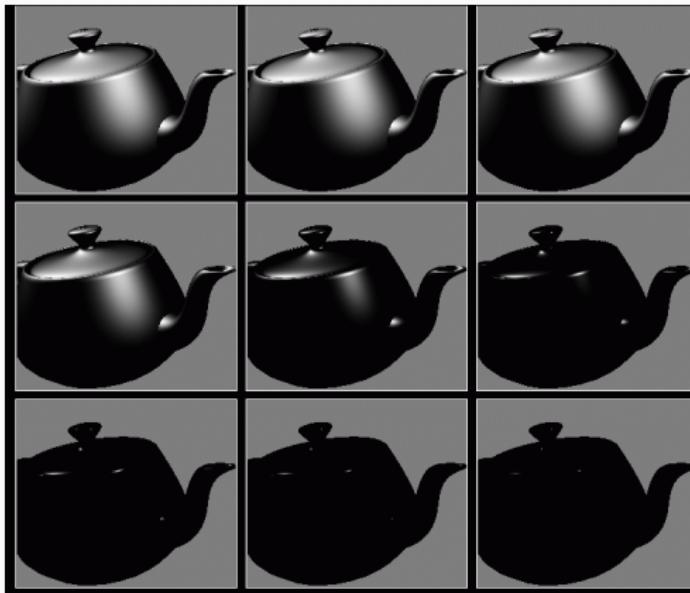
## Ejemplo de material pseudo-especular

Aquí  $M_A(\mathbf{p}) = M_D(\mathbf{p}) = 0$ , y  $e = 5.0$ :



# Efecto del exponente de brillo

Aquí crece de izquierda a derecha y de arriba abajo:



## La expresión del modelo completo:

Sustituyendo la expresiones de  $f_{ra}$ ,  $f_{rd}$  y  $f_{rs}$  en la ecuación 4 obtenemos el modelos simplificado completo:

$$L(\mathbf{p}, \mathbf{v}) = M_E(\mathbf{p}) + A_G(\mathbf{p}) + \sum_{i=0}^{n-1} S_i C_i$$

donde:

$$\begin{aligned} C_i &= M_A(\mathbf{p}) \\ &\quad + M_D(\mathbf{p}) \max(0, \mathbf{n} \cdot \mathbf{l}_i) \\ &\quad + M_S(\mathbf{p}) d_i [\max(0, \mathbf{r}_i \cdot \mathbf{v})]^e \end{aligned}$$

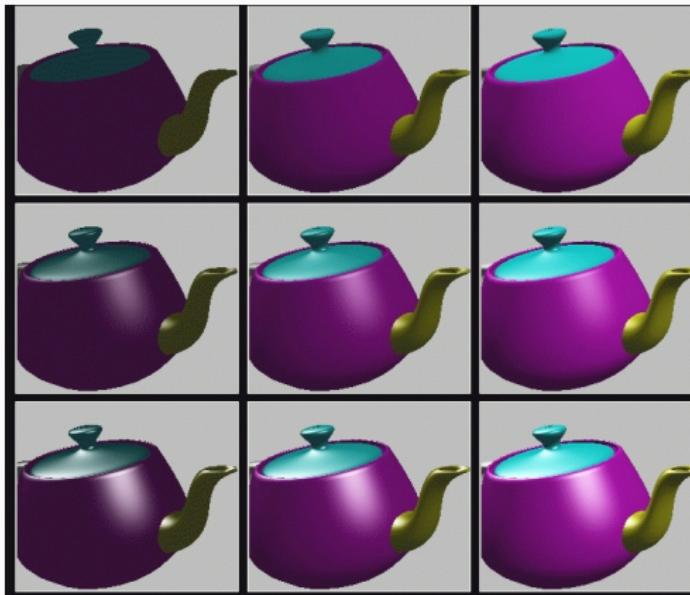
# Ejemplo de material combinado

Combinación ambiental, más difusa, más pseudo.especular:



# Combinaciones material difuso + pseudo especular

$M_D$  crece de izquierda a derecha y  $M_S$  de arriba abajo:



## Problema 3.9.

Suponemos que visualizamos una esfera de radio unidad centrada en el origen. Se ilumina con una fuente de luz puntual situada en el punto  $\mathbf{p} = (0, 2, 0)$ . El observador está situado en  $\mathbf{o} = (2, 0, 0)$ . En estas condiciones:

- ▶ Describe razonadamente en que punto de la superficie de la esfera el brillo será máximo si el material es puramente difuso ( $M_D = (1, 1, 1)$ , y el resto de reflectividades a 0) ¿es ese punto visible para el observador ?
- ▶ Repite el razonamiento anterior asumiendo ahora que el material es puramente pseudo-especular ( $M_S = (1, 1, 1)$ , resto a cero). Indica si dicho punto es visible para el observador.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.4.

Texturas.

## Detalles a pequeña escala

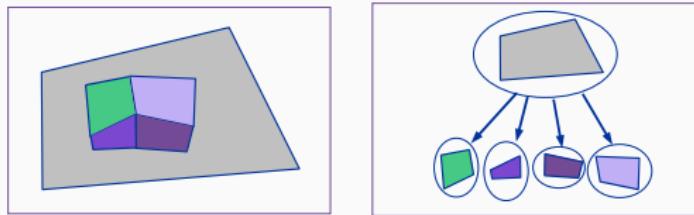
Los objetos reales presentan a veces detalles a pequeña escalar, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia:



- ▶ Estas variaciones se pueden modelar como funciones que asignan a cada punto de la superficie de un objeto un valor diferente para algunos parámetros del MIL.
- ▶ Lo más usual es variar las reflectividades difusa y ambiente, pero se hace también con la normal (rugosidades), o a veces otros parámetros.

## Implementación de detalles: polígonos de detalle

Para reproducir detalles a pequeña escala se pueden usar **polígonos de detalle**, son polígonos pequeños adicionales a los que definen la geometría de la escena, pero con materiales y/o orientación distintos entre ellos:



La desventaja de esta opción es su enorme complejidad en espacio (necesario para almacenar muchos polígonos pequeños) y tiempo (empleado en su visualización).

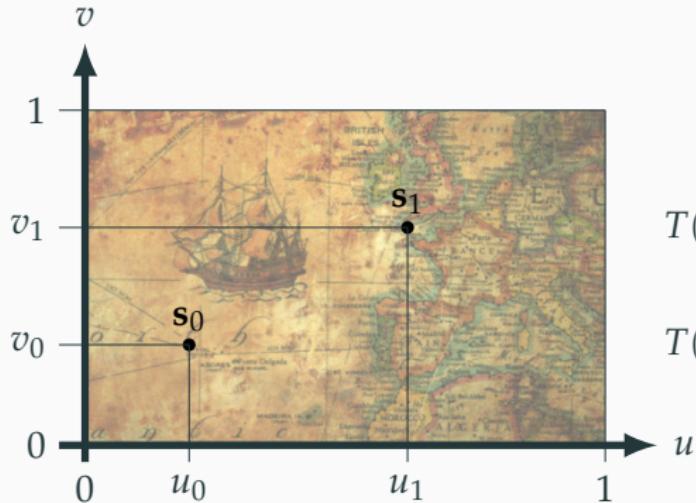
# Texturas

Una opción mejor (mucho más eficiente) es usar imágenes para representar las funciones antes citadas. A estas imágenes se les llama (en el contexto de la Informática Gráfica) **texturas**:

- ▶ Una textura se puede interpretar como una función  $T$  que asocia a cada punto  $\mathbf{s}$  de un dominio  $D$  (usualmente  $[0, 1] \times [0, 1]$ ) un valor para un parámetro del MIL (típicamente  $M_D$  y  $M_A$ ). La función  $T$  determina como varía el parámetro en el espacio.
- ▶ La función  $T$  puede estar representada en memoria como una matriz de pixels RGB (una imagen discretizada), a cuyos pixels se les llama **texels** (*texture elements*). A esta imagen se le llama **imagen de textura**.
- ▶ La función  $T$  puede tambien representarse como un subprograma que calcula los valores a partir de  $\mathbf{s}$  (que se le pasa como parámetro). A este tipo de texturas le llamamos **texturas procedurales**.

# La textura como una función

En este ejemplo vemos una imagen de textura (bidimensional). El dominio  $D$  es  $[0, 1]^2$ . Cada punto del dominio es una par  $\mathbf{s} = (u, v)$ . Los valores  $T(\mathbf{s}) = T(u, v)$  son ternas RGB.



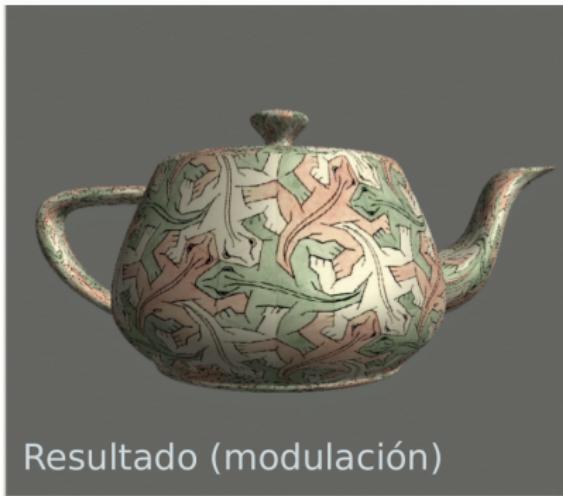
$$T(\mathbf{s}_1) = T(u_1, v_1) = (r_1, g_1, b_1)$$

$$T(\mathbf{s}_0) = T(u_0, v_0) = (r_0, g_0, b_0)$$

# Aplicación de texturas

Vemos varias formas de asignar colores a puntos del objeto:

- ▶ evaluación del MIL con reflectividades blancas (izq. abajo)
- ▶ uso directo de colores de la textura (izq. arriba)
- ▶ evaluación del MIL con reflectividades obtenidas de la textura (der.)



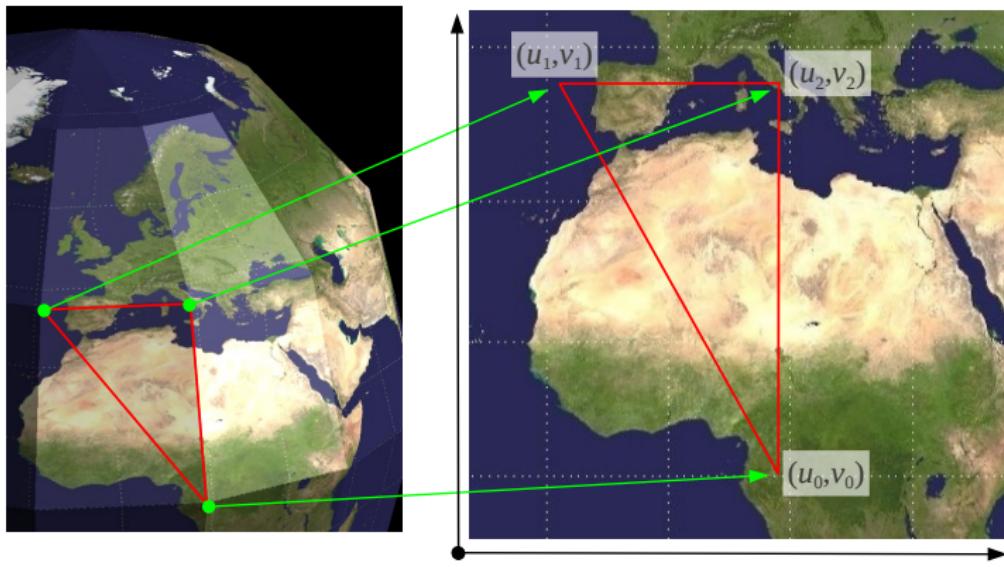
## Coordenadas de textura

Para poder aplicar una textura a la superficie de un objeto, es necesario hacer corresponder cada punto  $\mathbf{p} = (x, y, z)$  de su superficie con un punto  $(u, v)$  del dominio de la textura:

- ▶ Debe existir una función  $f$  tal que  $(u, v) = f(x, y, z)$
- ▶ Si  $(u, v) = f(x, y, z)$  entonces decimos que  $(u, v)$  son las **coordenadas de textura** del punto  $\mathbf{p} = (x, y, z)$ .
- ▶ Normalmente  $f$  se descompone en dos componentes  $f_u, f_v$ , de forma que  $u = f_u(x, y, z)$  y  $v = f_v(x, y, z)$
- ▶ La función  $f$  puede implementarse usando una tabla de coordenadas de textura de los vértices, o bien calcularse proceduralmente con un subprograma.

# Ejemplo de coordenadas de textura.

Vemos como a cada vértice de un triángulo del modelo se le asignan sus coordenadas de textura  $(u_i, v_i)$  (donde  $i$  es el índice del vértice en la tabla de vértices).



# Asignación explícita o procedural

La asignación de coord. de text. se puede hacer usando:

- ▶ **Asignación explícita a vértices:** las coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada al cauce gráfico, en forma de un vector o tabla de coordenadas de textura de vértices  $(v_0, u_0), (v_1, u_1), \dots (u_{n-1}, v_{n-1})$ .
  - ▶ se puede hacer manualmente en objetos sencillos, o bien
  - ▶ de forma asistida usando software para CAD.
- ▶ hace necesario realizar una interpolación de coords. de text. en el interior de los polígonos.
- ▶ **Asignación procedural:**  $f$  se implementa como un subprograma `CoordText(p)` que calcula las coordenadas de textura (para un punto  $p$  devuelve el par  $(u, v) = f(p)$  con las coords. de textura de  $p$ ).

## Ejemplo de asignación explícita.

Esto es posible en objetos sencillos como este cubo construido con triángulos. En este ejemplo se busca una asignación que de cc.t. que sea continua en las aristas:

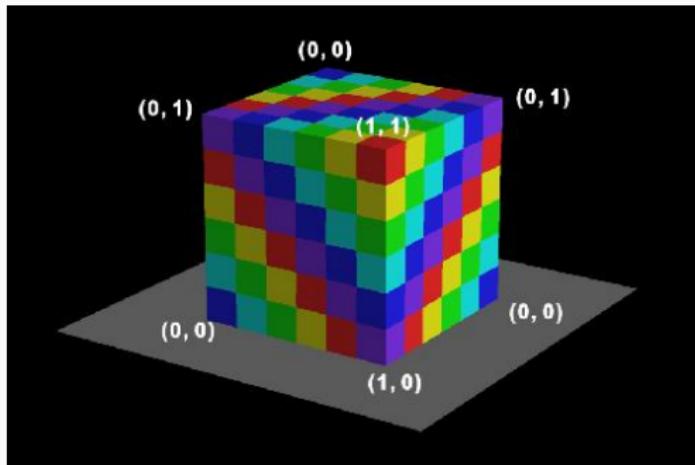
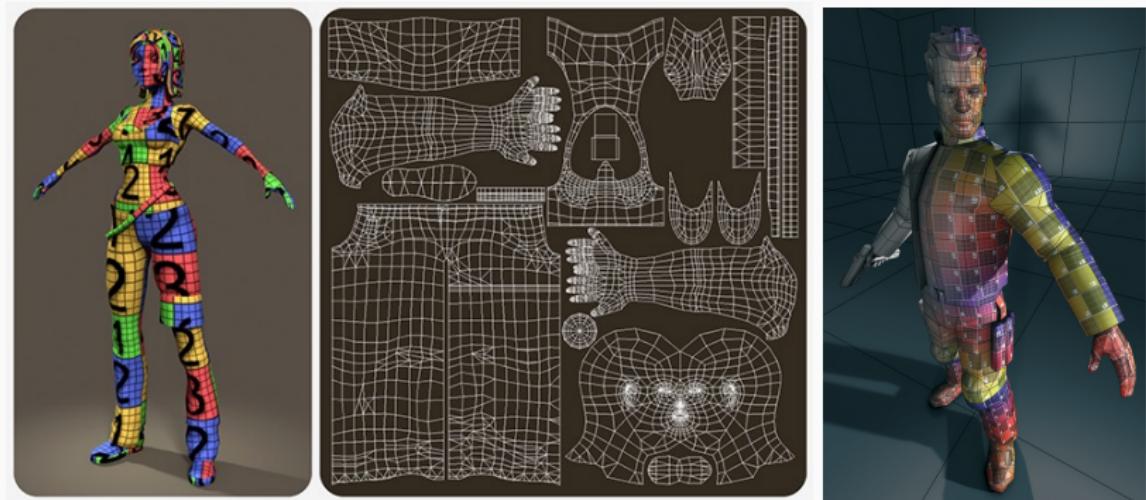


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

# Ejemplo de uso de herramientas CAD.

En objetos complejos es necesario el uso de herramientas CAD:



Imágenes de Sean Dixon (izquierda, centro) y Mayan Escalante (derecha).

# Tipos de asignación procedural

Hay dos opciones:

- ▶ **Asignación procedural a vértices:** se invoca `CoordText( $v_i$ )` para calcular las coordenadas de textura en cada vértice  $v_i$ , y las coordenadas obtenidas se almacenan y después se interpolan linealmente en el interior de los polígonos de la malla.
  - ▶ Funciona de forma totalmente correcta (exacta) solo cuando  $f$  es lineal, en otro caso es una aproximación lineal a trozos.
- ▶ **Asignación procedural a puntos:** se invoca `CoordText( $p$ )` cada vez que sea necesario evaluar el MIL en un punto de la superficie  $p$ .
  - ▶ Permite exactitud incluso aunque  $f$  sea no lineal.
  - ▶ En OpenGL, esto requiere programación del cauce gráfico, invocando a `CoordText` en cada pixel desde el *fragment shader*.

## Funciones para asignación procedural:

Los tipos de funciones  $f$  más frecuentes son:

- ▶ **Funciones lineales** de la posición (proyección en un plano): el punto  $\mathbf{p} = (x, y, z)$  se proyecta sobre un plano y se expresa como un par  $(x', y')$  de coordenadas en dicho plano, que se interpretan como coordenadas de textura.
- ▶ **Coordenadas paramétricas**: se pueden usar si la malla aproxima una superficie paramétrica (p.ej. la tetera, hecha de superficies paramétricas tipo B-spline). Se usa asignación procedural a vértices. Se trata de funciones no lineales de la posición.

## Otras opciones para asignación procedural

Otras opciones (no lineales) son estas dos:

- ▶ **Coordenadas polares** (proyección en una esfera): el punto  $\mathbf{p}$  se expresa en coordenadas polares como una terna  $(\alpha, \beta, r)$ , los valores  $u$  y  $v$  se obtienen de  $\alpha$  y  $\beta$ .
- ▶ **Coordenadas cilíndricas** (proyección en un cilindro): el punto  $\mathbf{p}$  se expresa en coordenadas cilíndricas como una terna  $(\alpha, y, r)$ , los valores  $u$  y  $v$  se obtienen de  $\alpha$  e  $y$ .

Es muy complicado usarlas con asignación a vértices ( $\alpha$  puede pasar de 360 a 0 en un triángulo, la textura se vería mal), y por tanto requieren usar asignación procedural a puntos (invocar `CoordText` desde los *fragment shaders*).

## Funciones lineales (proyección).

En este caso el punto  $\mathbf{p} = (x, y, z)$  se proyecta en un plano, y se usan las coordenadas del punto proyectado (en el sistema de referencia del plano), como coordenadas de textura.

El plano estará definido por un punto por el que pasa ( $\mathbf{q}$ ) y por dos vectores libres ( $\mathbf{e}_u$  y  $\mathbf{e}_v$ , de longitud unidad y perpendiculares entre sí). En estas condiciones:

$$u = f_u(\mathbf{p}) = (\mathbf{p} - \mathbf{q}) \cdot \mathbf{e}_u \quad v = f_v(\mathbf{p}) = (\mathbf{p} - \mathbf{q}) \cdot \mathbf{e}_v$$

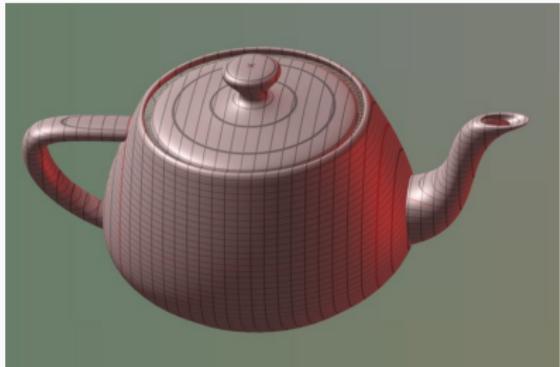
como casos particulares, y a modo de ejemplo, podemos hacer  $\mathbf{q}$  igual al origen  $(0, 0, 0)$ ,  $\mathbf{e}_u = \mathbf{x} = (1, 0, 0)$  y  $\mathbf{e}_v = \mathbf{y} = (0, 1, 0)$ , y en este caso es una proyección paralela el eje Z, sobre el plano XY (descarta la Z)

$$u = x = \mathbf{p} \cdot \mathbf{x} = (x, y, z) \cdot (1, 0, 0)$$

$$v = y = \mathbf{p} \cdot \mathbf{y} = (x, y, z) \cdot (0, 1, 0)$$

## Ejemplo de proyección paralela a Z.

Las coordenadas de **p** que se usan en las funciones lineales pueden ser las coordenadas de objeto (izquierda) o bien o las coordenadas de mundo (derecha). Aquí vemos un ejemplo de una proyección paralela al eje Z:



este método funciona mejor (menor deformación) cuando la normal es aproximadamente paralela a la dirección de proyección (parte frontal en el ejemplo de la izquierda).

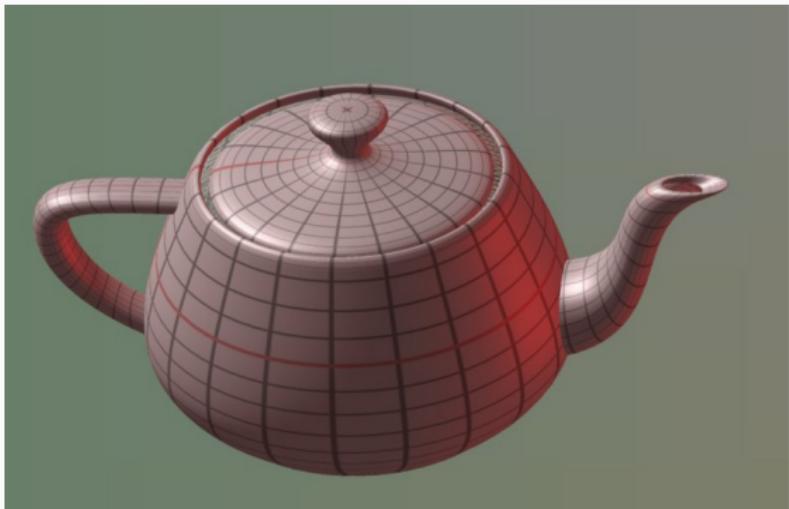
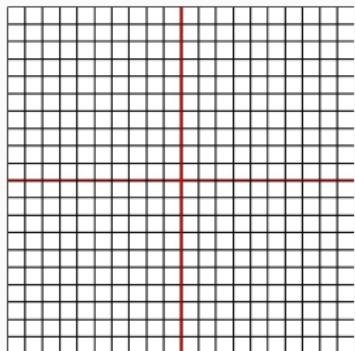
## Coordenadas paramétricas.

Una **superficie paramétrica** es una variedad plana de dos dimensiones (que puede ser abierta o cerrada), para la cual existe una función  **$\mathbf{g}$**  (con dominio en  $[0, 1] \times [0, 1]$ ) tal que, si  **$\mathbf{p}$**  es un punto de la superficie, entonces existen  $(s, t)$  tales que  $\mathbf{p} = \mathbf{g}(s, t)$ :

- ▶ En este caso, al par  $(s, t)$  se le llaman **coordenadas paramétricas** del punto  **$\mathbf{p}$** , y a la función  **$\mathbf{g}$**  se le llama **función de parametrización** de la superficie.
- ▶ Usando la capacidad de evaluar  **$\mathbf{g}$** , podemos construir una malla que aproxima cualquier superficie paramétrica. La posición  **$\mathbf{p}_i$**  del  $i$ -ésimo vértice se obtiene como  **$\mathbf{g}(s_i, t_i)$** , donde los  $(s_i, t_i)$  forman una rejilla en  $[0, 1] \times [0, 1]$ .
- ▶ En estas condiciones, podemos hacer  $(u, v) = f(\mathbf{p}) = (s, t)$ , es decir, podemos usar las coordenadas paramétricas como coordenadas de textura.

# Ejemplo de coordenadas paramétricas

Vemos un ejemplo de textura (izq.) y su aplicación a la tetera (der.)



Esta imagen se ha generado asignando explicitamente en el programa a cada vértice sus coordenadas de textura, usando para ello sus coordenadas paramétricas.

# Coordenadas esféricicas

Se basa en usar las coordenadas polares (longitud, latitud y radio) del punto **p**:

- ▶ Equivale a una proyección radial en una esfera.
- ▶ Las coordenadas  $(\alpha, \beta, r)$  se obtienen a partir de las coordenadas cartesianas  $(x, y, z)$  (normalmente coordenadas de objeto, con el origen en un punto central de dicho objeto).

Hacemos:

$$\alpha = \text{atan2}(z, x) \quad \beta = \text{atan2}\left(y, \sqrt{x^2 + z^2}\right)$$

- ▶ Se obtiene  $\alpha$  en el rango  $[-\pi, \pi]$  y  $\beta$  en el rango  $[-\pi/2, \pi/2]$ . Por tanto, podemos calcular  $u$  y  $v$  como sigue:

$$u = \frac{1}{2} + \frac{\alpha}{2\pi} \quad v = \frac{1}{2} + \frac{\beta}{\pi}$$

el valor de  $r$  no se usa y por tanto no es necesario calcularlo.

# Ejemplo de coordenadas esféricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

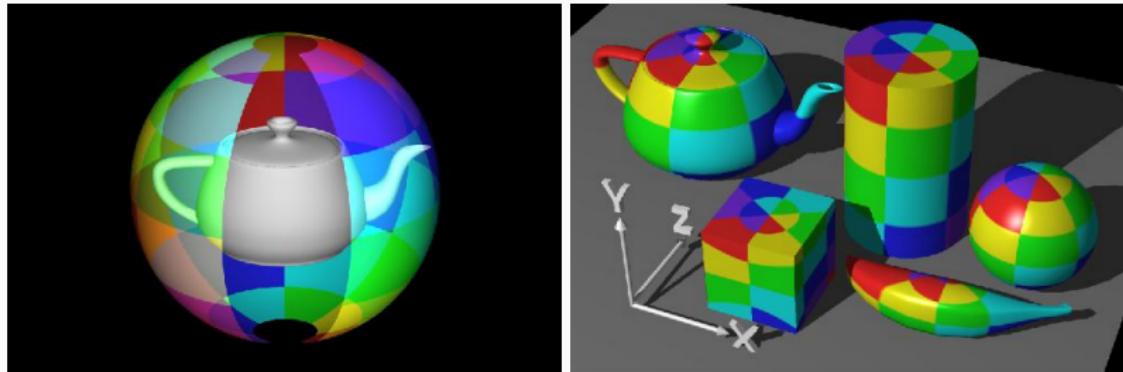


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

# Coordenadas cilíndricas

Se usan las coordenadas polares (ángulo y altura) del punto **p**:

- ▶ Equivale a una proyección radial en un cilindro (cuyo eje es usualmente un eje vertical central al objeto).
- ▶ Las coordenadas  $(\alpha, h, r)$  se obtienen a partir de las coordenadas cartesianas  $(x, y, z)$  (también con origen en el centro del objeto). Hacemos:

$$\alpha = \text{atan2}(z, x) \quad h = y$$

- ▶ El valor de  $\alpha$  está en el rango  $[-\pi, \pi]$  y  $h$  en el rango  $[y_{min}, y_{max}]$  (el rango en Y del objeto). Por tanto, podemos calcular  $u$  y  $v$  como:

$$u = \frac{1}{2} + \frac{\alpha}{2\pi} \quad v = \frac{y - y_{min}}{y_{max} - y_{min}}$$

tampoco el valor de  $r$  se usa ahora y por tanto no es necesario calcularlo.

# Ejemplo de coordenadas cilíndricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

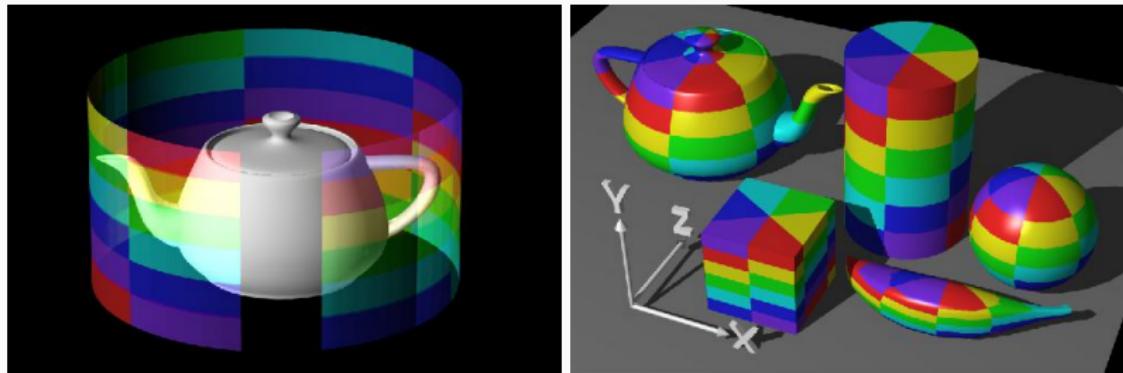


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

## Consulta de texels en texturas de imagen.

En una textura de imagen con  $n_x$  columnas de texels y  $n_y$  filas, podemos interpretar que cada texel tiene asociada un pequeño rectángulo contenido en  $[0, 1]^2$ . El texel en la columna  $i$ , fila  $j$  tendrá un área con centro en el punto  $(c_i, d_j)$

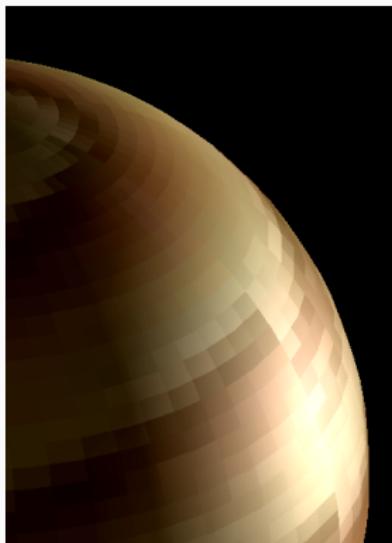
La consulta del color de la textura en un punto  $(u, v)$  puede hacerse de dos formas:

- ▶ **más cercano:** usar el color del texel cuyo centro sea más cercano a la posición  $(u, v)$ , es equivalente a seleccionar el texel cuya área contiene a  $(u, v)$ .
- ▶ **interpolación** realizar un interpolación (bilineal) entre los colores de los cuatro texels con centros más cercanos al punto  $(u, v)$ .

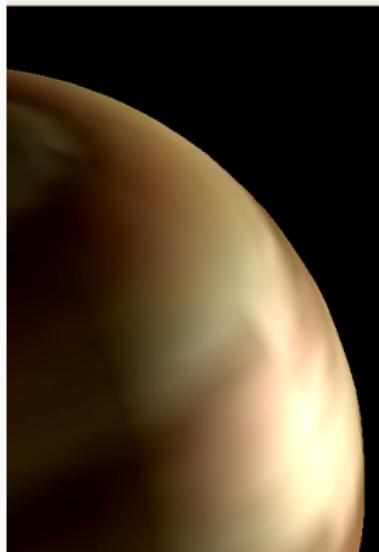
las diferencias entre ambos métodos son visibles cuando la proyección en la ventana de un texel ocupa muchos pixels.

# Interpolación bilineal

Aquí vemos una textura de baja resolución, vista de cerca, que se visualiza usando los dos métodos:



más cercano

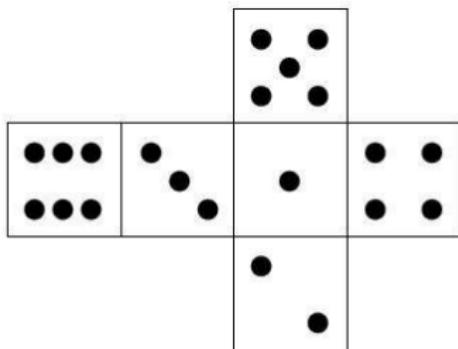


interpolación bilineal

# Problemas: coordenadas de textura (1/3)

## Problema 3.10.

Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar un textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3. La imagen aparece aquí:



(continua en la siguiente transparencia)

## Problema 3.10. (continuación)

Responde a estas cuestiones:

- (a) Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
- (b) Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de textura, y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en  $(1/2, 1/2, 1/2)$ . Dibuja un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.

## Problemas: coordenadas de textura (2/3)

### Problema 3.11.

Considera de nuevo el cubo y la textura del problema anterior. Ahora supón que queremos visualizar con OpenGL el cubo usando sombreado de Gouraud o de Phong, para lo cual debemos de asignar normales a los vértices. Responde a estas cuestiones

- ▶ Describe razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que has escrito en el problema anterior, o es necesario usar otra tabla distinta.
- ▶ Si has respondido que no es posible usar las mismas tablas, escribe la nueva tabla de vértices, la nueva tabla de coordenadas de textura. Asimismo, escribe como sería la tabla de normales.

## Problema 3.12.

Considera un cubo (de nuevo de lado unidad, y con centro en  $(1/2, 1/2, 1/2)$ ) que se quiere visualizar con una textura a partir de una única imagen (cuadrada) que se replicará en las 6 caras de dicho cubo. Asume que no se va a usar iluminación (no es necesario calcular la tabla de normales). Escribe ahora la tabla de coordenadas de vértices y la tabla de coordenadas de textura.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.5.

Métodos de sombreado para rasterización..

## Alternativas

En el algoritmo de Z-buffer, la evaluación del MIL puede hacerse en tres puntos distintos del cauce gráfico:

- ▶ **Sombreado plano:** (*flat shading*) una vez por cada polígono que forma el modelo, asignando el resultado (una terna RGB única) a todos los pixels donde se proyecta el polígono.
- ▶ **Sombreado de vértices:** (*smooth shading* o *Gouroud shading*) una vez por vértice, cada color RGB obtenido se usa para interpolar los colores de los pixels en cada polígono.
- ▶ **Sombreado de pixel:** (*pixel shading* o *Phong shading*) una vez por cada pixel donde se proyecta el polígono

## Sombreado plano

Este método de sombreado es muy eficiente en tiempo si el modelo es sencillo ( $\equiv$  el número de polígonos es pequeño en comparación con el de pixels).

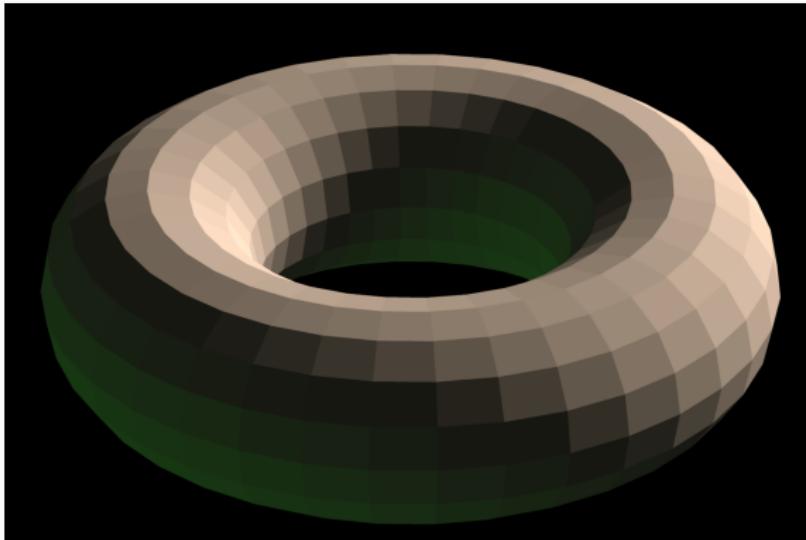
- ▶ Se debe seleccionar un punto cualquiera  $\mathbf{p}$  de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- ▶ Se usa la normal al polígono  $\mathbf{n}_p$ .
- ▶ Se calcula el vector al observador  $\mathbf{v}$  en  $\mathbf{p}$ .

Las desventajas son:

- ▶ Puede no ser deseable que se aprecien los polígonos del modelo.
- ▶ Produce discontinuidades en el brillo de los pixels en las aristas.
- ▶ No es realista si el tamaño del polígono es grande en comparación con la distancia que lo separa al observador, en proyección perspectiva y/o brillos pseudo-especulares.

## Resultados del sombreado plano.

Aquí vemos un objeto curvo aproximado con caras planas y visualizado con sombreado plano (MIL difuso).



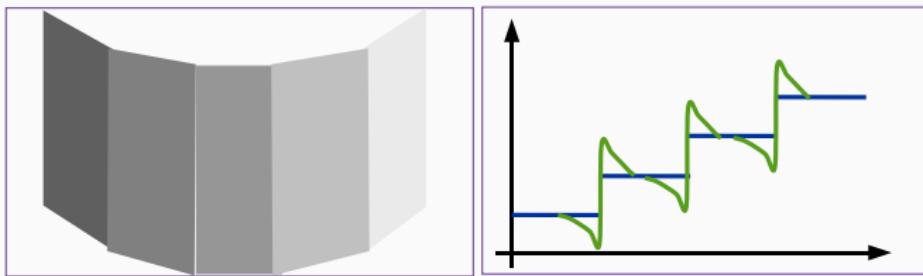
# Resultados del sombreado plano

Aquí se observa la tetera, con sombreado plano, a distintas resoluciones. En este caso el MIL tiene una componente pseudo-especular no nula.



# Bandas Mach

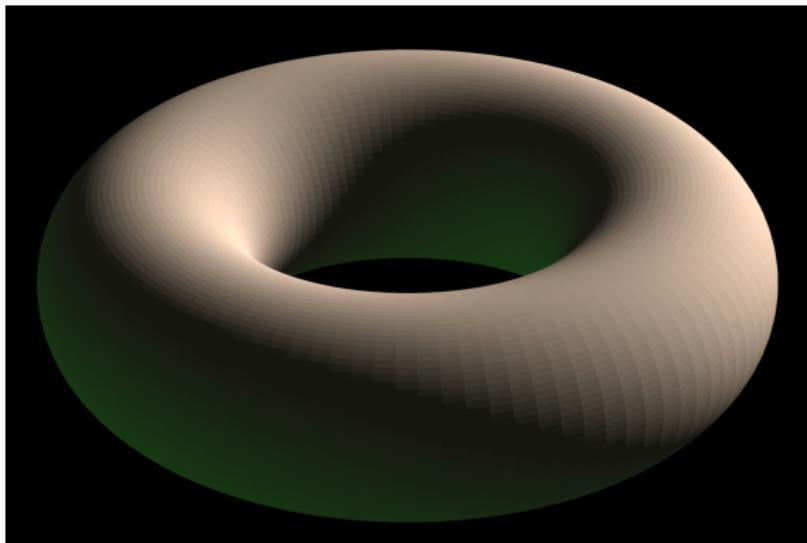
La **Bandas Mach** son una ilusión visual producida por la *inhibición lateral de las neuronas de la retina*, que es un mecanismo desarrollado evolutivamente para resaltar el contraste en aristas entre colores planos:



si no se quiere modelar un objeto formado realmente por caras planas, esta forma de visualizar produce resultados pobres. En algunos casos (objetos hechos de caras planas, iluminación puramente difusa) puede ser muy eficiente y realista.

## Ejemplo de bandas Mach

En este objeto las bandas Mach son fácilmente apreciables:



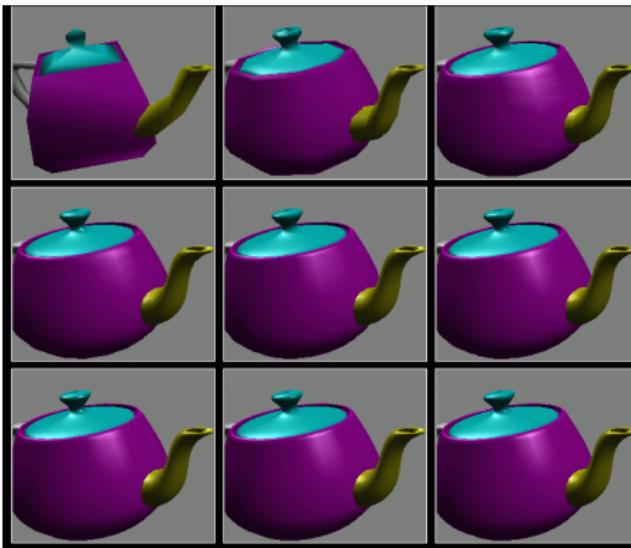
## Sombreado en los vértices

En esta modalidad (*vertex shading*), el MIL se evalua una vez en cada vértice del modelo.

- ▶ La posición  $\mathbf{p}$  coincide con la posición del vértice.
- ▶ Si la malla de polígonos aproxima un objeto curvo, la normal  $\mathbf{n}_p$  puede calcularse como el promedio de las normales de los polígonos adyacentes al vértice.
- ▶ La evaluación del MIL produce un color único para cada vértice
- ▶ Los valores en los vértices se usan como valores extremos para interpolar los colores de los pixels donde se proyecta el polígono
- ▶ La eficiencia en tiempo es parecida al sombreado plano.
- ▶ Los resultados son muchas veces más realistas que con sombreado plano.
- ▶ Pueden persistir problemas de bandas Mach y poco realismo.

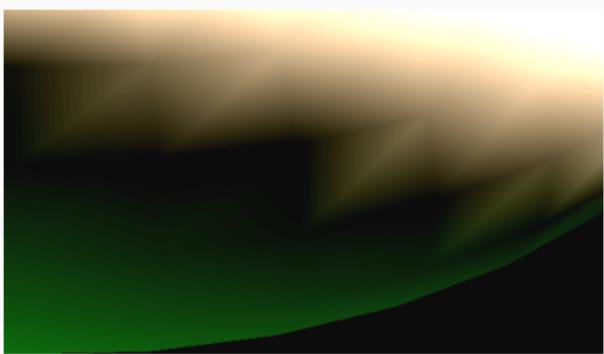
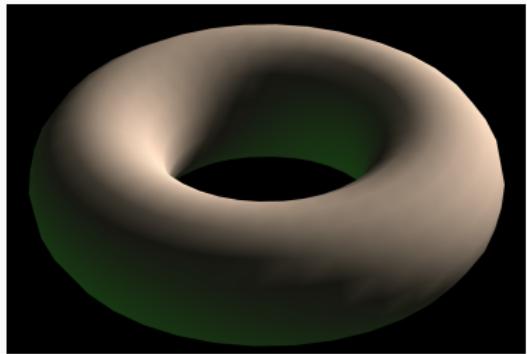
## Pérdida de zonas brillantes

Los resultados mejoran, pero puede haber problemas de pérdida de zonas brillantes (componente pseudo-especular), en modelos con pocos polígonos que aproximan objetos curvos:



# Discontinuidades en la derivada

A veces pueden aparecer problemas parecidos a las bandas Mach, en este caso por exageración en la retina de las discontinuidades de primer orden (cambios bruscos en la pendiente de la iluminación)



(al izquierda aparece una ampliación, con el brillo y contraste aumentado)

## Sombreado en los píxeles.

En esta modalidad (*pixel shading*), el MIL se evalua en cada pixel del viewport en el que se proyecta un polígono

- ▶ Requiere interpolar las normales asociadas a los vértices.
- ▶ Es computacionalmente más costoso que los anteriores, pero no cuando el número de polígonos visibles es del orden del número de pixels del viewport (o superior).
- ▶ Produce resultados de más calidad, hay muchos menos defectos por discontinuidades.
- ▶ Los resultados son más realistas incluso con pocos polígonos.
- ▶ La evaluación del MIL es la última etapa del cauce.

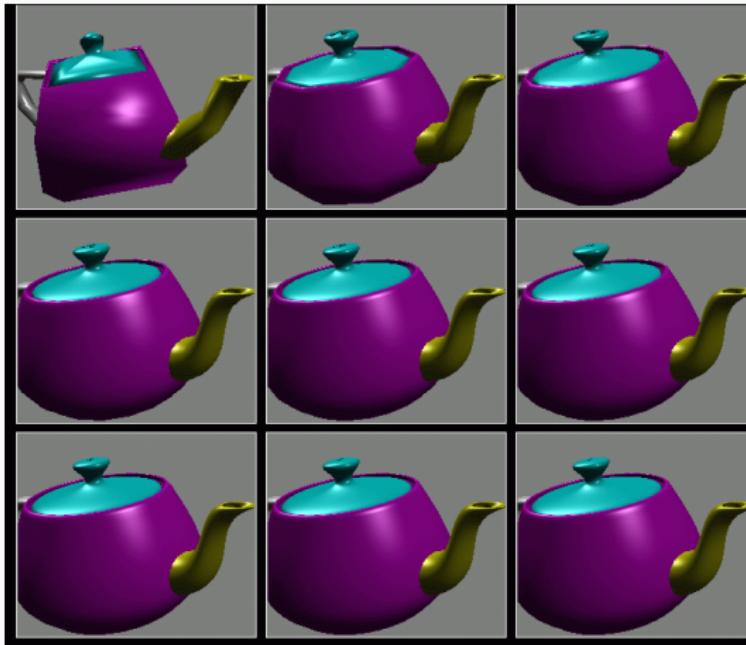
# Ejemplo de sombreado

Esta imagen se ha creado con sombreado en los pixels:



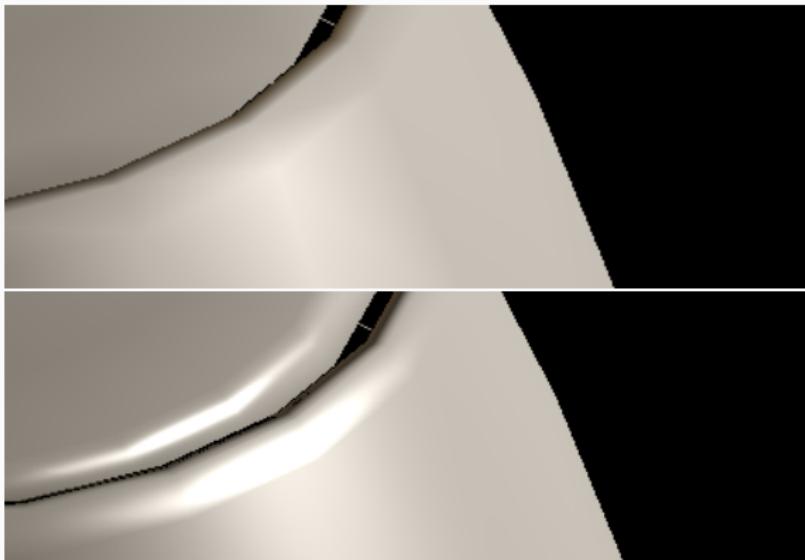
# Reproducción de zonas de brillo

Con este sombreado se reproducen los brillos incluso a baja resolución:



# Comparación de sombreado vértices y píxeles

Sombreado de vértices (arriba) y de píxeles (abajo), en iguales condiciones de iluminación, observador y atributos material:



## Sección 3. Iluminación y texturas con el cauce fijo.

- 3.1. Introducción: activación, iluminación versus colores prefijados.
- 3.2. Definición de fuentes de luz: tipos y atributos.
- 3.3. Definición de atributos de materiales.
- 3.4. Configuración de sombreado en el cauce fijo.
- 3.5. Carga de texturas en el sistema gráfico.
- 3.6. Configuración de texturas en el cauce fijo.
- 3.7. Implementación de la clase **CauceFijo**

# Introducción

La librería OpenGL como parte de la funcionalidad fija (pre-programada), incluye una implementación de un modelo de iluminación similar al ya introducido

- ▶ Es necesario usar la orden **glEnable/glDisable** para activar o desactivar la funcionalidad de iluminación:

```
glEnable(GL_LIGHTING); // activa evaluacion del MIL  
glDisable(GL_LIGHTING); // desactiva evaluacion del MIL
```

- ▶ Esta funcionalidad está por defecto desactivada en el estado inicial de OpenGL.

**Nota** toda esta funcionalidad fue declarada **obsoleta (deprecated)** en la versión 3.0 de la API de OpenGL (Septiembre, 2008), y fue **eliminada** de la versión 3.1 en adelante (Mayo, 2009) (se usa programación del cauce gráfico).

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.1.

Introducción: activación, iluminación versus colores prefijados..

# Cálculo de iluminación

El comportamiento de OpenGL depende de si la evaluación del MIL está activada o no lo está:

- ▶ Con la iluminación desactivada, el color de las primitivas dibujadas depende de una terna RGB del estado interno, que se modifica con **glColor**.
- ▶ Con la iluminación activada el MIL se evalua usando unos parámetros incluidos en el estado de OpenGL, y el color resultante obtenido se usa en lugar del especificado con **glColor**.

(activar o desactivar el MIL es independiente del *modo de sombreado* activo en cada momento).

## Parámetros del MIL.

En su estado interno, OpenGL mantiene un conjunto de ternas RGB que constituyen los parámetros más importantes del MIL. Son los siguientes:

$M_E$   $\equiv$  emisividad del material.

$A_G$   $\equiv$  término ambiente global.

$M_A, M_D, M_S$   $\equiv$  reflectividad difusa, ambiente y pseudo-especular del material.

$e$   $\equiv$  exponente de la componente pseudo-especular.

$S_{iA}, S_{iD}, S_{iS}$   $\equiv$  luminosidad de la  $i$ -ésima fuente de luz (para las componentes ambiental, difusa o pseudo-especular).

$\mathbf{q}_i, \mathbf{l}_i$   $\equiv$  posición o dirección de la  $i$ -ésima fuente de luz (en EC).

estos parámetros se pueden modificar en cualquier momento, afectando su nuevo valor a las evaluaciones del MIL posteriores.

## El modelo de la funcionalidad fija

Es muy similar al ya visto, excepto que en lugar de usar una sola terna  $S_i$  para el color de una fuente de luz, se usan tres de ellas,  $(S_{iA}, S_{iD}, S_{iS})$ , una por cada componente del modelo:

$$I = M_E + A_G + \sum_{i=0}^{n-1} C_i$$

donde:

$$C_i = S_{iA}M_A + S_{iD}M_D \max(0, \mathbf{n} \cdot \mathbf{l}_i) + S_{iS}M_S d_i [\mathbf{n_p} \cdot \mathbf{h}_i]^e$$

el resultado  $I = (r, g, b)$  es un color que se asigna al pixel. Si  $r, g$  o  $b$  tienen un valor superior a 1, dicho valor se trunca a 1.

# Evaluación del MIL

La evaluación se hace en coordenadas de ojo (relativa al sist. de ref. de la cámara), usando:

$\mathbf{p}$   $\equiv$  posición del vértice donde se evalua el MIL, en EC.

$\mathbf{v}$   $\equiv$  vector normalizado desde  $\mathbf{p}$  hacia el observador en EC. Si la proyección es perspectiva, se usa  $\mathbf{v} = -\mathbf{p}/\|\mathbf{p}\|$ , si es ortogonal se usa  $\mathbf{v} = (0, 0, 1)$ .

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.2.

Definición de fuentes de luz: tipos y atributos..

## Activación y desactivación.

Las implementaciones de OpenGL están obligadas a gestionar un número mínimo de 8 fuentes de luz.

- ▶ Cada una de ellas se referencia por un valor entero, el valor de las constantes **GL\_LIGHT0**, **GL\_LIGHT1**, ..., **GL\_LIGHT8** (tienen valores consecutivos).
- ▶ Cada una de estas fuentes de luz puede activarse y desactivarse de forma individual, con:

```
glEnable(GL_LIGHTi) ; // activa la i-esima fuente de luz  
glDisable(GL_LIGHTi) ; // desactiva la i-esima fuente de luz
```

- ▶ Solo las fuentes activas intervienen en el MIL (por defecto están todas desactivadas)

## Configuración de colores de una fuente.

Se hace con la función **glLightf** (en todos los casos, el primer parámetro identifica la fuente de luz cuyos atributos queremos modificar)

Los valores de  $S_{iA}, S_{iD}, S_{iS}$  se fijan con estas llamadas

```
const float
    caf[4] = { ra, ga, ba, 1.0 }, // color ambiental de la fuente
    cdf[4] = { rd, gd, bd, 1.0 }, // color difuso de la fuente
    csf[4] = { rs, gs, bs, 1.0 }; // color especular de la fuente

glLightfv( GL_LIGHTi, GL_AMBIENT, caf ) ; // hace  $S_{iA} := (r_a, g_a, b_a)$ 
glLightfv( GL_LIGHTi, GL_DIFFUSE, cdf ) ; // hace  $S_{iD} := (r_d, g_d, b_d)$ 
glLightfv( GL_LIGHTi, GL_SPECULAR, csf ) ; // hace  $S_{iS} := (r_s, g_s, b_s)$ 
```

# Configuración de posición/dirección de una fuente.

Los posición (en luces posicionales) o dirección (en direccionales) se especifica con una llamada a **glLightfv**, de esta forma:

```
// fuentes posicionales:  $\mathbf{p}_i = (p_x, p_y, p_z)$ 
const GLfloat posf[4] = { p_x, p_y, p_z, 1.0 } ;
glLightfv( GL_LIGHTi, GL_POSITION, posf );

// fuentes direccionales  $\mathbf{l}_i = (v_x, v_y, v_z)$ 
const GLfloat dirf[4] = { v_x, v_y, v_z, 0.0 } ;
glLightfv( GL_LIGHTi, GL_POSITION, dirf );
```

- ▶ El valor de w determina el tipo de fuente de luz.
- ▶ A la tupla  $(x, y, z, w)$  se le aplica la matriz *modelview M* activa en el momento de la llamada, y el resultado se almacena y se interpreta en coordenadas de cámara.

## Posición u orientación de la fuente.

La tupla  $(x, y, z, w)$  puede especificarse en varios marcos de coordenadas:

- ▶ Coordenadas de cámara: si se especifica cuando  $M = \text{Ide}$ .
- ▶ Coordenadas del mundo: si se especifica cuando  $M$  contiene la matriz de vista  $V$ .
- ▶ Coordenadas maestras (de algún objeto  $O$ ): si se especifica cuando  $M = VN$  (donde  $N$  es la matriz de modelado del objeto  $O$ )

En todos los casos se pueden usar (adicionalmente) transformaciones específicas para esto, situando dichas transformaciones, seguidas del **glLightfv**, entre **glPushMatrix** y **glPopMatrix**.

# Dirección en coordenadas polares

Por ejemplo, para establecer la dirección a una fuente de luz usando coordenadas polares (dos ángulos  $\alpha$  y  $\beta$  de longitud y latitud, respectivamente, en grados), podríamos hacer:

```
const float[4] ejeZ = { 0.0, 0.0, 1.0, 0.0 } ;
glMatrixMode( GL_MODELVIEW ) ;
glPushMatrix() ;

glLoadIdentity() ;      // hacer M = Ide
glMultMatrix( A ) ;   // A podría ser Ide, V o VN

// (3) rotación  $\alpha$  grados en torno a eje Y
glRotatef(  $\alpha$ , 0.0, 1.0, 0.0 ) ;
// (2) rotación  $\beta$  grados en torno al eje X-
glRotatef(  $\beta$ , -1.0, 0.0, 0.0 ) ;
// (1) hacer  $\mathbf{l}_i := (0,0,1)$  (paralela eje Z+)
glLightf( GL_LIGHTi, GL_POSITION, ejeZ ) ;

glPopMatrix()
```

## Observador local o en el infinito

OpenGL debe calcular el vector  $\mathbf{v}$  que va desde el punto  $\mathbf{p}$  hacia el observador (en EC), esto debe hacerse en función del tipo de proyección activo. La función **glLightModel** permite configurar este comportamiento:

- ▶ Si la proy. es ortogonal,  $\mathbf{v}$  debe ser  $(0,0,1)$  (el observador está en el infinito), es necesario hacer esta llamada:

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE );
```

- ▶ Si la proy. es perspectiva,  $\mathbf{v}$  debe ser  $-\mathbf{p}/\|\mathbf{p}\|$  (se dice que el observador es **local**, no está en el infinito) en este caso debemos de hacer:

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
```

## Normalización del vector normal.

Los vectores normales especificados con **glNormal** se transforman por la matriz *modelview* ( $M$ ) activa en el momento de la llamada, y se almacenan en coordenadas de cámara. Es necesario que OpenGL use normales de longitud unidad para evaluar el MIL. Para lograrlo hay tres opciones:

- ▶ Enviar normales de longitud unidad (solo válido si  $M$  no incluye cambios de escala ni cizallas).
- ▶ Enviar normales de longitud unidad, y habilitar **GL\_RESCALE\_NORMAL** (solo válido si  $M$  no incluye cizallas, aunque puede tener cambios de escala).

```
glEnable(GL_RESCALE_NORMAL); // deshabilitado por defecto
```

- ▶ Enviar normales de longitud arbitraria, y habilitar **GL\_NORMALIZE** (válido para cualquier  $M$ ) (preferible).

```
glEnable(GL_NORMALIZE); // deshabilitado por defecto
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.3.

Definición de atributos de materiales..

# Termino ambiente global y emisividad

El término ambiente global ( $A_G$ ) es una terna RGB que forma parte del estado de OpenGL y que se cambia con la función **glLightModel**, como sigue:

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
// hace  $A_G := (r,g,b)$ , inicialmente es (0.2,0.2,0.2)  
glLightModelf( GL_LIGHT_MODEL_AMBIENT, color ) ;
```

Las propiedades del material también forman parte del estado y se modifican con llamadas a la función **glMaterial**, para modificar la emisividad del material ( $M_E$ ), hacemos:

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
// hace  $M_E := (r,g,b)$ , inicialmente es (0,0,0)  
glMaterialf( GL_FRONT_AND_BACK, GL_EMISSION, color ) ;
```

## Colores del material

Además de la emisión, el resto de colores que definen el material ( $M_A, M_D, M_S$ ) y el exponente de brillo  $e$  también se cambian con **glMaterial**, como se indica aquí:

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
  
// hace  $M_A := (r,g,b)$ , inicialmente (0.2,0.2,0.2)  
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, color ) ;  
  
// hace  $M_D := (r,g,b)$ , inicialmente (0.8,0.8,0.8)  
glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, color ) ;  
  
// hace  $M_S := (r,g,b)$ , inicialmente (0,0,0)  
glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, color ) ;  
  
// hace  $e := v$ , inicialmente 0.0 (debe estar entre 0.0 y 128.0)  
glMaterialf( GL_FRONT_AND_BACK, GL_SHININESS, v ) ;
```

# Asociación de colores del material al color actual

**glColor** actualiza una terna RGB (la llamamos  $C$ ) en el estado de OpenGL. Con la función **glColorMaterial** podemos hacer que el valor de alguna de las reflectividades del material se haga igual a  $C$  cada vez que  $C$  cambie:

```
// asociar  $M_E$  (emisión) con  $C$  :  
glColorMaterial( GL_FRONT_AND_BACK, GL_EMISSION ) ;  
// asociar  $M_A$  (refl. ambiente) con  $C$  :  
glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT ) ;  
// asociar  $M_D$  (refl. difusa) con  $C$  :  
glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE ) ;  
// asociar  $M_S$  (refl. pseudo-especular) con  $C$  :  
glColorMaterial( GL_FRONT_AND_BACK, GL_SPECULAR ) ;  
// asociar  $M_A$  y  $M_D$  con  $C$  :  
glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE ) ;
```

Por defecto, esta funcionalidad está activada, y OpenGL asocia siempre  $M_A$  y  $M_D$  con  $C$ . Se puede activar o desactivar con:

```
 glEnable( GL_COLOR_MATERIAL );  
 glDisable( GL_COLOR_MATERIAL );
```

## Atributos de material de caras delanteras y traseras

El estado interno de OpenGL contiene dos juegos de reflectividades del material: uno para polígonos **delanteros** (*front-facing polygons*), y otro para polígonos **traseros** (*back-facing polygons*).

- ▶ Por defecto, se consideran polígonos delanteros aquellaos en cuya proyección los vértices aparecen en sentido anti-horario al recorrellos en el orden en el que se proporcionan a OpenGL con **glVertex**. El resto son traseras (este comportamiento es configurable).
- ▶ Todas las llamadas que permiten cambiar el material tienen un primer parámetro que permite discriminar sobre que juego de ternas RGB se está actuando. Los valores son:

```
GL_FRONT          // atrib. del material de caras delanteras  
GL_BACK          // atrib. del material de caras traseras  
GL_FRONT_AND_BACK // ambos juegos de atributos
```

## Ejemplo de material delantero/trasero

Aquí vemos un ejemplo de diferencias entre los atributos del material para las caras traseras y las delanteras, en un objeto no cerrado:

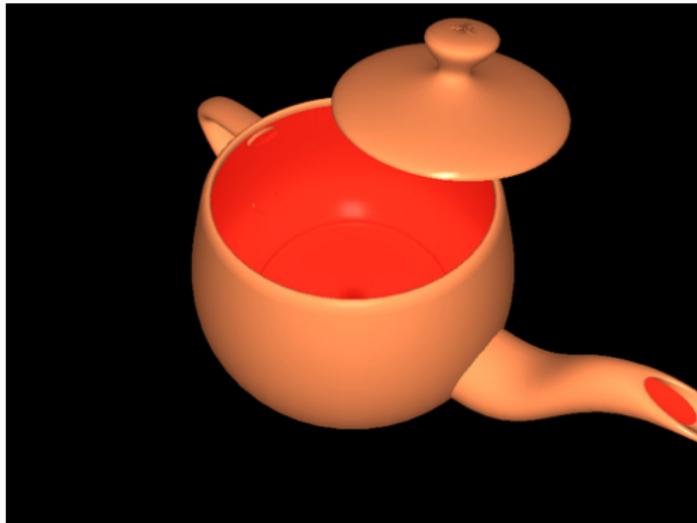


Imagen obtenida de:  editorial Packt

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.4.

Configuración de sombreado en el cauce fijo..

# Tipos de sombreado en OpenGL

La función **glShadeModel** permite seleccionar el método de sombreado activo en cada momento (afecta a todas las primitivas posteriores)

- ▶ **Sombreado plano** se usa el color del último vértice para todo el polígono:

```
glShadeModel(GL_FLAT); // activa sombreado plano
```

- ▶ **Sombreado de vértices** el color de cada vértice es interpolado en el interior de los polígonos:

```
glShadeModel(GL_SMOOTH); // activa sombreado de vértices
```

- ▶ **Sombreado de píxeles**: no está disponible en OpenGL con el cauce fijo.

initialmente, el método de sombreado es el sombreado de vértices (**GL\_SMOOTH**).

## Sombreado y iluminación

El método de sombreado en OpenGL puede cambiarse incluso si la iluminación está desactivada. En cualquier caso (con ilum. activada o desactivada), OpenGL siempre asocia un color a cada vértice:

- ▶ Sin iluminación activada el color asociado a cada vértice es el color especificado en la tabla de colores o con **glColor**.
- ▶ Con iluminación activada el color asociado a cada vértice es el resultado de evaluar el modelo de iluminación local en dicho vértice.

*nota:* la función **glShadeModel** fue declarada *obsoleta (deprecated)* en OpenGL 3.0 y eliminada de OpenGL 3.1 y posteriores, al igual que lo relacionado con iluminación.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.5.

Carga de texturas en el sistema gráfico..

## Identificadores de textura. Creación.

OpenGL puede gestionar más de una textura a la vez. Para diferenciarlas usa un valor entero único para cada una de ellas, que se denomina **identificador de textura** (*texture name*) (de tipo **GLuint**).

- ▶ Para crear o generar un nuevo identificador de textura único (distinto de cualquiera ya existente) usamos:

```
GLuint idTex ;  
glGenTextures( 1, &idTex ); // idTex = nuevo identificador
```

- ▶ Para crear  $n$  nuevos identificadores de textura (en un array de **GLuint**), hacemos:

```
GLuint arrIdTex[n] ; // n es una constante entera (n > 0)  
glGenTextures( n, arrIdTex ); // crea n nuevos idents. en arrIdTex
```

# Unidades de textura

OpenGL permite configurar distintas *unidades de textura*, y activar distintas texturas en las distintas unidades.

Para cambiar la unidad de texturas activa podemos usar:

```
// activa textura con identificador 'idTex' :  
glActiveTexture( GL_TEXTUREi );
```

- ▶ el parámetro debe ser **GL\_TEXTURE0**, **GL\_TEXTURE1**,  
**GL\_TEXTURE2**, etc....
- ▶ inicialmente la unidad activa es la unidad 0
- ▶ nosotros siempre usaremos la unidad 0

# Unidades de textura y textura activa

En el estado interno de OpenGL, por cada unidad de textura, hay en cada momento un identificador de textura activa

- ▶ Cualquier operación de visualización de primitivas usará la textura asociada a dicho identificador.
- ▶ Cualquier operación de configuración de la funcionalidad de texturas se referirá a dicha textura activa.

Para cambiar el identificador de textura activa podemos hacer:

```
// activa textura con identificador 'idTex' :  
glBindTexture( GL_TEXTURE_2D, idTex );
```

# Alojamiento en RAM de imágenes de textura

Antes de usar una textura en OpenGL (de tamaño  $n_x \times n_y$ ), es necesario alojar en la memoria RAM una matriz con los colores de sus texels:

- ▶ Cada texel se representa (usualmente) con tres bytes (enteros sin signo entre 0 y 255), que codifican la proporción de rojo, verde y azul, respecto al valor máximo (255).
- ▶ Los tres bytes de cada texel se almacenan contiguos, usualmente en orden RGB.
- ▶ Los  $3n_x$  bytes de cada fila de texels se almacenan contiguos, de izquierda a derecha.
- ▶ Las  $n_y$  filas se almacenan contiguas, desde abajo hacia arriba.
- ▶ Se conoce la dirección de memoria del primer byte, que llamamos **texels** (es un puntero de tipo **void \***)

con este esquema la imagen ocupará, lógicamente,  $3n_xn_y$  bytes consecutivos en memoria.

# Especificación de los texels de la imagen de textura

En cualquier momento podemos especificar cual será la imagen de textura asociada al identificador de textura activa, con

## `glTexImage2D:`

```
glTexImage2D( GL_TEXTURE_2D,  
    0,           // nivel de mipmap (para imágenes multiresolución)  
    GL_RGB,      // formato interno  
    ancho,       // núm. de columnas (potencia de dos:  $2^n$ ) (GLsizei)  
    alto,        // núm de filas (potencia de dos:  $2^m$ ) (GLsizei)  
    0,           // tamaño del borde, usualmente es 0  
    GL_RGB,      // formato y orden de los texels en RAM  
    GL_UNSIGNED_BYTE,  
                // tipo de cada componente de cada texel  
    texels       // puntero a los bytes con texels (void *)  
);
```

Al llamar a esta función, OpenGL leerá los bytes de la RAM y los copiará en otra memoria (típicamente la memoria de vídeo o de la GPU, en un formato interno).

# Especificación de la imagen con GLU

Si es posible usar GLU, hay una alternativa preferible a **glTexImage2D** que no requiere imágenes de tamaño potencia de dos, y que además genera automáticamente versiones a múltiples resoluciones (*mip-maps*)

```
gluBuild2DMipmaps( GL_TEXTURE_2D,  
    GL_RGB,      // formato interno  
    ancho,       // núm. de columnas (arbitrario) (GLsizei)  
    alto,        // núm de filas (arbitrario) (GLsizei)  
    GL_RGB,      // formato y orden de los texels en RAM  
    GL_UNSIGNED_BYTE,  
                // tipo de cada componente de cada texel  
    texels       // puntero a los bytes con texels (void *)  
);
```

(esta función hace copias escaladas de la imagen para adaptarla a tamaños potencias de dos a distintas resoluciones)

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.6.

Configuración de texturas en el cauce fijo..

## Texturas en el cauce fijo

Cuando se habilitan las texturas hay una textura activa en cada momento, que se consulta cada vez que un polígono se proyecta en un pixel, antes de calcular el color de dicho pixel. El color obtenido es  $T$ . Además, en dicho pixel se calcula un color interpolado  $C$ , a partir de **glColor** o la tabla de colores de vértice. Entonces:

- ▶ Con iluminación **desactivada**, el color del pixel puede ser:
  - ▶ igual a  $T$ : se ignora el color  $C$  (usaremos siempre esta opción)
  - ▶ igual a  $T \cdot C$ : el color  $C$  modula al color de la textura.
- ▶ Con iluminación **activada**, el color de la textura  $T$  sustituye a las reflectividades del material (usualmente a la difusa y la ambiental) en la evaluación del MIL.

todas las operaciones de texturas en el cauce fijo requieren que esta funcionalidad esté activada.

# Activación y desactivación

Las ordenes **glEnable** y **glDisable** se pueden usar para activar o desactivar, en el cauce de funcionalidad fija, toda la funcionalidad de OpenGL relacionada con las texturas

```
glEnable( GL_TEXTURE_2D ) ; // habilita texturas  
glDisable( GL_TEXTURE_2D ) ; // deshabilita texturas
```

(las texturas están inicialmente desactivadas en el cauce fijo).

## Parámetros de texturas.

En el estado de OpenGL, hay un conjunto de atributos o parámetros que determinan la apariencia de las texturas. Estos parámetros determinan, entre otros aspectos:

- ▶ Como se usa el color de los texels en el MIL con ilum. activada (que reflectividades del material son obtenidas de la textura activa)
- ▶ Como se selecciona el texel o texels a partir de una coords. de textura (más cercano o interpolación).
- ▶ Como se selecciona el texel cuando las coords. de textura no están en el rango  $[0, 1]$  (replicado o truncamiento).
- ▶ Si se asignan explícitamente coordenadas o bien OpenGL las genera proceduralmente, y en este caso como se hace.

## Texturas e iluminación.

La función **glLightModel** puede usarse para determinar como los colores de la textura afectan al MIL, cuando la iluminación y la texturas están activadas. Hay dos opciones:

- ▶ El color de la textura se use en lugar de todas las reflectividades del material,  $M_A, M_D$  y  $M_S$ ,

```
glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL,  
               GL_SINGLE_COLOR ) ; // inicialmente activado
```

- ▶ El color de la textura se usa en lugar de  $M_A$  y  $M_D$ , pero no  $M_S$ , esto permite brillos especulares de color blanco cuando hay texturas de color.

```
glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL,  
               GL_SEPARATE_SPECULAR_COLOR ) ;
```

## Selección de texels

OpenGL permite especificar como se seleccionarán los texels en cada consulta posterior de la textura activa, cuando el pixel actual es igual o más pequeño que el texel que se proyecta en él:

- ▶ seleccionar el texel con centro más cercano al centro del pixel:

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_MAG_FILTER, GL_NEAREST );
```

- ▶ hacer interpolación bilineal entre los cuatro texels con centros más cercanos al centro del pixel:

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_MAG_FILTER, GL_LINEAR );
```

(la opción inicialmente activada es la segunda de ellas).

## Generación procedural de coordenadas de textura.

OpenGL puede generar proceduralmente las coordenadas de textura  $(s, t)$  en cada pixel, cada vez que se consulte la textura, a partir de las coordenadas de objeto (o de mundo) del punto de la superficie  $\mathbf{p}$  que se proyecta en el centro del pixel.

Para habilitar esta posibilidad, hacemos:

```
glEnable( GL_TEXTURE_GEN_S ); // desactivado inicialmente  
glEnable( GL_TEXTURE_GEN_T ); // desactivado inicialmente
```

igualmente podemos usar **glDisable** para desactivar.

Es necesario hacer ambas llamadas ya que OpenGL permite generar únicamente la coordenada  $s$  o únicamente la  $t$  (normalmente se generan ambas o ninguna).

## Tipo de función para generación procedural.

Con OpenGL podemos usar funciones lineales (proyección en un plano) para la generación automática de coords. de textura.

- ▶ Para hacerlo en **coordenadas de objeto** (antes de modelview):

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR ) ;  
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR ) ;
```

- ▶ Para **coordenadas de ojo** (después de modelview), haríamos:

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR ) ;  
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR ) ;
```

- ▶ No es posible usar para esto las coordenadas del mundo en el cauce fijo.

(la opción inicial es usar coordenadas de objeto).

# Parámetros de la función lineal

La función lineal que se usa para calcular  $(s, t)$  a partir de  $(x, y, z)$  es de la forma:

$$\begin{aligned}s &= a_s x + b_s y + c_s z + d_s \\t &= a_t x + b_t y + c_t z + d_t\end{aligned}$$

Si queremos proyectar en un plano que pasa por  $\mathbf{q}$  y contiene los vectores  $\mathbf{e}_s = (s_x, s_y, s_z)$  y  $\mathbf{e}_t = (t_x, t_y, t_z)$  (de longitud unidad y perpendiculares entre sí), entonces debemos de hacer:

$$\begin{array}{lll}a_s &= s_x & a_t &= t_x \\b_s &= s_y & b_t &= t_y \\c_s &= s_z & c_t &= t_z \\d_s &= -\mathbf{q} \cdot \mathbf{e}_s & d_t &= -\mathbf{q} \cdot \mathbf{e}_t\end{array}$$

# Especificación de coeficientes

Para especificar los coeficientes de las funciones lineales, podemos usar **glTexGenfv**. OpenGL guarda en su estado dos juegos de parámetros, cada uno usado para un modo de generación de coordenadas.

```
GLfloat coefsS[4] = { as, bs, cs, ds } ,  
                      coefsT[4] = { at, bt, ct, dt } ;  
  
// para el modo de coords. de objeto:  
glTexGenfv( GL_S, GL_OBJECT_PLANE, coefsS ) ;  
glTexGenfv( GL_T, GL_OBJECT_PLANE, coefsT ) ;  
  
// para el modo de coords. del ojo:  
glTexGenfv( GL_S, GL_EYE_PLANE, coefsS ) ;  
glTexGenfv( GL_T, GL_EYE_PLANE, coefsT ) ;
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.7.

Implementación de la clase **CauceFijo**.

# Iluminación usando CauceFijo

La clase **CauceFijo** implementa el interfaz **Cauce** para facilitar implementaciones portables:

El método **fijarEvalMIL** activa o desactiva la iluminación:

```
void CauceFijo::fijarEvalMIL( const bool nue_eval_mil )  
{  
    if ( nue_eval_mil ) glEnable( GL_LIGHTING );  
    else                 glDisable( GL_LIGHTING );  
}
```

El método **fijarModoSombrPlano** permite activar sombreado plano o desactivarlo (activa modo suave)

```
void CauceFijo::fijarModoSombrPlano  
    ( const bool nue_sombr_plano )  
{  
    const GLenum modo = nue_sombr_plano ? GL_FLAT : GL_SMOOTH ;  
    glShadeModel( modo );  
}
```

## Fijar parámetros de las fuentes (1/2)

El método **fijarFuentesLuz** se invoca una vez por cuadro:

```
void CauceFijo::fijarFuentesLuz
    ( const std::vector<Tuple3f> & color,
      const std::vector<Tuple4f> & pos_dir_wc )
{
    const GLfloat color_negro[4] = { 0.0, 0.0, 0.0, 1.0 };

    // (1) fijar parámetros de iluminación y materiales
    glEnable( GL_NORMALIZE );
    glEnable( GL_COLOR_MATERIAL );
    glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE );

    glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
    glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL,
                    GL_SEPARATE_SPECULAR_COLOR );
    glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
    glLightModelfv( GL_LIGHT_MODEL_AMBIENT, color_negro );
    ....
```

## Fijar parámetros de las fuentes (2/2)

```
.....  
// (2) habilitar fuentes, desde 0 hasta color.size()-1,  
// para cada una de ellas, se envía su posición (el color de la fuente se enviará  
// cuando se especifiquen los parámetros del material, más adelante)  
for( unsigned i = 0 ; i < color.size() ; i++ )  
{ glEnable( GL_LIGHT0+i );  
    glLightfv( GL_LIGHT0+i, GL_POSITION, pos_dir_wc[i] );  
}  
  
// (3) deshabilitar restantes fuentes, hasta 'maxNumFuentesLuz' (no incluido).  
for( unsigned i = color.size() ; i < maxNumFuentesLuz(); i++ )  
    glDisable( GL_LIGHT0 + i );  
  
// (4) registrar los colores de las fuentes de luz, se usarán más adelante cuando  
// se active el material y se conozcan todos los parámetros del M.I.L.  
colores_fuentes.clear();  
for( unsigned i = 0 ; i < color.size() ; i++ )  
    colores_fuentes.push_back( color[i] );
```

## Fijar parámetros del MIL

El método **fijarParamsMIL** configura las reflectividades del MIL:

```
void CauceFijo::fijarParamsMIL( const Tupla3f &amb,
                                const Tupla3f &dif, const Tupla3f &esp,
                                const float exp )
{
    // definir emisividad (nula), comp. especular (1,1,1) y exponente
    glMaterialfv( GL_FRONT_AND_BACK, GL_EMISSION, Tupla4f{0,0,0,1});
    glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, Tupla4f{1,1,1,1});
    glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, exp );
    // definir los colores de las fuentes de luz en OpenGL: son el producto de
    // los colores de las fuentes que hay registradas por las reflectividades
    for( unsigned i = 0 ; i < colores_fuentes.size() ; i++ )
    { glLightfv( GL_LIGHT0+i, GL_AMBIENT, pro(amb,colores_fuentes[i]));
      glLightfv( GL_LIGHT0+i, GL_DIFFUSE, pro(dif,colores_fuentes[i]));
      glLightfv( GL_LIGHT0+i, GL_SPECULAR, pro(esp,colores_fuentes[i]));
    }
}
```

**pro** acepta dos tuplas y devuelve la tupla producto comp. a comp.

## Habilitar una textura

El método **fijarEvalText** permite habilitar (o deshabilitar) las texturas, y en el primer caso requiere el identificador de la textura a activar:

```
void CauceFijo::fijarEvalText( const bool nue_eval_text,
                               const int   nue_text_id  )
{
    if ( nue_eval_text )
    {
        glEnable( GL_TEXTURE_2D );
        glBindTexture( GL_TEXTURE_2D, nue_text_id );
        glColor3f( 1.0, 1.0, 1.0 );
    }
    else
        glDisable( GL_TEXTURE_2D );
}
```

## Fijar parámetros de generación de cc.t. (1/2)

El método **fijarTipoGCT** fija los parámetros relacionados con generación automática de coordenadas de textura (generación activada sí/no, tipo de generación, coeficientes):

```
void CauceFijo::fijarTipoGCT( const int nue_tipo_gct,
                               const float * coefs_s, const float * coefs_t )
{
    switch ( nue_tipo_gct )
    {
        case 0 : // ==> generación de cc.t. desactivada
            glDisable( GL_TEXTURE_GEN_S );
            glDisable( GL_TEXTURE_GEN_T );
            break ;
        .....
    }
}
```

## Fijar parámetros de generación de cc.t. (2/2)

```
.....  
  
case 1 : // ==> generación activada, en coords de objeto.  
    glEnable( GL_TEXTURE_GEN_S );  
    glEnable( GL_TEXTURE_GEN_T );  
    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR );  
    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR );  
    glTexGenfv( GL_S, GL_OBJECT_PLANE, coefs_s ) ;  
    glTexGenfv( GL_T, GL_OBJECT_PLANE, coefs_t ) ;  
    break ;  
  
case 2 : // ==> generación activada, en coordenadas de cámara  
    glEnable( GL_TEXTURE_GEN_S );  
    glEnable( GL_TEXTURE_GEN_T );  
    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );  
    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );  
    glTexGenfv( GL_S, GL_EYE_PLANE, coefs_s ) ;  
    glTexGenfv( GL_T, GL_EYE_PLANE, coefs_t ) ;  
    break ;  
}  
}
```

## Sección 4.

### Iluminación y texturas en el cauce programable.

- 4.1. Estructura del *vertex shader*.
- 4.2. Estructura del *fragment shader*.
- 4.3. Implementación de la clase **CauceProgramable**.

## Illum. y texturas en el cauce programable

En OpenGL moderno la evaluación del Modelo de Iluminación Local (MIL) sencillo y la consulta de texturas se hacen principalmente en los *shaders*

- ▶ En el *vertex shader* se hace:
  - ▶ Generación de coordenadas de textura (si está activada, en otro caso se pasan las coords. de textura del vértice)
- ▶ En el *fragment shader* se hace:
  - ▶ Obtención del vector normal y el vector al observador
  - ▶ Obtención de los parámetros de las fuentes de luz.
  - ▶ Obtención de los parámetros del material.
  - ▶ Consulta de la textura
  - ▶ Evaluación del MIL.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 4. Iluminación y texturas en el cauce programable

Subsección 4.1.

Estructura del *vertex shader*..

# Generación de coords. de textura en el Vertex Shader

Aquí vemos los parámetros de entrada y las variables relacionadas:

```
.....
// parámetros de entrada (uniforms)
uniform int eval_text; // 0 -> no evaluar texturas, 1 -> sí eval. texturas
uniform int tipo_gct; // tipo gen.cc.tt. (0=desact, 1=objeto, 2=camara)
uniform vec4 coefs_s; // coefficientes para G.CC.TT. (coordenada 's')
uniform vec4 coefs_t; // coefficientes para G.CC.TT. (coordenada 't')
// variables calculadas en el vertex shader (varying)
varying vec4 var_posic_ec; // posicion del vert. en EC (coords.cámara)
varying vec2 var_coord_text; // coords. de textura del vértice
// función de cálculo de las coords. de textura
vec2 CoordsTextura() { .... }
...
void main()
{
    ...
    var_posic_ec = ..... ; // calcula posición en EC
    var_coord_text = CoordsTextura(); // calcula coords. de text.
    gl_Position = ..... ; // calcula posición en CC
}
```

# Generación de coordenadas de textura

La función **CoordsTextura** se encarga de devolver las coordenadas de textura según los disntintos *uniform* y la posición en coordenadas de cámara o de objeto:

```
vec2 CoordsTextura() // calcula las coordenadas de textura
{
    if ( eval_text == 0 )           // si no se están evaluando las cc.t.
        return vec2( 0.0, 0.0 );   // devuelve cualquier cosa
    if ( tipo_gct == 0 )           // texturas activadas, generación desact:
        return gl_MultiTexCoord0.st; // devuelve las cc.t. del vértice

    vec4 pos_ver ;
    if ( tipo_gct == 1 )           // generacion en coordenadas de objeto
        pos_ver = gl_Vertex;      // usar las coords originales (objeto)
    else                           // generacion en coords de cámara
        pos_ver = var_posic_ec;   // usar las coordenadas de cámara

    return vec2( dot(pos_ver,coefs_s), dot(pos_ver,coefs_t) );
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 4. Iluminación y texturas en el cauce programable

Subsección 4.2.

Estructura del *fragment shader*..

## Evaluación del MIL en el *fragment shader*. Parámetros.

El *fragment shader* se encarga de evaluar el Modelo de Iluminación Local (MIL) sencillo que hemos introducido antes (el mismo que el cauce fijo). La evaluación produce como resultado el color del pixel.

Usa estos parámetros:

- ▶ Un valor lógico que indica si se debe evaluar el MIL o no. Otro que indica si se debe usar la textura o no.
- ▶ Vector normal (normalizado), el vector al observador (posición en EC negada), coordenadas de textura.
- ▶ El número de fuentes de luz activas.
- ▶ Parámetros de las fuentes de luz: para cada una, su posición o dirección y el color, (en sendos arrays).
- ▶ Parámetros del material: reflectividades difusa, ambiental y especular, color y exponente de brillo.
- ▶ Textura activa actualmente, coordenadas de textura calculadas en el fragment shader.

# Eval. MIL: parámetros de entrada *uniform*

Son los siguientes:

```
// MIL y texturas
uniform int      eval_mil;    // evaluar el MIL: sí (1) o no (0)
uniform int      sombr_plano; // Sombreado: 0 -> Gouroud, 1 -> plano
uniform int      eval_text;   // usar textura sí (1) o no (0)
uniform sampler2D tex;       // textura activa en la unidad 0

// datos del material activo
uniform vec3     mil_ka;     // reflectividad ambiental ( $M_a$ )
uniform vec3     mil_kd;     // reflectividad difusa
uniform vec3     mil_ks;     // reflectividad pseudo-especular
uniform float    mil_exp;    // exponente de brillo

// datos de las fuentes de luz activas
const int max_num_luces = 8 ; // máx. númer. de fuentes de luz activas
uniform int      num_luces;  // número de luces activas
uniform vec4     pos_dir_luz_ec[max_num_luces]; // posic./direc.
uniform vec3     color_luz[max_num_luces] ;      // colores
```

## Evaluación del MIL: parámetros de entrada *varying*

En el *fragment shader*: las variables *varying* son parámetros de entrada, con datos interpolados desde el *vertex shader*. Usaremos las siguientes declaraciones:

```
varying vec4 var_posic_ec ;      // posicion del punto  
varying vec3 var_normal_ec;     // normal del punto (no norm.)  
varying vec3 var_vec_obs_ec ;   // vector hacia el observador  
varying vec4 var_color;         // color de la primitiva en el pixel  
varying vec2 var_coord_text;    // coordenadas de textura
```

La posición, la normal (no normalizada), y el vector al observador se expresan en coordenadas de cámara (EC)

# Vector hacia el observador y normal

Se calculan usando estas funciones:

```
vec3 VectorHaciaObs() // devuelve el vector hacia el observador normalizado
{
    return normalize( var_vec_obs_ec );
}

vec3 NormalTriangulo() // calcula la normal al triángulo
{
    vec4 tx = dFdx( var_posic_ec ); // tangente al tri. en horizontal
        ty = dFdy( var_posic_ec ); // tangente al tri. en vertical
    return normalize( cross( tx.xyz, ty.xyz ) ); // producto vectorial
}

vec3 Normal() // devuelve la normal que se debe usar en el MIL
{
    vec3 nor = (sombr_plano == 1) ? NormalTriangulo()
                                    : normalize( var_normal_ec );
    return gl_FrontFacing ? nor // es una cara delantera
                          : -nor; // es una cara trasera
}
```

# Vector hacia una fuente de luz

El vector hacia la  $i$ -ñesima fuente de luz (normalizado), en coordenadas de cámara, se calcula en función de la componente W de la posición o dirección a dicha fuente.

Usamos esta función:

```
vec3 VectorHaciaFuente( int i )
{
    return ( pos_dir_luz_ec[i].w == 1.0 ) ?
        normalize( pos_dir_luz_ec[i].xyz - var_posic_ec.xyz ) :
        normalize( pos_dir_luz_ec[i].xyz ) ;
}
```

# Evaluación del MIL

```
vec3 EvalMIL( vec3 col ) // col ≡ color interpolado o de textura
{
    vec3 n = Normal(),                      // normal (EC)
        v = VectorHaciaObs(),                // vector hacia observador (EC)
        s = vec3( 0.0, 0.0, 0.0 );           // suma color debido a cada fuente

    for( int i = 0 ; i < num_luces ; i++ ) // para cada fuente
    {
        vec3 l = VectorHaciaFuente( i ); // vector hacia fuente (EC)
        float nl = dot( n, l ); // coseno de ángulo entre n y l

        if ( 0.0 < nl ) // si normal está de cara a la luz
        {
            float hn = max( 0.0, dot( n, normalize( l+v ) ) );
            vec3 c = mil_kd*col*nl + mil_ks*pow(hn,mil_exp);
            s = s + c*color_luz[i]; // sumar componentes difusa + especular
        }
        s = s + mil_ka*col*color_luz[i]; // sumar componente ambiental
    }
    return s ; // devolver la suma de todas las fuentes
}
```

# Función principal

La función **main** calcula el color del pixel en **gl\_FragColor**, invocando **EvalMIL** si es necesario:

```
void main()
{
    vec4 color_obj = ( eval_text == 1 ) ?
        texture2D( tex, var_coord_text ) : // color de textura
        var_color ;                      // color de la primitva

    gl_FragColor = ( eval_mil == 1 ) ?
        vec4( EvalMIL( color_obj.rgb ), 1.0 ) : // ilum. activada
        color_obj ;                          // ilum desactivada
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 4. Iluminación y texturas en el cauce programable

Subsección 4.3.

Implementación de la clase **CauceProgramable..**

# Iluminación usando CauceProgramable

La clase **CauceProgramable** implementa el interfaz **Cauce** para facilitar el uso de los shaders.

El método **fijarEvalMIL** actualiza el uniform **eval\_mil**:

```
void CauceProgramable::fijarEvalMIL( const bool nue_eval_mil )  
{  
    glUseProgram( id_prog );  
    glUniform1i( loc_eval_mil, eval_mil ? 1 : 0 );  
}
```

El método **fijarModoSombrPlano** actualiza **sombr\_plano**:

```
void CauceProgramable::fijarModoSombrPlano  
    ( const bool nue_sombr_plano )  
{  
    glUseProgram( id_prog );  
    glUniform1i( loc_sombr_plano, sombr_plano ? 1 : 0 );  
}
```

# Parámetros de las fuentes

El método **fijarFuentesLuz** se invoca una vez por cuadro:

```
void CauceProgramable::fijarFuentesLuz
    ( const std::vector<Tuple3f> & color,
      const std::vector<Tuple4f> & pos_dir_wc )
{
    const unsigned nl = color.size(); // número de fuentes
    std::vector<Tuple4f> pos_dir_ec; // vector con pos./dir.
    for( unsigned i = 0 ; i < nl ; i++ ) // para cada fuente
        pos_dir_ec.push_back( mat_vista * pos_dir_wc[i] ); // añadir p/d
    // copiar vectores en los uniforms:
    glUseProgram( id_prog );
    glUniform1i( loc_num_luces, nl );
    glUniform3fv( loc_color_luz, nl, (const float *)color.data() );
    glUniform4fv( loc_pos_dir_luz_ec, nl,
                  (const float *)pos_dir_ec.data() );
}
```

transforma las direcciones o posiciones por la matriz de vista actual,  
así que las interpreta en coords. de mundo y produce E.C.

## Parámetros del MIL (material)

El método **fijarParamsMIL** fija los parámetros del material  
(reflectividades ambiente, difusa y especular, y el exponente)

```
void CauceProgramable::fijarParamsMIL
    ( const Tupla3f & mil_ka, const Tupla3f & mil_kd,
      const Tupla3f & mil_ks, const float exp )
{
    glUseProgram( id_prog );
    glUniform3fv( loc_mil_ka, 1, mil_ka );
    glUniform3fv( loc_mil_kd, 1, mil_kd );
    glUniform3fv( loc_mil_ks, 1, mil_ks );
    glUniform1f ( loc_mil_exp, exp );
}
```

se debe invocar cada vez que se quiera activar un nuevo material

## Habilitar una textura

El método **fijarEvalText** permite habilitar (o deshabilitar) las texturas, y en el primer caso requiere el identificador de la textura a activar:

```
void CauceProgramable::fijarEvalText( const bool nue_eval_text,
                                      const int   nue_text_id )
{
    glUseProgram( id_prog );
    if ( eval_text ) // activar
    {
        glBindTexture( GL_TEXTURE0 );
        glActiveTexture( GL_TEXTURE_2D, nue_text_id );
        glUniform1i( loc_eval_text, 1 );
        glColor3f( 1.0, 1.0, 1.0 );
    }
    else // desactivar
    {
        glUniform1i( loc_eval_text, 0 );
    }
}
```

## Fijar parámetros de generación de cc.t.

El método **fijarTipoGCT** fija los parámetros relacionados con generación automática de coordenadas de textura (generación activada sí/no, tipo de generación, coeficientes):

```
void CauceProgramable::fijarTipoGCT( const int nue_tipo_gct,
                                      const float * coefs_s, const float * coefs_t )
{
    glUniform1i( loc_tipo_gct, tipo_gct );

    if ( tipo_gct == 1 || tipo_gct == 2 )
    { glUniform4fv( loc_coefs_s, 1, coefs_s );
      glUniform4fv( loc_coefs_t, 1, coefs_t );
    }
}
```

## Sección 5.

### Representación de materiales, texturas y fuentes..

- 5.1. Las clases **Textura** y **Material**
- 5.2. Fuentes de luz. La clase **ColeccionFuentes**
- 5.3. Materiales en el grafo de escena

# Clases relacionadas con iluminación y texturas.

En esta sección veremos como representar en una aplicación los diversos parámetros relacionados con la iluminación y texturas:

- ▶ Clase **Textura**: incluye un puntero a los texels en RAM, y los parámetros de generación de coords. de text.
- ▶ Clase **Material**: incluye los parámetros del MIL (reflectividades ambiente, difusa y especular, exponente de brillo), y opcionalmente un puntero a una instancia de una textura.
- ▶ Clase **FuenteLuz** y **ColFuentesLuz**: parámetros que definen cada fuente de luz (posición o dirección, color), y conjunto de fuentes de luz a usar en una escena.
- ▶ Clase **NodoGrafoEscena**: en esta clase se podrán incluir entradas de tipo material.
- ▶ Clase **Escena**: contendrá una colección de fuentes, y un material inicial.

## Activación

Las instancias de estas las clases **Textura**, **Material** y **ColFuentesLuz** incorporan un método para *activarlas*:

- ▶ La **activación** es el proceso por el cual el cauce se configura para que en la siguientes operaciones de visualización se use una textura, un material, o una colección de fuentes.
- ▶ En todos los casos se usa un método llamado **activar**, que tiene como único parámetro una referencia al cauce actual.
- ▶ Veremos como se implementa la activación usando el interfaz de la clase **Cauce**.

Para las texturas, materiales y colecciones de fuentes, se definen clases derivadas con constructores que implementan distintas variantes.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 5. Representación de materiales, texturas y fuentes.

Subsección 5.1.

Las clases Textura y Material.

# Clase Textura

La clase textura se declara así:

```
class Textura
{
public:
    Textura( const std::string & nombreArchivoJPG ) ; // constructor
    ~Textura() ; // libera memoria ocupada por texels
    void activar( Cauce & cauce ) ; // activación

protected:
    void enviar() ; // envia la imagen a la GPU (gluBuild2DMipmaps)
    unsigned char * imagen = nullptr; // texels en mem. dinámica
    bool enviada = false; // true si enviada
    GLuint ident_textura = -1 ; // 'nombre' o identif. de text.
    unsigned ancho = 0, // número de columnas
            alto = 0 ; // número de filas de la imagen
    ModoGenCT modo_gen_ct = mgct_desactivada ; // modo gen. cc.t.
    float coefs_s[4] = {1.0,0.0,0.0,0.0}, // coeficientes (S)
          coefs_t[4] = {0.0,1.0,0.0,0.0}; // coeficientes (T)
};
```

# Clase Material

Encapsula una textura y los cuatro parámetros del MIL (reales)

```
class Material
{
public:
    Material() {} ; // usa valores por defecto (en las declaraciones)
    // constructores (sin textura y con textura)
    Material( const float p_k_amb, const float p_k_dif,
               const float p_k_pse, const float p_exp_pse );
    Material( Textura * p_textura,
               const float p_k_amb, const float p_k_dif,
               const float p_k_pse, const float p_exp_pse );
    void activar( Cauce & cauce ) ; // activación
    ~Material() ; // libera la textura, si hay alguna
protected:
    Textura * textura = nullptr; // textura, si != nullptr
    float k_amb = 0.2f, // coeficiente de reflexión ambiente
          k_dif = 0.8f, // coeficiente de reflexión difusa
          k_pse = 0.0f, // coeficiente de reflexión pseudo-especular
          exp_pse = 0.0f; // exponente de brillo para reflexion pseudo-especular
};
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 5. Representación de materiales, texturas y fuentes.

Subsección 5.2.

Fuentes de luz. La clase **ColecciónFuentes**.

# Clase FuenteLuz

Ejemplo de fuente de luz direccional y manipulable interactivamente:

```
class FuenteLuz
{
public:
    FuenteLuz( GLfloat p_longi_ini, GLfloat p_lati_ini,
                const Tupla3f & p_color ) ;
    bool gestionarEventoTeclaEspecial( int key ); // procesa L+tecla
    void actualizarLongi( const float incre ); // actualiza longi.
    void actualizarLati( const float incre ); // actualiza lati.

protected:
    float    longi,           // longitud actual en WC (grados, 0 a 360)
            lati ;          // latitud actual en WC (grados, -90 a +90)
    float    longi_ini,        // valor inicial de 'longi'
            lati_ini,        // valor inicial de 'lati'
    Tupla3f  col_ambiente,   // color de la fuente para componente ambiental
            col_difuso,     // color de la fuente para componente difusa
            col_especular; // color de la fuente para componente especular
    friend class ColFuentesLuz ;
};
```

# Clase ColFuentesLuz

Colección de fuentes manipulable (con una fuente actual)

```
class ColFuentesLuz
{
public:
    ColFuentesLuz() ; // crea la colección vacía
    ~ColFuentesLuz() ; // libera memoria ocu
    void insertar( FuenteLuz * pf ) ; // inserta nueva fuente (copia ptr)
    void activar( Cauce & cauce ) ; // activación fuentes e ilum.
    void sigAntFuente( int d ) ; // cambiar fuente act. (d==+1 o -1)
    FuenteLuz * fuenteLuzActual() ; // devuelve (puntero a) fuente act.

private:
    std::vector<FuenteLuz *> vpf ; // vector de punteros a fuentes
    GLint max_num_fuentes = 0 ; // máximo número de fuentes
    unsigned i_fuente_actual = 0 ; // índice de fuente actual
};
```

# Visualización con materiales y luces

El código de la aplicación, para visualizar una escena con iluminación activada, debe:

- ▶ Antes de visualizar los objetos:
  1. activar la evaluación del MIL
  2. activar una colección de fuentes de luz
  3. activar un material por defecto
- ▶ Durante la visualización, para visualizar un objeto que tiene un material específico, se debe
  1. guardar un puntero al material activado antes
  2. activar el material específico
  3. visualizar el objeto
  4. activar el material anterior

Para recordar un puntero al material activo, podemos usar el contexto de visualización.

## Ejemplo de visualización

Si queremos visualizar un objeto (**objeto**) usando un puntero a un material (**material**), podemos usar la variable **material\_act** del contexto de visualización (**cv**):

```
// (1) registrar material activo actualmente (suponemos que no es nullptr)
Material * material_previo = cv.material_act ;
// (2) activar material actual:
cv.material_act = material ;
cv.material_act->activar( cv.cauce_act );
// (3) visualizar el objeto
objeto->visualizarGL( cv );
// (4) reactivar material activo al inicio
cv.material_act = material_previo ;
cv.material_act->activar( cv.cauce_act );
```

Antes de visualizar una escena, debemos de activar algún material por defecto y guardar el puntero en el contexto de visualización.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 5. Representación de materiales, texturas y fuentes.

Subsección 5.3.

Materiales en el grafo de escena.

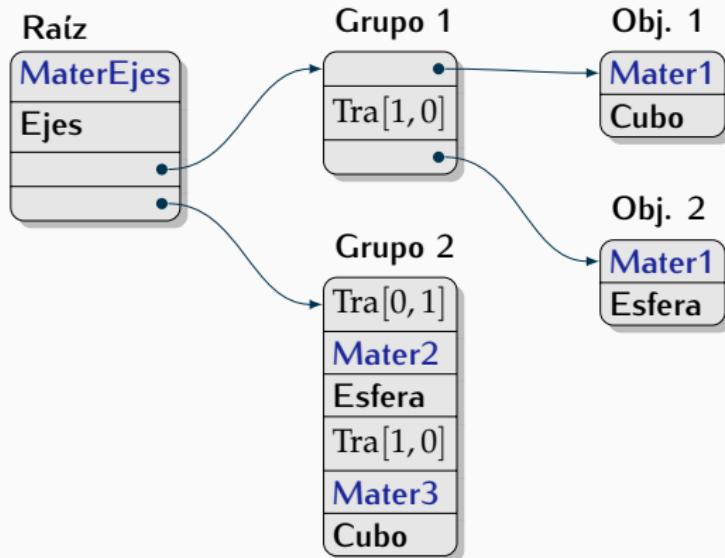
## Materiales en los grafos de escena

En los grafo de escena vamos a incorporar la posibilidad de asignar distintos materiales a distintas partes o nodos del grafo:

- ▶ Hay un nuevo tipo de entrada en los nodos: las **entradas de tipo material**: es un puntero a una instancia de **Material**.
- ▶ Un material en una entrada de un nodo afecta a todas las entradas posteriores de dicho nodo, hasta el final del nodo o bien hasta otra entrada posterior del nodo también de tipo material.
- ▶ Por tanto, toda entrada está afectada del primer material encontrado en el camino desde esa entrada hasta la primera entrada del nodo raíz (si no hay ninguno, se usaría uno por defecto).

# Ejemplos de materiales en el grafo de escena

Disponemos de las primitivas **Cubo**, **Esfera** y **Ejes**, y cuatro materiales posibles (se muestran en azul):



# Entradas de tipo material

Ahora las entradas de los nodos del tipo grafo de escena pueden contener un puntero a un material cualquiera:

```
struct EntradaNGE
{
    unsigned char tipoE ; // 0 => objeto, 1 => transformacion, 2 => material
    union
    {
        Objeto3D * objeto ; // ptr. a un objeto
        Matriz4f * matriz ; // ptr. a matriz 4x4 transf. (propriet.)
        Material * material ; // ptr. a material
    } ;
    // constructores (uno por tipo)
    EntradaNGE( Objeto3D * pObjeto ) ; // (copia únicamente el puntero)
    EntradaNGE( const Matriz4f & pMatriz ) ; // (crea copia de la matriz)
    EntradaNGE( Material * pMaterial ) ; // (copia únicamente el puntero)
};
```

Habrá una nueva versión del método **agregar** de los nodos:

```
class NodoGrafoEscena : public Objeto3D
{
    .....
    void agregar( Material * pMaterial ) ; // añadir material al final
};
```

# Procesamiento de nodos con materiales

En el método **visualizarGL** de la clase **NodoGrafoEscena**:

- ▶ Al inicio, registrar material activo

```
Material * material_pre = cv.iluminacion ?  
                                cv.material_act : nullptr;
```

- ▶ En el bucle, al encontrar una entrada de tipo material, se activa:

```
case TipoEntNGE::material : // si la entrada es de tipo 'material'  
    if ( cv.iluminacion ) // y si está activada la iluminación  
    { cv.material_act = entradas[i].material ; // registrar material  
      cv.material_act->activar( cauce ); // activar material  
    }  
    break ;
```

- ▶ Al finalizar, reactivamos el material activo originalmente

```
if ( material_pre != nullptr )  
{ cv.material_act = material_previo ; // copiar el previo en 'cv'  
  cv.material_act->activar( cauce ); // activar el previo  
}
```



UNIVERSIDAD  
DE GRANADA

# Informática Gráfica: Teoría. Tema 4. Interacción. Animación.

---

Carlos Ureña

2021-22

**Grado en Informática y Matemáticas**  
Dpt. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

## Teoría. Tema 4. Interacción. Animación.

### Índice.

1. Introducción
2. Eventos en GLFW
3. Posicionamiento
4. Control de cámaras
5. Selección
6. Animación

Sección 1.  
Introducción.

# Sistemas Gráficos Interactivos

Un **Sistema Gráfico Interactivo** (SGI) es un sistema software cuya respuesta a cada acción del usuario

- ▶ ocurre (por lo general) en un tiempo corto (del orden de décimas de segundo como mucho) desde dicha acción del usuario.
- ▶ se presenta al usuario en forma de visualización gráfica 2D o 3D

Un sistema SGI, por lo general, mantiene en memoria una estructura de datos (un modelo) y ejecuta un ciclo infinito, en cada iteración

1. espera o detecta una acción del usuario.
2. obtiene los datos que caracterizan dicha acción.
3. modifica el estado del modelo según dichos datos.
4. visualiza una nueva imagen obtenida a partir del nuevo estado del modelo

# Interactividad

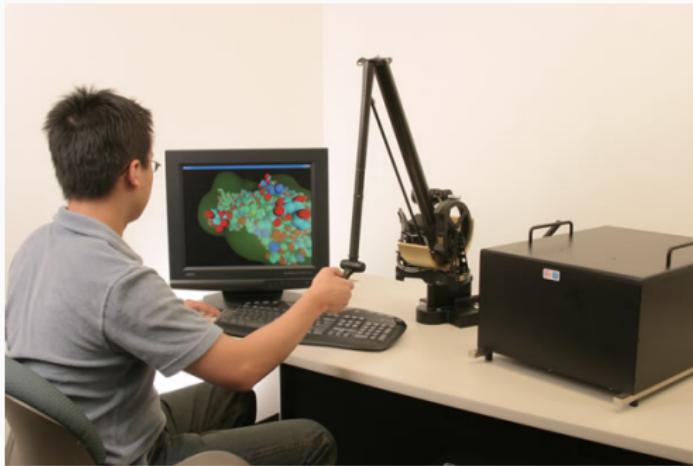
La incorporación de interactividad permite realizar aplicaciones que respondan ágilmente a las acciones de los usuarios y les ofrezcan retroalimentación sobre el efecto de dichas acciones.

La mayor parte de los sistemas gráficos son interactivos. Es esencial en:

- ▶ Videojuegos (*Videogames*) y Juego Serios (*Serious Games*).
- ▶ Sistemas de Diseño Asistido por Ordendor (CAD: *Computer Aided Design*)
- ▶ Sistemas de Realidad Virtual (VR: *Virtual Reality*)
- ▶ Sistemas de Realidad Aumentada (AR: *Augmented Reality*)
- ▶ Simuladores de aprendizaje (de conducción, de aviones, de barcos, etc...)

# Dispositivos de entrada y salida

En un SGI el usuario debe disponer de al menos un dispositivo de entrada (p.ej. teclado, ratón) y un dispositivo de visualización (típicamente un monitor).



Hay otros dispositivos de entrada: tabletas digitalizadoras, sistemas de posicionamiento 3D.

# Sistemas interactivos y de tiempo real

Los sistemas gráficos interactivos no siempre son **sistemas de tiempo real**:

- ▶ En un sistema interactivo se requiere que el retardo (latencia) entre la acción del usuario y la respuesta del sistema sea suficientemente pequeño como para que el usuario perciba una relación de causa-efecto.
- ▶ No obstante, eventualmente la respuesta puede demorarse algo más.
- ▶ En un **sistema de tiempo real** la latencia debe ser menor o igual que un tiempo máximo de respuesta prefijado en las especificaciones del sistema. Un retraso superior a ese límite se considera un fallo del sistema.

## Realimentación: utilidad

La realimentación es el mecanismo mediante el cual el sistema da información al usuario útil para permitir al usuario

- ▶ conocer más fácilmente el estado interno del sistema.
- ▶ ayudar a decidir la siguiente acción a realizar.

La información de realimentación que el sistema genera en cada momento depende lógicamente del estado del sistema y de la información previamente entrada por el usuario. Se puede usar con diferentes fines específicos:

- ▶ mostrar el estado del sistema
- ▶ como parte de una función de entrada
- ▶ para reducir la incertidumbre del usuario

# Funciones de entrada en un SGI

Un sistema gráfico interactivo necesita normalmente funciones de entrada usuales en todo tipo de aplicaciones, p.ej:

- ▶ Entrada de una cadena de texto.
- ▶ Entrada de un valor numérico.
- ▶ Selección de un dato en una lista.

Además en un SGI se suelen necesitar otros tipos de entrada más específicos, por ejemplo, en un sistema CAD 3D podemos encontrar, entre otras, estas funciones:

- ▶ Lectura de posiciones 3D (selección de unas coordenadas específicas en el espacio de coordenadas del mundo)
- ▶ Selección de una componente de un modelo jerárquico 3D
- ▶ Entrada de los ángulos de rotación que determinan la orientación de un objeto.

# Dispositivos físicos de entrada

El usuario introduce la información por medio de **dispositivos físicos de entrada**. Estos pueden ser

- ▶ de propósito general: p.ej. el teclado, o
- ▶ específicos para datos geométricos: p.ej. digitalizador.

Podemos clasificar los dispositivos de entrada gráfica atendiendo a la información que generan de forma directa. Esta puede ser:

- ▶ **Posiciones 2D**: tableta digitalizadora, lápiz óptico, pantalla táctil.
- ▶ **Posiciones 3D**: digitalizador, tracking.
- ▶ **Desplazamientos 2D**: ratón, trackball, joystick.
- ▶ **Imágenes o videos**: cámaras de fotografía o vídeo.

# Dispositivos lógicos de entrada

Un dispositivo lógico de entrada es una componente software que usa uno o varios dispositivos físicos de entrada para producir información de más alto nivel o más elaborada, obtenida a partir de los datos recibidos directamente de los dispositivos físicos (o indirectamente de otros dispositivos lógicos). Ejemplos:

- ▶ **Puntero del ratón:** permite entrar puntos en pantalla a partir de los desplazamientos físicos del ratón y del estado de sus botones.
- ▶ **Selector de componentes (Picker):** permite seleccionar un componente de un modelo 3D usando el puntero de ratón.
- ▶ **Fotografía 3D:** permite entrar imágenes con información de profundidad, a partir de pares de imágenes obtenidas con dos cámaras de fotografía convencionales.

# Lectura de datos dispositivos de entrada: modos de entrada.

Un **modo de entrada** es un método que usa una aplicación para decidir cuando debe consultar los datos relacionados con el estado (y los cambios de estado) de un dispositivo físico o lógico.

- ▶ Distintos tipos de dispositivos pueden tener asociados distintos modos de funcionamiento.
- ▶ Algunos tipos de dispositivos se pueden usar con más de un modo de funcionamiento.

Veremos los tres modos básicos más frecuentes:

- ▶ **modo de muestreo:** la aplicación consulta del estado actual en instantes arbitrarios.
- ▶ **modo de petición:** la aplicación espera hasta que se produzca un cambio de estado.
- ▶ **modo de cola de eventos:** la aplicación recibe una lista de cambios de estado no procesados aún.

# Estado y eventos de un dispositivo

Los cambios de estado que ocurren en un dispositivo de entrada (físico o lógico) se denominan **sucesos** o **eventos**.

- ▶ El **estado** de un dispositivo en un instante es el conjunto de valores de las variables gestionadas por el driver del dispositivo, y que representan en memoria su estado físico. P.ej:
  - ▶ En un teclado: un vector de valores lógicos que indican, para cada tecla, si dicha tecla está pulsada o no está pulsada.
  - ▶ En un ratón: estado de los dos botones (dos lógicos) y posición actual en pantalla del cursor (dos enteros).
- ▶ Un evento tiene asociados ciertos datos:
  - ▶ Instante de tiempo en el que el cambio ha ocurrido o se ha registrado
  - ▶ Información sobre: el estado inmediatamente después del evento, y sobre como ha cambiado el estado respecto al anterior al evento.

# Modo de muestreo

Un dispositivo puede usarse **modo de muestreo**: (sample):

- ▶ El software del dispositivo mantiene en memoria variables que representan el estado actual del dispositivo.
- ▶ La aplicación puede consultar dichas variables en cualquier momento, sin espera alguna.

## Ventajas/Desventajas

- ▶ Es muy eficiente en tiempo y memoria, y simple.
- ▶ Requiere a la aplicación emplear tiempo de CPU en muestrear a una frecuencia suficiente como para no perderse posibles cambios de estado relevantes.
- ▶ No hay información de cuando ocurrió el último cambio de estado.

Ejemplo: en un teclado, array de valores lógicos que indica, para cada tecla, si está pulsada o levantada.

## Modo de petición.

Para evitar perder eventos relevantes, la aplicación puede usar el **modo petición (request)**:

- ▶ La aplicación hace una petición y espera a que se produzca determinado tipo de evento.
- ▶ Cuando se produce, la aplicación recibe datos del evento.

### Ventaja/Desventajas

- ▶ Nunca se perderá el siguiente evento tras hacer una petición.
- ▶ Puede perderse un evento si no se hace una petición antes de que ocurra.
- ▶ Se puede perder mucho tiempo esperando (no se puede hacer otras cosas).

**Ejemplo:** en un teclado, esperar hasta que se pulse una tecla alfanumérica, y entonces saber de qué tecla se trata.

# Modos cola de eventos

## En el modo **cola de eventos**

- ▶ Cada vez que ocurre un evento, el software del dispositivo lo añade a una cola FIFO de eventos pendientes de procesar.
- ▶ La aplicación accede a la cola, extrae cada evento y lo procesa.

## Ventajas:

- ▶ No se pierde ningún evento.
- ▶ La aplicación no está obligada a consultar con cierta frecuencia, ni antes de cada evento.
- ▶ La aplicación no pierde tiempo en esperas si es necesario hacer otras cosas (se funciona en modo asíncrono).

Ejemplo: en un teclado, acceder a la lista de pulsaciones de teclas, ocurridas desde la última vez que se consultó.

Sección 2.  
Eventos en GLFW.

# Modelo de eventos de GLFW

GLFW gestiona los dispositivos de entrada en modo *cola de eventos*:

- ▶ Para cada tipo de evento se puede registrar una función de la aplicación que procesará eventos de ese tipo (cada función se llama **callback** o **función gestora de eventos**, FGE).
- ▶ Las *callbacks* se ejecutan al producirse un evento del tipo.
- ▶ Los eventos cuyo tipo no tiene *callback* asociado se ignoran.
- ▶ Los parámetros de un *callback* proporcionan información del evento.
- ▶ La aplicación debe incluir explícitamente un bucle de gestión de eventos, en cada iteración se espera hasta que se han procesado todos y luego se visualiza un frame o cuadro de imagen, si es necesario (si ha cambiado el estado de la aplicación).

## Funciones para registrar callbacks

Para cada tipo de evento, GLFW contiene una función para registrar el *callback* asociado a dicho tipo. Las funciones de registro y los tipos de eventos asociados son:

- ▶ `glfwSetMouseButtonCallback`: pulsar/levantar de botones del ratón.
- ▶ `glfwSetCursorPosCallback`: movimiento del cursor del ratón.
- ▶ `glfwSetWindowSizeCallback`: cambio de tamaño de la ventana.
- ▶ `glfwSetKeyCallback`: pulsar/levantar una tecla física.
- ▶ `glfwSetCharCallback`: pulsar una tecla (o una combinación de ellas) que produce un único carácter unicode.

# Eventos de botones del ratón

Son los eventos que ocurren cuando se pulsa o se levanta un botón del ratón. Para registrar el callback asociado, usamos esta llamada:

```
glfwSetMouseButtonCallback( ventana, FGE_PulsarLevantarBotonRaton )
```

El callback debe estar declarado así:

```
void FGE_PulsarLevantarBotonRaton( GLFWwindow* window, int button,  
                                    int action, int mods );
```

Los parámetros permiten conocer información del evento:

- ▶ **button**: botón afectado (valores: **GLFW\_MOUSE\_BUTTON\_LEFT**, **GLFW\_MOUSE\_BUTTON\_MIDDLE**, **GLFW\_MOUSE\_BUTTON\_RIGHT**)
- ▶ **action**: estado posterior del botón afectado, indica si se ha pulsado o levantado (valores: **GLFW\_PRESS**, **GLFW\_RELEASE**)
- ▶ **mod**: estado de teclas de modificación en el momento de levantar o pulsar (*shift*, *control*, *alt* y *super*).

## Ejemplo de callback de botón del ratón

Este callback (**FGE\_BotonRaton**) se encarga de procesar una pulsación del botón izquierdo o derecho del ratón

```
void FGE_PulsarLevantarBotonRaton( GLFWwindow* window, int button,
                                    int action, int mods )
{
    if ( action == GLFW_PRESS ) // si se ha pulsado un botón
    {
        double x,y ;
        glfwGetCursorPos( window, &x, &y ); // leer posición del ratón
        if ( button == GLFW_MOUSE_BUTTON_LEFT ) // si se ha pulsado izquierdo:
            RatonClickIzq( int(x), int(y) ); // hacer acción asociada
        else if ( button == GLFW_MOUSE_BUTTON_RIGHT )
        { xclick_der = int(x); // registra coord. X de pulsación bot. der.
          yclick_der = int(y); // idem coord. Y
        }
    }
}
```

Usamos **glfwGetCursorPos** para leer la posición. La función **RatonClickIzq** se encarga de procesar el click izquierdo como sea necesario.

## Eventos de movimiento del cursor del ratón

Son los eventos que ocurren cada vez que se mueve el ratón. Se pueden registrar con esta llamada:

```
glfwSetCursorPosCallback( ventana, FGE_RatonMovido )
```

El callback debe estar declarado con tres parámetros enteros:

```
void FGE_MovimientoRaton( GLFWwindow* window, double xpos, double ypos )
```

Los parámetros permiten conocer información del evento:

- ▶ **xpos,ypos**: posición del cursor en coordenadas de pantalla, en el momento del evento.

## Ejemplo de callback de movimiento activo del ratón

Si se registra este *callback*, podemos, por ejemplo, registrar el desplazamiento cada vez que se mueve el ratón estando pulsado el botón derecho (en combinación con **glfwGetMouseButton** para saber si está pulsado el botón derecho)

```
void FGE_MovimientoRaton( GLFWwindow* window, double xpos, double ypos )
{
    if ( glfwGetMouseButton( window, GLFW_MOUSE_BUTTON_RIGHT ) == GLFW_PRESS )
    { const int
        dx = int(xpos) - xclick_der , // calcular desplazamiento en X desde click
        dy = int(ypos) - yclick_der ; // calcular desplazamiento en Y desde click
        RatonArrastradoDer( dx, dy );
    }
}
```

La función **RatonArrastradoDer** podría ser cualquier función que se ejecuta cuando se mueve el ratón con el botón derecho pulsado. Recibe como parámetros los desplazamientos desde que se pulsó dicho botón derecho.

## Eventos de scroll. Ejemplo.

Este *callback* sirve para detectar movimientos de la rueda del ratón o de otros mecanismos de scroll (por ejemplo, gestos en un *touchpad*). Se pueden detectar movimientos tanto verticales como horizontales, en función del dispositivo usado.

```
glfwSetScrollCallback( ventana, FGE_Scroll );
```

El callback debe estar declarado con estos parámetros

```
void FGE_Scroll( GLFWwindow* window, double xoffset, double yoffset )
```

A modo de ejemplo, si queremos detectar únicamente scroll vertical

```
void FGE_Scroll( GLFWwindow* window, double xoffset, double yoffset )
{
    if ( fabs( yoffset ) < 0.05 ) // poco movimiento vertical -> ignorar evento
        return ;
    const int direccion = 0.0 < yoffset ? +1 : -1 ;
    ScrollVertical( direccion ); // función que procesa scroll (+1 o -1)
}
```

# El bucle de gestión de eventos

Se encarga de procesar eventos y visualizar:

```
terminar = false ; // escrito en los callbacks para señalar que se debe terminar.  
redibujar = true ; // escrito en los callbacks para señalar que hay que redibujar.  
while ( ! terminar ) // hasta que no sea necesario terminar...  
{ if ( redibujar )  
    { VisualizarEscena(); // visualizar la ventana si es necesario  
        redibujar = false; // no volver a visualizar si no es necesario  
    }  
    glfwWaitEvents(); // esperar y procesar eventos  
    terminar = terminar || glfwWindowShouldClose( glfw_window ) ;  
}
```

- ▶ **glfwWaitEvents** procesar todos los eventos pendientes, o bien, si no hay ninguno, espera bloqueada hasta que haya al menos un evento pendiente y entonces procesarlo (llama a los *callbacks* que haya registrados).
- ▶ **glfwWindowShouldClose** devuelve **true** solo si el usuario ha realizado alguna acción de cierre de ventana en el gestor de ventanas.

# Bucle de gestión de eventos con animaciones

Para animaciones, se ejecuta una función de actualización de estado cuando no hay eventos pendientes de procesar:

```
terminar = false ; // escrito en los callbacks para señalar que se debe terminar.  
redibujar = true ; // escrito en los callbacks para señalar que hay que redibujar.  
while ( ! terminar ) // hasta que no sea necesario terminar...  
{ if ( redibujar )  
{ VisualizarEscena(); // visualizar la ventana si es necesario  
    redibujar = false ; // no volver a visualizar si no es necesario  
}  
glfwPollEvents(); // procesar eventos pendientes (sin esperar)  
if ( ! terminar && ! redibujar ) // si no terminamos ni redibujamos  
    ActualizarEscena(); // actualizar el estado del modelo  
terminar = terminar || glfwWindowShouldClose( glfw_window ) ;  
}
```

- ▶ **glfwPollEvents**: si hay eventos pendientes, los procesa, en otro caso no hace nada.
- ▶ **ActualizarEscena**: se invoca continuamente, actualiza el modelo al siguiente estado de la animación.

# Bucle de gestión de eventos genérico

En realidad las animaciones pueden activarse o desactivarse:

```
terminar = false ; // escrito en los callbacks para señalar que se debe terminar.
redibujar = true ; // escrito en los callbacks para señalar que hay que redibujar.
animacion = false ; // true solo cuando están activadas las animaciones
while ( ! terminar )
{ if ( redibujar )           // si ha cambiado algo:
    { VisualizarEscena();   // dibujar la escena
      redibujar = false;    // evitar que se redibuje continuamente
    }
  if ( animacion )          // si la animación está activada:
    { glfwPollEvents();     // procesar eventos pendientes (sin esperar)
      if ( ! terminar && ! redibujar ) // si no terminamos ni redibujamos:
        ActualizarEscena();       // actualizar estado
    }
  else                      // si las animaciones está desactivadas:
    glfwWaitEvents();        // esperar que se produzcan eventos y procesarlos
  terminar = terminar || glfwWindowShouldClose( ventana_glfw ) ;
}
```

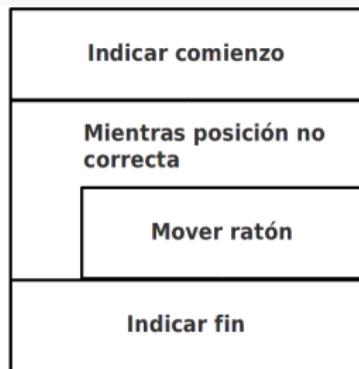
- **animacion**: tiene el estado de las animaciones.

## Sección 3. Posicionamiento.

# Posicionamiento: acciones del usuario.

El **posicionamiento** es la operación que permite al usuario seleccionar fácilmente un punto en el espacio de coordenadas del mundo de una aplicación 2D o 3D.

- ▶ Esto se lleva a cabo usando un dispositivo lógico (de tipo cursor) que permite seleccionar pixels en pantalla.
- ▶ El usuario opera mejorando iterativamente su selección hasta que la juzga correcta. La realimentación es esencial.



# Posicionamiento: pasos de la aplicación.

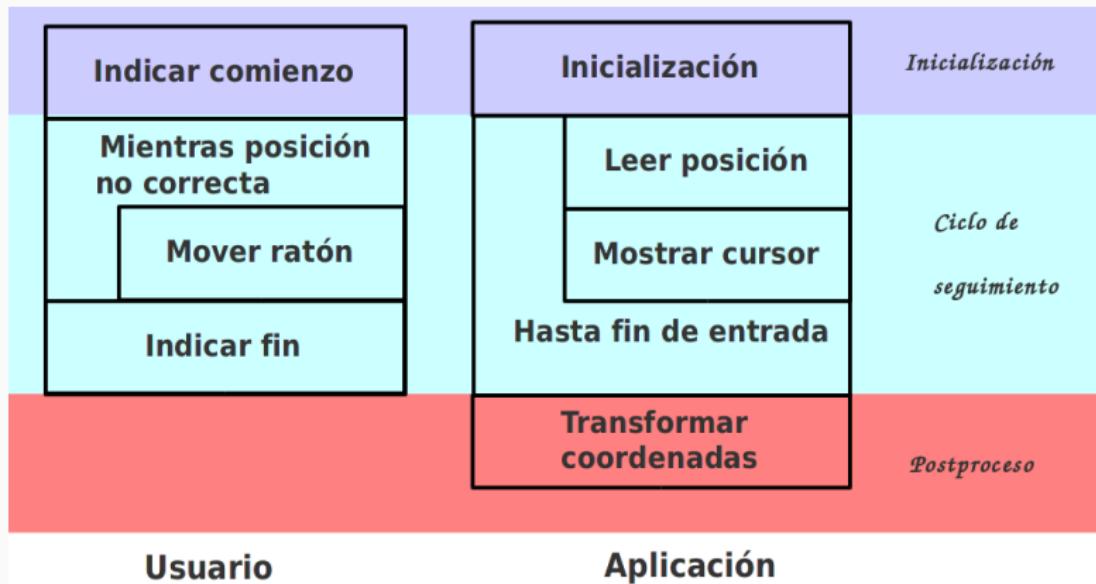
El proceso en el software de tratamiento se realiza en una estructura cíclica, que se corresponde con el ciclo de búsqueda que realiza el usuario. En este proceso podemos distinguir tres etapas:

- ▶ **Inicialización:** inicialización de estado.
- ▶ **Ciclo de seguimiento:** hasta que el usuario no indica que está satisfecho, se ejecuta un bucle en el cual: (a) se procesan los eventos de entrada y (b) se actualiza la información visual de realimentación y la posición seleccionada.
- ▶ **Postproceso:** se transforma la posición final.



# Acciones del usuario y del sistema.

Correspondencia entre operaciones del usuario y del sistema



## Modos de introducción de coordenadas

En visualización 2D de modelos planos (con una matriz de proyección ortogonal  $O$  y una matriz de dispositivo  $D$ ), usamos:

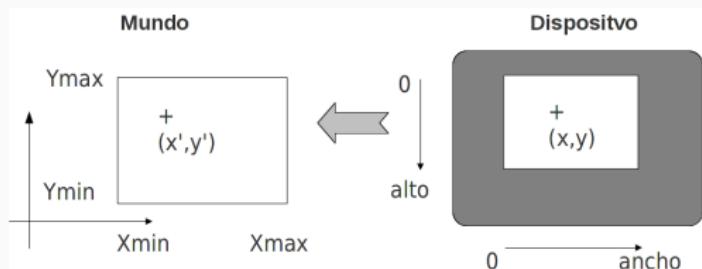
- ▶ **Transformación lineal:** se usa la matriz  $D^{-1}$  para pasar de DC a NDC, y luego  $O^{-1}$  para pasar de NDC a WC (la matriz usada es  $M = O^{-1}D^{-1}$ ).

En visualización de modelos 3D en pantalla, hay varias estrategias

- ▶ **Restricción a un plano:** el punto seleccionado se obtiene proyectando un punto 2D (obtenido con un cursor convencional 2D) sobre un plano 3D.
- ▶ **Cursor virtual 3D:** se manipulan las tres coordenadas en una vista ortográfica.
- ▶ **Tres cursores 2D ligados:** se usan tres vistas ortográficas perpendiculares.

# Posicionamiento 2D: transformación lineal

Usamos una transformación lineal para convertir desde  $(x_d, y_d)$  (en DC) hacia  $(x_w, y_w)$  (en WC) (las coordenadas DC son enteras, proporcionadas por el gestor de ventanas):



$$x_w = l + (r - l) \left( \frac{x_d + 1/2}{n_x} \right) \quad y_w = t - (t - b) \left( \frac{y_d + 1/2}{n_y} \right)$$

- ▶  $l, r$  son los límites del view-frustum 2D en X.
- ▶  $b, t$  son los límites del view-frustum 2D en Y.
- ▶  $n_x, n_y$  son el ancho y el alto (en pixels) del viewport.

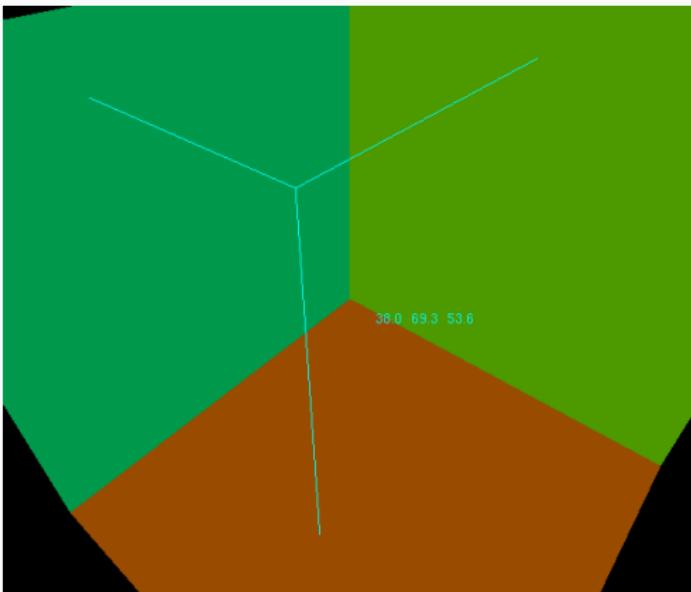
# Posicionamiento en 3D: tres cursos 2D ligados

Se usan tres proyecciones ortográficas perpendiculares:



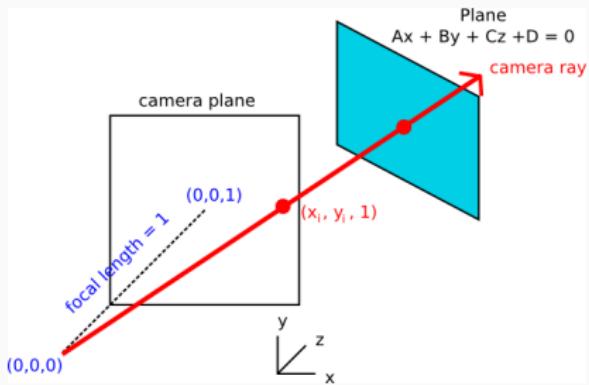
# Posicionamiento 3D: Cursor virtual 3D

Se usa un cursor virtual en 3D. El desplazamiento se realiza en el plano X-Z o en el X-Y en función de los botones del ratón que estén pulsados.



# Posicionamiento 3D: restricción a un plano

Si se restringe la posición a un plano que no sea perpendicular al de proyección se puede obtener una posición 3D por intersección de la recta que pasa por el punto introducido (en el plano de proyección y el centro de proyección con el plano



# Entrada de transformaciones

Las transformaciones geométricas se pueden definir a partir de puntos.

- ▶ Una traslación se puede definir por el vector que va de la posición original a la posición nueva
- ▶ Una rotación por el ángulo formado por el vector que va del punto de referencia a la posición actual con la horizontal

## Sección 4. Control de cámaras.

- 4.1. Modelo y operaciones de cámaras
- 4.2. Modos de cámara. La cámara de 3 modos.
- 4.3. Problema: animación de cámaras

## Control interactivo de la cámara: modos

Un modo de cámara es una de modificar interactivamente (con retroalimentación) los parámetros de la transformación de vista. Hay varios:

- ▶ **Modo orbital (o examinar):** útil para visualización de objetos, la cámara mantiene como punto de atención el origen de un objeto, y rota alrededor del mismo.
- ▶ **Modo primera persona:** útil para exploración de escenarios, manipulamosla la cámara cambiando su
  - ▶ **posición:** se desplaza la posición del observador en el sentido de los ejes de la cámara.
  - ▶ **orientación:** se rotan los ejes de la cámara entorno a la posición del observador (que no cambia).

esto da lugar a tres modos distintos de control de cámara.

## Cámara en modo orbital o examinar.

En este modo el usuario puede manipular la cámara virtual, pero siempre se mantiene el **punto de atención** proyectado en el centro de la imagen:

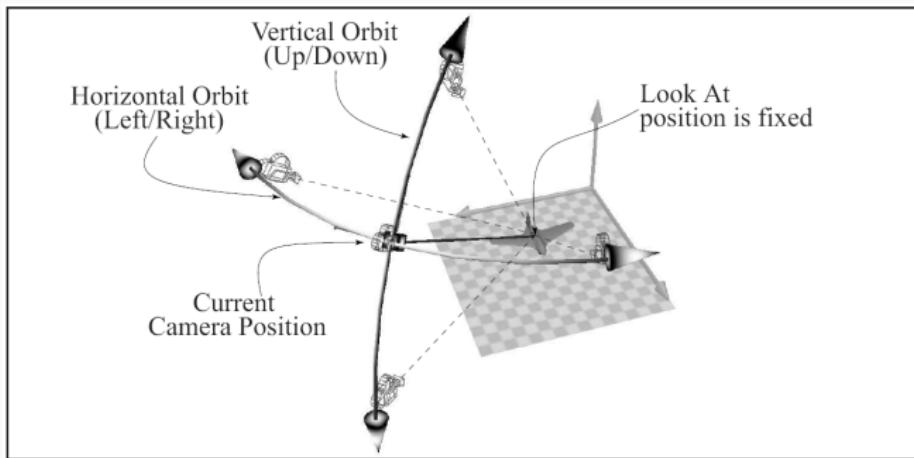


Figura obtenida de: K.Sung, P.Shirley, S.Baer **Essentials of Interactive Computer Graphics: Concepts and Implementation.**

Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 4. Control de cámaras

Subsección 4.1.  
Modelo y operaciones de cámaras.

## Parámetros de la cámara: Marco $\mathcal{V}$ y matriz de vista $V$ .

Suponemos que el marco de coordenadas de la vista  $\mathcal{V}$  tiene como componentes  $[\vec{x}_{\text{ec}}, \vec{y}_{\text{ec}}, \vec{z}_{\text{ec}}, \vec{o}_{\text{ec}}]$ , y dichas componentes están representadas en memoria por sus coordenadas homogéneas  $\mathbf{x}, \mathbf{y}_{\text{ec}}, \mathbf{z}_{\text{ec}}$  y  $\mathbf{o}_{\text{ec}}$  en el marco de coordenadas del mundo  $\mathcal{W}$ :

$$\begin{array}{rcl} \mathbf{x}_{\text{ec}} & = & (a_x, a_y, a_z, 0)^t \\ \mathbf{z}_{\text{ec}} & = & (c_x, c_y, c_z, 0)^t \end{array} \quad \begin{array}{rcl} \mathbf{y}_{\text{ec}} & = & (b_x, b_y, b_z, 0)^t \\ \mathbf{o}_{\text{ec}} & = & (o_x, o_y, o_z, 1)^t \end{array}$$

La matriz de vista  $V$  es la composición de una matriz de traslación por  $(-o_x, -o_y, -o_z)$  seguida de una matriz cuyas filas son las coordenadas de los ejes de  $\mathcal{V}$ , es decir:

$$V = \begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

## Parámetros de cámara: origen y punto de atención.

Podemos determinar una cámara usando el **punto de atención** y el **vector al origen**

- ▶ **Punto de atención  $\vec{a}_t$ :** es un punto del espacio que se va a proyectar en el centro de la imagen (está en el la rama negativa del eje Z de la cámara).
- ▶ **Vector al origen  $\vec{n}$ :** es el vector que va desde el punto de atención hasta el origen del marco de coordenadas de la cámara, es decir  $\vec{n} = \vec{o}_{ec} - \vec{a}_t$

Podemos expresar el marco de cámara en función de  $\vec{n}$  y  $\vec{a}_t$ :

$$\begin{aligned}\vec{z}_{ec} &= \vec{n} / \|\vec{n}\| & \vec{y}_{ec} &= \vec{z}_{ec} \times \vec{x}_x \\ \vec{x}_{ec} &= (\vec{y}_w \times \vec{z}_{ec}) / \|\vec{y}_w \times \vec{z}_{ec}\| & \vec{o}_{ec} &= \vec{a}_t + \vec{n}\end{aligned}$$

( $\vec{z}_{ec}$  es paralelo a  $\vec{n}$ , y el vector VUP siempre es  $\vec{y}_w$ ).

## Modelo de cámaras. Coordenadas esféricas.

Para representar una cámara podemos usar las tuplas  $\mathbf{a}_t$ ,  $\mathbf{s}$  y  $\mathbf{n}$ :

- ▶ Coordenadas del punto de atención ( $\mathbf{a}_t$ ), en WC.
- ▶ Coordenadas esféricas y cartesianas del vector al origen. Las coordenadas esféricas forman una tupla  $\mathbf{s} = (\alpha, \beta, r)$ , y las coordenadas cartesianas son una terna  $\mathbf{n} = (n_x, n_y, n_z)$ , ambas en WC.

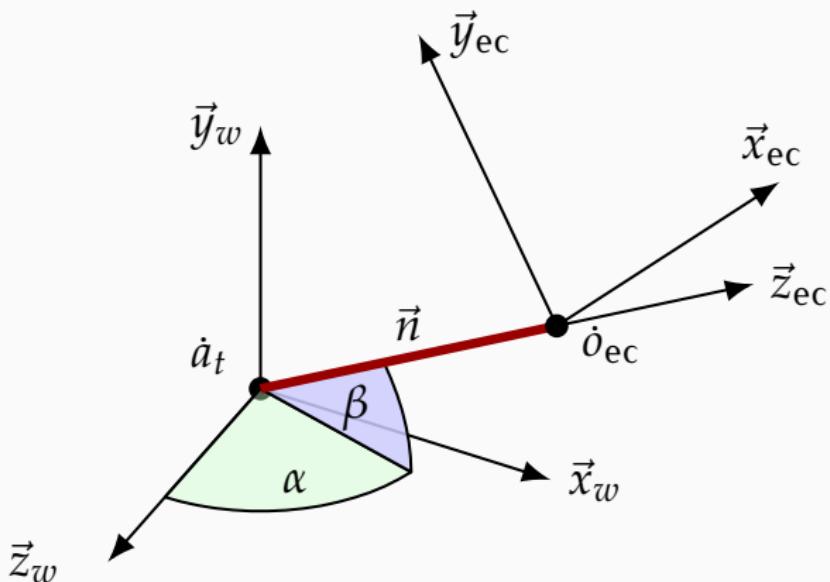
Se usa una representación redundante por comodidad. Se cumple:

$$\begin{array}{lcl} n_x & = & r(\sin \alpha)(\cos \beta) \\ n_y & = & r(\sin \beta) \\ n_z & = & r(\cos \alpha)(\cos \beta) \end{array} \quad \begin{array}{lcl} \alpha & = & \text{atan2}(n_x, n_z) \\ \beta & = & \text{atan2}(n_y, r_h) \\ r & = & \|\mathbf{n}\| \end{array}$$

donde:  $r_h = \sqrt{n_x^2 + n_z^2}$  y  $\text{atan2}(a, b)$  es como  $\arctan(a/b)$  pero teniendo en cuenta los signos y con valores en  $(-\pi, \pi]$ .

## Elementos del modelo de cámara

En esta figura se observan el marco de la cámara  $\mathcal{V}$  con origen en  $\dot{o}_{\text{ec}} = \dot{a}_t + \vec{n}$ , junto con el marco del mundo  $\mathcal{W}$  (trasladado a  $\dot{a}_t$ ):



## Cámaras interactivas de 3 modos

En general podemos hacer tres operaciones de modificación interactiva de una cámara:

- ▶ **Izquierda/derecha:** rotaciones en torno a  $\vec{y}_w$  o traslaciones paralelas a  $\vec{x}_{ec}$
- ▶ **Arriba/abajo:** rotaciones en torno a  $\vec{x}_{ec}$  o traslaciones paralelas a  $\vec{y}_{ec}$
- ▶ **Adelante/detrás:** traslaciones paralelas a  $\vec{n}$ .

Cada vez que se modifica el estado de una cámara:

1. se actualizan las tuplas  $\mathbf{a}_t, \mathbf{n}$  y  $\mathbf{s}$
2. se recalcula el marco de cámara (tuplas  $\mathbf{o}_{ec}, \mathbf{x}_{ec}, \mathbf{y}_{ec}$  y  $\mathbf{z}_{ec}$ ).

A las cámaras que se pueden actualizar así las llamamos **cámaras interactivas**.

# Clase base Camara: declaración

La clase base para cualquier cámara es la siguiente:

```
class Camara
{
public:
    // fija las matrices model-view y projection en el cauce
    void activar( Cauce & cauce ) ;
    // cambio el valor de 'ratio_vp' (alto/ancho del viewport)
    void fijarRatioViewport( const float nuevo_ratio ) ;
    // lee la descripción de la cámara (y probablemente su estado)
    virtual std::string descripcion() ;

protected:
    bool      matrices_actualizadas = false; // true si matrices actualizadas
    Matriz4f  matriz_vista = MAT_Ident(),      // matriz de vista
              matriz_proye = MAT_Ident();      // matriz de proyección
    float     ratio_vp     = 1.0 ;                // ratio viewport (alto/ancho)

    // actualiza matriz_vista y matriz_proye a partir de los parámetros
    // específicos de cada tipo de cámara
    virtual void actualizarMatrices() ;
};
```

# Representación de cámaras interactivas

La clase **CamaraInteractiva** es una clase derivada de **Camara** que puede ser manipulada con estas operaciones:

```
class CamaraInteractiva : public Camara
{
public:
    // operación izq/der (da) y arriba/abajo (db)
    virtual void desplRotarXY( const float da, const float db ) = 0 ;
    // operación acercar/alejar (dz)
    virtual void moverZ( const float dz ) = 0 ;
    // cambiar punto de atención manteniendo origen de cámara
    virtual void mirarHacia( const Tupla3f & paten ) ;
    // cambiar el modo de la camara al siguiente modo o al primero
    virtual void siguienteModo() ;
};
```

Esta clase define un interfaz pero no se puede instanciar. Se definen clases derivadas de ella. Usaremos dos: **CamaraOrbitalSimple** y **Camara3Modos**. Estudiaremos la segunda.

Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 4. Control de cámaras

Subsección 4.2.  
Modos de cámara. La cámara de 3 modos..

# La clase para cámaras de tres modos

La clase **Camara3Modos** implementa los tres modos:

```
class Camara3Modos : public CamaraInteractiva
{
public:
    Camara3Modos(); // cámara perspectiva por defecto
    Camara3Modos( ..... ) ; // cámara con parámetros iniciales específicos
    virtual void desplRotarXY( const float da, const float db ) ;
    virtual void moverZ( const float dz ) ;
    virtual void mirarHacia( const Tupla3f & nuevo_punto_aten );//pasa a m.ex.
    virtual void siguienteModo(); // ir al siguiente modo
    virtual Tupla3f puntoAtencion() ; // devuelve el punto de atención actual
private:
    virtual void actualizarMatrices(); // actualiza matriz V y P
    void actualizarEjesMCV() ; // actualiza ejes del MCV
    ModoCam modo_actual = ModoCam::examinar; // modo actual
    Tupla3f punto_atencion = { 0.0,0.0,0.0 } ; // punto de atención
    Tupla3f org_polares = { 0.0,0.0,r } ; // vector origen (esféricas)
    Tupla3f org_\ecartesianas = { 0.0,0.0,r } ; // vector origen (cartesianas)
    Tupla3f eje[3] = { {1,0,0},{0,1,0},{0,1,0} } ; // ejes del MCV
};
```

Aquí **r** es el radio inicial.

## Cámara en primera persona con traslaciones

En el modo de primera persona con traslaciones, la actualización de la cámara supone simplemente trasladar el origen del marco de cámara  $\mathbf{o}_{ec}$  y el punto de atención  $\mathbf{a}_t$  de forma solidaria:

- ▶ La operación **desplRotarXY**( $\Delta_a, \Delta_b$ ) supone:

1.  $\mathbf{a}_t = \mathbf{a}_t + \Delta_x \mathbf{x}_{ec} + \Delta_y \mathbf{y}_{ec}$
2.  $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$

- ▶ La operación **moverZ**( $\Delta_z$ ) supone:

1.  $\mathbf{a}_t = \mathbf{a}_t + \Delta_z \mathbf{z}_{ec}$
2.  $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$

Las tuplas  $\mathbf{s}, \mathbf{n}, \mathbf{x}_{ec}, \mathbf{y}_{ec}, \mathbf{z}_{ec}$  no cambian.

## Cámara primera persona con rotaciones

En este caso se usan rotaciones entorno al origen de la cámara  $\mathbf{o}_{ec}$ . Dichas rotaciones se implementan modificando los ángulos  $\alpha$  y  $\beta$  que hay en  $\mathbf{s}$ . El movimiento en Z es similar al anterior

- ▶ La operación **desplRotarXY**( $\Delta_a, \Delta_b$ ) supone:
  1.  $\mathbf{s} = \mathbf{s} + (\Delta_a, \Delta_b, 0)$  (incrementa o decrementa  $\alpha$  y  $\beta$ )
  2.  $\mathbf{n}' = \text{Cartesianas}(\mathbf{s})$  (es el nuevo valor de  $\mathbf{n}$ ).
  3.  $\mathbf{a}_t = \mathbf{a}_t + (\mathbf{n}' - \mathbf{n})$
  4.  $\mathbf{n} = \mathbf{n}'$
  5. actualizar  $\mathbf{x}_{ec}, \mathbf{y}_{ec}$  y  $\mathbf{z}_{ec}$  (el origen  $\mathbf{o}_{ec}$  no cambia)
- ▶ La operación **moverZ**( $\Delta_z$ ) supone:
  1.  $\mathbf{a}_t = \mathbf{a}_t + \Delta_z \mathbf{z}_{ec}$
  2.  $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$  (los vectores  $\mathbf{x}_{ec}, \mathbf{y}_{ec}$  y  $\mathbf{z}_{ec}$  no cambian)

Este modo es típico en videojuegos FPS (*First Person Shooter*), normalmente sin movimientos de rotación en vertical ( $\Delta_b = 0$ )

## Cámara en modo orbital o examinar

En este caso se usan rotaciones entorno al punto de atención  $\mathbf{a}_t$ . El movimiento en Z nos acerca o aleja a dicho punto (cambia el valor de  $r$  que hay en  $\mathbf{s}$ ):

- ▶ La operación **desplRotarXY**( $\Delta_a, \Delta_b$ ) supone:
  1.  $\mathbf{s} = \mathbf{s} + (\Delta_a, \Delta_b, 0)$  (incrementa o decrementa  $\alpha$  y  $\beta$ )
  2.  $\mathbf{n} = \text{Cartesianas}(\mathbf{s})$  (es el nuevo valor de  $\mathbf{n}$ ).
  3.  $\mathbf{o}_{\text{ec}} = \mathbf{a}_t + \mathbf{n}$ , actualizar  $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}, \mathbf{z}_{\text{ec}}$
- ▶ La operación **moverZ**( $\Delta_z$ ) supone:
  1.  $r = r_{\min} + (r - r_{\min})(1 + \epsilon)^{\Delta_z}$
  2.  $\mathbf{n} = \text{Cartesianas}(\mathbf{s})$
  3.  $\mathbf{o}_{\text{ec}} = \mathbf{a}_t + \mathbf{n}$  (los vectores  $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$  y  $\mathbf{z}_{\text{ec}}$  no cambian)

El radio nunca es inferior a  $r_{\min} > 0$ . Para  $\Delta_z \geq 1$  aleja, y para  $\Delta_z \leq -1$  acerca ( $\Delta_z$  nunca debe estar en  $(-1, 1)$ ).

## Apuntar la cámara hacia un punto

Esta operación supone hacer que el punto de atención se fije a unas coordenadas  $\mathbf{c}$  dadas, sin modificar el origen de cámara

1.  $\mathbf{n} = \mathbf{n} + \mathbf{c} - \mathbf{a}_t$
2.  $\mathbf{s} = \text{Esfericas}(\mathbf{n})$
3.  $\mathbf{a}_t = \mathbf{c}$
4. actualizar  $\mathbf{x}_{\text{ec}}$ ,  $\mathbf{y}_{\text{ec}}$  y  $\mathbf{z}_{\text{ec}}$  (el origen  $\mathbf{o}_{\text{ec}}$  no cambia)

La función Esfericas produce las coordenadas esféricas a partir de las cartesianas (es la inversa de **Cartesianas**).

Esta operación permite seleccionar un objeto y que pase a ocupar el centro de la imagen (fijando el punto de atención a un punto central de dicho objeto).

Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 4. Control de cámaras

Subsección 4.3.  
Problema: animación de cámaras.

# Problema: animación de cámaras

## Problema 4.1.

Supongamos que queremos visualizar una secuencia de frames, en los cuales la cámara va cambiando. Para ellos queremos escribir el código de una función que fija la matriz de vista en el cauce. La función acepta como parámetro un valor real  $t$ , que es el tiempo en segundos transcurrido desde el inicio de la animación. Suponemos que la animación dura  $s$  segundos en total.

En ese tiempo el observador de cámara se desplaza con un movimiento uniforme desde un punto de coordenadas de mundo  $\mathbf{o}_0$  (para  $t = 0$ ) hasta un punto destino  $\mathbf{o}_1$  (para  $t = 1$ ). Además el punto de atención de la cámara también se desplaza desde  $\mathbf{a}_0$  hasta  $\mathbf{a}_1$ . Durante toda la animación, el vector VUP es  $(0, 1, 0)$ .

Escribe el pseudo-código de la citada función.

## Sección 5. Selección.

- 5.1. Introducción. Métodos de selección
- 5.2. Problemas: selección por intersecciones
- 5.3. Modo de selección de OpenGL
- 5.4. Selección con *frame-buffer object* invisible.

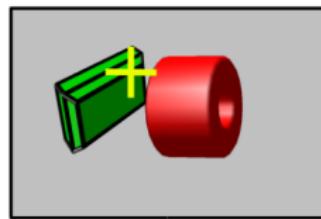
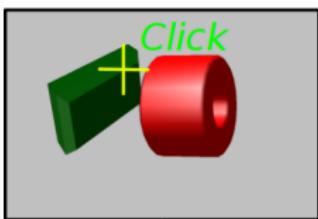
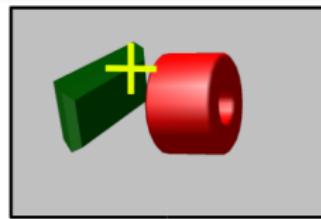
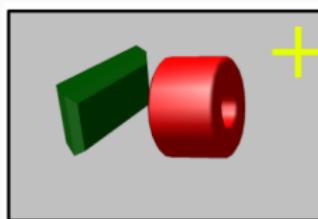
Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 5. Selección

Subsección 5.1.  
Introducción. Métodos de selección.

# Selección de objetos

La selección permite al usuario identificar componentes u objetos de la escena:

- ▶ Se necesita para identificar el objeto sobre el que actuan las operaciones de edición.
- ▶ Suele realizarse como posicionamiento seguido de búsqueda.



-> Id: 23

# Identificadores de objetos

Para poder realizar la selección los componentes de la escena deben tener asociados identificadores numéricos (enteros), a distintos niveles:

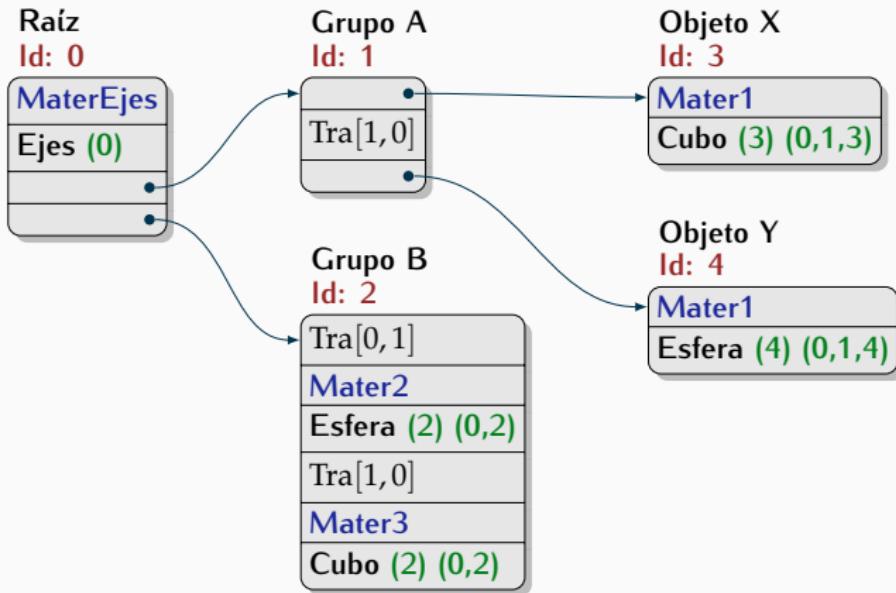
- ▶ **Triángulos:** cada triángulo o cara tiene asociado un entero, permite seleccionarlos para operaciones de edición de bajo nivel en las mallas.
- ▶ **Mallas:** cada malla de la escena puede tener un identificador único.
- ▶ **Grupos de objetos:** los grupos de mallas (o, en general, grupos de objetos arbitrarios), pueden tener asociados identificadores, permiten operar con partes complejas de la escena.

Para ediciones de alto nivel en objetos o partes de la escena, lo más simple es **asignar identificadores enteros a los nodos del grafo de escena**.

# Grafo de escena con identificadores

Cada nodo del grafo guarda un identificador entero (no negativo).

Cada primitiva u objeto terminal tendrá siempre asociado un identificador (o bien una lista de identificadores), en la figura se han anotado en verde:



# Procedimiento y métodos de selección

Para hacer la selección se pueden dar estos pasos:

1. El usuario selecciona un pixel en pantalla .
2. Se buscan los identificadores de los elementos (triángulos, mallas u objetos) que se proyectan en el centro de dicho pixel (o de pixels cercanos).

La búsqueda se puede hacer de varias formas:

- ▶ **Ray-casting:** calculando intersecciones de un rayo (semirecta con origen en  $\mathbf{o}_c$  y pasando por el centro del pixel) con los objetos de la escena.
- ▶ **Clipping:** (*recortado*) calculando que objetos están parcial o totalmente dentro de un *view-frustum* pequeño centrado en el pixel.
- ▶ **Rasterización:** visualizar la escena por rasterización usando identificadores en el lugar de los colores. Permite obtener el color (un identificador de objeto) del pixel en cuestión.

## Selección por rasterización en OpenGL (1/2)

En OpenGL podemos usar visualización (por rasterización), de dos formas:

- ▶ **Modo selección de OpenGL:** se usa una funcionalidad de OpenGL, específica para este fin:
  - ▶ Visualizar con el **modo de selección** activado, OpenGL usa identificadores en lugar de colores (tomados de la **pila de nombres**).
  - ▶ Los identificadores visualizados se registran en un **buffer de selección** (en memoria) específicamente destinado a contenerlos.

(funcionalidad obsoleta desde OpenGL 3.0).

## Selección por rasterización en OpenGL (2/2)

En OpenGL podemos usar visualización (por rasterización), de dos formas:

- ▶ **Frame-buffer no visible.** Se usa algún **frame buffer object** (array de colores de pixels en la memoria de la GPU) distinto del que se está visualizando en la ventana:
  - ▶ Se debe desactivar iluminación y texturas, y visualizar con colores planos (colores obtenidos de los identificadores de los objetos).
  - ▶ Al final se leen los colores (identificadores de objetos proyectados) en el pixel y alrededores.

OpenGL ofrece la posibilidad de definir y activar *Frame Buffer Objects* (FBOs) del tamaño y características que queramos. Una vez activado un FBO, se rasteriza sobre esa memoria, no sobre pantalla.

Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 5. Selección

Subsección 5.2.  
Problemas: selección por intersecciones.

# Problema: intersección rayo-triángulo (1/2)

## Problema 4.2.

Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un *rayo* (una semirrecta que pasa por el centro de un pixel) y cada uno de los triángulos de la malla.

Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

- ▶ El rayo tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla **o**, y como vector de dirección la tupla **d** (la suponemos normalizada).
- ▶ Las coordenadas del mundo de los vértices del triángulo son **v<sub>0</sub>**, **v<sub>1</sub>** y **v<sub>2</sub>**.
- ▶ El algoritmo debe de indicar si hay intersección o no, y, en caso de que la haya, calcular las coordenadas del mundo del punto de intersección.

(ver la siguiente transparencia).

# Problema: intersección rayo-triángulo (2/2)

## Problema 4.2. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe  $t > 0$  tal que el punto  $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$  está en dicho plano. Equivale a decir que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es perpendicular a la normal al plano.
2. El punto  $\mathbf{p}_t$  citado arriba está dentro del triángulo. Es decir, hay dos valores reales no negativos  $a$  y  $b$  (con  $0 \leq a + b \leq 1$ ) tales que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es igual a  $a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$ .

(a los tres valores  $a$ ,  $b$  y  $c \equiv 1 - b - a$  se les llama *coordenadas baricéntricas* de  $\mathbf{p}_t$  en el triángulo, se usan en ray-tracing).

# Problema: calculo de rayos para selección

## Problema 4.3.

Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por centro del pixel donde se ha hecho click. Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

- ▶ Tenemos una vista perspectiva, y conocemos los 6 valores  $l, r, t, b, n, f$  usados para construir la matriz de proyección.
- ▶ También conocemos el marco de coordenadas de vista, es decir, las tuplas  $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$  y  $\mathbf{o}_{\text{ec}}$  con los versores y la tupla  $\mathbf{\delta}_{\text{ec}}$  con el punto origen (todos en coordenadas del mundo).
- ▶ El viewport tiene  $w$  columnas y  $f$  filas de pixels. Se ha hecho click en el pixel de coordenadas enteras  $x_p$  e  $y_p$

El algoritmo debe producir como salida las tuplas  $\mathbf{o}$  y  $\mathbf{d}$  (normalizado) que definen el rayo.

Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 5. Selección

Subsección 5.3.  
*Modo de selección de OpenGL.*

## Modo selección de OpenGL (1/2)

OpenGL contempla un modo de visualización llamado **modo de selección**.

- ▶ Se puede activar con **glRenderMode( GL\_SELECT )** (se debe usar la constante **GL\_RENDER** para volver al modo habitual).
- ▶ Con este modo activo, al enviar primitivas no se visualizan en pantalla.
- ▶ OpenGL mantiene una variable interna con una lista de enteros, llamada **pila de nombre**, manipulable por la aplicación.
- ▶ Cuando se envia una primitiva desde la aplicación, se le asocia la lista de enteros en la pila de nombre actual de OpenGL. A la lista se le llama **nombre** de la primitiva.

## Modo selección de OpenGL (2/2)

OpenGL contempla un modo de visualización llamado **modo de selección**.

- ▶ OpenGL registra cada nombre distinto de cada primitiva que se visualiza en cualquier pixel. Para cada nombre se almacenan en el **buffer de selección**:
  - ▶ Lista de enteros que forman el nombre.
  - ▶ Rango de profundidades en Z para todas las apariciones de ese nombre.

# Manipulación de la pila del nombre

La aplicación debe asegurarse que la pila (LIFO) del nombre contiene los identificadores adecuados antes de visualizar cada objeto, para ello se pueden usar estas operaciones sobre dicha pila:

- ▶ **glLoadName( i )**  
sustituye entero en el tope de la pila por **i**
- ▶ **glPushName( i )**  
apila el entero **i**
- ▶ **glPopName()**  
desapila un entero (si la lista no está vacía)
- ▶ **glInitNames()**  
vacia la pila de nombres

en todos los casos, **i** es de tipo **GLenum** (normalmente equivalente a **unsigned int**)

# Ejemplo de manipulación de la pila del nombre

Dibujo de un tablero de ajedrez, el nombre cada casilla es el par (número de fila, número de columna).

```
for( num_fila = 0 ; num_fila < 8 ; num_fila++ )
{
    glTranslatef( 1.0, 0.0, 0.0 );
    glPushMatrix();
    glPushName( num_fila );

    for( num_colu = 0 ; num_colu < 8 ; num_colu++ )
    {
        glTranslatef( 0.0, 1.0, 0.0 );
        glPushName( num_colu );
        DibujaCasilla(); // aquí, la pila contiene: ( num_fila, num_colu )
        glPopName();
    }
    glPopName();
    glPopMatrix();
}
```

Suponemos que **DibujaCasilla** dibuja un cuadrado de lado unidad en el plano XY ( $z = 0$ ), con centro en  $(0.5, 0.5)$ .

## Contenido del buffer de selección

El buffer de selección es un vector de entradas de tipo **GLsizei** (normalmente equivalente a **int**). Las entradas se dividen en bloques de tamaño variable, cada bloque corresponde a un nombre distinto aparecido el visualizar. En cada bloque se incluye esta información del nombre:

- ▶ El número de enteros que forman el nombre.
- ▶ El intervalo de profundidades del nombre: mínimo seguido de máximo (dos entradas).
- ▶ La lista de enteros que forman el nombre (uno por entrada).

0	1	2	3	...	$2+n_1$	$3+n_1$	$4+n_1$	$5+n_1$	$6+n_1$	...	$5+n_1+n_2$	$6+n_1+n_2$	$7+n_1+n_2$	...
$n_1$	MinZ1	MaxZ1	$id_1$	...	$id_1$	$n_2$	MinZ2	MaxZ2	$id_2$	...	$id_2$	$n_3$	MinZ3	...

Durante la rasterización, OpenGL registra nombres (y profundidades) en el buffer de selección, mientras haya memoria suficiente para ello.

# Buffer de selección: implementación

Podemos usar una clase (singleton) para representar el buffer de selección, de esta forma:

```
class BufferSeleccion
{
public:
    const GLsizei tamB = 1024 ; // tamaño del buffer (== 4 Kb, p.ej.)
    int vec[ tamB ];          // vector de entradas enteras (buffer)
    // variables calculadas en finModoSel:
    int numNombres ;          // número de nombres (bloques) en el buffer
    int * nombreMin ;          // puntero al nombre más cercano (dentro de vec)
    int longNombreMin ;        // longitud de nombreMin
    // métodos:
    BufferSeleccion() {}      // constructor por defecto (no hace nada)
    void inicioModoSel();     // fijar buffer en OpenGL, pasar al modo selección
    void finModoSel();         // analizar buffer, pasa a modo render
};
```

La constante **tamB** determina la cantidad de memoria disponible para el buffer, de forma que OpenGL nunca produce desbordamiento al escribir durante la rasterización en modo selección.

# Procesamiento del buffer de selección

El número total de bloques se obtiene al llamar a **glRenderMode** estando en modo **GL\_SELECT**. Se puede hacer al volver al modo *render*.

```
void BufferSeleccion::inicioModoSel() // inicio del modo de selección
{
    glSelectBuffer( vec, tamB ); // decirle a OpenGL donde está el buffer de selección
    glRenderMode( GL_SELECT ); // entrar en modo selección
    glInitNames(); // vacía la pila de nombres
}
void BufferSeleccion::finModoSel( ) // fin del modo de selección
{
    int profZmin; // mínima profundidad encontrada hasta el momento
    int * bloque = vec; // puntero a la base del bloque actual en el buffer
    numNombres = glRenderMode( GL_RENDER );// leer núm. de bloques, volver modo render
    for( i = 0 ; i < numNombres ; i++ ) // para cada nombre (bloque) en el buffer:
    {
        if ( i == 0 || bloque[1] < profZmin ) // si la profundidad es menor:
        {
            profZmin = bloque[1]; // guardar nueva prof. mínima
            longNombreMin = bloque[0]; // registrar longitud del nombre
            nombreMin = &(bloque[3]); // registrar puntero a nombre mínimo.
        }
        bloque += bloque[0]+3; // avanzar puntero a la base del bloque siguiente
    }
}
```

# El view-frustum centrado en el pixel

Esta función fija un view-frustum de 5x5 pixels centrado en el pixel cuyas coordenadas se pasan como parámetro:

```
void FijarViewFrustumSel( int xpix, int ypix )
{
    GLint viewport[4];
    glGetIntegerv( GL_VIEWPORT, viewport );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPickMatrix( xpix, viewport[3]-ypix, 5.0, 5.0, viewport[0] );
    glFrustum( left, right, bottom, top, near, far ); // fijar view-frustum
}
```

- ▶ El efecto neto es fijar la matriz de proyección de OpenGL.
- ▶ Se deben usar los mismos parámetros del view-frustum usados para la visualización de la escena (**left,right,bottom**, etc....).
- ▶ Se hace uso de la librería GLU, para multiplicar por una matriz que restringe el tamaño del view-frustum

# Gestión de eventos de click

Si queremos (por ejemplo) hacer selección con el botón izquierdo del ratón, podemos definir la función **Seleccion** que aparece abajo, e invocarla desde la función gestora de evento de botón de ratón:

```
BufferSeleccion buffer ;  
  
void SeleccionBufferTrasero( int xpix, int ypix )  
{  
    buffer.inicioModoSel();           // entrar en modo selección  
    FijarViewFrustumSel( xpix, ypix );// fijar view-frustum centrado en pixel  
    VisualizarEscena();             // visualizar la escena  
    buffer.finModoSel();            // analizar buffer de selec., volver modo render  
    // imprimir nombre seleccionado (lista de enteros)  
    cout << "nombre seleccionado: " ;  
    for( i = 0 ; i < buffer.longNombreMin ; i++ )  
        cout << " " << buffer.nombreMin[i] ;  
}  
void FGE_BotonRaton( GLFWwindow* window, int button, int action, int mod)  
{  
    if ( button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS )  
    {  
        double xf,yf ; glfwGetCursorPos( window, &xf, &yf );  
        SeleccionBufferTrasero( int(xf), int(yf) );  
    }  
}
```

Informática Gráfica, curso 2021-22.  
Teoría. Tema 4. Interacción. Animación.  
Sección 5. Selección

Subsección 5.4.  
Selección con *frame-buffer object* invisible..

## Visualización sobre un frame-buffer object invisible

Si no se quiere usar funcionalidad obsoleta, se puede utilizar directamente visualización sobre un frame-buffer distinto del que se está viendo en pantalla. Se puede hacer de dos forma:

- ▶ Creando un objeto OpenGL de tipo **frame-buffer object** (FBO), y haciendo rasterización con ese objeto como imagen de destino (*rendering target*).
- ▶ Usando el modo de **doble buffer**:
  - ▶ En este modo siempre existen dos FBOs creados por OpenGL: un *buffer trasero* (*back buffer*), que es donde se visualizan las primitivas, y un *buffer delantero* (*front buffer*), que es el que se visualiza en pantalla.

Usaremos la primera opción al no depender de la existencia de doble buffer.

# Visualización identificadores

La visualización sobre el frame-buffer no visible requiere:

- ▶ **Codificación de identificadores:** se necesita codificar los identificadores como colores (R,G,B), y usar esos colores en lugar de los materiales de los objetos.
- ▶ **Cambio de colores:** durante la visualización es necesario cambiar el color actual de OpenGL (antes de cada objeto), usando esos colores, con total precisión numérica.
- ▶ **Modo de visualización:** es necesario desactivar iluminación y texturas, activar sombreado plano y visualizar con todas las primitivas (triángulos) llenos.

Esto constituye un nuevo modo de visualización, lo llamaremos **modo de identificadores**.

# Codificación y cambio de color

El cambio de color de OpenGL debe hacerse evitando errores numéricos que podrían darse si se usan ternas RGB en coma flotante.

- ▶ Para evitar errores de precisión, se usa una variante de **glColor** que acepta 3 datos de tipo **GLubyte** ( $\equiv \text{unsigned char}$ ).
- ▶ Los identificadores son enteros sin signo (tipo **unsigned** de C/C++, usualmente 4 bytes), con valores entre 0 y  $2^{24} - 1$  (el byte más significativo es 0, se usan los tres menos significativos).

Se puede hacer el cambio de color con una función como esta:

```
void FijarColorIdent( const unsigned ident ) // 0 ≤ ident < 224
{
    const unsigned char
        byteR = ( ident           ) % 0x100U,    // rojo = byte menos significativo
        byteG = ( ident / 0x100U ) % 0x100U,    // verde = byte intermedio
        byteB = ( ident / 0x10000U ) % 0x100U;  // azul = byte más significativo

    glColor3ub( byteR, byteG, byteB );          // cambio de color en OpenGL.
}
```

## Lectura de colores

Para leer los colores se puede usar la función **glReadPixels**, que lee los colores de un bloque de pixels en el framebuffer activo para escritura.

- ▶ Leeremos un bloque con un único pixel.
- ▶ Se leen tres valores **unsigned char** en orden R,G,B.
- ▶ Se reconstruye el identificador **unsigned**, conocidos los tres bytes.

Lo podemos hacer así:

```
unsigned LeerIdentEnPixel( int xpix, int ypix )
{
    unsigned char bytes[3] ; // para guardar los tres bytes
    // leer los 3 bytes del frame-buffer
    glReadPixels( xpix,ypix, 1,1, GL_RGB,GL_UNSIGNED_BYTE, (void *)bytes );
    // reconstruir el identificador y devolverlo:
    return bytes[0] + ( 0x100U*bytes[1] ) + ( 0x10000U*bytes[2] ) ;
}
```

## Frame-buffer objects (FBOs)

OpenGL permite crear y gestionar FBOs alojados en la memoria de la GPU

- ▶ Cada FBO tiene asociado un identificador entero único, no negativo.
- ▶ El FBO inicial (que se visualiza en la ventana de la aplicación) tiene asociado el identificador 0 y está creado al inicio
- ▶ Es posible crear un FBO, indicando su tamaño.
- ▶ Un FBO puede tener asociado
  - ▶ Un array de colores de pixels (*color buffer*): es una textura de OpenGL alojada en la GPU.
  - ▶ Un array de profundidades (*render buffer*): contiene la profundidad del objeto proyectado en cada pixel (es el Z-buffer usado para EPO)

# La clase Framebuffer

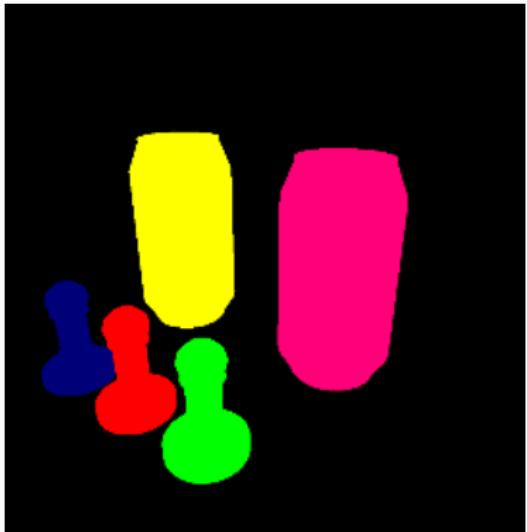
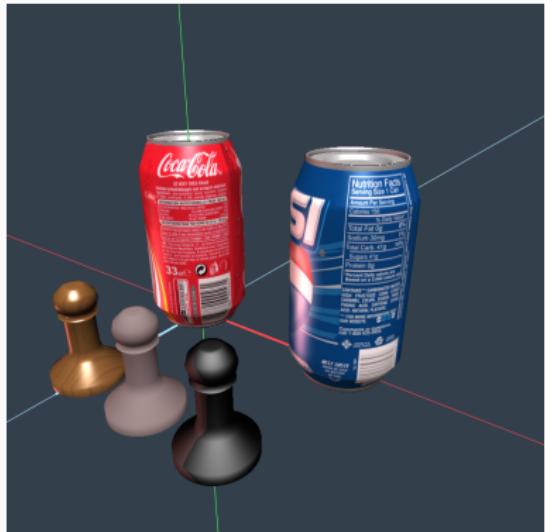
Encapsula un identificador de FBO, y su ancho y alto. Permite activarlo, redimensionarlo y consultar un pixel (**leerPixel**). Al activarlo, se rasteriza sobre este FBO.

```
class Framebuffer
{
public:
    // crear un FBO de este tamaño
    Framebuffer( const int pancho, const int palto );
    // activar, recreando el FBO si hay cambio de tamaño
    void activar( const int pancho, const int palto );
    void desactivar(); // desactivar (activa el 0)
    ~Framebuffer(); // llama a 'destruir'

private:
    void inicializar( const int pancho, const int palto );
    void destruir(); // libera memoria en la GPU
    GLuint fboId = 0, // identificador del framebuffer (0 si no creado)
           textId = 0, // identificador de la textura de color
           rbId = 0; // identificador del z-buffer
    int    ancho = 0, // ancho del framebuffer, en pixels
          alto = 0; // alto del framebuffer, en pixels
};
```

# Escena y FBO con identificadores

Se visualiza una escena en modo normal (izquierda) y en modo selección (derecha), con la misma cámara. Se han seleccionado los identificadores para que se correspondan con colores RGB distinguibles. Los ejes no son seleccionables.



## Sección 6. Animación.

# Animación

Generar animaciones implica:

- ▶ Regenerar la imagen periodicamente (al menos 30 veces por segundo)
- ▶ Modificar la pose de los objetos de la escena

# Keyframe

El animador define dos configuraciones del modelo.

El sistema calcula los fotogramas intermedios (in-betweens) por interpolación.



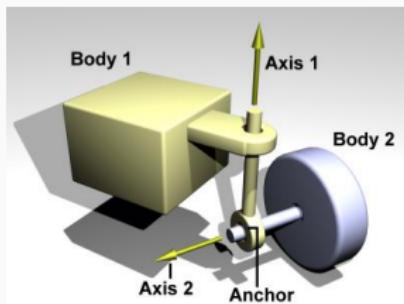
(c) Walt Disney Company, from "The Illusion of Life"

# Simulación física

Para escenas con modelos físicos simples se puede calcular la configuración del escenario en cada fotograma usando las leyes de la mecánica clásica.

Existen librerías específicas para realizar esta simulación:

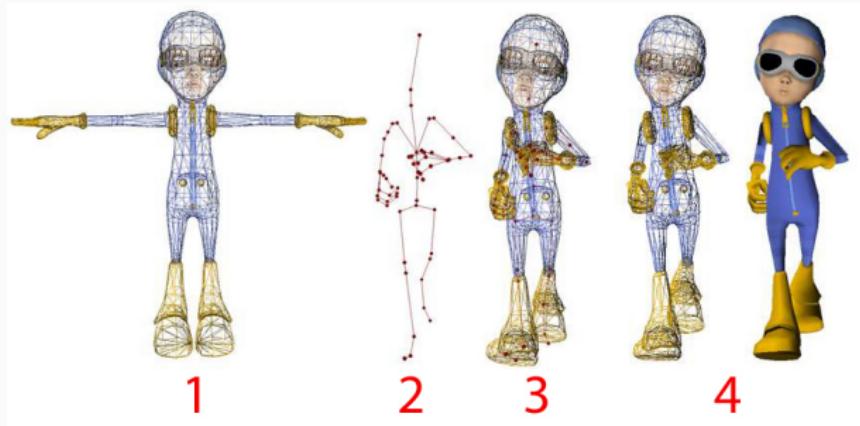
- ▶ ODE: Open Dynamic Engine.
- ▶ Newton: Newton Physics Engine
- ▶ Bullet: Physics Library



# Esqueletos

Para conseguir que la animación de personajes resulte plausible se suelen usar esqueletos.

Un esqueleto es un modelo simplificado del personaje, formado por segmentos rígidos unidos por articulaciones.



# Animación procedural

El comportamiento del objeto se describe mediante un procedimiento.

Es útil cuando el comportamiento es fácil de generar pero difícil de simular físicamente (p.e. la rotura de un vidrio).



Fin de la presentación.



UNIVERSIDAD  
DE GRANADA

# Informática Gráfica:

## Teoría. Tema 5. Realismo en Rasterización.

### Ray-tracing.

---

Carlos Ureña

2021-22

Grado en Informática y Matemáticas  
Dpt. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

## Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

### Índice.

1. Técnicas realistas en rasterización
2. Ray tracing

## Sección 1.

### Técnicas realistas en rasterización.

- 1.1. Mipmaps.
- 1.2. Perturbación de la normal
- 1.3. Sombras arrojadas
- 1.4. Superficies transparentes. Refracción.
- 1.5. Superficies especulares

Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 1. Técnicas realistas en rasterización

Subsección 1.1.

Mipmaps..

# La resolución de las texturas.

En muchos casos la resolución a la que se ve la textura no coincide con la de la imagen sintetizada:

- ▶ Si la resolución es menor (objeto lejano), en un pixel se proyectan muchos texels.
- ▶ Si la resolución es mayor, un texel se proyecta en muchos pixels.

en ambos casos el efecto es una pérdida de realismo. El primer problema se puede solucionar usando anti-aliasing, o de forma mucho más eficiente usando la técnica de *mipmaps*

## Creación de los *mipmaps*

Un *mipmap* (de *multum in parvo maps*) es una serie de  $n + 1$  texturas (bitmaps) obtenida a partir de una imagen o textura de  $2^n \times 2^n$  texels.

- ▶ La primera imagen (imagen  $M_0$ ) coincide con la original
- ▶ La  $i$ -ésima imagen ( $M_i$ ) tiene como resolución  $2^{n-i} \times 2^{n-i}$  texels.
- ▶ Cada texel de la imagen  $i + 1$  ( $M_{i+1}$ ) se obtiene a partir de cuatro texels de la imagen número  $i$ , promediándolos:

$$M_{i+1}[j, k] = \frac{1}{4} ( M_i[2j, 2k] + M_i[2j + 1, 2k] + M_i[2j, 2k + 1] + M_i[2j + 1, 2k + 1] )$$

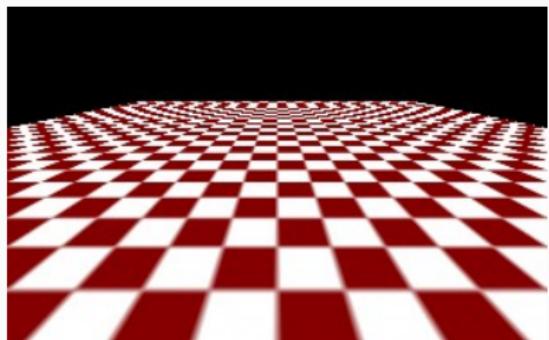
## Acceso a los *mipmaps*

Durante el sombreado, en cada punto  $\mathbf{p}$  a sombrear es necesario saber que versión de la trextura debemos de leer:

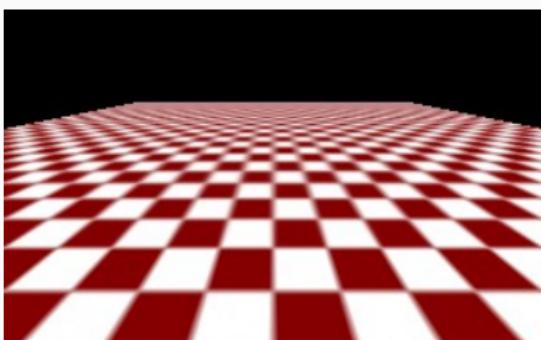
- ▶ Se usará la textura  $M_i$ , donde  $i$  crece linealmente con el logaritmo de la distancia  $d$  entre  $\mathbf{p}$  y el observador (menos resolución a mayor distancia).
- ▶ Esta solución puede presentar cambios bruscos de la resolución al pasar bruscamente de una resolución a otra en pixels cercanos. La solución consiste en interpolar entre las dos texturas más apropiadas en función de  $\log(d)$

## Ejemplo de *mipmapping*

En la parte más cercana se usa en ambos casos la textura original. Con mipmaps, a distancias mayores se usan sucesivamente texturas de menos resolución.



sin mipmapping



con mipmapping

[http://www.flipcode.com/archives/Advanced\\_OpenGL\\_Texture\\_Mapping.shtml](http://www.flipcode.com/archives/Advanced_OpenGL_Texture_Mapping.shtml)

Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 1. Técnicas realistas en rasterización

Subsección 1.2.

Perturbación de la normal.

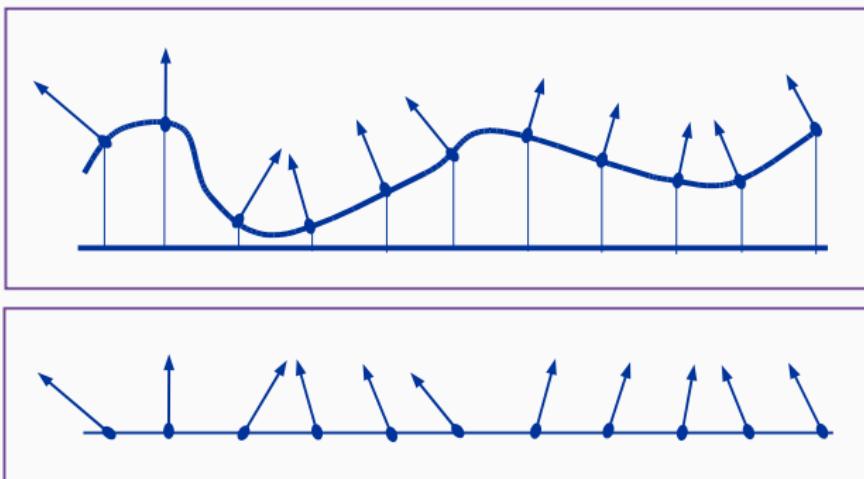
## Rugosidades a pequeña escala

Algunos tipos de superficies presentan cambios de orientación a pequeña escala (rugosidades)

- ▶ Esto se puede reproducir con mallas de polígonos con muchos polígonos pequeños, o con polígonos de detalle de diferente orientación. En cualquier caso, la complejidad en tiempo y espacio del proceso de rendering es muy alta.
- ▶ Una solución consiste en usar un textura para modificar a pequeña escala el vector normal que se usa en el MIL, a esto se le llama *mapas de perturbación de la normal (bump-maps)*.

# Rugosidades definidas por un campo de alturas

Es necesario usar una función real  $f_h$ , tal que para cada par de coordenadas de textura  $(u, v)$ , el valor real  $f_h(u, v)$  se interpreta como la altura de la superficie rugosa respecto del plano del polígono en el punto de coordenadas de textura  $(u, v)$



## Codificación del campo de alturas

Para evaluar  $f_h(u, v)$  dados  $u$  y  $v$  se pueden usar dos opciones:

- ▶  $f_h$  puede representarse como una función con una expresión analítica conocida y evaluable con algún algoritmo que tiene a  $u$  y  $v$  como datos de entrada (se llaman *texturas procedurales*).
- ▶ la opción más usual es que  $f_h$  este codificada como una textura cuyos texels son valores escalares (tonos de gris) que codifican la altura. Para evaluar  $f_h(u, v)$  se usa el mismo método visto para acceso a texturas en la sección anterior (se usan los texels más cercanos a  $(u, v)$  en el espacio de coords. de textura).

# Derivadas del campo de alturas

El procedimiento de perturbación de la normal usa como parámetros las derivadas parciales de  $f_h$  ( $d_u$  y  $d_v$ ):

$$d_u = \frac{\partial f_t(u, v)}{\partial u} \quad d_v = \frac{\partial f_t(u, v)}{\partial v}$$

- ▶ si  $f_h$  está definido por una función analítica conocida y derivable, estas derivadas se pueden conocer evaluando las expresiones de las derivadas parciales de  $f_h$ .
- ▶ si  $f_h$  está codificada con una textura, se usan diferencias finitas

## Aproximación de las derivadas por diferencias finitas

Cuando el campo de alturas  $f_t$  se codifica con una textura de grises, los valores de  $d_u$  y  $d_v$  se deben aproximar por diferencias finitas:

$$d_u \approx k \frac{f_h(u + \Delta, v) - f_h(u - \Delta, v)}{2\Delta}$$

$$d_v \approx k \frac{f_h(u, v + \Delta) - f_h(u, v - \Delta)}{2\Delta}$$

donde:

- ▶  $\Delta$  es usualmente del orden de  $1/n_t$  ( $n_t$  = resol. de la textura).
- ▶  $k$  es un valor real que sirve para atenuar o exagerar el relieve

## La superficie como una función de las c.t.

Los puntos de los polígonos que forman las superficies de los objetos pueden interpretarse como una función  $f_p$  de las coordenadas de textura, es decir, si las coord. de textura de un punto  $q$  son  $(u, v)$ , entonces:

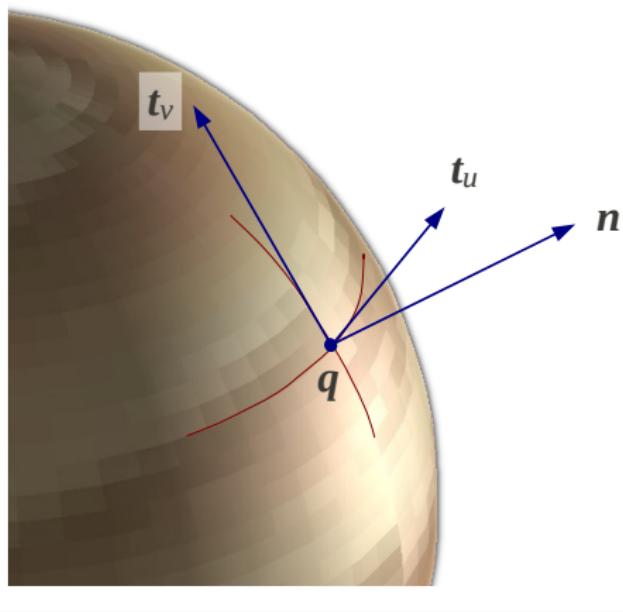
$$q = f_p(u, v)$$

para calcular la normal modificada es necesario conocer las derivadas parciales de  $f_p$  (dos vectores  $t_u$  y  $t_v$ )

$$t_u = \frac{\partial f_p(u, v)}{\partial u} \quad t_v = \frac{\partial f_p(u, v)}{\partial v}$$

# Las tangentes y la normal

A los vectores  $t_u$  y  $t_v$  se les suele llamar *tangente* y *bitangente*. Ambos definen un plano tangente, perpendicular a la normal.



## Cálculo de los vectores tangentes modificados

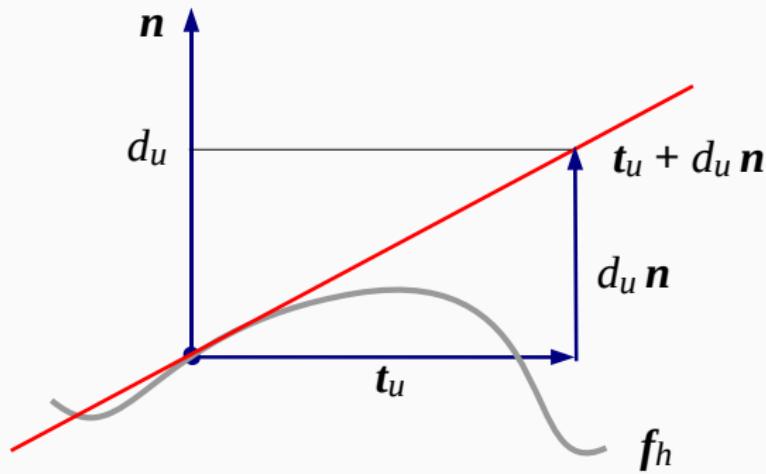
Estos vectores son tangentes a la superficie del objeto, ya que la normal original  $\mathbf{n}$  es colineal con  $\mathbf{t}_u \times \mathbf{t}_v$ . Existen varias alternativas para obtenerlos:

- ▶ Para objetos sencillos, los vectores tangentes son constantes o muy fáciles de calcular
- ▶ Para mallas de polígonos:
  - ▶ Se pueden calcular como constantes en cada polígono, a partir de las coordenadas de textura.
  - ▶ Se pueden asignar a los vértices (igual que las c.t.) y realizar una interpolación en el interior de los polígonos (igual que se interpola la normal).

## Obtención de las tangentes modificadas

Los vectores tangentes  $t'_u$  y  $t'_v$  a la superficie rugosa son:

$$t'_u = t_u + d_u \mathbf{n} \quad t'_v = t_v + d_v \mathbf{n}$$



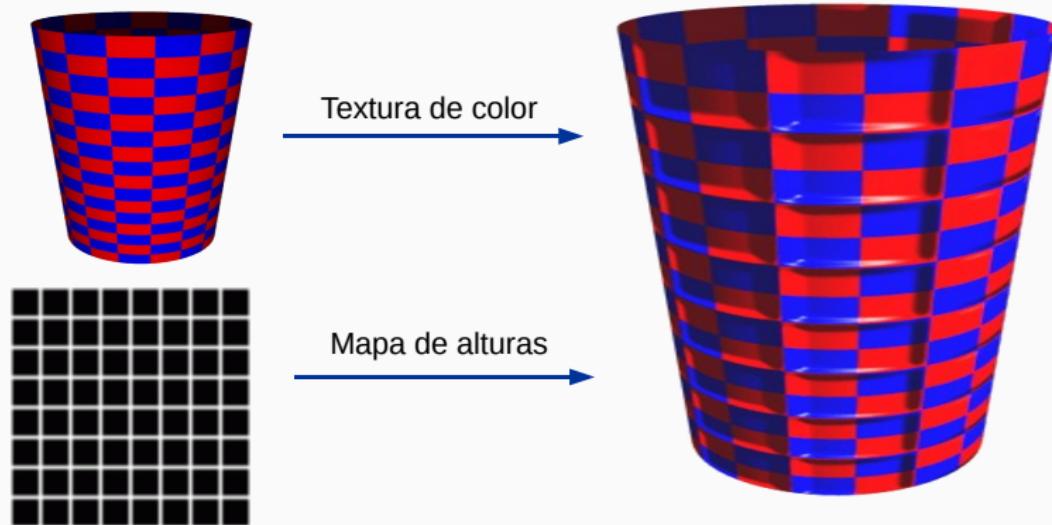
la normal modificada  $\mathbf{n}'$  es perpendicular a estos dos vectores, por tanto se calcula usando su producto vectorial (y normalizando)

$$\mathbf{n}' = \frac{\mathbf{n}''}{\|\mathbf{n}''\|} \quad \text{donde: } \mathbf{n}'' = \mathbf{t}_u' \times \mathbf{t}_v'$$

# Ejemplo de texturas + perturbación de la normal (1)

Imágenes de Fredo Durand y Barb Curtler:

<http://groups.csail.mit.edu/graphics/classes/6.837>



## Ejemplo de texturas + perturbación de la normal (2)



Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 1. Técnicas realistas en rasterización

Subsección 1.3.

Sombras arrojadas.

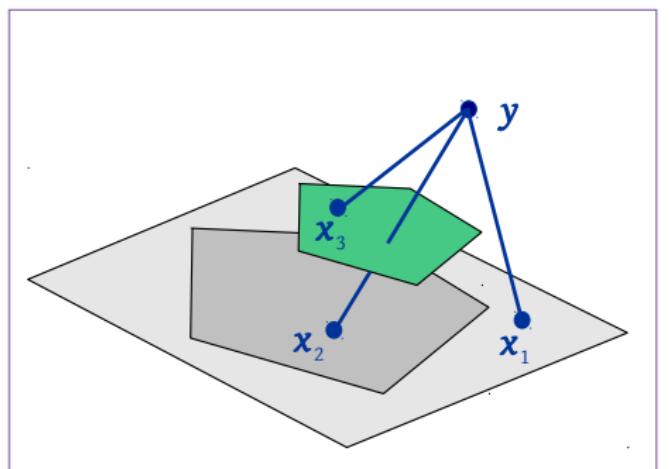
## Sombras arrojadas y el MIL

Ninguna de las técnicas anteriores tiene en cuenta la existencia de sombras arrojadas.

- ▶ Se supone que todas las fuentes son visibles desde todos los puntos de la superficie, lo cual no siempre es cierto.
- ▶ Si asumimos que los polígonos son opacos, y las fuentes puntuales (o direccionales), para cada punto en una superficie y para cada fuente de luz, el punto y la fuente pueden ser mutuamente visibles o no.
- ▶ Cuando la fuente no ilumina el punto, el sumando del MIL correspondiente a la fuente no debe añadirse para obtener el color reflejado.

# La función de visibilidad $V$

La visibilidad de la fuente de luz (en  $y$ ) está controlada por la función  $V$ :



$$V(x_1, y) = 1 \quad V(x_2, y) = 0 \quad V(x_3, y) = 1$$

## Sombras arrojadas y visibilidad

El problema de las sombras arrojadas es, por tanto, semejante al problema de la visibilidad:

- ▶ Se pueden usar algoritmos con precisión de objetos: se producen en la salida los polígonos (parte de los originales) iluminados por (visibles desde) las fuentes de luz.
- ▶ Se pueden usar algoritmos con precisión de imagen: se obtiene el primer punto visible en el centro de cada pixel de un plano de visión asociado a una fuente de luz.

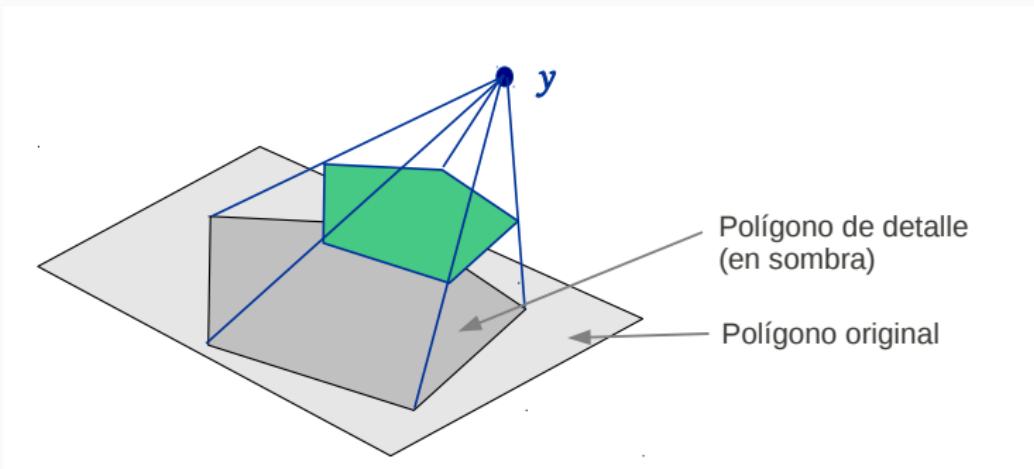
El papel del observador lo juega la fuente de luz. Puede ser posicional (observador a distancia finita) o direccional (observador a distancia infinita: proyección ortogonal).

## Algoritmo de fuerza bruta

Supondremos escenas formadas por poliedros opacos delimitados por caras planas o polígonos planos individuales.

- ▶ El algoritmo más sencillo consiste en proyectar todos los polígonos contra todos, usando la fuente de luz como foco.
- ▶ Para cada par de polígonos  $P$  y  $Q$  se calcula el polígono de sombra arrojada  $S$  que proyecta  $P$  sobre  $Q$  (si hay alguna), y se recorta  $S$  usando  $Q$  como polígono de recorte.
- ▶ Los polígonos producidos se tratan como polígonos de detalle. Son polígonos superpuestos a los originales en los cuales la fuente de luz no es visible.

## Algoritmo de fuerza bruta (2)



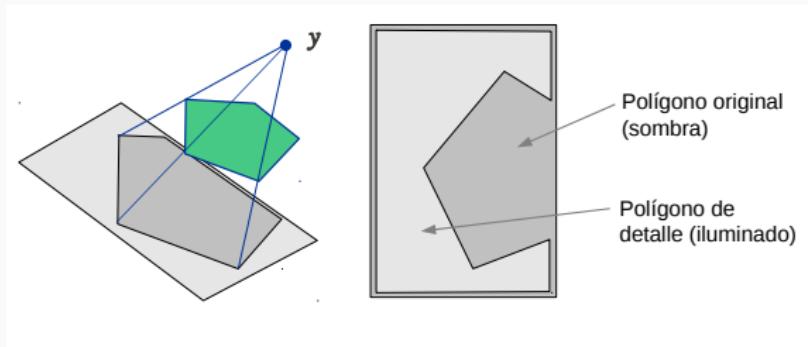
- ▶ tiene complejidad cuadrática con el número de polígonos
- ▶ se puede usar solo para un único polígono receptor y unos pocos que arrojan sombras.

## Algoritmo de Weiler-Atherton-Greenberg

Otros algoritmos de sombras arrojadas (más eficientes) están basados en algoritmos de eliminación de partes ocultas ya existentes. Un ejemplo es el algoritmo de Weiler-Atherton-Greenberg (1978) para sombras arrojadas:

- ▶ Se usa el algoritmo de Weiler-Atherton para eliminación de partes ocultas
- ▶ Se produce un modelo con polígonos iluminados asociados a los originales (son también polígonos de detalle).
- ▶ La complejidad en tiempo es mucho menor que cuadrática en el caso medio.

## Algoritmo de Weiler-Atherton-Greenberg (2)

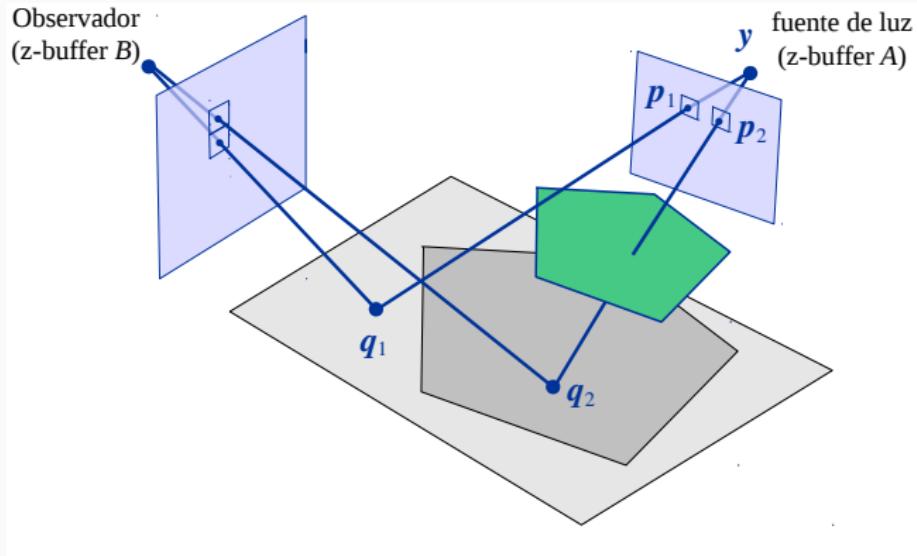


En general, los algoritmos con precisión de objetos para sombras son:

- ▶ muy complejos en tiempo para escenas complejas
- ▶ para algunas aplicaciones son los más idóneos (cuando se necesita un resultado en forma de dibujo vectorial).

# Z-buffer para sombras arrojadas

Otra posibilidad (mucho más eficiente) es usar Z-buffer para sombras arrojadas:

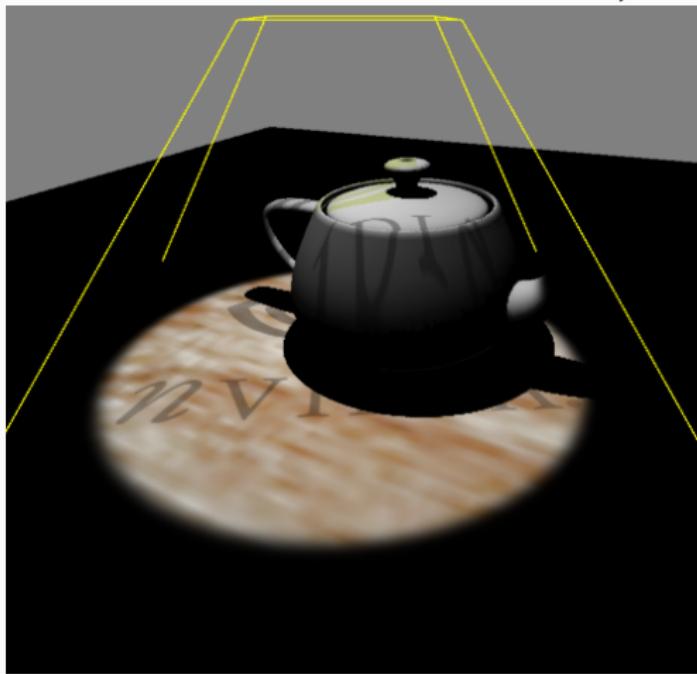


## Z-buffer para sombras arrojadas (2)

- ▶ En la primera pasada se calcula el Z-buffer  $A$  asociado a la fuente de luz (se proyectan los objetos contra la fuente)
- ▶ La segunda pasada es semejante al Z-buffer normal, se calcula el Z-buffer  $B$ , para cada punto visible  $\mathbf{q}_i$  desde el observador en un pixel, se debe calcular el color con el que se ve  $\mathbf{q}_i$ , y por tanto es necesario comprobar si es visible desde la fuente, para ello:
  - ▶ se calcula  $\mathbf{p}_i$  (la proyección de  $\mathbf{q}_i$  en el plano de visión asociado a la fuente de luz)
  - ▶ se accede al pixel del Z-buffer  $A$  correspondiente a  $\mathbf{p}_i$ , que contiene una distancia  $d$
  - ▶ si  $d < \|\mathbf{q}_i - \mathbf{y}\|$ , entonces  $\mathbf{q}_i$  no está iluminado (este es el caso de la figura), en otro caso  $\mathbf{q}_i$  sí está iluminado.

# Ejemplo de Z-buffer para sombras

(se proyecta una textura desde la fuente en los objetos):



## Errores de Z-buffer para sombras

son visibles si el observador está cerca de ellas en comparación con la distancia a la fuente de luz:



Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

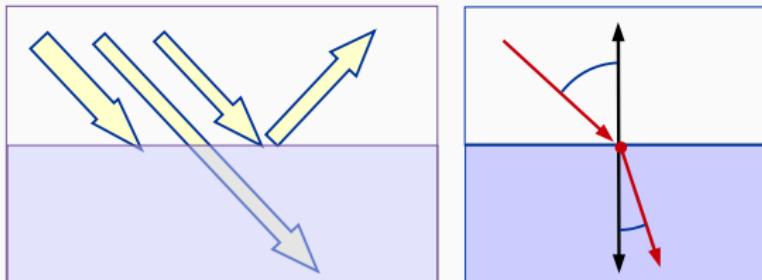
Sección 1. Técnicas realistas en rasterización

Subsección 1.4.

Superficies transparentes. Refracci'on..

# Materiales transparentes

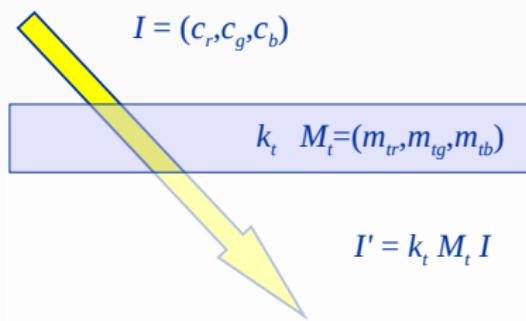
Hay objetos sólidos o líquidos que permiten pasar (además de reflejar o absorber) algunos fotones de la luz que los alcanza. Su estructura atómica permite a los rayos de luz viajar en línea recta.



la luz cambia de dirección debido a su progreso más lento en estos medios (debido al retraso por múltiples eventos de dispersión de fotones)

# Cambio del color en la refracción

Al pasar por un objeto delgado transparente, la cantidad de luz que no es absorbida en el medio (y atraviesa el objeto) depende de la longitud de onda:



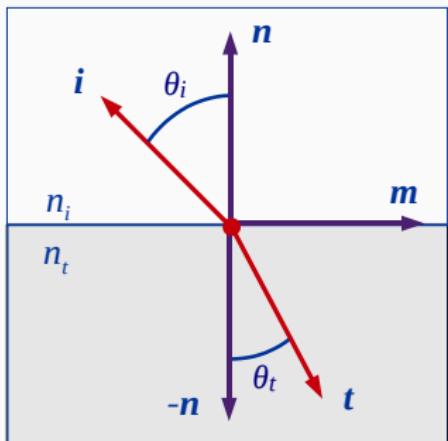
- ▶ La fracción global de luz refractada es  $k_t$ , que está entre 0 y 1
- ▶ En cada longitud de onda se refracta una fracción distinta, en RGB estas fracciones son un color  $M_t = (m_{tr}, m_{tg}, m_{tb})$

Por tanto, estos materiales están caracterizados por  $k_t$  y  $M_t$

# Cambio de dirección en la refracción

El vector  $\mathbf{t}$  puede calcularse a partir de  $\mathbf{n}$ ,  $\mathbf{i}$ , y los índices de refracción  $n_i$  y  $n_t$ , teniendo en cuenta la ley de Snell:

$$n_i \sin \theta_i = n_t \sin \theta_t$$



$$\mathbf{t} = (r \cos \theta_i - \cos \theta_t) \mathbf{n} - r \mathbf{i}$$

$$\cos \theta_i = \mathbf{i} \cdot \mathbf{n}$$

$$\cos \theta_t = \sqrt{1 - r^2 [1 - (\mathbf{i} \cdot \mathbf{n})^2]}$$

$$r = n_i / n_t$$

Si  $1 < r^2 [1 - (\mathbf{i} \cdot \mathbf{n})^2]$  entonces no hay refracción (hay *reflexión interna total*). Solo puede ocurrir cuando  $n_t < n_i$ , para  $\theta_i > \theta_{\max}$ .

# Superficies transparentes en Z-buffer

El método de Z-buffer solo puede tener en cuenta, para un pixel, los colores de los puntos en el proyector o rayo que pasa por el centro del pixel hacia el observador.

- ▶ Cuando  $n_i \neq n_t$  los rayos se desvían, y esto no puede reproducirse con Z-buffer
- ▶ Si suponemos que  $n_i = n_t$ , entonces no hay cambio de dirección debida a la refracción, y por tanto  $t = -i$
- ▶ Con esta simplificación, se puede adaptar el método de Z-Buffer para incluir polígonos transparentes.

a continuación vemos un esbozo del método

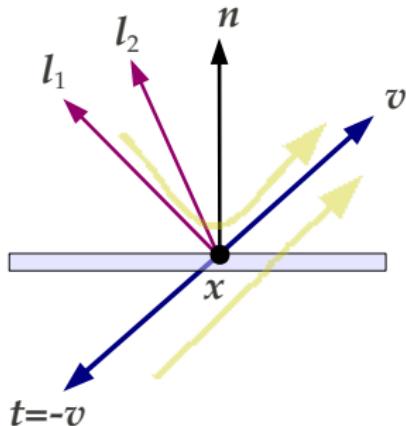
# Z-buffer adaptado a superficies transparentes

El algoritmo dibuja primero los polígonos opacos, y después los transparentes o semi transparentes (en cualquier orden)

- 
- 1: Inicializar Z-buffer ( $Z$ ) e Imagen ( $I$ )
  - 2: **for** cada polígono opaco  $P$  **do**
  - 3:     Rasterizar  $P$ , actualizando  $Z$  e  $I$
  - 4: **for** cada polígono transparente  $Q$  **do**
  - 5:      $k_t$  = fraccion de luz refractada de  $Q$
  - 6:      $M_t$  = color transparente de  $Q$
  - 7:     **for** cada pixel  $(x, y)$  ocupado por  $Q$  **do**
  - 8:         **if**  $Q$  es visible en  $(x, y)$  **then**
  - 9:              $I[x, y] = k_t M_t I[x, y]$
-

# Combinación de reflexión y refracción

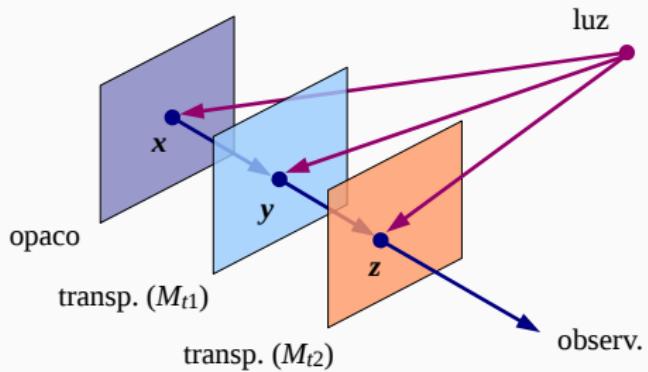
En la superficie entre una lámina de material transparente y el espacio (vacío) entre objetos puede también reflejarse la luz:



Si  $k_t > 0$ , al MIL debe sumársele la luz refractada proveniente del otro lado del polígono, en la dirección de  $v$

# Dependencia del orden

El color  $I$  que percibe el observador depende de los colores  $I_x$ ,  $I_y$  y  $I_z$  reflejados en los puntos  $x, y$  y  $z$ :



Este color depende del orden de los polígonos:

$$I = I_z + M_{t2}I_y + M_{t1}M_{t2}I_x \neq I_z + M_{t1}I_x + M_{t2}M_{t1}I_y$$

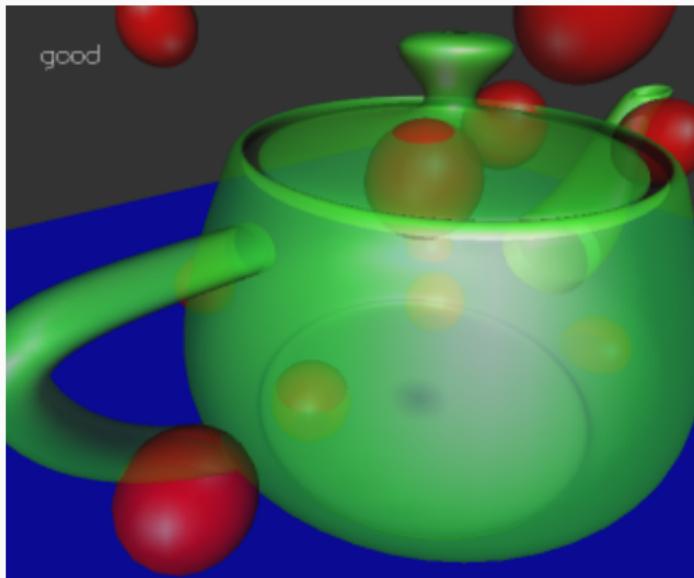
# Z-buffer en superf. transparentes y reflectantes

Los polígonos transp. deben dibujarse en orden de Z:

---

- 1: Inicializar Z-buffer ( $Z$ ) e Imagen ( $I$ )
  - 2: **for** cada polígono opaco  $P$  **do**
  - 3:     Rasterizar  $P$ , actualizando  $Z$  e  $I$
  - 4: **for** cada polígono transparente  $Q$  (en orden de Z) **do**
  - 5:      $k_t$  = fraccion de luz refractada de  $Q$
  - 6:      $M_t$  = color transparente de  $Q$
  - 7:     **for** cada pixel  $(x,y)$  ocupado por  $Q$  **do**
  - 8:         **if**  $Q$  es visible en  $(x,y)$  **then**
  - 9:              $I_m$  = resultado de evaluar MIL
  - 10:             $I[x,y] = I_m + k_t M_t I[x,y]$
-

# Ejemplo de materiales transparentes



Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 1. Técnicas realistas en rasterización

Subsección 1.5.  
Superficies especulares.

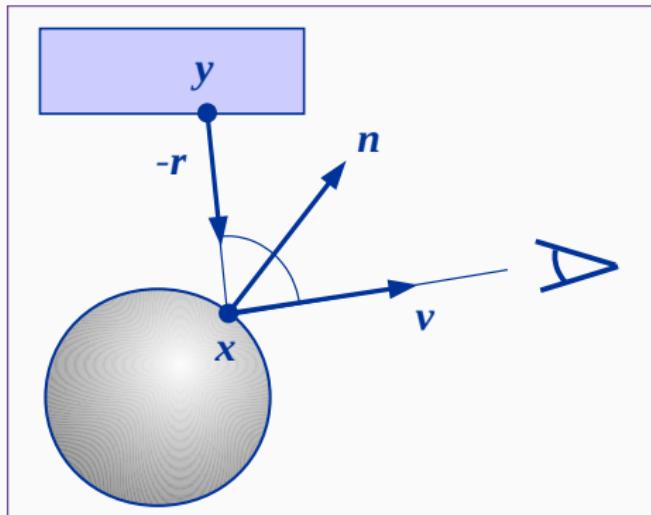
## Reflexión especular de la luz

Algunos objetos pulidos reflejan la luz de forma especular perfecta, como los espejos planos

- ▶ La componente especular perfecta es una componente más del modelo de iluminación local, que se suma a las anteriores (se suele dar en combinación con la refractada en los objetos de cristal).
- ▶ Este efecto no puede reproducirse con ninguno de los métodos que hemos visto, pues la iluminación no procede de la dirección del rayo central a un pixel, sino de otras direcciones.

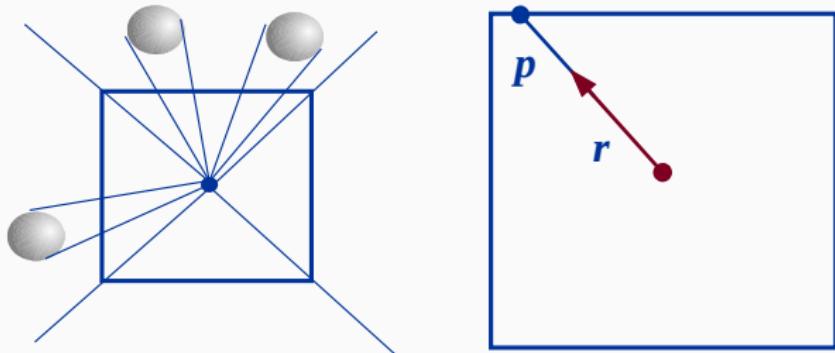
## Reflexión especular de la luz

Si la esfera es perfectamente reflectante, el color de  $x$  visto desde  $v$  es igual al color de  $y$  visto desde la dirección  $-r$  (el vector reflejado  $r$ , cambiado de signo).



## Mapas de entorno tipo caja (*box map*)

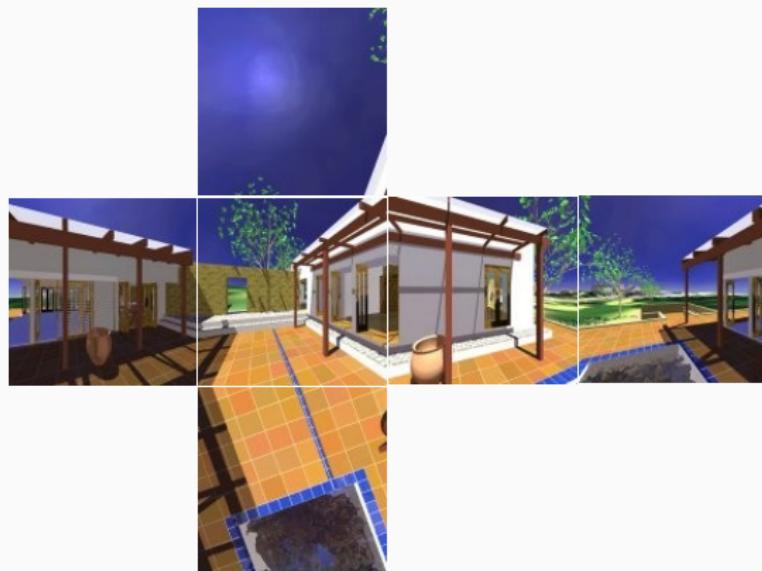
En esta técnica, el entorno se proyecta en las 6 caras de un cubo centrado en el objeto reflectante, obteniéndose 6 texturas.



En tiempo de rendering, el vector  $r$  se proyecta sobre la cara que corresponda (se calcula  $p$ ), y se obtienen el color RGB del texel que contiene a  $p$ .

# Texturas para mapas de entorno tipo caja

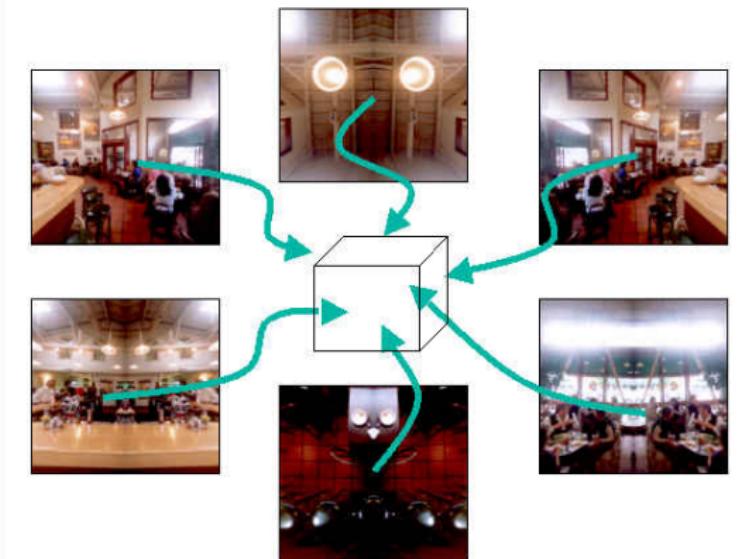
Ejemplo de 6 texturas para mapas de entorno



[http://developer.nvidia.com/object/cube\\_map\\_ogl\\_tutorial.html](http://developer.nvidia.com/object/cube_map_ogl_tutorial.html)

# Uso de fotografías de un entorno real

Tambien es posible usar fotografías de un entorno real



[http://developer.nvidia.com/object/cube\\_map\\_ogl\\_tutorial.html](http://developer.nvidia.com/object/cube_map_ogl_tutorial.html)

# Mapa de entorno esférico

Una sola imagen codifica el color reflejado para todas las posibles orientaciones de la normal



se asume una proyección ortográfica fija, en la cual el vector  $v$  es constante y paralelo al eje Z.

# Panoramas equirectangulares

Es una sola imagen que codifica, en cada texel  $(u, v)$ , la irradiancia en una dirección de coordenadas polares  $\alpha = au$  y  $\beta = bv$ :



se suelen obtener a partir de múltiples fotografías de un entorno. A su vez, pueden servir para crear mapas de entorno esféricos.

# Mapa de entorno esférico

Ejemplo del mapa de entorno esférico anterior en la tetera:



## Mapa de entorno

Ejemplo de combinación con texturas y perturbación de la normal.  
Además, la tetera se muestra en el entorno que refleja:

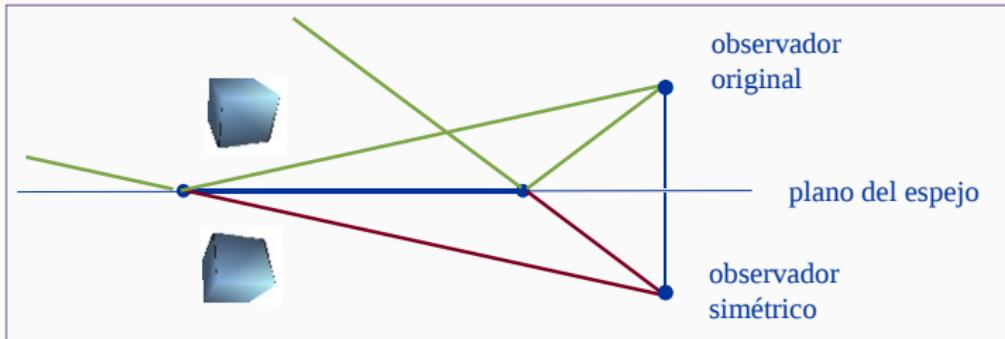


## Ejemplo en cine de animación



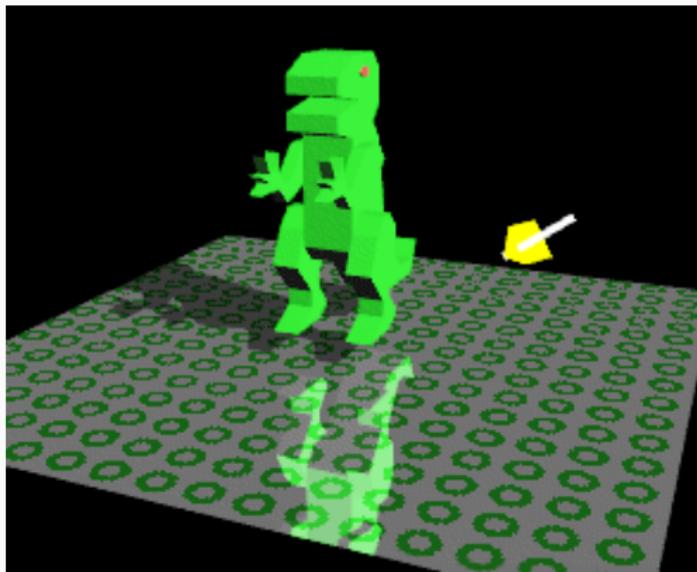
# Reflexión en espejos planos

En estos objetos, la escena reflejada es simétrica respecto de la original respecto del plano del espejo:



Se pueden reproducir las reflexiones sintetizando la imagen vista por una cámara *simétrica* respecto de la original.

## Ejemplo de duplicación de entorno:



## Sección 2. Ray tracing.

- 2.1. El algoritmo de Ray-Tracing
- 2.2. Intersecciones rayo-objeto y rayo-escena
- 2.3. Problemas: Cálculo de intersecciones
- 2.4. Ejemplos

Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 2. Ray tracing

Subsección 2.1.

El algoritmo de Ray-Tracing.

# Introducción

Hemos visto bastantes efectos:

- ▶ Superficies difusas y pseudo-especulares
- ▶ Texturas y mapas de perturbación de la normal
- ▶ Sombras arrojadas
- ▶ Superficies transparentes
- ▶ Superficies especulares perfectas

Considerarlos todos en visualización por rasterización lleva a software que es bastante complicado de implementar. Además los tiempos de síntesis de imágenes pueden hacerse bastante altos.

# La técnica de *Ray-Tracing*

Existe un algoritmo no muy eficiente en tiempo, pero bastante sencillo, que tiene en cuenta todos los efectos anteriores:

- ▶ Este algoritmo es el algoritmo de *Ray-Tracing* (*seguimiento de rayos*, usualmente traducido por *trazado de rayos*)
- ▶ Descrito completamente por primera vez por Turner Whitted en 1979-80.
- ▶ Está basado en la EPO por Ray-Casting, combinada con evaluación del MIL
- ▶ Es conceptualmente muy sencillo, y fácil de implementar.
- ▶ Obtiene un grado de realismo muy superior a Z-buffer, a costa de tiempos de cálculo usualmente más altos.

## Ventajas de la técnica

Frente a rasterización, Ray-Tracing puede visualizar objetos

- ▶ curvos (con superficie definida por ecuaciones implícitas).
- ▶ especulares perfectos (que reflejan el entorno).
- ▶ transparentes (con índices de refracción arbitrarios).
- ▶ con sombras arrojadas por otros.

Existen diversos algoritmos basados en Ray-Tracing con funcionalidad adicional:

- ▶ **Ray-Tracing distribuido**: fuentes de luz extensas, profundidad de campo, desenfoque de movimiento (*motion-blur*).
- ▶ **Path-Tracing**: iluminación indirecta o global.

El algoritmo de **Path-Tracing** y derivados se usa en generación por ordenador de películas y efectos especiales.

# Un M.I.L. extendido para Ray-Tracing

Ray-Tracing calcula la radiancia  $L_{\text{in}}(\mathbf{q}, \mathbf{u})$  incidente sobre un punto  $\mathbf{q}$  proveniente de una dirección  $\mathbf{u}$  (un vector unitario).

$$\begin{aligned} L_{\text{in}}(\mathbf{q}, \mathbf{u}) &= L(\mathbf{p}, \mathbf{v}) \\ &= M_E(\mathbf{p}) + L_{\text{ind}}(\mathbf{p}, \mathbf{v}) + \sum_{i=0}^{n_L-1} v_i \cdot S_i \cdot L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) \end{aligned}$$

donde:

- ▶ El punto  $\mathbf{p}$  es el primero visible desde  $\mathbf{q}$  en la dirección de  $\mathbf{u}$ .
- ▶ El vector  $\mathbf{v}$  es la dirección de salida de radiancia (es  $-\mathbf{u}$ ).
- ▶  $M_E(\mathbf{p})$  es la emisividad en  $\mathbf{p}$ .
- ▶ El valor  $v_i$  es la visibilidad de la  $i$ -ésima fuente de luz (vale 0 o 1).
- ▶ Las funciones  $L_{\text{dir}}$  y  $L_{\text{ind}}$  representan la **iluminación directa** y la **iluminación indirecta**, respectivamente.

## MIL de Ray-Tracing. Iluminación directa.

La iluminación directa  $L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l})$  es la radiancia reflejada en  $\mathbf{p}$  hacia  $\mathbf{v}$  debida a iluminación directa proveniente de una fuente de luz que está en la dirección  $\mathbf{l}$ . Se define así:

$$\begin{aligned} L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l}) &\equiv k_a \cdot M_A(\mathbf{p}) \\ &+ k_d \cdot M_D(\mathbf{p}) \cdot \max(0, \mathbf{n} \cdot \mathbf{l}) \\ &+ k_s \cdot M_S(\mathbf{p}) \cdot d_i \cdot [\max(0, \mathbf{r} \cdot \mathbf{v})]^e \end{aligned}$$

esta fórmula incorpora las componentes ambiental, difusa y pseudo-especular o *glossy*, de forma similar a como vimos en el tema 3 para rasterización. Aquí:

- ▶  $\mathbf{r} \equiv$  vector reflejado en  $\mathbf{p}$  (depende de  $\mathbf{v}$  y  $\mathbf{n}$ ).
- ▶  $\mathbf{n} \equiv$  normal en  $\mathbf{p}$  (depende de  $\mathbf{p}$  y  $O$ )
- ▶  $k_a, M_A, k_d, M_D, k_s, M_S, \equiv$  parámetros del material.

# MIL de Ray-Tracing. Iluminación indirecta.

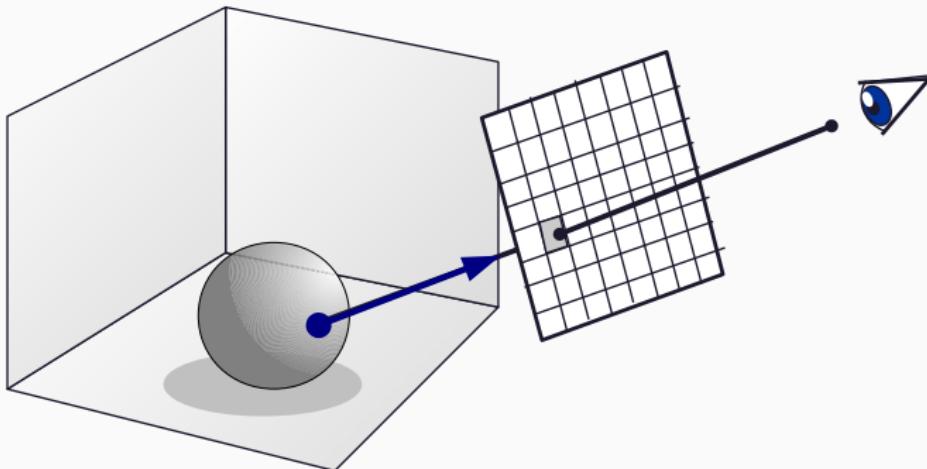
El término  $L_{\text{ind}}(\mathbf{p}, \mathbf{v})$  incluye la radiancia ambiente global, la reflejada perfectamente y la refractada. Se define **recursivamente** en términos de  $L_{\text{in}}$ :

$$L_{\text{ind}}(\mathbf{p}, \mathbf{v}) \equiv A_G(\mathbf{p}) + k_{\text{ps}}(\mathbf{p})M_{\text{PS}}(\mathbf{p})L_{\text{in}}(\mathbf{p}, \mathbf{r}) + k_t(\mathbf{p})M_T(\mathbf{p})L_{\text{in}}(\mathbf{p}, \mathbf{t})$$

- ▶  $A_G(\mathbf{p}) \equiv$  radiancia ambiente global (suple ilum. indirecta).
- ▶  $k_t(\mathbf{p}) \equiv$  fracción de luz refractada en  $\mathbf{p}$ .
- ▶  $k_{\text{ps}}(\mathbf{p}) \equiv$  fracc. de luz refl. de forma especular perfecta en  $\mathbf{p}$ .
- ▶  $M_{\text{PS}}(\mathbf{p})$  y  $M_T(\mathbf{p})$  son ternas RGB (con valores en  $[0, 1]$ ) que permiten modular el color de la componente refejada perfectamente o refractada.
- ▶  $\mathbf{r} \equiv$  vector reflejado en  $\mathbf{p}$  (depende de  $\mathbf{v}$  y  $\mathbf{n}$ ).
- ▶  $\mathbf{t} \equiv$  vector refractado en  $\mathbf{p}$  (depende de  $\mathbf{v}$  y  $\mathbf{n}$ ).

# Generación de Rayos-primarios

Los pixels se procesan secuencialmente, en cada uno se crea un rayo (llamado *rayo primario* o *rayo de cámara*) y se determina el primer objeto visible por Ray-Casting:



# El procedimiento principal de Ray-Tracing

El pseudocódigo del algoritmo, que recorre todos los pixels, puede quedar así:

- 
- 1: **o** := posición del observador, en coords. del mundo
  - 2: **for** cada pixel  $(i, j)$  de la imagen **do**
  - 3:   **q** := punto central (en WCC) del pixel  $(i, j)$
  - 4:   **u** := vector desde **o** hasta **q** normalizado
  - 5:   **rad** := RAYTRACING(**o,u,1**)
  - 6:   fijar el pixel  $(i, j)$  al valor **rad**
- 

- ▶ La función RAYTRACING es recursiva, devuelve un color, y tiene un parámetro que sirve para que la recursión no se haga infinita.
- ▶ Los colores de los pixels no están acotados, así que puede ser necesario un paso de post-proceso para normalizar la imagen.

# La función RAYTRACING

Esta función calcula la radiancia incidente sobre el punto **o**, proveniente de la dirección **u** (como una terna RGB). El entero *n* es el nivel de profundidad en las llamadas recursivas.

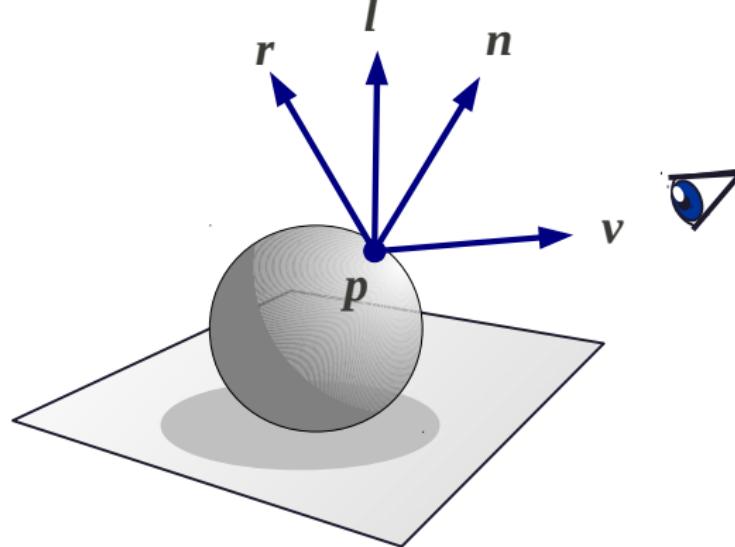
---

```
1: function RAYTRACING( punto o, vector u, entero n )
2:   if n > max then    // si se ha superado máximo nivel de recursión
3:     return (0,0,0)           // devolver radiancia nula
4:   O := primer objeto visible desde o en la dir. u    // (por ray-casting)
5:   if no existe ningun objeto visible then
6:     return radiancia de fondo correspondiente a u
7:   p := punto de O intersecado
8:   return EVALUAMILREC( O, p, -u, n )
```

---

# Evaluación del MIL

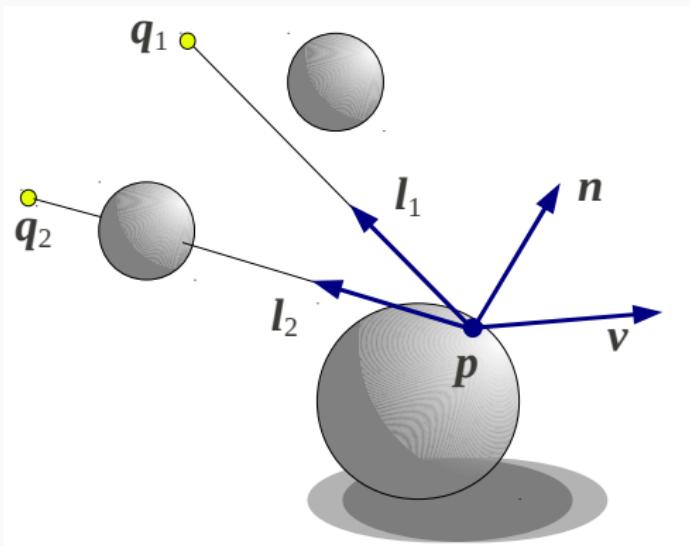
Una vez se conoce el punto  $p$ , se obtienen la normal  $n$ , y los parámetros del MIL, que es evaluado:



podemos considerar objetos curvos (esferas, cilindros, conos, etc..)

# Sombras arrojadas

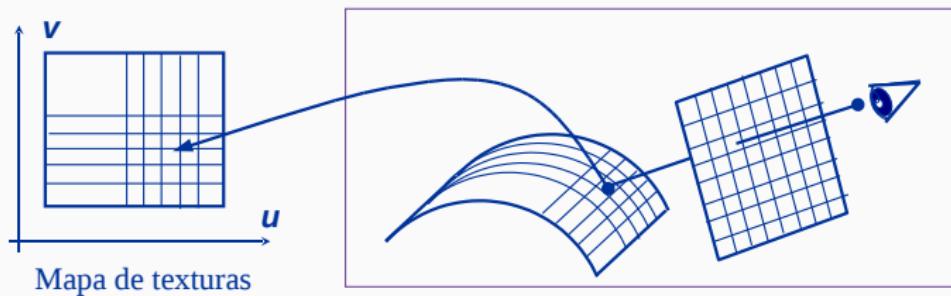
Este método permite incorporar sombras arrojadas, usando Ray-Casting para visibilidad. Se comprueba si un segmento de recta desde  $p$  hasta (o hacia) la fuente interseca algún objeto de la escena, es decir, se evalua  $V(p, q_i)$



# Detalles de las superficies

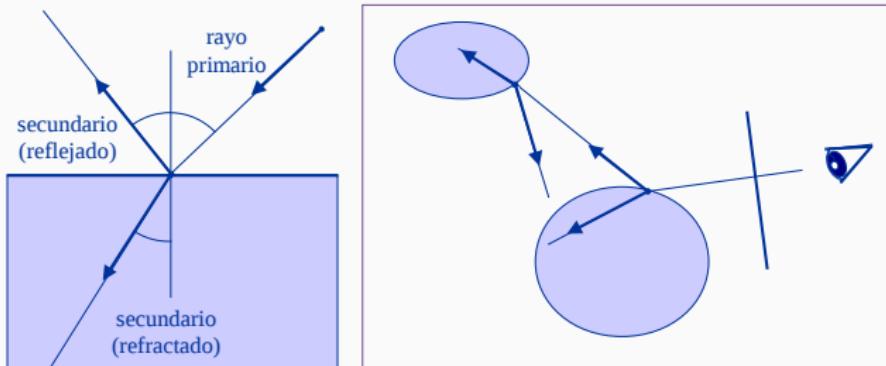
El objeto en el que está  $p$  puede tener asociadas texturas (o mapas de pert. de la normal)

- ▶ A partir de  $p$  se obtienen las coordenadas  $(u, v)$  en el espacio de la textura
- ▶ A partir de  $(u, v)$  se consulta la textura o texturas asociadas al objeto.



# Rayos secundarios y recursividad

También es posible tener en cuenta superficies perfectamente especulares y/o perfectamente transparentes. Esto se hace creando *rayos secundarios*, e invocando recursivamente al algoritmo.



Da lugar a un árbol de rayos asociado al árbol de llamadas recursivas.

# La función EVALUAMILREC

La función que evalua el MIL (EVALUAMILREC) llama recursivamente a la función de RAYTRACING,

---

```
1: function EVALUAMILREC( O, p, v, n )
2:   Obtener parámetros del material de O (n,  $k_a$ ,  $k_d$ ,  $k_s$ ,  $k_t$ ,  $u$ ,  $v$ , etc ... )
3:   Obtener parámetros de fuentes de luz ( $n_L$ , li,  $S_i$ , etc...)
4:   rad :=  $M_E(\mathbf{p}) + A_G(\mathbf{p})$            // emisividad y comp. ambiente global
5:   for i := 0 to  $n_L - 1$  do                 // para cada fuente de luz
6:     if la fuente i es visible desde p then      // (que sea visible)
7:       rad := rad +  $S_i \cdot \text{DIRECTA}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$           // ilum. directa
8:     if  $k_t > 0$  and n < max then
9:       t := vector refractado respecto de v
10:      rad := rad +  $k_t \cdot M_T \cdot \text{RAYTRACING}(\mathbf{p}, \mathbf{t}, n + 1)$     // componente refractada
11:      if  $k_{ps} > 0$  and n < max then
12:        r := vector reflejado respecto de v
13:        rad := rad +  $k_{ps} \cdot M_{PS} \cdot \text{RAYTRACING}(\mathbf{p}, \mathbf{r}, n + 1)$     // componente reflejada
14:   return rad
```

---

# La función DIRECTA

Esta función evalua  $L_{\text{dir}}$  (las componentes ambiental, difusa y especular correspondientes a una fuente de luz).

---

```
1: function DIRECTA( p, v, l )
2:     rad :=  $k_a \cdot M_A(p)$ 
3:     if  $k_d > 0$  then
4:         rad := rad +  $k_d \cdot M_D(p) \cdot \max(0, n \cdot l)$ 
5:     if  $k_s > 0$  then
6:         rad := rad +  $k_s \cdot M_S(p) \cdot d_i \cdot [\max(0, r \cdot v)]^e$ 
7:     return rad
```

---

Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 2. Ray tracing

Subsección 2.2.

Intersecciones rayo-objeto y rayo-escena.

# Cálculo de intersecciones: rayo-objeto y rayo-escena

El algoritmo de Ray-Tracing emplea la mayor parte de su tiempo en:

- ▶ Evaluación de MIL avanzados: a los subprogramas que los evaluan se les llama *shaders*.
- ▶ Cálculo de intersecciones. Dado un punto  $\mathbf{o}$  y un vector unitario  $\mathbf{v}$  (en coordenadas de mundo, codificando una **semirrecta** o **rayo**), se trata de calcular el menor valor real  $t > 0$  tal que el punto  $\mathbf{p}(t) \equiv \mathbf{o} + t\mathbf{v}$  está en la frontera o superficie de algún objeto de la escena.

Hay dos algoritmos fundamentales:

- ▶ Intersección rayo-objeto: cada tipo de geometría posible tiene asociado un algoritmo.
- ▶ Intersección rayo-escena: podemos recorrer exhaustivamente los objetos, pero es  $O(n_o)$ . Se reducen los tiempos con *indexación espacial*, que es  $O(\log n_o)$ .

## Intersecciones rayo-objeto: método general

Calcular la intersección de un rayo  $(\mathbf{o}, \mathbf{v})$  con un objeto cuya geometría es  $O \subseteq \mathbb{R}$  consiste en obtener el mínimo valor de  $t > 0$  (si hay alguno) tal que  $\mathbf{o} + t\mathbf{v} \in \partial O$ . El objeto  $O$  puede estar caracterizado por estas dos funciones:

- ▶ Ecuación implícita: un campo escalar  $F$  tal que si  $\mathbf{p} \in \partial O$  entonces  $F(\mathbf{p}) = 0$ .
- ▶ Condiciones adicionales: un predicado o función lógica  $C$  tal que  $\mathbf{p} \in \partial O$  si y solo si  $F(\mathbf{p}) = 0$  y  $C(\mathbf{p})$ .

El algoritmo requiere:

1. Calcular el conjunto  $S = \{t_0, t_1, \dots, t_{n-1}\}$  con las raíces de la ecuación  $F(\mathbf{o} + t\mathbf{v}) = 0$ .
2. Eliminar de  $S$  los  $t_i$  que no cumplen  $C(\mathbf{o} + t_i\mathbf{v})$ .
3. Si  $S \neq \emptyset$  la solución es  $t = \min(S)$ . Si  $S = \emptyset$  no hay inters.

# Intersecciones rayo-objeto: métodos de solución

Hay dos tipos de algoritmos para obtener la menor raíz  $t$ :

- ▶ Directos:  $t$  se obtiene con una fórmula explícita.

Por ejemplo: si  $\partial O$  es un subconjunto de una **superficie cuádrica**, entonces

$$F(\mathbf{o} + t\mathbf{v}) = 0 \iff \exists a, b, c \in \mathbb{R} \quad \text{t.q: } at^2 + bt + c = 0$$

- ▶ Iterativos:  $t$  se obtiene por sucesivas aproximaciones

Por ejemplo: se puede usar si  $F$  es la ***signed distance function*** (SDF) de  $O$ , es decir, si

$$F(\mathbf{p}) = s \cdot \left( \min_{\mathbf{q} \in \partial O} \|\mathbf{p} - \mathbf{q}\| \right) \quad \text{donde: } s \equiv \begin{cases} -1 & \text{si } \mathbf{p} \in O \\ +1 & \text{si } \mathbf{p} \notin O \end{cases}$$

Si no se dispone de SDF existen alternativas (Bisección, Newton, cotas de distancia, etc...).

# Indexación espacial para métodos directos.

Para calcular las intersecciones rayo-escena podemos usar los algoritmos directos de intersección rayo-objeto para las distintos tipos de objetos que forman la escena:

- ▶ Las escenas usualmente consisten en mallas de triángulos.
- ▶ Solución de *fuerza bruta*: comprobar la intersección con cada triángulo de la escena (tiempo en  $O(n_t)$ ).
- ▶ La **indexación espacial** se basa en descartar partes de la escena si el rayo no interseca un volumen englobante  $V$  de todos los triángulos de esa parte de la escena.
- ▶ La intersección rayo- $V$  es muy rápida de calcular.
- ▶ Se hace jerárquicamente (recursivamente), y se obtiene un árbol de volúmenes englobantes, con conjuntos de triángulos en los nodos terminales. Se obtiene tiempo en  $O(\log n_t)$ .

## Intersecciones rayo-escena: método iterativo basado en SDFs.

Si se disponen de SDFs  $F_0, \dots, F_{n-1}$  de los objetos  $O_0, \dots, O_{n-1}$  se puede usar este algoritmo (donde  $0 \lesssim \epsilon$  es la tolerancia):

---

```
1: function INTERSECCIONSDF( o, v,  $O_0, \dots, O_{n-1}$  )           //  $\|\mathbf{v}\| = 1$ 
2:   p = o // mejor punto de intersección encontrado hasta ahora
3:    $d = 0$  //  $d \equiv$  máxima distancia que se puede avanzar desde p
4:   repeat
5:     p = p +  $d\mathbf{v}$  // avanzamos a lo largo del rayo
6:      $d = \min \{F_0(\mathbf{p}), \dots, F_{n-1}(\mathbf{p})\}$  // actualizamos  $d$  en p
7:   until  $\|d\| \leq \epsilon$  (o hasta un número máximo de iteraciones)
8:   if  $\|d\| \leq \epsilon$  then // si p está ya muy cerca de la superficie
9:     Hay intersección con  $O_i$  en p (con  $i$  tal que  $d \equiv F_i(\mathbf{p})$ )
10:  else
11:    No hay intersección con ningún objeto
```

---

Hay numerosos ejemplos de esta técnica en  Shadertoy

Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 2. Ray tracing

Subsección 2.3.

Problemas: Cálculo de intersecciones.

# Problema: intersección rayo-disco (1/2)

## Problema 5.1.

Supongamos que un *rayo* (una semirecta en 3D) tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla **o**, y como vector de dirección la tupla **d** (la suponemos normalizada).

Además sabemos que un *disco* de radio  $r$  tiene como centro el punto de coordenadas de mundo **c** y está en el plano perpendicular al vector **n**.

Con estos datos de entrada, diseña un algoritmo para calcular si hay intersección entre el rayo y el disco.

(ten en cuenta las indicaciones que hay en la siguiente transparencia).

# Problema: intersección rayo-disco (2/2)

## Problema 5.1. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe  $t > 0$  tal que el punto  $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$  está en dicho plano. Equivale a decir que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es perpendicular a la normal al plano  $\mathbf{n}$ .
2. El punto  $\mathbf{p}_t$  citado arriba está dentro del disco, es decir, su distancia a  $\mathbf{c}$  es inferior al radio.

# Problema: intersección rayo-esfera

## Problema 5.2.

Diseña un algoritmo para calcular la primera intersección entre un rayo (con origen en  $\mathbf{o}$  y vector  $\mathbf{d}$ , normalizado) y una esfera de radio unidad y centro en el origen, si hay alguna.

Ten en cuenta que un punto cualquiera  $\mathbf{p}$  está en esfera si y solo si el módulo de  $\mathbf{p}$  es la unidad, es decir, si y solo si  $F(\mathbf{p}) = 0$ , donde  $F$  es el campo escalar definido así:

$$F(\mathbf{p}) \equiv \mathbf{p} \cdot \mathbf{p} - 1$$

Describe como podría usarse ese mismo algoritmo para calcular la intersección con una esfera con centro y radio arbitrarios (este problema puede reducirse al anterior si el rayo se traslada a un espacio de coordenadas donde la esfera tiene centro en el origen y radio unidad).

# Problema: intersección rayo-cilindro y rayo-cono

## Problema 5.3.

Describe como podemos definir el campo escalar cuyos ceros son los puntos en un cilindro con altura unidad y radio unidad (sin considerar los discos que forman la base ni la tapa).

Usando esa definición diseña el algoritmo para calcular la intersección rayo-cilindro.

Describe asimismo el campo escalar y el algoritmo correspondientes a un cono de altura unidad y radio de la base unidad (sin considerar el disco de la base).

Informática Gráfica, curso 2021-22.

Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

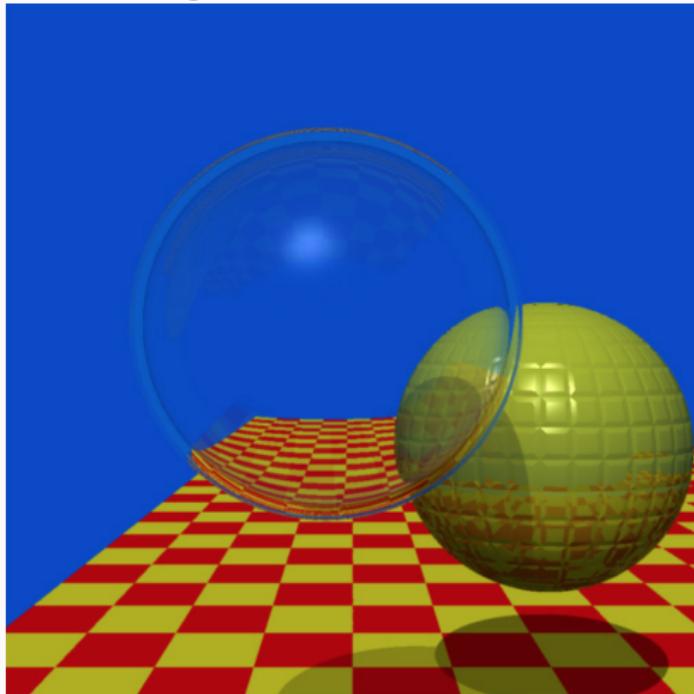
Sección 2. Ray tracing

Subsección 2.4.

Ejemplos.

# Ejemplos: primeras imágenes

Una de las primeras imágenes creadas con esta técnica (en 1980)



Turner Whitted (1980) [An Improved Illumination Model for Shaded Display \(CACM\)](#)  
Entrevista a T. Whitted, vídeo [sitio web de nVidia](#)

# Ejemplos: reflexión y refracción

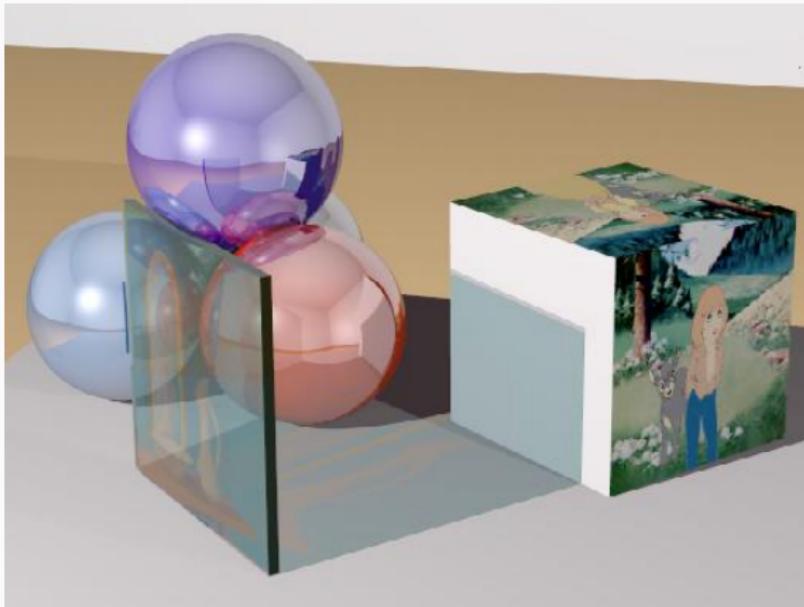
Incluye: texturas, sombras arrojadas, reflexiones, *bump-mapping*.



Universidad de Cornell (1998)  Computer Graphics programme: Reflection and Transparency.

# Ejemplos: reflexiones especulares, sombras arrojadas.

Incluye: reflexiones, texturas, sombras arrojadas con texturas.



Universidad de Cornell (1998)  Computer Graphics programme: Reflection and Transparency.

# Ejemplos: escenas complejas, indexación espacial.

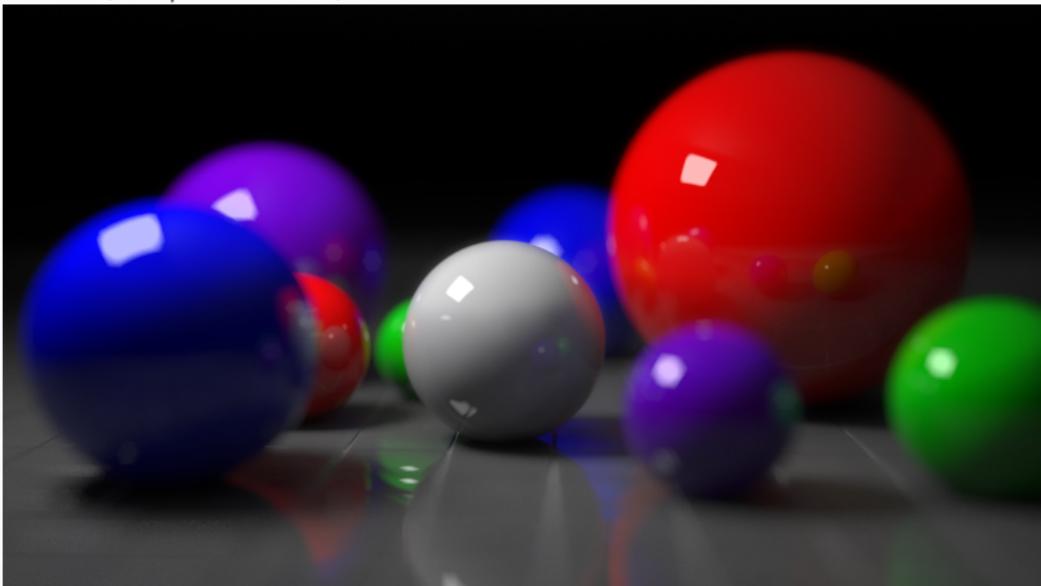
Escena compleja: múltiples objetos complejos. Indexación espacial.



Autor: Gilles Tran Sitio web Oyonale. Public Domain

## Ejemplos: *Distributed Ray-Tracing*

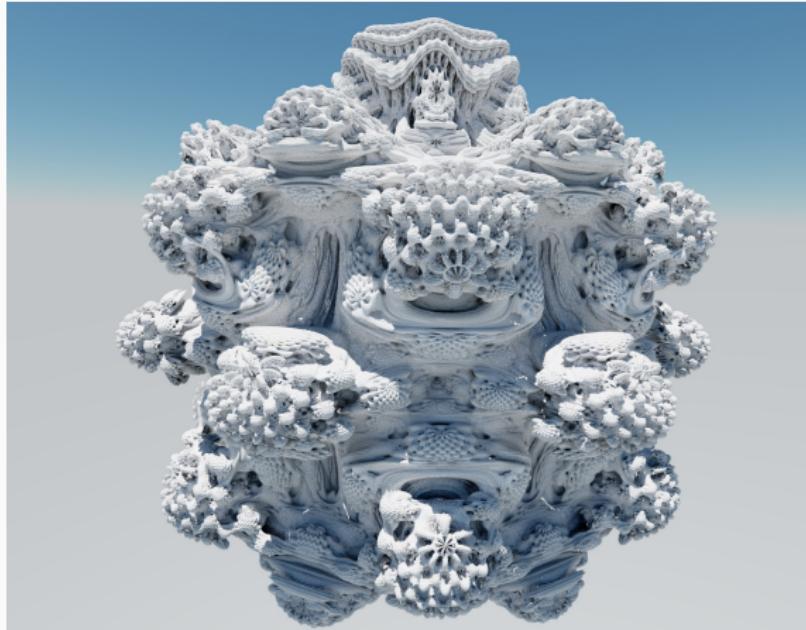
Se usa *Distributed Ray-Tracing* (Cook, 1984): se simula la profundidad de campo (*Depth of Field*) y las penumbras generadas por luces de extensas (no puntuales):



By Mimigu at [English Wikipedia: Ray Tracing \(Graphics\)](#). CC BY 3.0.

## Ejemplos: *Path-tracing*, fractales.

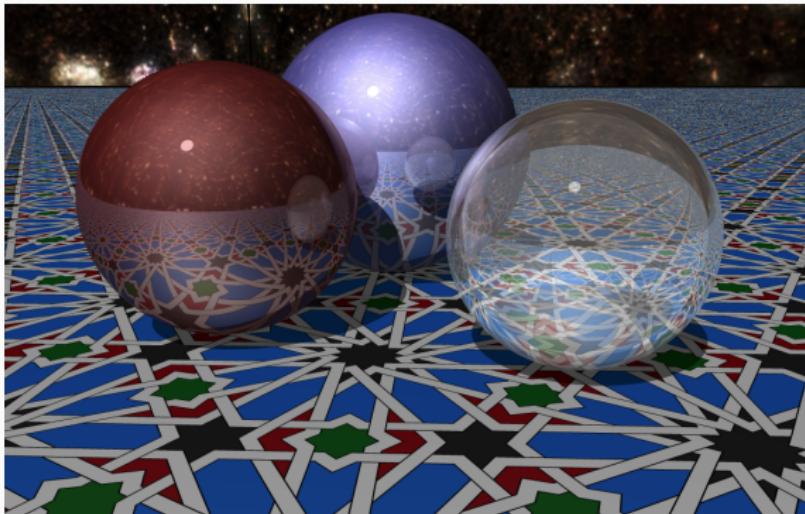
Se usa *Path-tracing* (Kajiya, 1986, *Global Illumination*), e intersecciones con un fractal (con métodos numéricos iterativos).



Por: Mikael Hvidtfeldt Christensen  Blog sobre arte generativo.  
 <https://www.flickr.com/photos/syntopia/>).

# Ejemplos: Ray-tracing sencillo en GPU

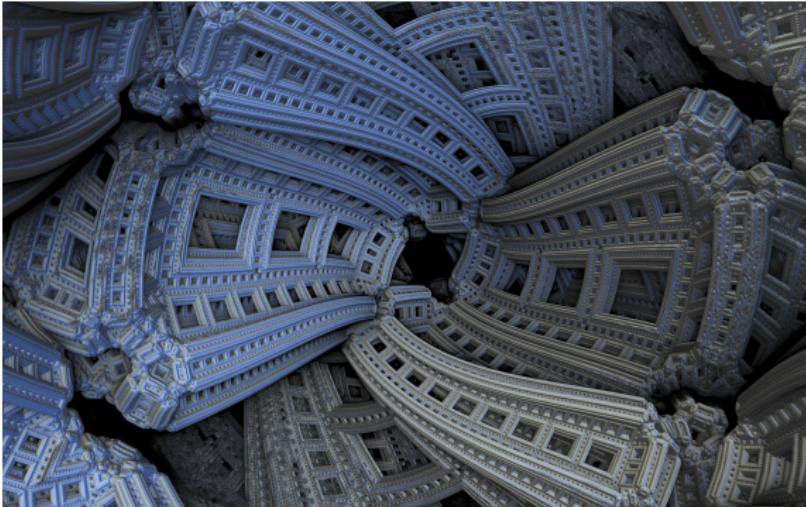
Ray-tracing en tiempo real en la GPU usando un *fragment shader*.  
Incluye: reflexiones, refracciones, texturas procedurales (grupos cristalográficos del plano: pmm6)



Animación disponible on-line en *Shadertoy*: <https://www.shadertoy.com/view/4sdfzn>

## Ejemplos: uso de SDF en GPU

Se usan objetos definidos por su SDF (*Signed Distance Function*) para calcular intersecciones iterativamente. Se hace en la GPU en un *fragment shader*.



*Menger Journey* por **Mikael Hvidtfeldt Christensen**. Animación on-line en [Shadertoy](#).

Fin de la presentación.