

Informática Gráfica:

Teoría. Tema 1. Introducción.

Carlos Ureña
2022-23

Grado en Informática y Matemáticas
Grado en Informática y Administración y Dirección de Empresas
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Teoría. Tema 1. Introducción.
Índice.

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

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.

Sección 1.
Introducción.

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.

Sección 1. Introducción

Subsección 1.1.

Concepto y metodologías.

- 1.1. Concepto y metodologías
- 1.2. Aplicaciones.

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:

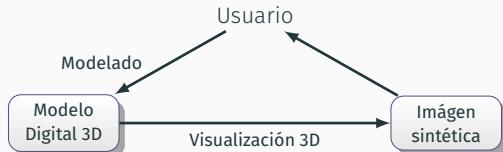
- ▶ Programación orientada a objetos y programación concurrente.
- ▶ Ingeniería del software.
- ▶ Geometría computacional.
- ▶ Hardware (hardware gráfico, dispositivos de interacción).
- ▶ Matemática aplicada (métodos numéricos).
- ▶ Física (óptica, dinámica).
- ▶ Psicología y medicina (percepción visual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

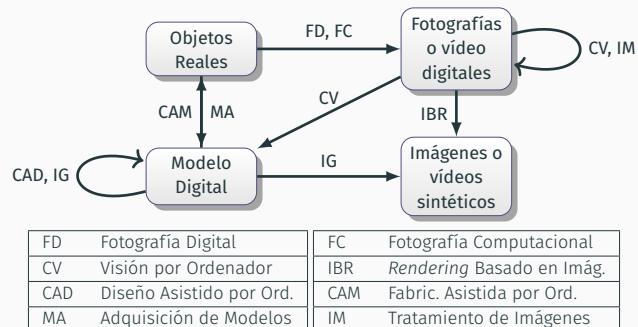
Los elementos esenciales de una aplicación gráfica son:

- ▶ **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:



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



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 7 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 8 de 256

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.
Sección 1. Introducción

Subsección 1.2. Aplicaciones..

Aplicaciones

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 10 de 256

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>

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 11 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 12 de 256

Películas y animaciones generadas por ordenador



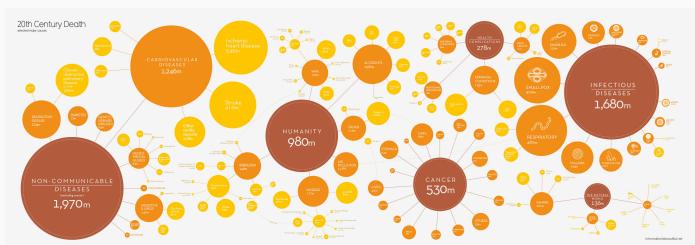
Image courtesy of Digid Pictures © 2013 Ubisoft Entertainment. All rights reserved. Watch Dogs and Ubisoft, and the Ubisoft logo are trademarks of Ubisoft Entertainment in the US and/or other countries.

Fotograma del tráiler cinematográfico del videojuego Watch Dogs. Imagen creada por Digid 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/>

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 13 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 14 de 256

Visualización en Medicina



Obtenido del sitio web MIT Technology Review

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 15 de 256

Cirugía asistida con Realidad Aumentada

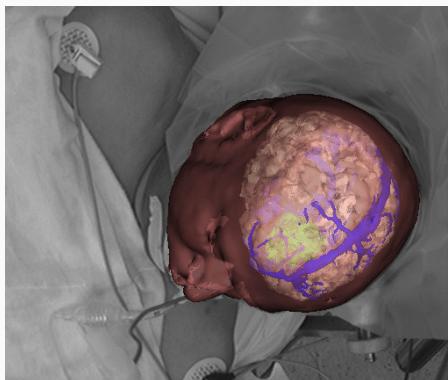
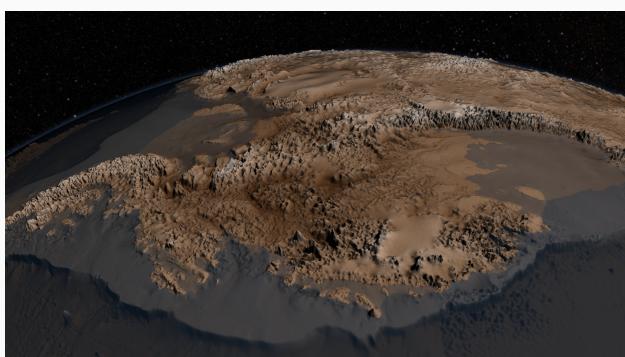


Imagen creada por Christopher Brown, Universidad de Rochester:

<http://www.cs.rochester.edu/u/brown/projects.html>

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 16 de 256

Visualización científica (geología)

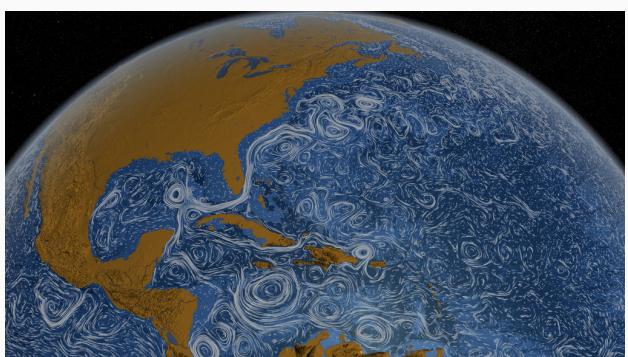


Visualización de la topografía del suelo de la Antártica (NASA):

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 17 de 256

Visualización científica (climatología)



Visualización de las corrientes oceánicas (NASA):

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 18 de 256



Simulador de conducción de Mercedes-Benz:

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 19 de 256



Fotografía (izquierda) y visualización 3D de un modelo (derecha).

Proyecto *The Digital Michelangelo*, de la Universidad de Standford.<http://graphics.stanford.edu/projects/mich/>

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 20 de 256

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.Sección 2.
El proceso de visualización.Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.

Sección 2. El proceso de visualización

Subsección 2.1.

Programas gráficos: interactivos versus off-line.

- 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

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 23 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 24 de 256

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.

Subsección 2.2.

El proceso de visualización.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 25 de 256

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)

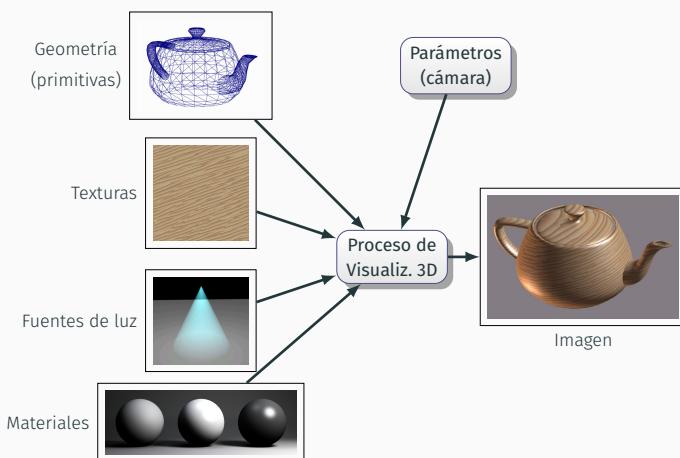
El proceso de visualización produce una imagen a partir de un **modelo de escena** y unos **parámetros**:

- ▶ **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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 27 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 28 de 256

El proceso de visualización 3D: esquema



Subsección 2.3.

Rasterización y ray-tracing..

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 29 de 256

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 $O(n \cdot p)$)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 31 de 256

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 $O(p \log n)$)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 32 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 33 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 34 de 256

El cauce gráfico: entradas y salidas

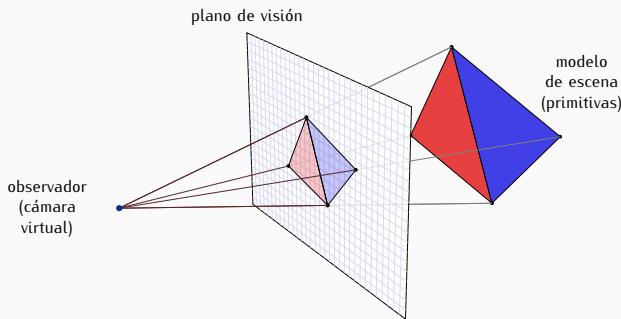
El **cauce gráfico** es el conjunto de etapas de cálculo que permiten la síntesis de imágenes por rasterización:

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.
Sección 2. El proceso de visualización
Subsección 2.4.
El cauce gráfico en 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.

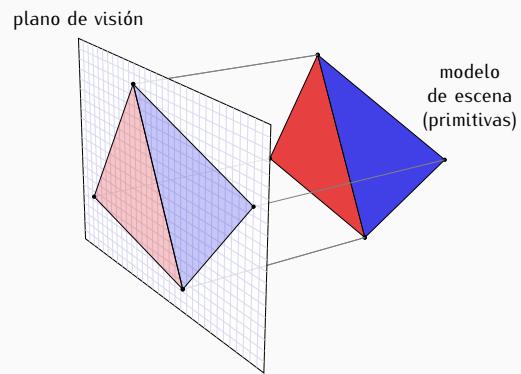
GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 36 de 256

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):



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 37 de 256

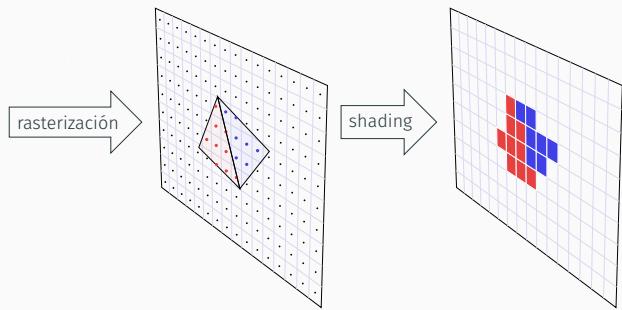
La proyección puede ser **perspectiva** (como en la transparencia anterior), o **paralela**, como aparece aquí:



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 38 de 256

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.



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 39 de 256

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)

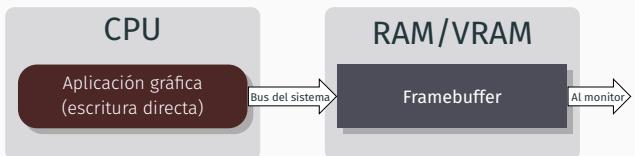


GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 40 de 256



Aplicaciones de escritura directa

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

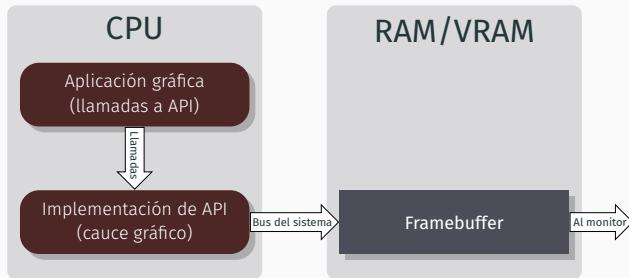


Desventajas

- ▶ 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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 42 de 256

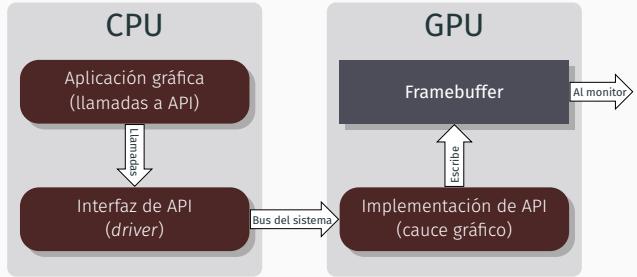
El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad y acceso simultáneo**



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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 43 de 256

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



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 44 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 45 de 256

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 Direct X, 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 aprovechar mejor el hardware.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 46 de 256

Etapas del cauce gráfico (1/2)

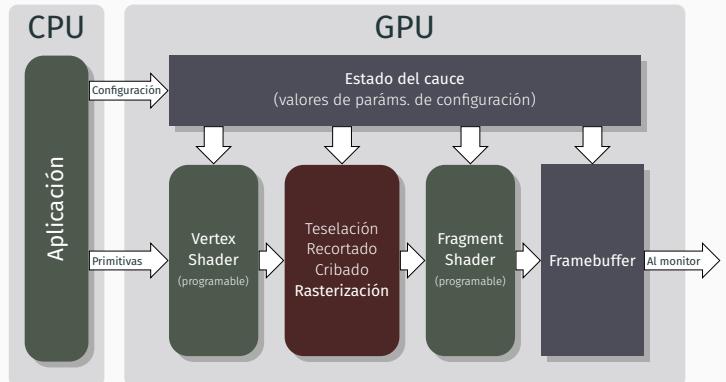
El cauce gráfico tiene estas etapas:

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.

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.

DFD simplificado de una aplicación gráfica y el cauce en GPU



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 49 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 50 de 256

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)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 51 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 52 de 256

- 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



- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 55 de 256

- ▶ 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).

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 56 de 256

Historia de OpenGL.

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 57 de 256

La librería GLFW

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 58 de 256

Repositorio público con ejemplo en OpenGL

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

<https://github.com/carlos-urena/opengl3-minimo>

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

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

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

Subsección 3.2.

Programación y eventos en GLFW.

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.

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(argc,argv); // crea una ventana
    Inicializa_OpenGL(); // inicializa estado del cauce
    BucleEventos(); // 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 explícitamente):

- ▶ 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,
                                    "Prácticas IG (19-20)", nullptr, nullptr );
    glfwMakeContextCurrent( ventana_glfw ); // necesario para OpenGL
}

....
```

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* 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);
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 67 de 256

```
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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 68 de 256

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.
Sección 3. La librería OpenGL (y GLFW). Visualización.
Subsección 3.3.
Tipos de primitivas..

Especificación de primitivas

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 70 de 256

Tipos de primitivas: puntos y segmentos

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

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

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

Tipos de primitivas: triángulos

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

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

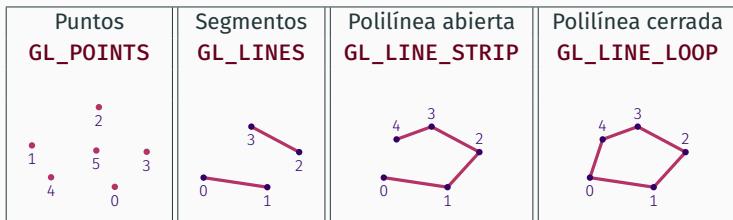
En las primeras versiones de OpenGL se contemplaban primitivas de tipo cuadrilátero y polígono, pero en las versiones actuales estas primitivas no se contemplan.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 71 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 72 de 256

Primitivas de tipo puntos y segmentos.

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



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 73 de 256

Triángulos delanteros y traseros. Cribado.

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

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

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

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

Esta clasificación tiene utilidad especialmente en visualización 3D.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 74 de 256

Modo de visualización de triángulos.

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 75 de 256

Primitivas tipo triángulos (no adyacentes)

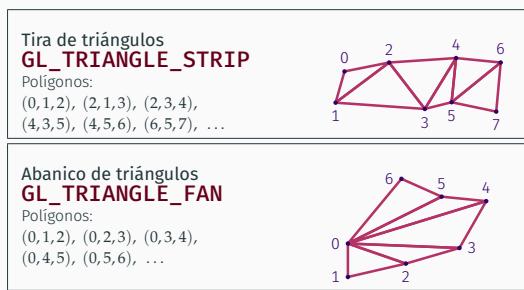
La visualización de una secuencia de n vértices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_{n-1}$ (que codifica $n/3$ triángulos) depende del modo de visualización de polígonos.



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 76 de 256

Primitivas tipo triángulos (adyacentes)

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



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 77 de 256

Polígonos de más de tres vértices

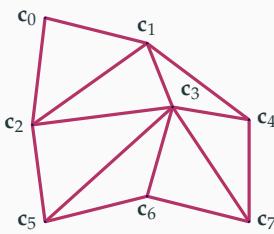
Respecto a la posibilidad de visualizar primitivas de más de tres vértices:

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 78 de 256

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:

$$\{ \begin{array}{l} \mathbf{c}_0, \mathbf{c}_2, \mathbf{c}_1, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \\ \mathbf{c}_1, \mathbf{c}_3, \mathbf{c}_4, \mathbf{c}_2, \mathbf{c}_5, \mathbf{c}_3, \\ \mathbf{c}_3, \mathbf{c}_5, \mathbf{c}_6, \mathbf{c}_3, \mathbf{c}_6, \mathbf{c}_7, \\ \mathbf{c}_3, \mathbf{c}_7, \mathbf{c}_4 \end{array} \}$$

Supone emplear más memoria y/o tiempo para visualizar del necesario. En este ejemplo necesitamos una secuencia de 21 coordenadas de vértices, de las cuales solo hay 8 distintas (p.ej., las coordenadas \mathbf{c}_3 aparecen repetidas 6 veces)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 79 de 256

Secuencias indexadas

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

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

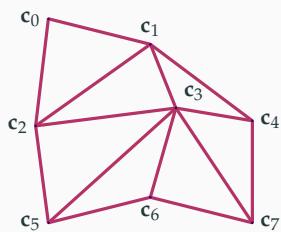
$$S_m = \{ \mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_{m-1}} \}$$

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 80 de 256

Ejemplo de secuencia indexada

En este ejemplo que hemos visto antes



Haríamos:

$$\begin{aligned} V_8 &= \{ \mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4, \mathbf{c}_5, \mathbf{c}_6, \mathbf{c}_7 \} \\ I_{21} &= \{0, 2, 1, 1, 2, 3, 1, 3, 4, 2, 5, 3, 3, 5, 6, 3, 6, 7, 3, 7, 4\} \end{aligned}$$

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 81 de 256

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

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

Subsección 3.4.
Atributos de vértices.

Atributos de vértices

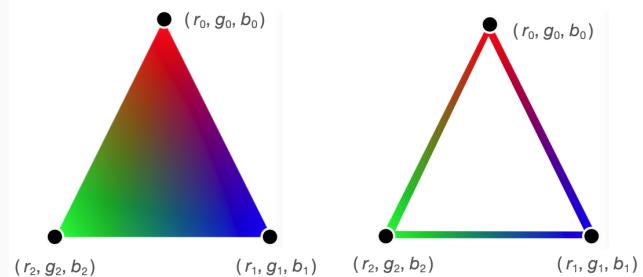
Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se pueden opcionalmente usar otros atributos, por ejemplo:

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

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

Atributos: colores de vértices

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

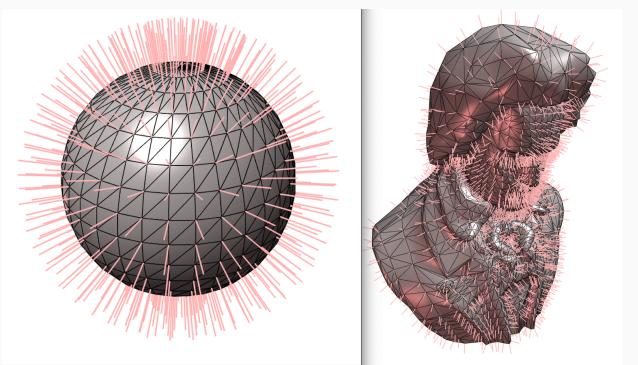


GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 83 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 84 de 256

Atributos: normales

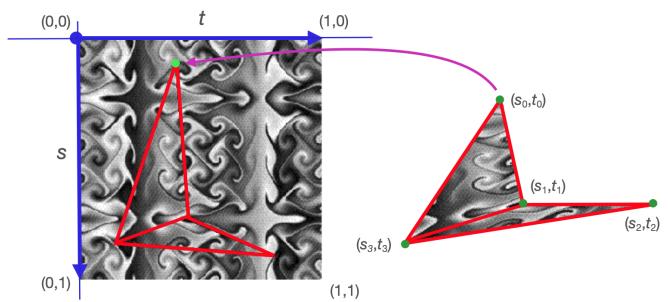
En visualización 3D, a cada vértice se le puede asociar un vector de 3 componentes (x, y, z) (su vector **normal**) que determina la orientación de la superficie en ese vértice y sirve para hacer el sombreado y la iluminación:



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 85 de 256

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):



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 86 de 256

Definición de valores de atributos

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 87 de 256

Índices de atributos

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 88 de 256

Modos de envío

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

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

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

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

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

Subsección 3.6.

Almacenamiento de vértices y atributos.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 91 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 93 de 256

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

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

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

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

posiciones	→	\overbrace{x_0, y_0, z_0}^{\text{posición 0}}, \overbrace{x_1, y_1, z_1}^{\text{posición 1}}, \dots, \overbrace{x_{n-1}, y_{n-1}, z_{n-1}}^{\text{posición n-1}}
colores	→	\overbrace{r_0, g_0, b_0}^{\text{color 0}}, \overbrace{r_1, g_1, b_1}^{\text{color 1}}, \dots, \overbrace{r_{n-1}, g_{n-1}, b_{n-1}}^{\text{color n-1}}
normales	→	\overbrace{n_{x0}, n_{y0}, n_{z0}}^{\text{normal 0}}, \overbrace{n_{x1}, n_{y1}, n_{z1}}^{\text{normal 1}}, \dots, \overbrace{n_{x,n-1}, n_{y,n-1}, n_{z,n-1}}^{\text{normal n-1}}
cc.textura	→	\overbrace{u_0, v_0, u_1, v_1}^{\text{cct. 0}}, \overbrace{\dots, u_{n-1}, v_{n-1}}^{\text{cct. 1}}, \overbrace{\dots, u_{n-1}, v_{n-1}}^{\text{cct. n-1}}

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 94 de 256

Almacenamiento de vértices en la aplicación

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

AOS

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

SOA

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 95 de 256

Almacenamiento de secuencias de vértices en la aplicación.

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 96 de 256

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 97 de 256

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

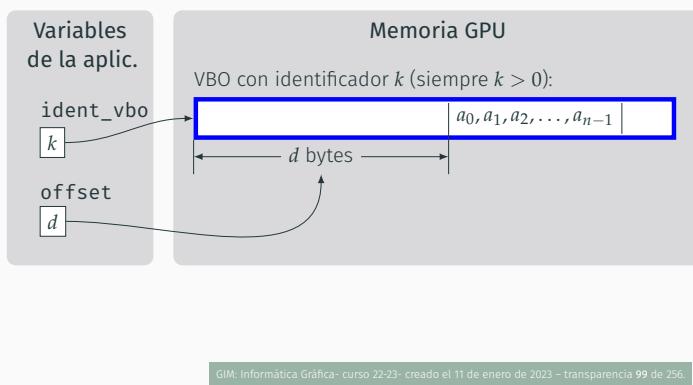
Un **Vertex Buffer Object** es una secuencia de bytes contiguos (un bloque de memoria) en la memoria de la GPU. Dicho bloque contiene una o varias tablas con coordenadas, colores u otros atributos de vértices.

- ▶ 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).

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 98 de 256

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:



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 99 de 256

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

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 100 de 256

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

3. **Número de valores por tupla o size** (**GLint num_vals_tupla**): número de valores reales por vértice, en general puede ser de 1 a 4 para las tablas de atributos y 1 en tablas de índices. En nuestro caso será 3 para las posiciones y resto de atributos, excepto para las coordenadas de textura, que vale 2.
4. **Número de tuplas (número de vértices) o count** (**GLsizei num_tuplas**): número de tuplas de datos, coincide con el número vértices, ya que debe haber exactamente una tupla por vértice.
5. **Puntero a datos o data (const void * datos)**: puntero a la dirección de memoria del programa donde está el primer byte de la tabla. No puede ser nulo y se usa simplemente para lectura.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 101 de 256

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 102 de 256

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

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

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

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

Otros parámetros de una tabla de índices

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

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

Almacenamiento de secuencias de vértices en GPU

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

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

Vertex Array Objects (VAO) en OpenGL

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

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

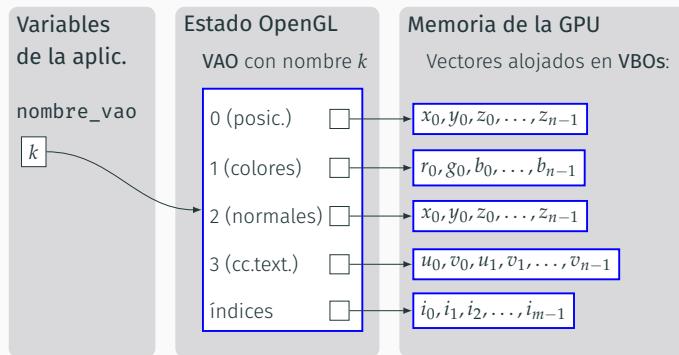
- ▶ Una aplicación debe crear una de estas estructuras por cada secuencia de vértices que quiera visualizar.
- ▶ Cada VAO forma parte del estado de OpenGL, y se usa o actualiza exclusivamente mediante llamadas a OpenGL (no directamente).

Vertex Array Objects (VAOs) de OpenGL

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

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

Esquema de un VAO



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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 109 de 256

Operaciones sobre VAOs

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 110 de 256

Operaciones sobre VBOs en el VAO activo

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 111 de 256

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 112 de 256

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

```
Glenum CrearVBOAtrib( ..... )
{
    .....

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

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 113 de 256

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 114 de 256

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

Esta función crea y deja activado un VBO de índices,

```
Glenum CrearVBOInd( Glenum tipo_indices, GLint num_indices,
                      const void * indices )
{
    // 1. comprobar la integridad de los parámetros
    assert( num_indices > 0 );
    assert( indices != nullptr );
    assert( tipo_indices == GL_UNSIGNED_BYTE || tipo_indices == GL_UNSIGNED_SHORT || tipo_indices == GL_UNSIGNED_INT );

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 115 de 256

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

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 116 de 256

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

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 117 de 256

Creación y activación de un VAO

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 118 de 256

Creación de VAOs para tablas de tuplas

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

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

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

Este código debe ejecutarse:

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

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

Problemas: generación de aristas de un polígono regular (1/2)

Problema 1.1.

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

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

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

(el enunciado continua en la siguiente transparencia)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 119 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 120 de 256

Problema 1.1. (continuación)

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 121 de 256

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 llenos que comparten un vértice en el centro del polígono (el origen). Usa ahora datos de simple precisión **float** para los vértices, con tres valores por vértice, siendo la Z igual a 0 en todos ellos.

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 122 de 256

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.
Sección 3. La librería OpenGL (y GLFW). Visualización.

Subsección 3.7.
Envío de vértices y atributos..

Funciones de visualización en modo diferido

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 124 de 256

La función glDrawArrays

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

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

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

La función glDrawElements

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 125 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 126 de 256

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 127 de 256

Problema 1.3.

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

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

(el enunciado continua en la siguiente transparencia)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 128 de 256

Problema: visualización polígono regular (2/2)**Problema 1.3. (continuación)**

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 129 de 256

Problema: visualización de polígono regular con VAO único**Problema 1.4.**

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 130 de 256

Problema: visualización de polígono regular con colores**Problema 1.5.**

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 131 de 256

Problema: visualización de polígono regular con colores**Problema 1.6.**

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 132 de 256

Estado de OpenGL y visualización de un frame.

Subsección 3.8.

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](#))

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 134 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 135 de 256

Color de vértices y modos de sombreado

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



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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 136 de 256

Cambio del modo de sombreado

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

Para cambiarlo usamos:

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

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

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

Eliminación de partes ocultas (EPO) con Z-buffer

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

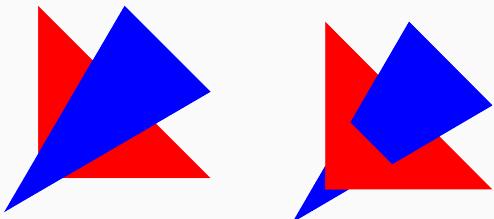
- ▶ Existe un buffer (llamado **Z-buffer**) donde se guarda la coordenada Z de lo que hay dibujado en cada pixel. Esto permite hacer el **test de profundidad** (*depth test*).
- ▶ Permite dibujar primitivas en 2D o 3D con posibles ocultaciones entre ellas.
- ▶ Inicialmente (por defecto) en un pixel,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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 137 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 138 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 139 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 140 de 256

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

Lectura del estado de OpenGL

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

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

```
GLfloat rango_lineas[2] ; // contendrá mínimo y máximo
glGetFloatv( GL_ALIASED_LINE_WIDTH_RANGE, rango_lineas );
```
- ▶ Si está habilitado el test de profundidad o no:

```
GLboolean test_habilitado = glIsEnabled( GL_DEPTH_TEST );
```
- ▶ El valor actual de algún atributo de vértices, identificado por su índice de atributo

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 141 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 142 de 256

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

Definición del *viewport*

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

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

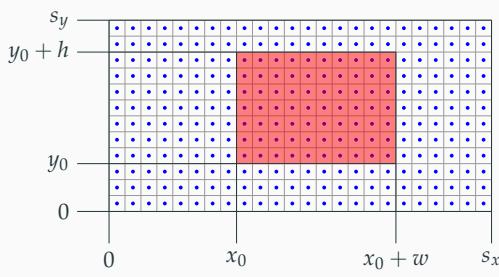
Los parámetros de la función (todo enteros, no negativos) son los siguientes (en orden)

- ▶ **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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 143 de 256

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):



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 144 de 256

La función gestora del cambio de tamaño de ventana

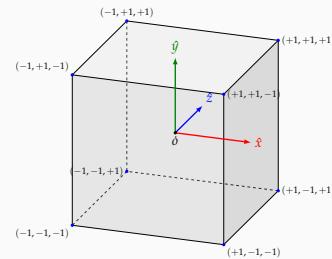
El evento de cambio de tamaño de la ventana se produce siempre una vez tras crear la ventana, y además siempre después de que se cambie su tamaño.

- ▶ 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):



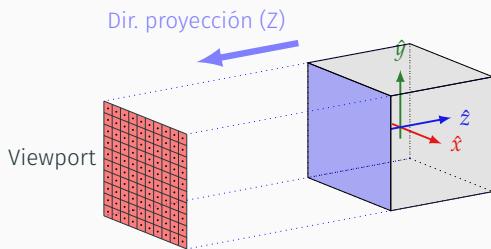
Las primitivas o partes de primitivas fuera de esta región no se dibujan (quedan *descartadas* o *recortadas*).

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 145 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 146 de 256

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:

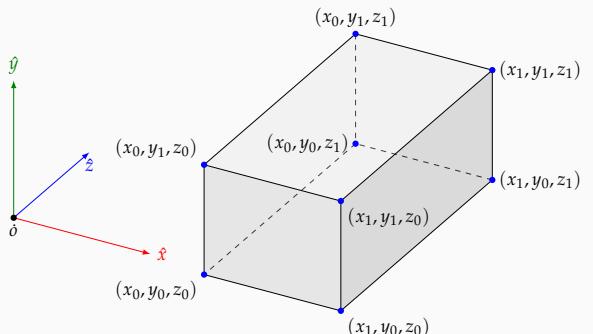


Nota: si se activa EPO, las primitivas con Z menor ocultan a las primitivas con Z mayor.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 147 de 256

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:



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 148 de 256

Cambiar región visible: la *matriz de proyección*

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

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

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

donde:

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

Cambiar la matriz de proyección: llamadas

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

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

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

```
glUniformMatrix4fv( loc_proyección, 1, GL_TRUE, matriz_proyección );
```

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 149 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 150 de 256

Problema 1.7.

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 151 de 256

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 rasterizan las primitivas en el *framebuffer* y después se visualiza en la ventana la imagen ya creada en dicho *framebuffer*.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 152 de 256

Ejemplo de función de redibujado

Un ejemplo sencillo para la función de redibujado es esta:

```
void VisualizarFrame()
{
    // comprobar si ha habido error, restablecer variable de error
    CError();
    // configurar estado de OpenGL (cámara, modo de polígonos, etc..)
    .....

    // limpiar la ventana: limpiar colores y limpiar Z-búffer
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // envío de secuencias de vértices (dibujamos varios objetos)
    .....

    // visualización de la imagen creada
    glfwSwapBuffers() ;

    // comprobar si ha habido error en esta función
    CError();
}
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 153 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 154 de 256

La macro CError

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 155 de 256

Documentación on-line sobre OpenGL y GLFW

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 156 de 256

- 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.

Transformación y sombreado

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

1. Transformación:

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

2. Sombreado:

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

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

Los shaders. Tipos.

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

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

Tipos de cauces gráficos:

Hay dos tipos de cauce gráfico:

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

Etapas del cauce gráfico y shaders (1/2)

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

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

(sigue).

Etapas del cauce gráfico y shaders (2/2)

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

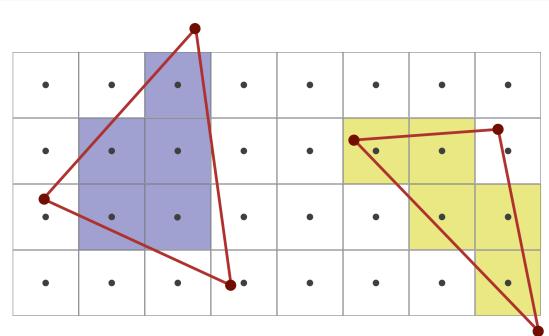
3. Rasterizado e interpolación de atributos (no programable).
4. Procesado de fragmentos: **Fragment Shader** (programable, obligatorio).
5. Procesado de muestras: etapas adicionales que se ejecutan después (algunas antes) del procesado de fragmentos, incluyendo, por ejemplo, el test de profundidad (no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

Más info en el sitio web del consorcio Khronos:

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

Ejemplo: Visualización de 2 triángulos

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

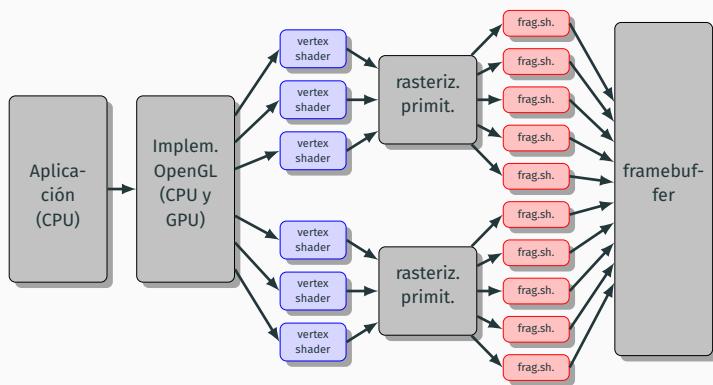


GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 163 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 164 de 256.

Cauce gráfico: DFD simplificado

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



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 165 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 166 de 256.

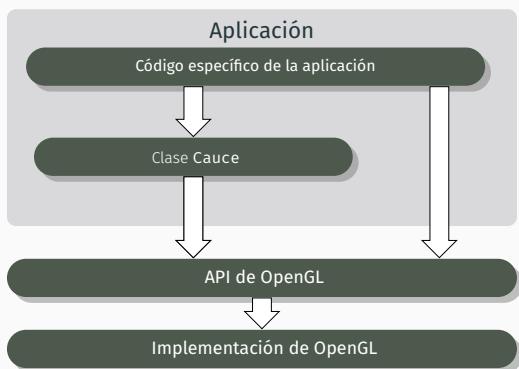
Implementación de cauce programable

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

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

Clase para interfaz de acceso al cauce

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



Esto permite abstraernos de los detalles de los shaders.

Ejemplos de métodos de configuración del cauce

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 167 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 168 de 256.

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 170 de 256

Elementos del fuente de los shaders: declaraciones

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 171 de 256

Entradas y salidas según el tipo de shader.

Vertex Shaders (se ejecutan una vez por vértice)

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

Fragment Shaders (se ejecutan una vez por pixel)

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 172 de 256

Ejemplo de *shaders* mínimos: declaraciones

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 173 de 256

Ejemplo de *shaders* mínimos: Vertex Shader

```
#version 330 core

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 174 de 256

```
#version 330 core

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 175 de 256

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 176 de 256

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.
Sección 4. Programación básica del cauce gráfico
Subsección 4.3.
Creación y ejecución de programas..

Identificación y activación de *shader programs*

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 178 de 256

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 ;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 180 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 179 de 256

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

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

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 181 de 256.

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
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 182 de 256.

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> // (innesesario en macOS)

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 183 de 256.

Uso de un programa.

Lo usual es que durante la inicialización de OpenGL

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 184 de 256.

Uso de la clase Cause y derivadas

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

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

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

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

Sección 4. Programación básica del cauce gráfico

Subsección 4.4.

Funciones auxiliares..

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 185 de 256.

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
}
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 187 de 256.

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
}
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 188 de 256.

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 ;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 189 de 256.

Inicialización de GLEW

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 190 de 256.

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

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

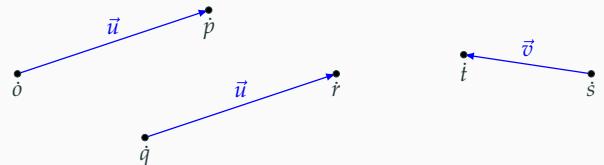


- ▶ Cada modelo 2D o 3D tiene asociado su propio espacio de puntos.
- ▶ Como veremos, los modelos que vamos a visualizar y almacenar en memoria se basan en conjuntos finitos de vértices, y cada uno de ellos se asocia a uno de estos puntos.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 193 de 256

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 194 de 256

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.

$$\dot{p} \quad \vec{v} \quad \dot{q} \quad \dot{q} - \dot{p} = \vec{v} \iff \dot{q} = \dot{p} + \vec{v}$$

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: $\dot{p} + \vec{w} = (\dot{p} + \vec{u}) + \vec{v}$.

$$\vec{w} = \vec{u} + \vec{v} \quad \vec{u} + \vec{v} = \vec{w} \iff \vec{v} = \vec{w} - \vec{u}$$

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 195 de 256

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 $s \neq 1$).

$$\vec{u} \quad \vec{v} = 2\vec{u} \quad \vec{w} = -1.5\vec{u}$$

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}

$$\dot{o} - 2\vec{v} \quad \dot{o} - 1\vec{v} \quad \dot{o} \quad \dot{o} + 1\vec{v} \quad \dot{o} + 2\vec{v}$$

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 196 de 256

Bases de vectores

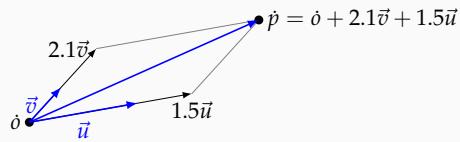
Usando dos vectores cualesquier \vec{u} y \vec{v} del plano (no paralelos ni nulos), podemos escribir cualquier otro vector \vec{w} como una combinación lineal de ellos:

$$\vec{w} = 1.5\vec{u} + 2.1\vec{v}$$

- ▶ 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).

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 198 de 256

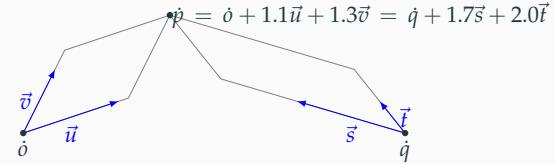
Si fijamos un punto \dot{o} (origen) y una base $\{\vec{u}, \vec{v}\}$, cualquier punto \dot{p} del plano se puede escribir como $\dot{p} = \dot{o} + a\vec{u} + b\vec{v}$:



- ▶ 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} .

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 199 de 256.

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 200 de 256.

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

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

Subsección 5.3. Coordenadas homogéneas.

Coordenadas homogéneas

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 202 de 256.

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 \mathbf{c} para denotar una tupla de coordenadas. Usaremos este tipo de símbolos ($\mathbf{a}, \mathbf{b}, \mathbf{c}, \dots$) 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:

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 203 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 204 de 256.

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

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

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

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

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})

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 205 de 256.

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:

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 206 de 256.

El marco de referencia especial

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 208 de 256.

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:

- ▶ Comutativa: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$
- ▶ 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 $\mathcal{E} = [\hat{x}, \hat{y}, \hat{z}, \delta]$, 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 $\mathcal{E} = [\hat{x}, \hat{y}, \delta]$, se cumple:

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 209 de 256.

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 210 de 256.

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 $\vec{u} \times \vec{v}$.

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 211 de 256

Marcos cartesianos en 2D

Sea $\mathcal{C} = [\vec{e}_x, \vec{e}_y, \dot{\delta}]$ 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 $\mathcal{E} = [\vec{x}, \vec{y}, \delta]$, 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 $\mathcal{E} = [\hat{e}_x, \hat{e}_y, \delta]$ (el marco de referencia \mathcal{E} es cartesiano por definición).

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 212 de 256

Marcos cartesianos en 3D

Sea $\mathcal{C} = [\vec{e}_x, \vec{e}_y, \vec{e}_z, \dot{\delta}]$ 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:

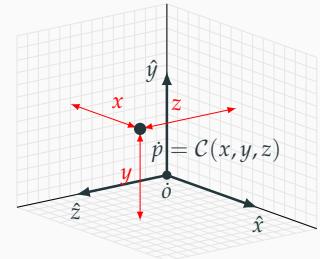
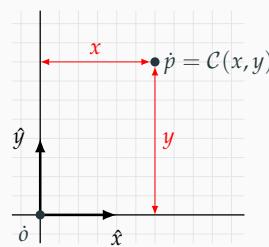
$$\vec{e}_x \times \vec{e}_y = \vec{e}_z \quad \vec{e}_y \times \vec{e}_z = \vec{e}_x \quad \vec{e}_z \times \vec{e}_x = \vec{e}_y$$

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 213 de 256

Marcos y coordenadas cartesianas

En un marco cartesiano $\mathcal{C} = [\hat{x}, \hat{y}, \hat{z}, \delta]$, los versores son paralelos a tres líneas (que pasan por el origen, $\dot{\delta}$) 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 vedor (en 2D).

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 214 de 256

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 $[\hat{e}_x, \hat{e}_y, \hat{e}_z, \dot{\delta}]$ puede tener la misma **orientación** que el marco $\mathcal{E} = [\hat{x}, \hat{y}, \hat{z}, \delta]$ 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**.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 215 de 256

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 usásemos 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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 216 de 256

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

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

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

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

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

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

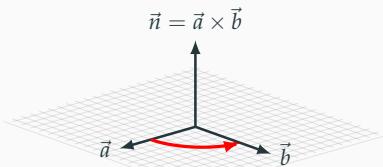
GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 217 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 218 de 256

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*)

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 219 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 220 de 256

Problemas: cálculo del producto vectorial

Problema 1.9.

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

Problema 1.10.

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

Problemas: cálculo del producto escalar

Problema 1.8.

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

Producto vectorial: longitud del vector

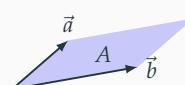
La longitud del vector obtenido como producto vectorial de otros dos está relacionada con el área entre esos otros dos vectores:

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

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

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

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



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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 221 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 222 de 256

Subsección 5.5. Transformaciones geométricas y afines.

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.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 224 de 256.

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:

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

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.

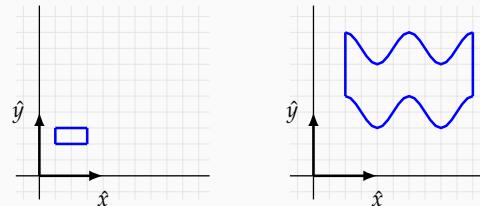
GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 225 de 256.

Ejemplo de transformación en 2D

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

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

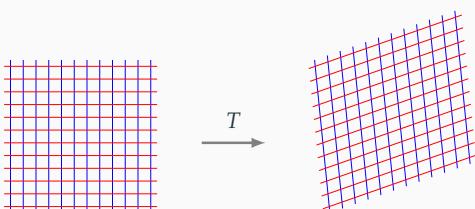
el efecto de T sobre los puntos de un polígono (el rectángulo azul) es el que se aprecia aquí:



GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 226 de 256.

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:

$$\begin{aligned} T(\dot{p} + \vec{u}) &= T(\dot{p}) + T(\vec{u}) \\ T(a\vec{u} + b\vec{v}) &= aT(\vec{u}) + bT(\vec{v}) \end{aligned}$$

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 227 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 228 de 256.

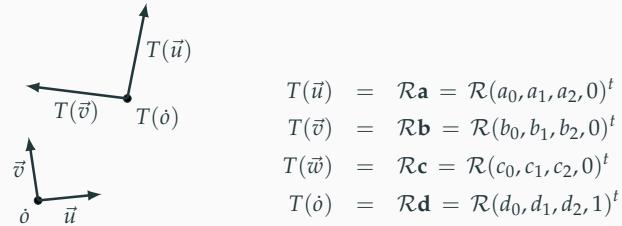
Subsección 5.6. Matrices de transformación.

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: $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$)



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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 230 de 256

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 $\mathcal{R}\mathbf{q} = \mathcal{R}(q_0, q_1, q_2, 1)^t$:

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

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 .

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 231 de 256

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 un 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**. Esta 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 4x4, mientras que en 2D será una matriz 3x3.
- Permite implementar en un programa una transformación afín, especificando su matriz asociada.
- La última fila siempre es 0,0,0,1

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 233 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 234 de 256

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 :

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 235 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 236 de 256.

Ventajas del uso de coords. homogéneas

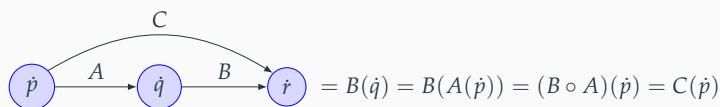
El uso de coordenadas homogéneas permite unificar la matriz R y la matriz D en una única matriz 4x4

- ▶ 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**

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 237 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 238 de 256.

Transformación inversa

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

$$T^{-1}(T(\dot{p})) = \dot{p} \xrightarrow{T} \dot{q} \xrightarrow{T^{-1}} \dot{q} = T(T^{-1}(\dot{q}))$$

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. Consideremos la transformación compuesta

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

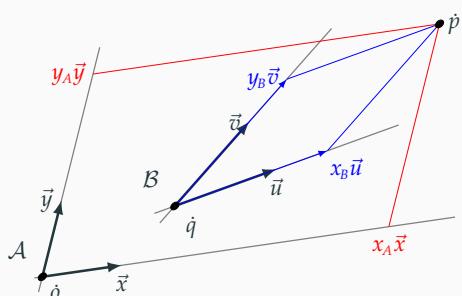
La matriz M_U asociada a U en un marco \mathcal{R} será el producto de las matrices M_i asociadas a T_i en ese marco: asociadas a cada una de las T_i :

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

(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 $\dot{p} = \mathcal{B}(x_B, y_B, z_B, 1)^t = \mathcal{A}(x_A, y_A, z_A, 1)^t$



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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 239 de 256.

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 240 de 256.

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:

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

esto se puede expresar matricialmente:

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

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}

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 241 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 242 de 256

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}

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 243 de 256

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 244 de 256

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 :

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

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

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

de cualquiera de estas dos igualdades se hace evidente que:

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

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}

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 245 de 256

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 $R_{\mathcal{A}, \mathcal{B}}$:

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

- 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 δ) se lleva hasta el origen de \mathcal{B} (el punto \dot{q}), el vector de desplazamiento es

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

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)

Descomposición de la inversa

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

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

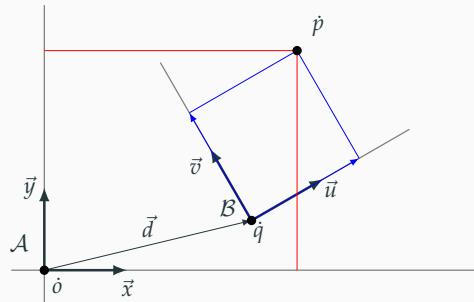
la matriz $D_{\mathcal{A}, \mathcal{B}}^{-1}$ es un desplazamiento que lleva \dot{q} a δ , es decir, un desplazamiento por el vector $\dot{q} - \delta = \mathcal{A}(-d_x, -d_y, -d_z, 0)^t$.

Matricialmente:

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 246 de 256

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 \vec{p} se puede expresar de dos formas:



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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 247 de 256

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} .

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 248 de 256

Informática Gráfica, curso 2022-23.
Teoría. Tema 1. Introducción.
Sección 5. Apéndice: puntos, vectores, marcos, coordenadas y matrices.
Subsección 5.7.
Representación y operaciones con tuplas y matrices.

Tuplas y matrices: el archivo `tup_mat.h`

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

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

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 250 de 256

Ejemplo de uso de las tuplas.

```
// Importar todas las definiciones:
#include <tup_mat.h>
using namespace tup_mat ;

// Tuplas adecuadas para coordenadas de puntos, vectores o normales en 3D
// también para colores (R,G,B)
Tupla3f t1 ; // tuplas de tres valores tipo float
Tupla3d t2 ; // tuplas de tres valores tipo double

// adecuadas para la tabla de caras en mallas indexadas
Tupla3i t3 ; // tuplas de tres valores tipo int
Tupla3u t4 ; // tuplas de tres valores tipo unsigned

// adecuadas para puntos o vectores en coordenadas homogéneas
// también para colores (R,G,B,A)
Tupla4f t5 ; // tuplas de cuatro valores tipo float
Tupla4d t6 ; // tuplas de cuatro valores tipo double

// adecuadas para puntos o vectores en 2D, y coordenadas de textura
Tupla2f t7 ; // tuplas de dos valores tipo float
Tupla2d t8 ; // tuplas de dos valores tipo double
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 251 de 256

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 ;
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 252 de 256

Operaciones entre tuplas y escalares.

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

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

Informática Gráfica, curso 2022-23.

Teoría. Tema 1. Introducción.

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

Subsección 5.8.

Representación y operaciones sobre matrices..

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 253 de 256.

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 <matrizg.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 ;
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 255 de 256.

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 ) ;
```

GIM: Informática Gráfica- curso 22-23- creado el 11 de enero de 2023 – transparencia 256 de 256.

Fin de la presentación.

Informática Gráfica:

Teoría. Tema 2. Modelos de objetos.

Carlos Ureña
2022-23

Grado en Informática y Matemáticas
Grado en Informática y Administración y Dirección de Empresas
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Teoría. Tema 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.

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.

Sección 1.
Modelos geométricos. Introducción..

Modelos geométricos formales

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

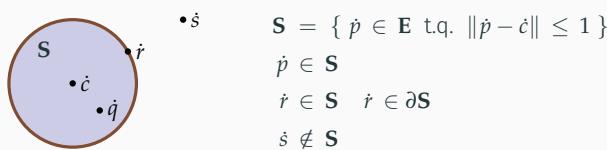
- ▶ Los modelos deben permitir la visualización computacional de los objetos que representan.
- ▶ Los más usados hoy en día son los **modelos de fronteras**: son estructuras de datos que representan la frontera del objeto de forma exacta o aproximada, normalmente mediante **mallas de triángulos**, pero hay otras posibilidades.
- ▶ Un modelo alternativo son los **modelos de volúmenes**.
- ▶ Ultimamente se usan bastante modelos basados en las **funciones de distancia con signo** (*signed distance functions*) o SDFs. Cada objeto se representa con un algoritmo que calcula la distancia (o una cota) desde cualquier punto al objeto.

En la asignatura nos centramos en las mallas de triángulos.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 4 de 184

Los conjuntos de puntos

Los modelos geométricos matemáticos abstractos más generales posibles son los **subconjuntos de puntos** de E , por ejemplo, una esfera:



Cada subconjunto o **región S** es **cerrado** (incluye a su propia **superficie o frontera**, ∂S), además:

- ▶ 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:

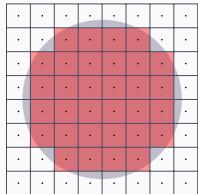
- ▶ permite representar matemáticamente cualquier objeto (es el modelo **más general** posible)
- ▶ pero **no se puede representar en la memoria** (finita, discreta) de un ordenador

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

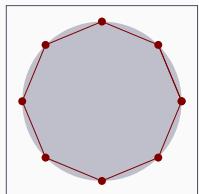
- ▶ **Enumeración espacial:** se partitiona el espacio en celdas (llamados **voxels**), y cada una se clasifica como interior o exterior al objeto.
- ▶ **Modelos de fronteras:** se representa la frontera (la superficie) en lugar de todo el interior, para ello se usan conjuntos finitos de polígonos planos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



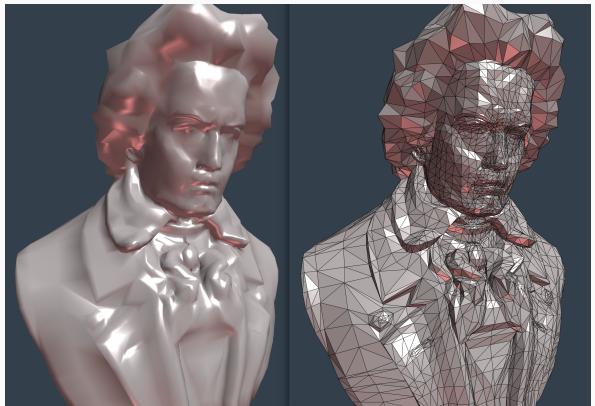
Modelos de fronteras

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 7 de 184.

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

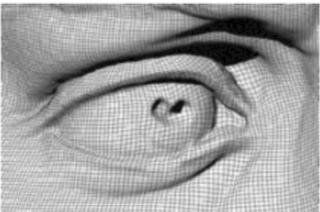
Ejemplo de una **mallas de polígonos** (de las prácticas). A la izquierda el modelo con iluminación, a la derecha vemos las caras que forman el modelo:



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 8 de 184.

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 Ángel en Florencia en 1999.

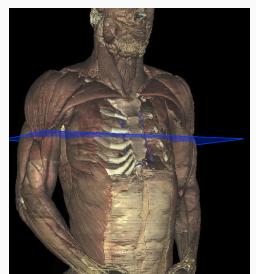
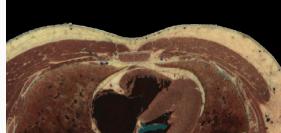
Marc Levoy et al. The Digital Michelangelo Project:

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 9 de 184.

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>

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 10 de 184.

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*.

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

Modelos de fronteras: mallas de polígonos..

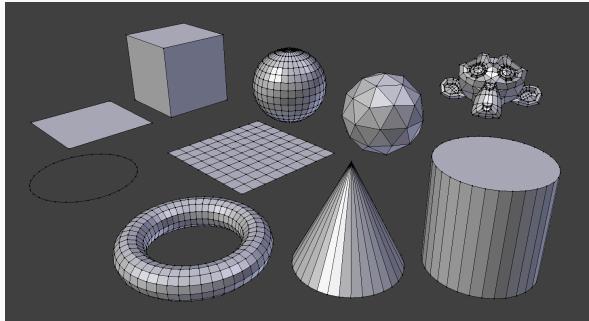
Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.

Sección 2.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 12 de 184.

Ejemplos de mallas

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

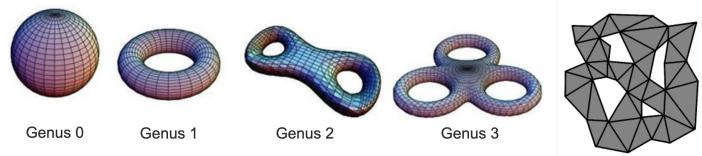


Catálogo de objetos predefinidos de la aplicación *Blender* para modelado 3D:
<https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html>

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 13 de 184.

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:



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

Derecha: [Stackoverflow](#)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 14 de 184.

Elementos de las mallas: vértices

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.
Sección 2. Modelos de fronteras: mallas de polígonos.

Subsección 2.1.
Elementos y adyacencia..

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 16 de 184.

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:

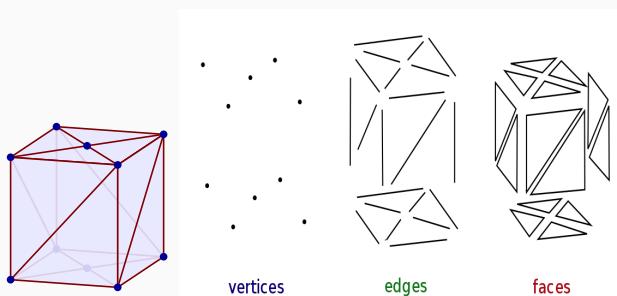
- ▶ la secuencia $\{p_0, p_1, \dots, p_{n-1}\}$ de posiciones de sus n vértices.
- ▶ la secuencia de caras, cada una de ellas representada como una secuencia de k índices de vértice: $\{i_0, \dots, i_{k-1}\}$ (k puede ser distinto en cada cara).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 17 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 18 de 184.

Vértices, caras y aristas

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



[Wikipedia: Polygon Mesh.](#)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 19 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 20 de 184.

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 triangulares coplanares).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 21 de 184.

Características de las 2-variedades

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

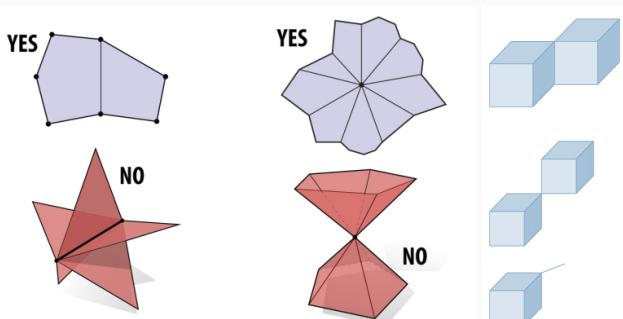
- ▶ Un vértice siempre es adyacente a dos aristas como mínimo (no hay vértices aislados).
- ▶ Una arista siempre es adyacente a una o a dos caras (no hay aristas aisladas, ni aristas adyacentes a 3 o más caras).
- ▶ Todas las caras adyacentes a un vértice se pueden ordenar en una secuencia en la cual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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 22 de 184.

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 et 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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 23 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 24 de 184.

Aristas y vértices de frontera

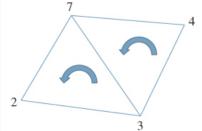
- ▶ Una arista es una **arista de frontera** (o de **borde**) (*boundary edge* o *border edge*) si es adyacente a una única cara.
- ▶ Un vértice es un vértice de frontera si es adyacente a alguna arista de frontera.
- ▶ Una malla es cerrada si y solo si no tiene aristas de frontera (todas las aristas 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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 25 de 184.

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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 26 de 184.

Atributos de las mallas

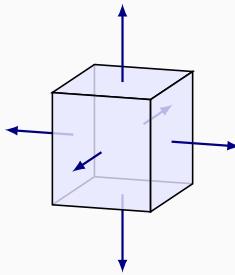
Como modelos de objetos reales, las mallas suelen incluir más información geométrica o del aspecto del objeto:

- ▶ **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...

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 30 de 184.

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:

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

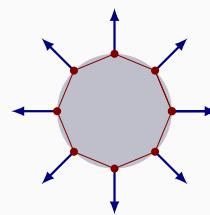
En estos casos la normal se puede precalcular y almacenar en la malla para lograr eficiencia en tiempo de visualización.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 31 de 184.

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:

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

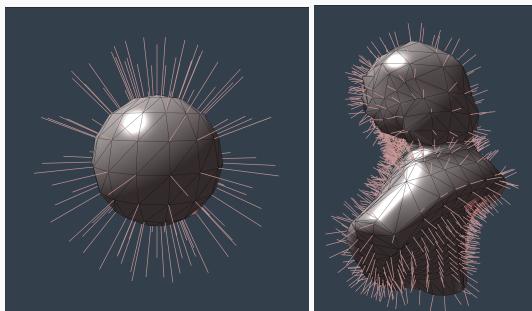
donde $\vec{m}_0, \vec{m}_1, \dots, \vec{m}_{k-1}$ son las normales de las caras adyacentes al vértice.

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 32 de 184.

Ejemplos de normales de vértices calculadas

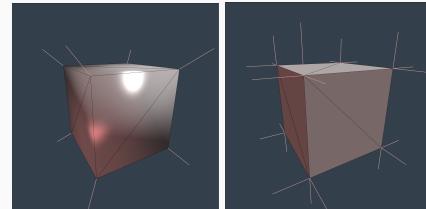
Aquí vemos visualizadas las normales de una esfera de baja resolución (calculadas analíticamente), y de una malla arbitraria (calculadas promediando normales de caras):



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 33 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 34 de 184.

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:

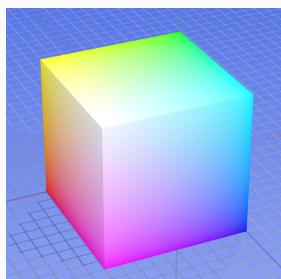


Imagen: http://en.wikipedia.org/wiki/File:RGB_color_solid_cube.png
(Wikimedia Commons)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 35 de 184.

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.

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

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 triángulares**, que es lo más común en la inmensa mayoría de las aplicaciones.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 37 de 184.

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 contiguas.
- ▶ En total, incluye $9n$ valores flotantes.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 39 de 184.

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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 40 de 184.

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,...)
    // ....
};
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 41 de 184.

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:

```
class MallaTA : public Objeto3D
{ protected:
    std::vector<Tupla3f> vertices ; // tabla de vértices (3nt tuplas)
    GLenum nombre_vao = 0 ; // VAO (creado en la visualización)
    // ..... (otros métodos)
public:
    virtual void visualizarGL( ContextoVis & cv ) ;
};
```

La visualización puede hacerse considerando la secuencia de vértices como una **secuencia no indexada** (no hay índices) con **glDrawArrays**, usando **GL_TRIANGLES** como tipo de primitiva. En la instancia se guarda el nombre del VAO.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 42 de 184.

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.

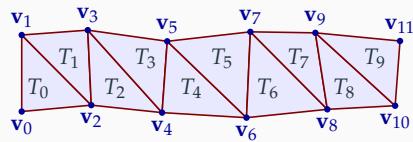
Sin embargo, esta representación

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 44 de 184.

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:

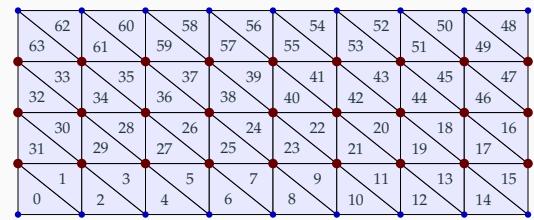


- ▶ Cada tira de n triángulos necesita $n + 2$ tuplas de coordenadas de vértices (tres para el primer triángulo y después una más por cada triángulo adicional)
- ▶ Se almacena una tabla que en la i -ésima entrada almacena las coordenadas del i -ésimo vértice.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 45 de 184.

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:

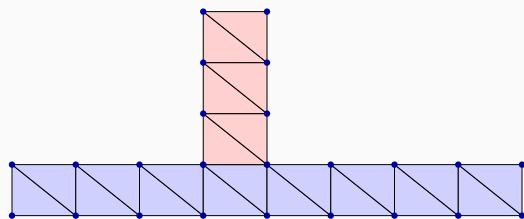


- ▶ las coords. de los vértices en rojo (grandes) se repiten dos veces.
- ▶ las de los vértices en azul (pequeños) aparecen una sola vez.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 46 de 184.

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_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_2}	y_{n_2}	z_{n_2}
x_{n_2+1}	y_{n_2+1}	z_{n_2+1}
x_{n_2+2}	y_{n_2+2}	z_{n_2+2}

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 47 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 48 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 49 de 184.

Tiras de triángulos: Implementación

Una malla de tiras de triángulos se puede representar con una instancia de **MallaTT**, que a su veces tiene una o varias tiras (en **TiraTri**):

```
struct TiraTri // Tira de triángulos (coords. de vértices)
{
    unsigned long num_tri; // número de triángulos en esta tira
    std::vector<Tupla3f> ver; // coords. de vert. (num_tri*2 entradas)
    GLuint nombre_vao = 0; // VAO (creado en la visu.)
};

class MallaTT : public Objeto3D // Malla compuesta de tiras de triángulos
{
protected:
    std::vector<TiraTri> tiras; // vector de tiras
    .....
public:
    virtual void visualizarGL( ContextoVis & cv );
};
```

Cada tira puede tener su propio VAO y se visualiza con **glDrawArrays** usando **GL_TRIANGLE_STRIP** como tipo de primitiva.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 50 de 184.

Informática Gráfica, curso 2022-23.
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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 52 de 184.

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.)			Tabla Vértices (m verts.)		
$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	0	x_0	y_0
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$	1	x_1	y_1
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	2	x_2	y_2
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$	3	x_3	y_3
:	:	:	4	x_4	y_4
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$:	:	:
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$	$m-2$	x_{m-2}	y_{m-2}
$0 \leq i_{jk} < m$			$m-1$	x_{m-1}	y_{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<Tupla3u> triangulos; // tabla de triángulos (índices)
    GLuint nombre_vao = 0; // VAO
    .....
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.

Se visualiza con **glDrawElements** usando un VAO y **GL_TRIANGLES** como tipo de primitiva.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 53 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 54 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 55 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 56 de 184.

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
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 57 de 184.

El formato OBJ se usa bastante hoy en día, es parecido a PLY, pero con las normales y coordenadas de textura indexadas:

- ▶ Incluye una tabla de vértices y una tabla de triángulos, igual que PLY
- ▶ Además, incluye tablas normales y coordenadas de textura. A diferencia de PLY, su tamaño no tiene porque coincidir con el de la tabla de vértices.
- ▶ En cada cara, cada vértice se representa por un índice de sus coordenadas de posición, y opcionalmente, otros índices independientes para su normal y sus coordenadas de textura.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 58 de 184.

Formato OBJ: valoración

La ventaja frente a PLY (malla indexada de triángulos) es una mayor flexibilidad lo que permite mayor eficiencia en memoria:

- ▶ dos vértices en posiciones distintas pueden compartir normal (por ejemplo, una malla plana puede tener una única normal, en lugar de tantas como vértices)
- ▶ un vértice único puede tener distintas coords. de textura o distintas normales en distintas caras, no es necesario replicarlo (por ejemplo, un cubo puede tener 8 vértices y solo 6 normales).

La principal desventaja es que OpenGL no puede visualizar directamente este tipo de tablas, así que es necesario convertirlas a una malla indexada de triángulos, replicando normales y coordenadas de textura.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 59 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 60 de 184.

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)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 61 de 184.

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)

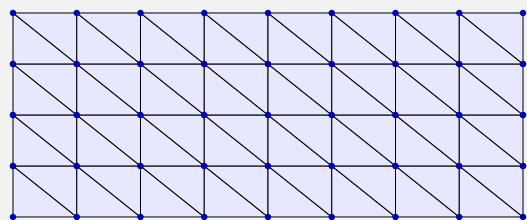
Compara los tamaños de ambas representaciones en ambos casos ($k = 16$ y $k = 1024$).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 62 de 184.

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)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 63 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 64 de 184.

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)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 65 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 66 de 184.

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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 67 de 184.

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)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 68 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 69 de 184.

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 \times , módulo $\| \cdot \|$, etc ...).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 70 de 184.

Representación con estructura de aristas aladas.

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.
Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.4.**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)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 72 de 184.

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-tríangulo**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 73 de 184.

Aristas siguiente y anterior

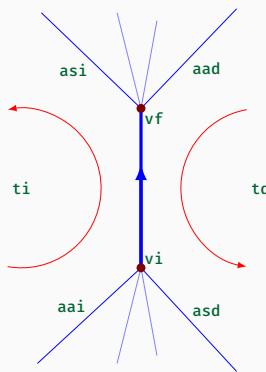
Además de los datos anteriores, se guarda, para cada arista, los índices de otras cuatro adyacentes a ella.

- ▶ 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)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 74 de 184.

Índices asociados a una arista

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



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 75 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 76 de 184.

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).

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.

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 programable
- 4.5. Implementación de *modelview* en la clase **Cauce**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 77 de 184.

Subsección 4.1. Concepto de transformación geométrica (TG)..

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 :

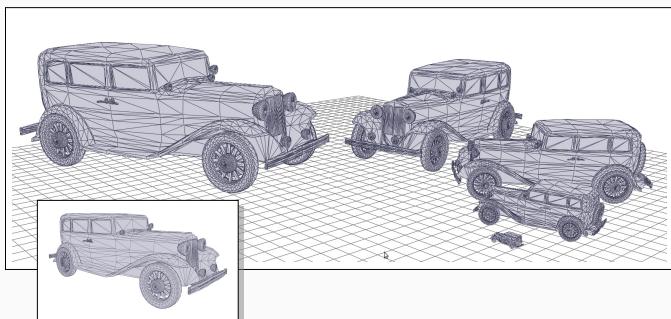
- ▶ En una escena con varias mallas (o, en general, varios objetos) todos los vértices deben aparecer referidos a un único sistema de referencia común
- ▶ Dicho sistema es el llamado **marco de coordenadas de la escena**, o **marco de coordenadas del mundo**, (*world coordinate system*), y es un marco 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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 80 de 184.

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:



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 81 de 184.

Transformación geométrica

Para lograr la instantiación, 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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 82 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 84 de 184.

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 \vec{p} y vector \vec{v} :

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

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

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

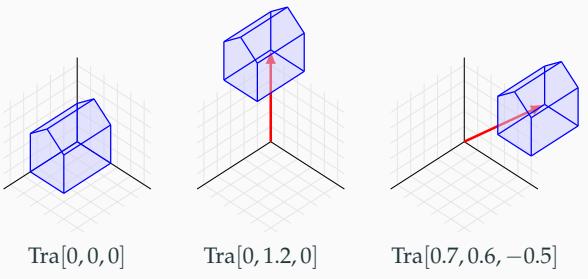
donde $(t_x, t_y, t_z, 0)^t$ son las coordenadas del \vec{t} en \mathcal{R} .

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 85 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 86 de 184.

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



Tra[0, 0, 0]

Tra[0, 1.2, 0]

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:

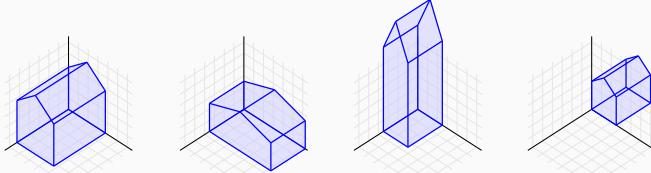
$$\left. \begin{array}{l} x' = e_x x \\ y' = e_y y \\ z' = e_z z \\ w' = w \end{array} \right\} \quad \text{Esc}[s_x, s_y, s_z] \equiv \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 87 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 88 de 184.

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 traslación 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} :

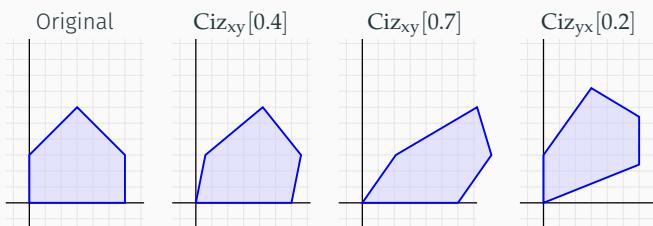
$$\left. \begin{array}{l} x' = x + ay \\ y' = y \\ z' = z \\ w' = w \end{array} \right\} \quad \text{Ciz}_{xy}[a] \equiv \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 89 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 90 de 184.

Ejemplos de transformaciones de cizalla.

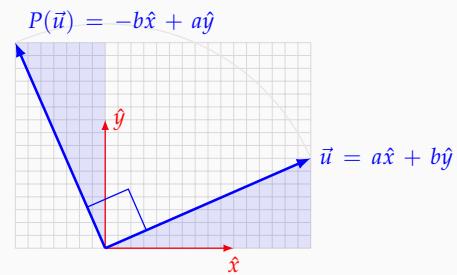
En el caso 2D, hay dos posibles cizallas. Aquí las vemos actuando sobre una figura simple:



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

Rotaciones de 90º 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 $P(\vec{u}) = \mathcal{C}(-b, a)$ perpendicular a \vec{u} (girado 90º a izquierdas).



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 90º entorno al origen de \mathcal{C} .

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 91 de 184.

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 90º es entorno al punto origen δ de \mathcal{C}).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 93 de 184.

Rotaciones de 90º 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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 94 de 184.

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:

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

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 90º entorno al origen de \mathcal{C}

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 95 de 184.

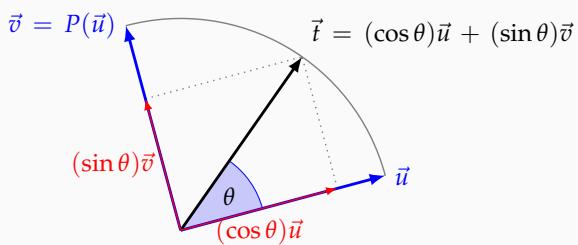
Rotaciones arbitrarias en 2D

Una transformación de rotación R_θ en 2D por un ángulo θ entorno a un punto δ transforma un vector \vec{u} o un punto \vec{p} de esta forma:

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

$$R_\theta(\vec{p}) = \delta + R_\theta(\vec{p} - \delta)$$

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



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 96 de 184.

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:

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 97 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 98 de 184.

Ejemplo de rotación arbitraria en 2D

Aquí vemos el ejemplo de una rotación por un ángulo θ (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 θ 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 θ y vector \vec{v} , se cumple:

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 99 de 184.

Transformaciones de rotación elementales en 3D.

En un espacio afín 3D consideramos un marco cartesiano $\mathcal{C} = [\hat{x}, \hat{y}, \hat{z}, \hat{o}]$ y una transformación de rotación de θ radianes, con eje en un vector \vec{e} no nulo.

- La rotación es en sentido anti-horario cuando $\theta > 0$ (según se mira hacia \hat{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:

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

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

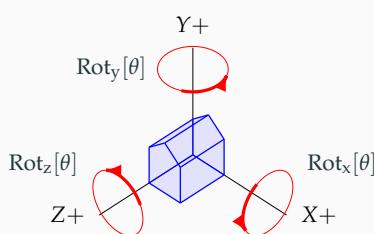
$$\text{Rot}_z[\theta] \equiv \text{Rot}[\theta, (0, 0, 1)]$$

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 100 de 184.

Transformaciones de rotación elementales en 3D.

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



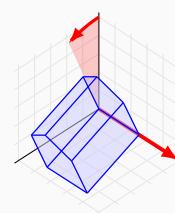
- Son rotaciones **siempre en sentido anti-horario** (para $\theta > 0$) cuando se mira hacia el origen desde la rama positiva del eje de giro).

Expresiones de las rotaciones elementales

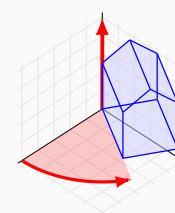
Definiendo $c \equiv \cos\theta$ y $s \equiv \sin\theta$, las expresiones son:

$$\begin{array}{lll} \text{Rot}_x[\alpha] & \text{Rot}_y[\alpha] & \text{Rot}_z[\alpha] \\ x' = x & x' = cx + sz & x' = cx - sy \\ y' = cy - sz & y' = y & y' = sx + cy \\ z' = sy + cz & z' = -sx + cz & z' = z \end{array}$$

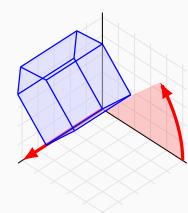
$$\text{Rot}_x[23^\circ]$$



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



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



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 101 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 102 de 184.

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.

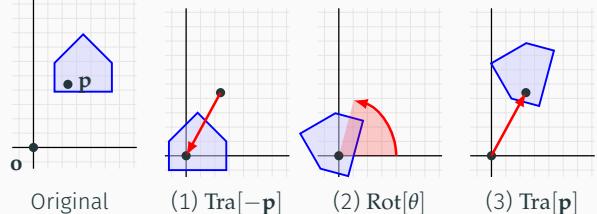
Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualquiera dos vectores \vec{a} y \vec{b} y un ángulo θ , se cumple:

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 103 de 184.

En un marco cartesiano C , la matriz de rotación entorno a un punto \dot{p} arbitrario, distinto del origen o del marco, se consigue componiendo (1) traslación al origen de C (por el vector $\dot{o} - \dot{p}$), (2) rotación por θ (en torno al origen 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 C)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 104 de 184.

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

En 3D una rotación R_θ 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.):

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 105 de 184.

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:

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

$$\begin{aligned} \vec{w} &= \vec{s} - \vec{u} \\ &= \vec{s} + P_{\hat{e}}^2(\vec{s}) \\ &= (\vec{s} \cdot \hat{e})\hat{e} \end{aligned}$$

$\vec{v} = \hat{e} \times \vec{s} = P_{\hat{e}}(\vec{s})$
 $\vec{u} = -\hat{e} \times \vec{v} = -P_{\hat{e}}(\vec{s})$
 $= \vec{s} - (\vec{s} \cdot \hat{e})\hat{e}$

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 106 de 184.

Fórmula de Rodrigues y expresión matricial.

En un marco cartesiano 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 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:

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 107 de 184.

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 C de \vec{e} , y

$$\begin{aligned} s &\equiv \sin \theta & a_{ij} &\equiv (1 - c)e_i e_j \\ c &\equiv \cos \theta & & \end{aligned}$$

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 108 de 184.

$$\text{Esc}[s_x, s_y, s_z] = \quad \text{Tra}[d_x, d_y, d_z] = \quad \text{Ciz}_{xy}[a] =$$

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rot}_x[\alpha] = \quad \text{Rot}_y[\alpha] = \quad \text{Rot}_z[\alpha] =$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 109 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 110 de 184.

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. $\neq 1$).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 111 de 184.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.
Sección 4. Transformaciones geométricas

Subsección 4.3.

Representación y operaciones sobre matrices..

El tipo Matriz4f en tup-mat.h. Operaciones.

El archivo fuente **tup-mat.h** incluye el tipo **Matriz4f**, que almacena los 16 valores (**float**) de forma contigua, proporciona operaciones para acceder a una matriz y multiplicarla por otra matriz

```
#include <tup_mat.h>
...
using namespace tup_mat ;
...

// 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 ;
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 113 de 184.

Operaciones entre matrices y tuplas

Una matriz se puede aplicar a una tupla de 4 flotantes, se puede convertir en un puntero, se puede imprimir en varias líneas, entre otras operaciones:

```
// multiplicación de matriz 4x4 por tupla de 4 floats
Tupla4f c4a, c4b = {1.0,2.0,3.5,1.0};
c4a = m2*c4b ;

// multiplicación por tuplas de 3 flotantes (añadiendo un 1)
Tupla3f c3a, c3b = {1.0,1.6,2.8};
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 ;
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 114 de 184.

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
inline Matriz4f MAT_Ident( ) ;
inline Matriz4f MAT_Traslacion( const Tupla3f & d ) ;
inline Matriz4f MAT_Escalado( const float sx, const float sy, const float sz ) ;
inline Matriz4f MAT_Rotacion( const float ang_gra, const Tupla3f & eje ) ;
inline Matriz4f MAT_Filas( const Tupla3f & fila0, const Tupla3f & fila1,
                          const Tupla3f & fila2 ) ;

// 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_Inversa( const Matriz4f & m );
Matriz4f MAT_Transpuesta3x3( const Matriz4f & org ) ;
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 115 de 184.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.4.

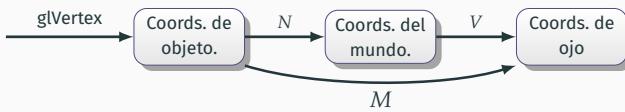
Transformaciones en OpenGL con el cauce programable.

La matriz *Modelview* en OpenGL.

Uno de los parámetros *uniform* de los shaders es 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**:



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 117 de 184.

Construcción de la matriz *modelview* en el caso general

En general, en cualquier aplicación OpenGL, debemos de usar funciones para generar la matriz de vista y la matriz de modelado, y usar esas matrices para configurar el cauce gráfico. Los pasos a dar para visualizar un frame son:

1. Usamos una variable M (matriz), inicialmente igual a la matriz identidad ($M := \text{Ide}$).
2. Componemos una matriz de vista V en M (hacemos $M := M \cdot V$). Podemos usar una función como **MAT_LookAt** u otras (en 2D puede ser simplemente la matriz identidad).
3. Componemos varias matrices de transformación T_1, T_2, \dots, T_n , para ello hacemos $M := M \cdot T_i$, en orden (para i desde 1 hasta n).

Al final, queda M como $V \cdot T_1 \cdot T_2 \cdot \dots \cdot T_n$, y usamos esa matriz M para darle valor a la matriz uniform de los shaders.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 118 de 184.

Matriz *Modelview* en opengl3-minimo: vertex shader

En el repositorio `opengl3-minimo` hay una variable *uniform* (**u_mat_modelview**) en el *vertex shader*, de tipo matriz de 4×4 floats, que se usa para transformar todos los vértices:

```
// variable uniform: matriz de transformación de posiciones
uniform mat4 u_mat_modelview;
// atributo 0: posición del vértice en coordenadas de mundo (salida)
layout( location = 0 ) in vec3 atrib_posicion ;
...

void main()
{
...
// calcular las posiciones del vértice en coords. de mundo y escribimos 'gl_Position'
gl_Position = u_mat_proyeccion *
              u_mat_modelview * vec4( atrib_posicion, 1.0 );
}
```

El código está escrito en GLSL (usa **mat4** para matrices 4×4 y **vec3,vec4** para tuplas de flotantes)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 119 de 184.

Asignaciones a *modelview* en opengl3-minimo

En el código de la aplicación debemos darle valores a la matriz *modelview*

- ▶ Usamos la función **glUniformMatrix4fv** (de la familia **glUniform**).
- ▶ Se necesita la variable entera con el *localizador* del parámetro (**loc_mat_modelview**).

A modo de ejemplo, aquí vemos como asignar una matriz de traslación a la *modelview*:

```
glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE,
                     MAT_Traslacion( { 0.4, 0.1, -0.1 } ) );
```

En otras aplicaciones se pueden usar objetos **Matriz4f** resultado de componer varias matrices.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 120 de 184.

Visualización con *modelview* compuesta.

Suponemos que **DibujarObjeto()** 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 M
Matriz4f M = MAT_Traslacion( {5.0,6.0,7.0} ); // M:=Tra[5,6,7]
M = M * MAT_Rotacion( 45.0, {1.0,0.0,0.0} ); // M:=M·Rot[45º,x]

// Dar valor al uniform del shader
glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, M );

// Visualizar con M ≡ V · Tra[5,6,7] · Rot[45º,x]
DibujarObjeto();
```

Es usual definir funciones auxiliares para gestión de la matriz *modelview*.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 121 de 184.

Funciones auxiliares en *opengl3-minimo*

En el repositorio **opengl3-minimo** hay una matriz (**modelview**) y estas funciones:

- La función **resetMM** sirve para fijar la matriz *modelview* como la matriz identidad

```
void resetMM()
{
    modelview = MAT_Ident();
    .....
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}
```

- La función **compMM** compone una matriz **m** con la matriz *modelview* (por la derecha):

```
void compMM( const Matriz4f & m )
{
    modelview = modelview * m ;
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 122 de 184.

Ejemplo usando las funciones auxiliares

El ejemplo anterior puede escribirse más fácilmente usando estas funciones:

```
// Hacer modelview igual a la identidad (al inicio del frame)
resetMM();

// Dibujar otros objetos (modelview puede cambiar)
.....
// Componer las matrices en la modelview
resetMM(); // M:=Ide
compMM( MAT_Traslacion( {5.0,6.0,7.0} ) ); // M:=Tra[5,6,7]
compMM( MAT_Rotacion( 45.0, {1.0,0.0,0.0} ) ); // M:=M·Rot[45º,x]

// Visualizar con M ≡ V · Tra[5,6,7] · Rot[45º,x]
DibujarObjeto();
```

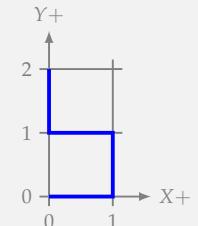
La función **resetMM** debe llamarse al inicio del frame, de forma que siempre comenzamos la visualización de los objetos con la *modelview* fijada a un valor conocido.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 123 de 184.

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).

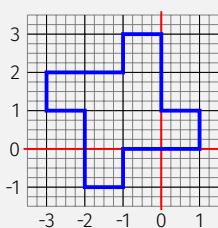
Usa la plantilla en el repositorio **opengl3-minimo** para esto.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 124 de 184.

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 **compMM** con **MAT_Traslacion** y **MAT_Rotacion**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 125 de 184.

Problema: cálculo directo de *modelview*

Problema 2.17.

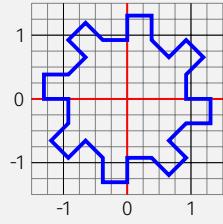
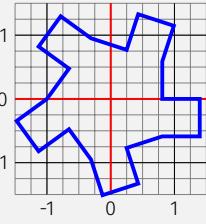
Escribe el pseudocódigo OpenGL otra función (**gancho_2p**) para dibujar esa misma figura, pero escalada y rotada de forma que sus extremos coincidan con dos puntos arbitrarios disintos p_0 y p_1 , puntos cuyas coordenadas de mundo son $p_0 = (x_0, y_0, 1)^t$ y $p_1 = (x_1, y_1, 1)^t$. Estas coordenadas 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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 126 de 184.

Problema 2.18.

Usa 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).



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 127 de 184.

Subsección 4.5.

Implementación de *modelview* en la clase Cauce..

Métodos en la clase Cauce

Para visualización 3D, la clase **Cauce** incluye como variables de instancia las matrices de modelado (*N*) y vista (*V*) por separado. Para manipular estas matrices, se usan estos métodos

► **fijarMatrizVista(*U*)**

fijar *V* como matriz de vista (se asume que *U* tiene una matriz de vista, de tipo **Matriz4f**), y reinicializa la matriz de modelado (*N*) a la identidad. Es decir, hace $V := U$ y $N := \text{Id}_e$. Se debe usar al inicio de cada frame para establecer la vista.

► **compMM(*T*):**

componer en la matriz de modelado actual *N* una matriz *T* por la derecha (también **Matriz4f**), es decir, hace: $N := N \cdot T$. Este método debe invocarse después de **fijarMatrizVista**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 129 de 184.

Ejemplo de manipulación de *modelview* con la clase Cauce

A modo de ejemplo, este código visualiza el objeto **obj** usando una matriz de modelado obtenida como una composición de una rotación (primero), seguida de una traslación después.

```
// Definir matriz 'modelview' en el Estado de OpenGL (usando clase Cauce)
cv.cauce->fijarMatrizVista( MAT_LookAt( o, a, u )); // M := V
cv.cause->compMM( MAT_Translacion( {5.0,6.0,7.0} )); // M := M · Tra[5,6,7]
cv.cause->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 );
```

► El objeto **cauce** forma parte de la estructura **cv** con el contexto de visualización.

► La función **MAT_LookAt** produce una *matriz de vista* (lo vemos después).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 130 de 184.

Implementación: código del *vertex shader*

En el código de prácticas se guardan por separado estas matrices:

- matriz de vista (uniform **u_mat_vista**)
- matriz de modelado (uniform **u_mat_modelado**).
- matriz de modelado de normales (uniform **u_mat_modelado_nor**).

La transformación de vértices en el *vertex shader* se hace así:

```
void main()
{
    vec4 posic_wc = u_mat_modelado * vec4( in_posicion_occ, 1.0 ) ;
    vec3 normal_wc = (u_mat_modelado_nor * vec4(in_normal,0.0)).xyz ;

    // calcular las variables de salida (en coords de vista)
    v_posic_ec = u_mat_vista * posic_wc ;
    v_normal_ec = (u_mat_vista * vec4(normal_wc,0.0)).xyz ;
    .....

    // calcular la posición del vértice en coords de dispositivo
    gl_Position = u_mat_proyeccion * v_posic_ec ;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 131 de 184.

Implementación de la clase Cauce

La clase **Cauce** 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** puede ser así:

```
void Cauce::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    mat_vista      = nue_mat_vista ;
    mat_modelado   = MAT_Ident();
    mat_modelado_nor = MAT_Ident();

    // fijar uniforms u_mat_vista, u_mat_modelado y u_mat_modelado_nor
    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 );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 132 de 184.

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 Cauce::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 u_mat_modelado y u_mat_modelado_nor
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_modelado,      1, GL_FALSE, mat_modelado );
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 133 de 184.

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.
Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.1. Modelos Jerárquicos y grafo de escena..

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.

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.

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}

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 136 de 184.

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**.

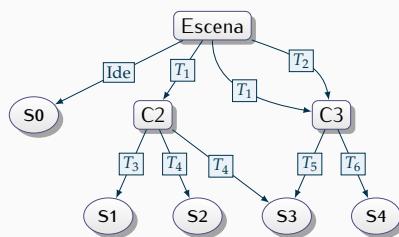
Cada objeto O_i del esquema anterior puede ser de **dos tipos** de objetos geométricos:

- ▶ **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**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 137 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 138 de 184.

Instanciaciones en el grafo

En el grafo anterior:

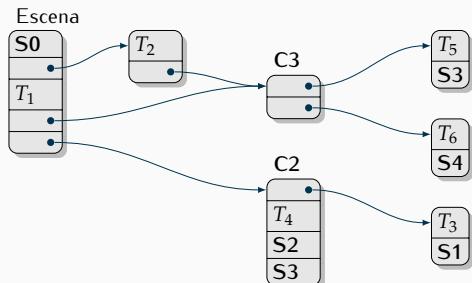
- ▶ Algunas transformaciones pueden ser la matriz identidad (Ide)
- ▶ Algunos pares de hermanos aparecen instanciados con la misma transformación.
- ▶ Algunos nodos (simples o compuestos) aparecen instanciados más de una vez (en el mismo o distinto padre)
- ▶ La transformaciones por las que se instancia 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: S_3 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).

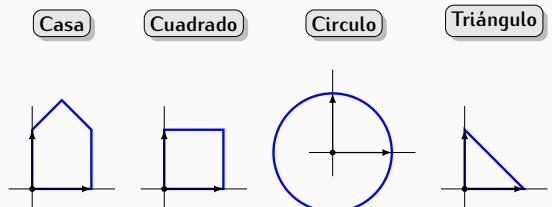
Notación inspirada en PHIGS para grafos de escena

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:



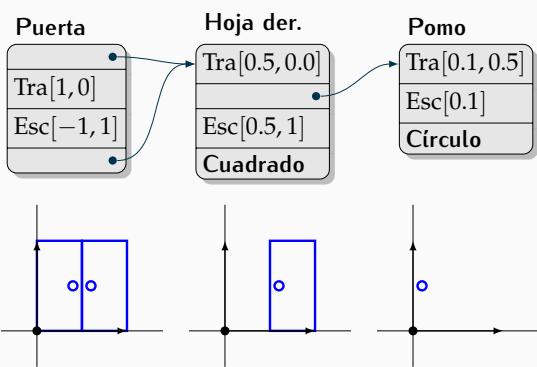
Ejemplo en 2D: objetos simples

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)



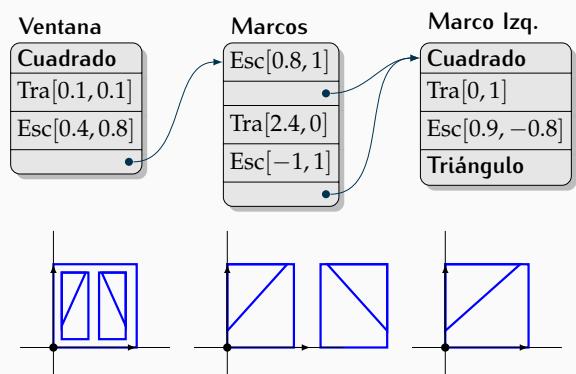
Objeto compuesto: Puerta

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



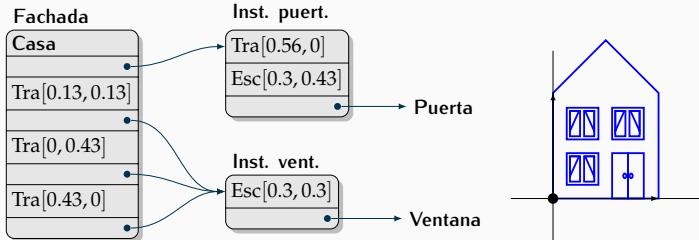
Objeto compuesto: Ventana

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:

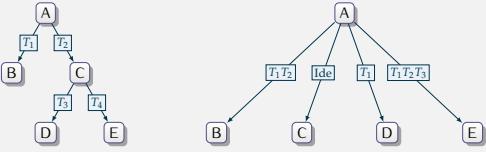


GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 145 de 184.

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 .

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 146 de 184.

Representación de grafos 3D

Informática Gráfica, curso 2022-23.
Teoría. Tema 2. Modelos de objetos.
Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.3. Representación de grafos..

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 148 de 184.

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)  
};
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 149 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 150 de 184.

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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 151 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 152 de 184.

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
};

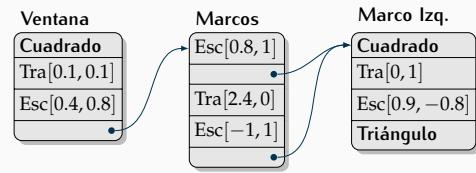
// clase para el nodo del grafo etiquetado como Marcos
class Marcos : public NodoGrafoEscena
{ public:
    Marcos() ; // constructor
};

// 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]
    agregar( MAT_Escalado( 0.4,0.8,1.0 ) ); // Esc[0.4,0.8]
    agregar( new Marcos ); // Marcos
}
```



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 153 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 154 de 184.

Implementación de constructores (2)

```
MarcoIzq::MarcoIzq()
{
    agregar( new Cuadrado ); // Cuadrado
    agregar( MAT_Traslacion( 0.0,1.0,0.0 ) ); // Tra[0,1,0]
    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) como 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.
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 155 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 156 de 184.

Subsección 5.4.

Visualización de grafos en OpenGL..

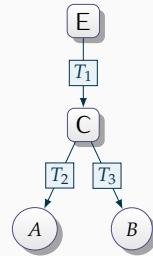
Visualización de varios objetos

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.



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 158 de 184.

Visualización con OpenGL (replicando sentencias)

Para hacer la visualización con OpenGL (usando **resetMM** y **compMM**) se debe usar este código:

```

resetMM();           // M := Ide
compMM( V );        // M := M · V
compMM( T1 );       // M := M · T1
compMM( T2 );       // M := M · T2
VisualizarObjetoA(); // visualizar A con M == V · T1 · T2

resetMM();           // M := Ide
compMM( V );        // M := M · V
compMM( T1 );       // M := M · T1
compMM( T3 );       // M := M · T3
VisualizarObjetoB(); // visualizar B con M == V · T1 · T3
  
```

Es decir: es necesario replicar llamadas para acumular en *M* las matrices *V* y *T₁*. En grafos de escena complejos:

- ▶ El código es poco legible y muy difícilmente modificable, ya que contiene líneas replicadas en muchos sitios.
- ▶ Es lento, ya que se repiten llamadas

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 159 de 184.

Guardar y recuperar la matriz Modelview

Para solventar los problemas descritos, en OpenGL se suelen usar funciones auxiliares que permiten guardar la matriz modelview *M* y restaurarla después. En el caso del repositorio **opengl3-minimo**, usamos las funciones **pushMM** y **popMM**:

resetMM() ;	// <i>M</i> := Ide (únicamente al inicio del frame)
compMM(<i>V</i>) ;	// <i>M</i> := <i>M</i> · <i>V</i>
compMM(<i>T₁</i>) ;	// <i>M</i> := <i>M</i> · <i>T₁</i>
pushMM() ;	// <i>C</i> := <i>M</i> (guarda una copia de <i>M</i> en <i>C</i>)
compMM(<i>T₂</i>) ;	// <i>M</i> := <i>M</i> · <i>T₂</i>
VisualizarObjetoA() ;	// visualizar A con <i>M</i> == <i>V</i> · <i>T₁</i> · <i>T₂</i>
popMM() ;	// <i>M</i> := <i>C</i> (restaura copia de <i>M</i>)
pushMM() ;	// <i>C</i> := <i>M</i> (guarda una copia de <i>M</i> en <i>C</i>)
compMM(<i>T₃</i>) ;	// <i>M</i> := <i>M</i> · <i>T₃</i>
VisualizarObjetoB() ;	// visualizar B con <i>M</i> == <i>V</i> · <i>T₁</i> · <i>T₃</i>
popMM() ;	// <i>M</i> := <i>C</i> (restaura copia de <i>M</i>)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 160 de 184.

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 se necesita tener en su estado una pila LIFO *P* de matrices de transformación *modelview* (initialmente vacía), en lugar de una sola matriz *C*.

```

// al inicio tope == 0
...
pushMM(); // P[tope] = M ; tope=tope+1
...
popMM(); // tope = tope-1 ; M = P[tope]
...
  
```

- ▶ La función **resetMM** reinicializa la pila (únicamente la invocamos al inicio del frame).
- ▶ Cada **pop** debe aparecer con igual indentación que su **push**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 161 de 184.

Implementación de push y pop

La implementación de **pushMM** y **popMM** en **opengl3-minimo** es sencilla, usa **pila_modelview** (un **std::vector** de **Matriz4f**):

```

void resetMM()
{
    modelview = MAT_Ident();
    pila_modelview.clear(); // <- aquí se vacía la pila de matrices
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}

void pushMM()
{
    pila_modelview.push_back( modelview );
}

void popMM()
{
    assert( 0 < pila_modelview.size() ); // falla si pila vacía
    modelview = pila_modelview[ pila_modelview.size()-1 ]; // lee tope
    pila_modelview.pop_back(); // elimina tope
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}
  
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 162 de 184.

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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 163 de 184.

```
void Fachada()
{ pushMM();
  Casa();
  pushMM(); // inst.puerta
  compMM( MAT_Traslacion( { 0.56, 0.0, 0.0 } ) );
  compMM( MAT_Escalado( 0.3, 0.43, 1.0 ) );
  Puerta();
  popMM();
  compMM( MAT_Traslacion( { 0.13, 0.13, 0.0 } ) );
  InstVentana();
  compMM( MAT_Traslacion( { 0.0, 0.43, 0.0 } ) );
  InstVentana();
  compMM( MAT_Traslacion( { 0.43, 0.0, 0.0 } ) );
  InstVentana();
  popMM();
}
```

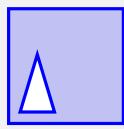
```
void InstVentana()
{ pushMM();
  compMM( MAT_Escalado( 0.3, 0.3, 1.0 ) );
  Ventana();
  popMM();
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 164 de 184.

Problemas: uso de *push* y *pop* (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).



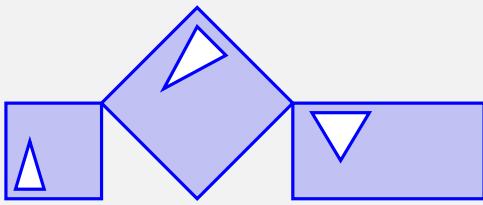
(usa el repositorio [opengl3-minimo](#))

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 165 de 184.

Problemas: uso de *push* y *pop* (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 (**pushMM** y **popMM**), junto con **MAT_Traslacion** y **MAT_Escalado**:

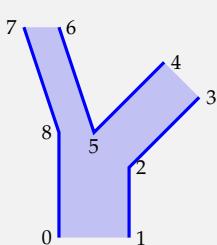


GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 166 de 184.

Problema: tronco de figura recursiva

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).



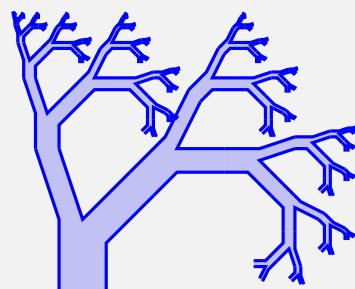
Índice	Coordenadas
0	(+0.0, +0.0)
1	(+1.0, +0.0)
2	(+1.0, +1.0)
3	(+2.0, +2.0)
4	(+1.5, +2.5)
5	(+0.5, +1.5)
6	(+0.0, +3.0)
7	(-0.5, +3.0)
8	(+0.0, +1.5)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 167 de 184.

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)



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 168 de 184.

Métodos para *push* y *pop* en la clase Cauce

La clase **Cauce**, para visualización 3D en las prácticas, incluye, como variables de instancia, dos pilas:

- ▶ una de matrices de modelado (**pila_mat_modelado**)
- ▶ otra de matrices de modelado de normales (**pila_mat_modelado_nor**)

Además, existen los métodos de manipulación de estas pilas:

- ▶ método **pushMM()** para apilar una copia de ambas matrices, cada una en su pila
- ▶ método **popMM()** para restaurar las matrices usando las de los topes, y eliminarlos

Además, hay que tener en cuenta que:

- ▶ el método **fijarMatrizVista(V)** ahora vacía ambas pilas (se llama una vez al inicio del frame).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 169 de 184.

Implementación de la pila en el cauce programable

La implementación de *push* y *pop* en la clase **Cause** sería así:

```
void Cause::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 Cause::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 de los shaders
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_modelado, 1, GL_FALSE, mat_modelado );
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 170 de 184.

Inicialización de las pilas en el cauce programable

Cuando se fija la matriz de vista (al principio de visualizar cada escena), se limpian las dos pilas de la instancia

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

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

    // fijar uniforms de los shaders
    ...
}
```

(en la práctica, si los *push* y *pop* están balanceados como deben, esto no sería necesario).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 171 de 184.

Visualización de grafos

El ejemplo de visualización de un grafos anteriores es un código específico para ese grafo:

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 172 de 184.

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.cause->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 :
            entradas[i].objeto->visualizarGL( cv ); // visualizar objeto
            break ;
            case TipoEntNGE::transformacion :
            cv.cause->compMM( *(entradas[i].matriz)); // componer matriz
            break ;
            case ....
            ....
        }
    // restaura modelview guardada
    cv.cause->popMM();
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 173 de 184.

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.



(continua en la siguiente transparencia)

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 174 de 184.

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**.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 175 de 184.

Modelos parametrizables

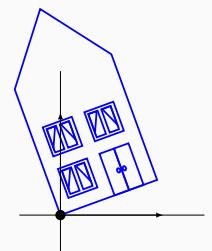
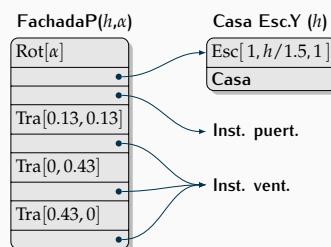
Un grafo de escena puede estar parametrizado respecto de ciertos valores reales:

- ▶ Dichos valores puede 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 (*α*):



GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 177 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 178 de 184.

Visualización directa de FachadaP

Se añaden parámetros a las funciones:

```

void FachadaP( float h, float alpha )
{
    pushMM();
    compMM( MAT_Rotacion( alpha, { 0.0,0.0,1.0 } );
    CasaEscY( h );
    InstPuerta();
    compMM( MAT_Traslacion( { 0.13, 0.13, 0.0 } );
    InstVentana();
    compMM( MAT_Traslacion( { 0.0, 0.43, 0.0 } );
    InstVentana();
    compMM( MAT_Traslacion( { 0.43, 0.0, 0.0 } );
    InstVentana();
    popMM();
}
void CasaEscY( float h )
{
    pushMM();
    compMM( MAT_Escalado( 1.0, h/1.5, 1.0 );
    Casa();
    popMM();
}
  
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 179 de 184.

Representación de grafos parametrizables en memoria

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 );
}
  
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 180 de 184.

Estos dos métodos recalcularán 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, 0.0, 1.0 } );
}
void FachadaP::fijarH( const float h_nuevo )
{
    *pm_escy_h = MAT_Escalado( 1.0, h_nuevo/1.5, 1.0 );
}
```

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_Translacion({0.13,0.13,0.13})); agregar( inst_ventana );
    agregar( MAT_Translacion({0.0,0.43,0.0})); agregar( inst_ventana );
    agregar( MAT_Translacion({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 );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 181 de 184.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 182 de 184.

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).

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 183 de 184.

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 (α y β) 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 (ϕ) 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.

GIM: Informática Gráfica- curso 22-23- creado el 17 de octubre de 2022 – transparencia 184 de 184.

Fin de la presentación.

Informática Gráfica:

Teoría. Tema 3. Visualización.

Carlos Ureña
2022-23

Grado en Informática y Matemáticas
Grado en Informática y Administración y Dirección de Empresas
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Teoría. Tema 3. Visualización.

Índice.

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

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.

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

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.1.

El cauce gráfico del algoritmo Z-buffer..

- 1.1. El cauce gráfico del algoritmo Z-buffer.
- 1.2. Transformación de vista 3D
- 1.3. Transformación de proyección 3D
- 1.4. Recortado y división por W
- 1.5. Transformación de viewport
- 1.6. Representación de cámaras

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.

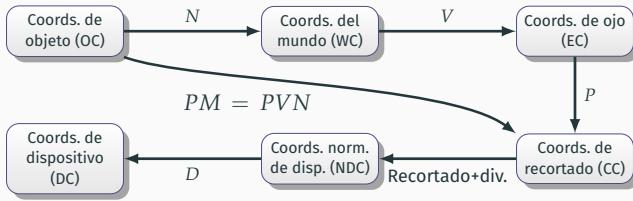
Los pasos del cauce gráfico suelen implementarse en secuencia, cada paso obtiene datos del anterior, los transforma de alguna manera y los entrega al siguiente paso. Los pasos (muy resumidos) son:

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).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 7 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 8 de 208.

Matrices de transformación

Las matrices de transformación (4×4) 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)

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 9 de 208.

Informática Gráfica, curso 2022-23.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.2.

Transformación de vista 3D.

La transformación de vista.

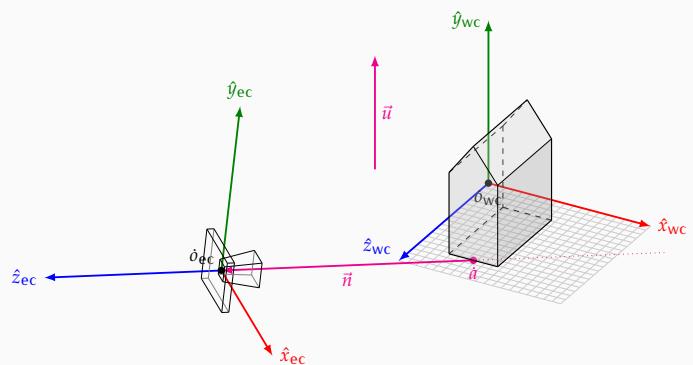
La transformación de vista es el cálculo que permite convertir coordenadas de mundo (world coordinates, WCC) en coordenadas de ojo (o de cámara) (eye or camera coordinates, ECC).

- ▶ Se usa un marco de referencia cartesiano $\mathcal{V} = [\hat{x}_{\text{ec}}, \hat{y}_{\text{ec}}, \hat{z}_{\text{ec}}, \hat{o}_{\text{ec}}]$, 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 $\hat{o}_{\text{wc}} - \hat{o}_{\text{ec}}$ seguida de una matriz (ortonormal) de rotación, que tiene las coordenadas de mundo de \hat{x}_{ec} , \hat{y}_{ec} y \hat{z}_{ec} en sus filas.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 11 de 208.

El marco de coordenadas de vista o de cámara.

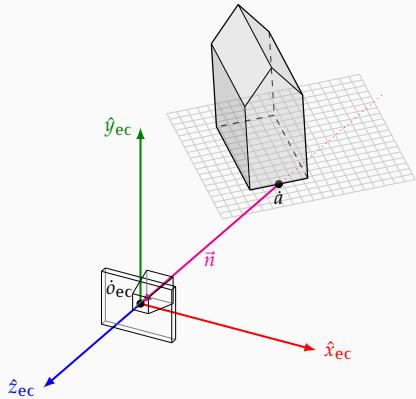
El marco (cartesiano) de coordenadas de vista se construye usando el punto \hat{o} , el punto \hat{o}_{ec} , el vector \hat{u} y el vector \hat{n} . El observador estaría situado en \hat{o}_{ec} mirando en la dirección de $-\hat{z}_{\text{ec}}$.



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 12 de 208.

Escena posterior a transformación de vista.

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



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 13 de 208.

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 $\dot{o}_{ec} = \dot{a} + \vec{n}$).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 14 de 208.

Cálculo del marco de vista.

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

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 15 de 208.

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:

$$\hat{x}_{ec} = \mathcal{W}(a_x, a_y, a_z, 0)^t = \mathcal{W}\mathbf{x}_{ec}$$

$$\hat{y}_{ec} = \mathcal{W}(b_x, b_y, b_z, 0)^t = \mathcal{W}\mathbf{y}_{ec}$$

$$\hat{z}_{ec} = \mathcal{W}(c_x, c_y, c_z, 0)^t = \mathcal{W}\mathbf{z}_{ec}$$

$$\dot{o}_{ec} = \mathcal{W}(o_x, o_y, o_z, 1)^t = \mathcal{W}\mathbf{o}_{ec}$$

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 \mathbf{u}, \mathbf{n} y \mathbf{o} , respectivamente.

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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 16 de 208.

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 $-\mathbf{o}_{ec}$) seguida de una matriz de rotación (de eje arbitrario) R :

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

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

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

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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 17 de 208.

Construcción directa de la matriz de vista 3D

Por tanto, para construir directamente la matriz de vista, el código C++ puede ser del estilo de este:

```
const GLfloat V[4][4] = {
    { 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 }
};
```

► Este matriz queda en memoria dispuesta **por filas** en memoria.

► Para fijar la matriz de vista en OpenGL, debemos de tener en cuenta que hay que indicar que está por filas, no por columnas.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 18 de 208.

Construcción de la matriz de vista con funciones

Se puede construir explícitamente usando las funciones de creación de matrices. Podemos usar **MAT_LookAt**, que tiene como parámetros las tuplas **o_{ec}**, **a** y **u**:

```
Matriz4f MAT_LookAt( const Tupla3f & o, const Tupla3f & a ,  
                      const Tupla3f & u );
```

O bien, si se conocen las tuplas **x_{ec}**, **y_{ec}**, **z_{ec}** y **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 ) ;
```

Las funciones **MAT_...** producen las matrices dispuestas en memoria por columnas.

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 3D.
- ▶ Declarar un parámetro uniform en el vertex shader que contiene la matriz de vista
- ▶ Escribir el código del *vertex shader* de forma que la posición de los vértices sea transformada usando la matriz de modelado y la matriz de vista.

Transformación de modelado y vista en el *vertex shader*

Usando la matriz de vista y la de modelado, se calculan la posición y la normal, en coordenadas de cámara, a partir de las coordenadas de objeto:

```
uniform mat4 u_mat_modelado ; // matriz de modelado  
uniform mat4 u_mat_modelado_nor; // matriz de modelado para normales  
uniform mat4 u_mat_vista ; // matriz de vista (mundo -> camara)  
  
.....  
  
void main()  
{  
    // calcular posición y normal en coords de mundo  
    vec4 posic_wcc = u_mat_modelado * vec4( in_posicion_occ, 1.0 ) ;  
    vec3 normal_wcc = (u_mat_modelado_nor * vec4(in_normal,0)).xyz ;  
  
    // calcular posición y normal en coordenadas de cámara  
    v_posic_ecc = u_mat_vista*posic_wcc ;  
    v_normal_ecc = (u_mat_vista*vec4(normal_wcc,0)).xyz ;  
  
    ....  
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 21 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 22 de 208.

Implementación de **fijarMatrizVista**

```
void Cauce::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();  
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 23 de 208.

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:

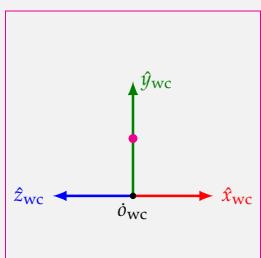
1. El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2. El punto de coordenadas (0,0,5,0) (aparece como un disco morado en la figura) debe aparecer en el centro del viewport
3. El observador (foco de la proyección) estará a 3 unidades de distancia del punto (0,0,5,0)

(continua en la siguiente transparencia).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 24 de 208.

Problema 3.1. (continuación)

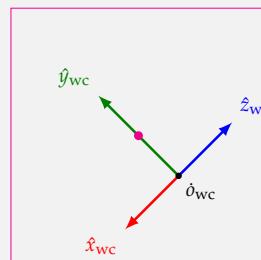
Escribe unos valores que podríamos usar para **a**, **u** y **n** de forma que se cumplan estos requisitos. En la figura se observa una vista esquemática de como quedaría la figura en un viewport cuadrado, no necesariamente a escala.



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 25 de 208.

Problema 3.2.

Repite el problema anterior, pero ahora para esta vista:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 26 de 208.

Problema: construcción de la matriz de vista**Problema 3.3.**

Escribe el código para calcular los vectores de coordenadas **x_{ec}**, **y_{ec}**, **z_{ec}** y **o_{ec}** que definen el marco de vista a partir de los vectores de coordenadas **a**, **u** y **n** (todos estos vectores de coordenadas son de tipo **Tupla3f**).

Problema 3.4.

Partiendo de los vectores de coordenadas **x_{ec}**, **y_{ec}**, **z_{ec}** y **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 **V** y luego asigna valor a **V(i,j)** para cada fila **i** y columna **j**, ambas entre 0 y 3).

Informática Gráfica, curso 2022-23.

Teoría. Tema 3. Visualización.

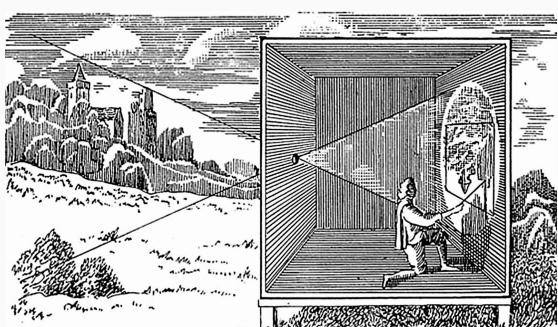
Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.3.**Transformación de proyección 3D.**

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 27 de 208.

Introducción

La transformación de proyección 3D (perspectiva) 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.



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 29 de 208.

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.

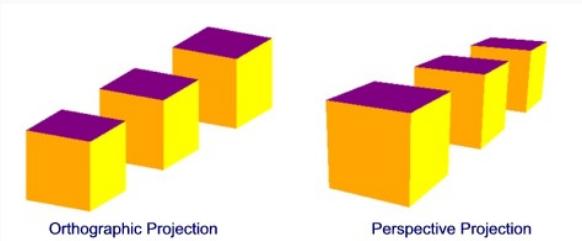
La proyección puede ser de dos tipos

- ▶ **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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 30 de 208.

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):



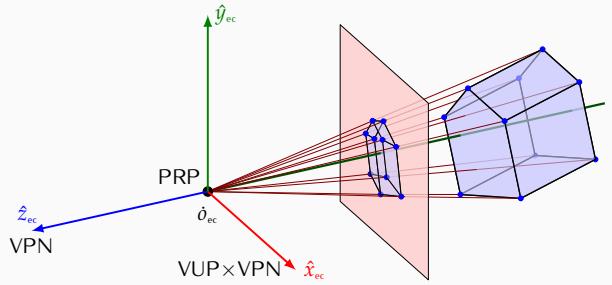
A la izquierda, se interpreta que el cubo más lejano es más grande que los otros, aunque en la imagen son los tres del mismo tamaño

Microsoft WPF documentation: 3D Graphics Overview

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 31 de 208.

Proyección perspectiva

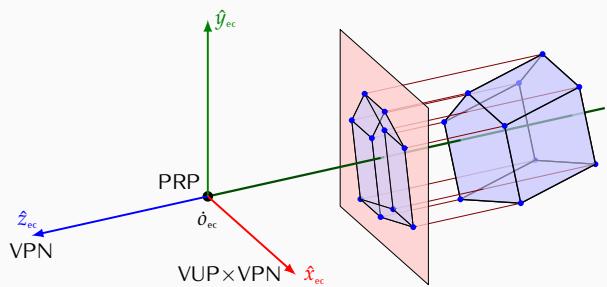
Cambia el tamaño de los objetos, usando un factor de escala s que crece de forma inv. proporcional a la distancia (d_z) en Z desde el objeto al foco (s es de la forma $1/(ad_z + b)$)



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 32 de 208.

Proyección paralela

En este caso, no hay transformación de escala, y la proyección se puede ver como una transformación afín:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 33 de 208.

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.).

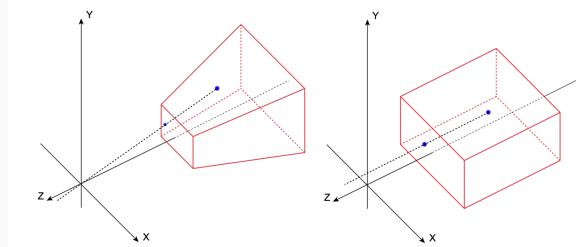


Imagen obtenida de: SGI® OpenGL Multipipe™ SDK User's Guide

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 34 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 35 de 208.

Parámetros del view-frustum. Extensión 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 NDC (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 $0 < n < f$.
- ▶ 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**

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 36 de 208.

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.

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.columns/núm.filas). Si esto no ocurre los objetos aparecerán deformados en la imagen.

En adelante supondremos que siempre seleccionamos $l < r$ y $b < t$.

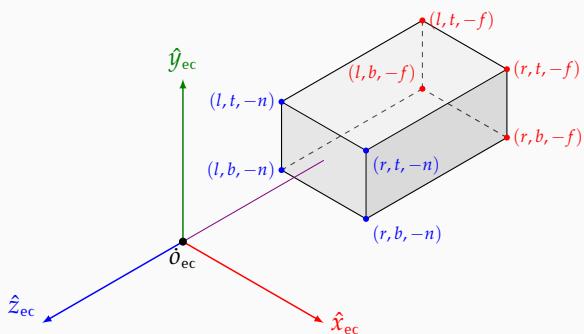
GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 37 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 38 de 208.

Parámetros en la proyección ortográfica.

En pr. ortográfica el view-frustum es un **orthoedro**. Contiene los puntos cuyas coordenadas de cámara (x_{ec}, y_{ec}, z_{ec}) cumplen:

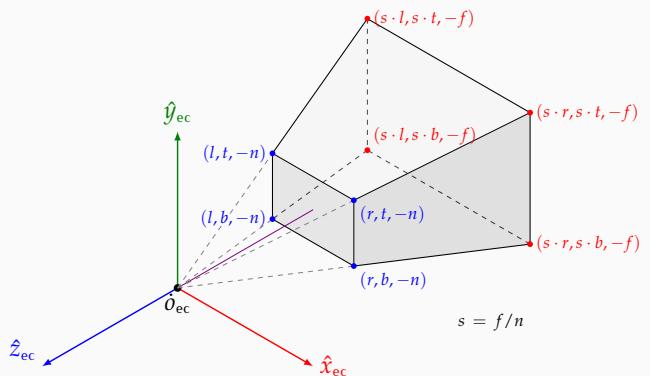
$$l \leq x_{ec} \leq r \quad b \leq y_{ec} \leq t \quad -f \leq z_{ec} \leq -n$$



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 39 de 208.

Parámetros en la proyección perspectiva (1/2)

En perspectiva, el view-frustum es una **pirámide rectangular truncada**:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 40 de 208.

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:

$$b \leq y_{ec} \left(\frac{n}{-z_{ec}} \right) \leq t$$

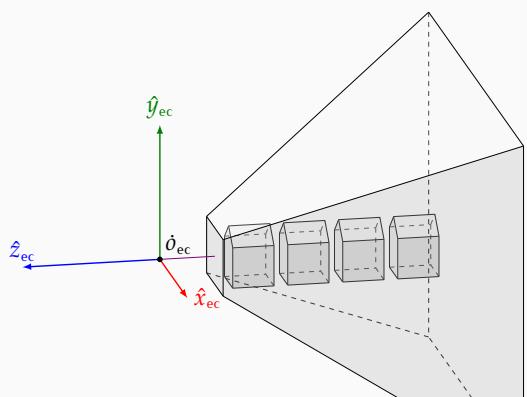
En el eje Z:

$$-f \leq z_{ec} \leq -n$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 41 de 208.

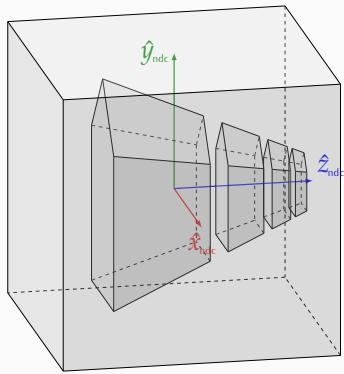
Escena de ejemplo para transformación perspectiva

Suponemos que partimos de una escena que vamos a proyectar usando perspectiva. En el espacio de coordenadas de cámara, la escena es esta:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 42 de 208.

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:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 43 de 208.

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 $-n$).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 44 de 208.

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:

$$a_0 \equiv \frac{2n}{r - l} \quad a_1 \equiv \frac{r + l}{r - l} \quad b_0 \equiv \frac{2n}{t - b} \quad b_1 \equiv \frac{t + b}{t - b}$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 45 de 208.

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 :

$$z'' \equiv \frac{c_0 z_{ec} + c_1}{-z_{ec}} \quad \text{donde: } c_0 \equiv \frac{n + f}{n - f} \quad c_1 \equiv \frac{2fn}{n - f}$$

- ▶ los dos valores c_0 y c_1 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 46 de 208.

Coordenadas cartesianas del punto proyectado

En resumen, tenemos estas tres igualdades:

$$x'' \equiv \frac{a_0 x_{ec} + a_1 z_{ec}}{-z_{ec}}$$

$$y'' \equiv \frac{b_0 y_{ec} + b_1 z_{ec}}{-z_{ec}}$$

$$z'' \equiv \frac{c_0 z_{ec} + c_1}{-z_{ec}} = \frac{c_0 z_{ec} + c_1 w_{ec}}{-z_{ec}}$$

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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 47 de 208.

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:

$$\begin{aligned} x_{cc} &\equiv a_0 x_{ec} + a_1 z_{ec} \\ y_{cc} &\equiv b_0 y_{ec} + b_1 z_{ec} \\ z_{cc} &\equiv c_0 z_{ec} + c_1 w_{ec} \\ w_{cc} &\equiv -z_{ec} \end{aligned}$$

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).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 48 de 208.

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:

$$Q \equiv \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 49 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 50 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 51 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 52 de 208.

Funciones de construcción de la matriz de proyección

Para construir una matriz de proyección a partir de los seis valores reales l, r, b, t, n y f , podemos usar estas dos funciones:

```
// construye matriz O
Matriz4f MAT_Ortografica( const float l, const float r,
                           const float b, const float t,
                           const float n, const float f );

// construye matriz Q
Matriz4f MAT_Frustum ( const float l, const float r,
                        const float b, const float t,
                        const float n, const float f );
```

Ambas funciones generan las matrices por columnas (como las espera OpenGL).

Transf. de proyección con MAT_Perspectiva

La función **MAT_Perspectiva** permite generar una matriz de proyección perspectiva de forma más fácil en ocasiones:

```
MAT_Perspectiva( GLdouble β, GLdouble a, // calcula Q a partir de β,a,n,f
                  GLdouble n, GLdouble f );
```

esta función equivale a **MAT_Frustum** centrado con:

$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

donde:

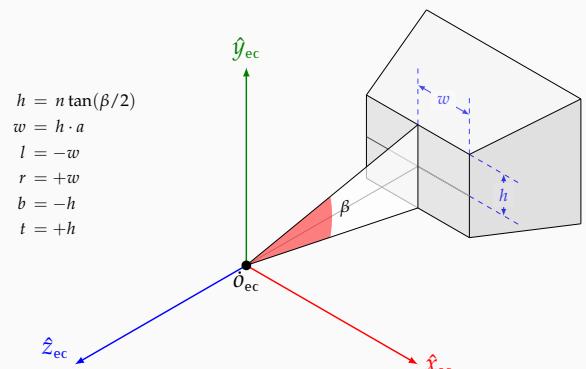
β ≡ es la **apertura vertical del campo de visión (fovy)**, es el ángulo en grados (entre 0 y 180.0) que hay entre la cara superior y la inferior del frustum

a ≡ **relación de aspecto (aspect ratio)** de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 53 de 208.

Parámetros de MAT_Perspectiva

El significado de los parámetros se aprecia en esta figura:



Esta perspectiva es *centrada*, ya que $r = -l$ y $t = -b$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 54 de 208.

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**.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 55 de 208.

Para la gestión de la matriz de proyección, en la clase **Cauce** y sus shaders se incluyen:

- ▶ Una variable de instancia con la matriz de proyección actual (**mat_proyeccion**).
- ▶ Un parámetro uniform con esa matriz (**u_mat_proyeccion**).
- ▶ El método **fijarMatrizProyección**, implementado de esta forma:

```
void Cauce::fijarMatrizProyección( const Matriz4f & nue_mat_pro )
{
    mat_proyeccion = nue_mat_pro; // no es estrictamente necesario
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_proyección, 1, GL_FALSE, mat_proyeccion );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 56 de 208.

Código del *vertex shader*

En el *vertex shader* se usa la matriz uniform **u_mat_proyección** para transformar la posición del vértice en coordenadas de vista (**v_posic_ecc**) hacia coordenadas de recortado (en **gl_Position**):

```
void main()
{
    // calcular posición y normal en coordenadas de mundo (transf. de modelado)
    vec4 posic_wcc = u_mat_modelado * vec4( in_posicion_occ, 1.0 );
    vec3 normal_wcc = (u_mat_modelado_nor * vec4(in_normal,0)).xyz ;

    // calcular posición y normal en coordenadas de vista (transf. de vista)
    v_posic_ecc = u_mat_vista*posic_wcc ;
    v_normal_ecc = (u_mat_vista*vec4(normal_wcc,0)).xyz ;

    ...

    // calcular posición en coords. de recortado (transf. de proyección)
    gl_Position = u_mat_proyección * v_posic_ecc ;
}
```

gl_Position es la variable de salida predefinida.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 57 de 208.

Informática Gráfica, curso 2022-23.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.4.

Recortado y división por *W*.

Recortado

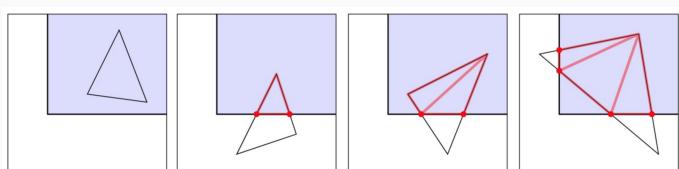
Una vez se tienen las coordenadas de recortado de los vértices, se comprueban que primitivas están dentro o fuera del viewfrustum (que en CC es un cubo de lado dos unidades centrado en el origen):

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 59 de 208.

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/>

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 60 de 208.

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 2022-23.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.5.

Transformación de viewport.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 61 de 208.

Transformación de Viewport

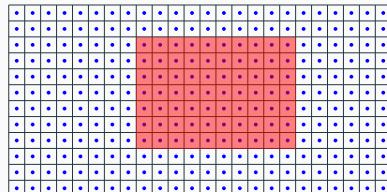
El siguiente paso consiste en calcular en qué posiciones de la imagen se proyecta cada vértice:

- ▶ 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*.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 63 de 208.

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:

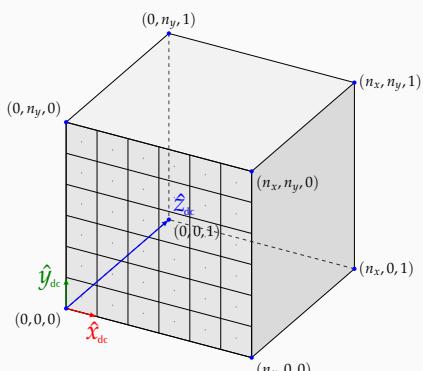


los centros de los pixels (puntos azules) tienen coordenadas de dispositivo con parte fraccionaria igual a 1/2. Los bordes entre pixels tienen coordenadas sin parte fraccionaria (enteras).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 64 de 208.

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:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 65 de 208.

Matriz del viewport y parámetros en OpenGL.

OpenGL tiene en su estado una matriz 4x4 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.

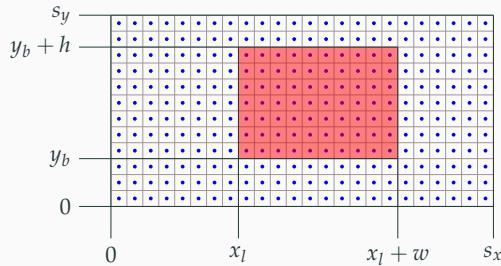
n_d, f_d rango de valores de salida en Z en DC. El valor n_d es la profundidad más cercana posible al observador, y f_d la más lejana. Por defecto $n_d = 0$ y $f_d = 1$.

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).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 66 de 208.

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: $\begin{cases} 0 \leq x_l < x_l + w \leq s_x \\ 0 \leq y_b < y_b + h \leq s_y \end{cases}$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 67 de 208.

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:

$$\begin{aligned} x_{dc} &= (x_{ndc} + 1)w/2 + x_l \\ y_{dc} &= (y_{ndc} + 1)h/2 + y_b \\ z_{dc} &= (z_{ndc} + 1)(f_d - n_d)/2 + n_d \end{aligned}$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 68 de 208.

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:

$$(x_{dc}, y_{dc}, z_{dc}, 1)^t = D(x_{ndc}, y_{ndc}, z_{ndc}, 1)^t$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 69 de 208.

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 x_l, GLint y_b, 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 70 de 208.

Representación de cámaras

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.
Sección 1. Cauce gráfico y definición de la cámara.
Subsección 1.6.
Representación de cámaras.

La clase **Cámara** 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 4x4 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 la 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}).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 72 de 208.

```

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}atio ) ;
    // 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 73 de 208.

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
}

```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 74 de 208.

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 ;
}

```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 75 de 208.

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) rotación 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
}

```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 76 de 208.

Problemas: parámetros de matriz de proyección (1)

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 sitúa el observador en el punto $\mathbf{o}_{\text{oc}} = (c_x, c_y, c_z + s + 2)$, el punto de atención **a** se hace igual a **c** (el centro del cubo se ve en el centro de la imagen), y el vector **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) .

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 77 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 78 de 208.

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 β 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 β .

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 β, s y $\mathbf{c} = (c_x, c_y, c_z)$).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 79 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 80 de 208.

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 82 de 208.

Informática Gráfica, curso 2022-23.
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 λ 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*).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 84 de 208.

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 λ 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)

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 85 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 86 de 208.

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 λ , 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 87 de 208.

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:

$$f(L_r) \approx (1, 0, 0) \quad f(L_g) \approx (0, 1, 0) \quad f(L_b) \approx (0, 0, 1) \quad (1)$$

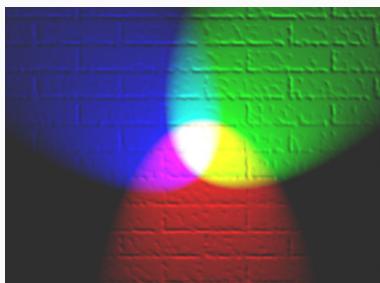
A una terna de distribuciones L_r, L_g y L_b que cumplen lo anterior se le denomina una terna de **primarios RGB**, ya que son percibidos como rojo, verde y azul, respectivamente.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 88 de 208.

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)

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 89 de 208.

Reproducción de ternas RGB

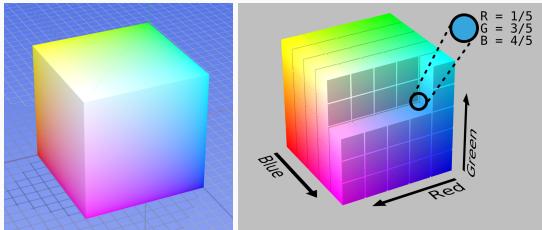
Un dispositivo de salida de color (monitor, impresora, proyector) tiene asociados tres primarios RGB (las distribuciones obtenidas cuando se muestra el rojo, verde y azul a máxima potencia en el dispositivo)

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 90 de 208.

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)

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](#))

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 91 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 92 de 208.

Informática Gráfica, curso 2022-23.
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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 94 de 208.

Modelo de la reflexión local en un punto

La radiancia $L(\lambda, \mathbf{p}, \mathbf{v})$ se puede escribir como suma de:

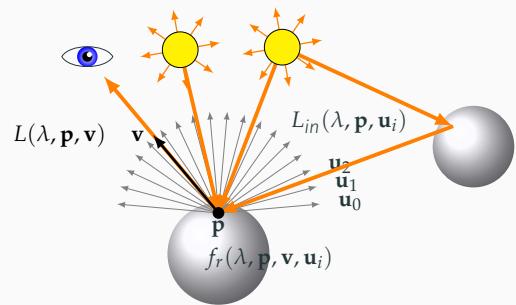
- ▶ la **radiancia emitida** desde \mathbf{p} en la dirección \mathbf{v} (0 si \mathbf{p} no está en una fuente de luz), que llamamos $L_{em}(\lambda, \mathbf{p}, \mathbf{v})$
- ▶ la **radiancia reflejada**, suma, para cada dirección \mathbf{u}_i del producto de:
 $L_{in}(\lambda, \mathbf{p}, \mathbf{u}_i) \equiv$ 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. λ)

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:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 95 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 96 de 208.

La ecuación anterior es complicada (larga) de calcular. Por tanto: en OpenGL básico se hacen varias simplificaciones:

1. No se considera la radiancia emitida.
2. No se considera la luz incidente que no provenga directamente de las fuentes de luz.
3. Las fuentes de luz son puntuales o unidireccionales, no extensas, y hay un número finito de ellas.
4. Los objetos o polígonos son totalmente opacos (no hay transparencias ni mat. translúcidos).
5. No se consideran sombras arrojadas (las fuentes son visibles desde cualquier cara delantera respecto de ellas).
6. El espacio entre los objetos no dispersa la luz (la radiancia se conserva en el espacio entre los objetos).
7. En lugar de considerar todas las longitudes de onda λ posibles, usamos el modelo RGB.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 98 de 208.

Efecto de las simplificaciones.

Aquí se observa una escena con iluminación compleja (izquierda) y simplificada (derecha)

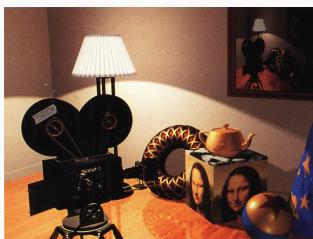


Imagen obtenida de: Computer Graphics: Principles and Practice in C (2nd Edition) Foley, van Dam, Feiner, Hughes.

Modelo simplificado

El modelo que hemos visto antes se simplifica:

- ▶ La iluminación indirecta se reduce a un término ambiente 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 son tuplas (r, g, b) (no acotadas)
- ▶ Los valores de reflectividad (f_r) son tuplas (r, g, b) (entre 0 y 1)

Por tanto:

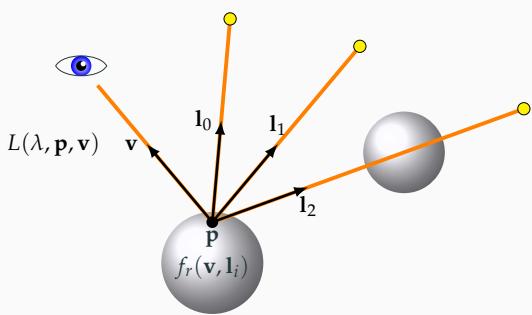
$$L(\mathbf{p}, \mathbf{v}) = \sum_{i=0}^{n-1} L_{in}(\mathbf{p}, \mathbf{l}_i) f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) \quad (2)$$

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 100 de 208.

Modelo simplificado (figura)

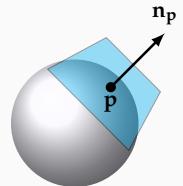
Ahora solo consideramos trayectorias desde las luminarias hacia \mathbf{p} , las luminarias se cuentan aunque la trayectoria esté bloqueada (no hay sombras arrojadas)



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 está 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



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 101 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 102 de 208.

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:

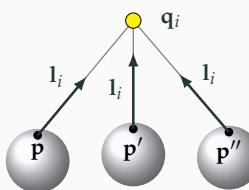
$$\mathbf{l}_i = \frac{\mathbf{q}_i - \mathbf{p}}{\|\mathbf{q}_i - \mathbf{p}\|}$$

- ▶ **Fuentes de luz direccionales:** están en un punto a distancia infinita, por tanto hay un vector \mathbf{l}_j que apunta a la fuente y que es el mismo para cualquier punto \mathbf{p} donde se quiera evaluar el MIL.

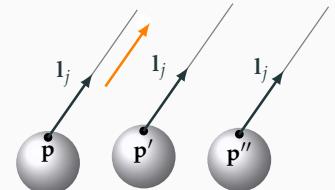
Además de esto, cada fuente de luz emite una radiancia $S_i = (r, g, b)$ (en general no acotada).

Posición o dirección de las luminarias

Fuente posicional (i)



Fuente direccional (j)



- ▶ **Posicional:** la dirección \mathbf{l}_i es distinta para cada punto \mathbf{p} considerado. Es necesario recalcularla cada vez que se evalua el MIL.
- ▶ **Direccional:** La dirección \mathbf{l}_j es igual para todos los puntos \mathbf{p} considerados. Es una constante.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 103 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 104 de 208.

Radiancia incidente y tipos de reflexión. Componentes del MIL.

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:

$$L(\mathbf{p}, \mathbf{v}) = \sum_{i=0}^{n-1} S_i [f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)] \quad (3)$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 105 de 208.

Color del objeto en un punto

En cada punto \mathbf{p} de la superficie de un objeto hay una terna RGB $C(\mathbf{p})$ con valores entre 0 y 1, que es el **color del objeto** en el punto \mathbf{p} .

- ▶ Para cada componente RGB, expresa la fracción de luz reflejada, y por tanto determina el color con el que apreciamos el objeto.
- ▶ Puede ser el mismo (constante) en todos los puntos \mathbf{p} de la superficie de un objeto.
- ▶ Puede variar de un punto a otro dentro del mismo objeto. En rasterización, esto puede ocurrir de dos formas:
 - ▶ Por el uso de **texturas** (las veremos más adelante).
 - ▶ Por el uso de una tabla de colores como atributos de vértice (en cada punto \mathbf{p} , $C(\mathbf{p})$ sería el color RGB interpolado).
- ▶ El color del objeto afecta únicamente a las componentes ambiental y difusa (no a la componente especular).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 106 de 208.

Componente ambiental.

Cada objeto puede reflejar más o menos cantidad de iluminación indirecta proveniente de la i -ésima fuente de luz.

- ▶ Esa iluminación indirecta es complicada de calcular y se ignora en este modelo.
- ▶ Por tanto, los objetos aparecerían negros donde no haya iluminación directa.
- ▶ Para suplir esto, se usa la **componente ambiental** f_{ra} .

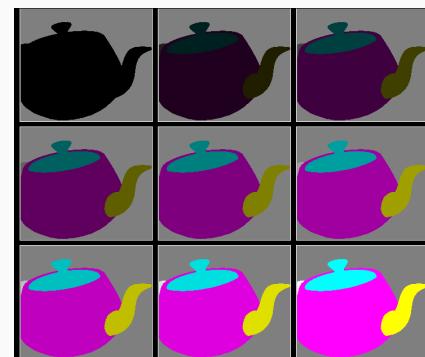
La componente ambiental, por tanto, no depende de \mathbf{v} ni \mathbf{l}_i , y se hace igual a:

$$f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_a(\mathbf{p}) \cdot C(\mathbf{p}) \quad (4)$$

Donde $k_a(\mathbf{p})$ es un valor real entre 0 y 1 que determina la fracción de luz reflejada de esta forma.

Reflectividad ambiental del objeto:

En este ejemplo, el color $C(\mathbf{p})$ depende de la parte de la tetera donde está \mathbf{p} , mientras que k_a es constante en toda la tetera (aunque crece en sucesivas imágenes).



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 107 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 108 de 208.

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 expresión concreta de f_{rd} es esta:

$$f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_d(\mathbf{p}) \cdot C(\mathbf{p}) \cdot \max(0, \mathbf{n}_p \cdot \mathbf{l}_i) \quad (5)$$

Donde $k_d(\mathbf{p})$ es un valor entre 0 y 1 que indica la fracción de luz reflejada de forma difusa.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 109 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 110 de 208.

Orientación de la superficie

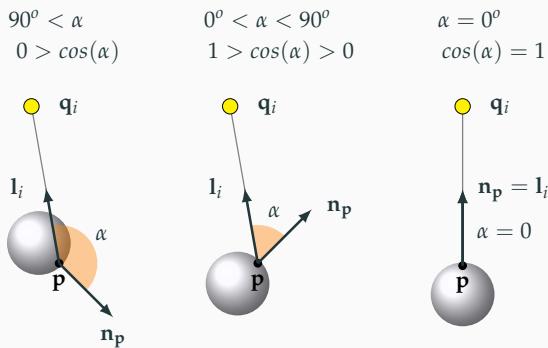
La orientación de la superficie respecto de la fuente de luz viene determinada por el valor α , que es el ángulo que hay entre los vectores \mathbf{n}_p y \mathbf{l}_i (el valor $\mathbf{n}_p \cdot \mathbf{l}_i$ es igual al coseno de α). Se pueden distinguir dos casos:

- 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 111 de 208.

Orientación de la superficie (2)

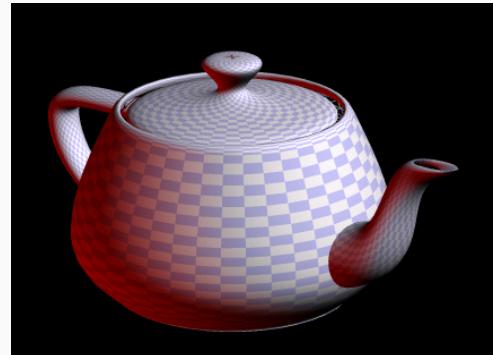
Aquí se ilustran tres posibles casos:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 111 de 208.

Material difuso

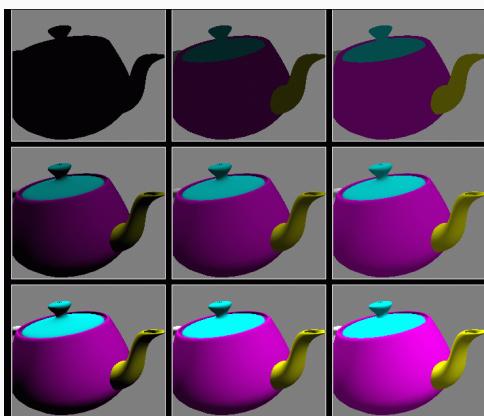
Ejemplo con dos fuentes de luz direccionales, $k_a(\mathbf{p}) = 0$ y $k_d(\mathbf{p}) = 1$ (solo hay componente difusa). Además, $C(\mathbf{p})$ varía de unos polígonos a otros:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 112 de 208.

Material difuso+ambiental

Aquí k_a crece de izquierda a derecha, y k_d de arriba abajo:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 113 de 208.

Componente 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 expresión ideada por *Bui Tuong Phong*, y conocida como **modelo de Phong** es esta:

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_s(\mathbf{p}) d_i [\max(0, \mathbf{r}_i \cdot \mathbf{v})]^e \quad (6)$$

Donde $k_d(\mathbf{p})$ es un valor real entre 0 y 1, representa la fracción de luz reflejada de forma pseudo-especular.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 114 de 208.

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:

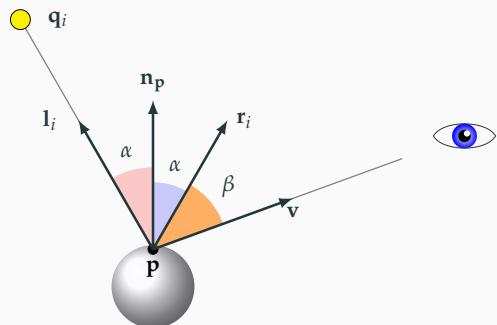
$$\mathbf{r}_i = 2(\mathbf{l}_i \cdot \mathbf{n}_p)\mathbf{n}_p - \mathbf{l}_i$$

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)

El valor $\mathbf{r}_i \cdot \mathbf{v}$ es el coseno del ángulo β 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:

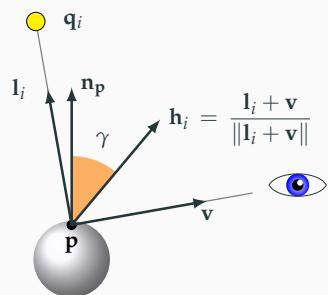


GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 115 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 116 de 208.

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 γ entre \mathbf{h}_i y \mathbf{n}_p (máximo cuando coinciden)



La expresión del **Modelo de Blinn-Phong** es la siguiente:

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_s(\mathbf{p}) d_i [\mathbf{n}_p \cdot \mathbf{h}_i]^e$$

Esta variante es más común que el modelo de Phong anterior.

Ejemplo de material pseudo-especular

Aquí $k_a(\mathbf{p}) = k_d(\mathbf{p}) = 0$, $k_s(\mathbf{p}) = 1$ y $e = 5.0$:

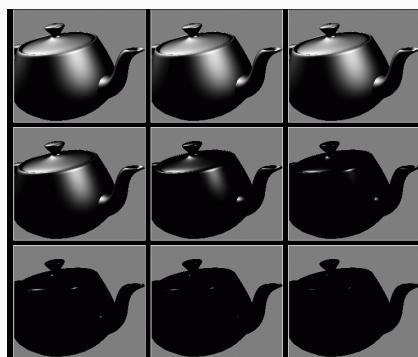


GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 117 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 118 de 208.

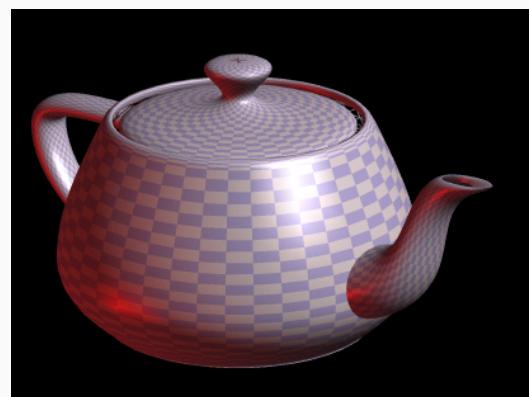
Efecto del exponente de brillo

Aquí el exponente e crece de izquierda a derecha y de arriba abajo:



Ejemplo de material combinado

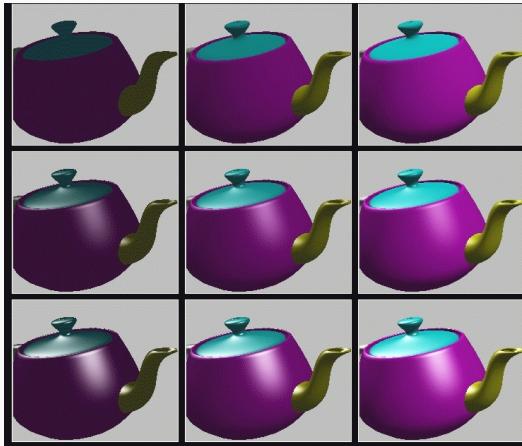
Combinación ambiental, más difusa, más pseudo especular:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 119 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 120 de 208.

Aquí k_d crece de izquierda a derecha y k_s de arriba abajo:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 121 de 208.

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 ($k_d = 1$ en todos los puntos, y k_a y k_s 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 122 de 208.

Informática Gráfica, curso 2022-23.
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 escala, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia:

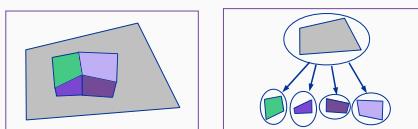


- ▶ Estas variaciones se pueden modelar como funciones que asignan a cada punto de la superficie de un objeto un valor diferente para algunos parámetros del MIL.
- ▶ Lo más usual es variar las reflectividades difusa y ambiente, pero se hace también con la normal (rugosidades), o a veces otros parámetros.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 124 de 208.

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

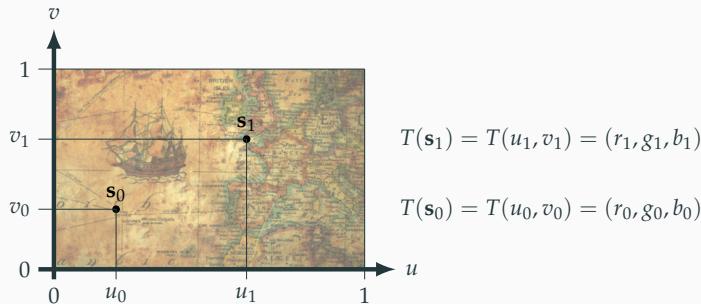
Una opción mejor (mucho más eficiente) es usar imágenes para representar las funciones antes citadas. A estas imágenes se les llama (en el contexto de la Informática Gráfica) **texturas**:

- ▶ 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 el color $C(\mathbf{p})$). 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 también 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**.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 125 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 126 de 208.

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.



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 127 de 208.

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.)



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 128 de 208.

Coordenadas de textura

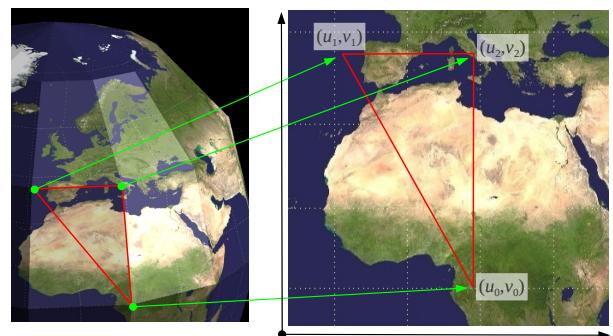
Para poder aplicar una textura a la superficie de un objeto, es necesario hacer corresponder cada punto $\mathbf{p} = (x, y, z)$ de su superficie con un punto (u, v) del dominio de la textura:

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 129 de 208.

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).



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 130 de 208.

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 $(v_0, u_0), (v_1, u_1), \dots (v_{n-1}, u_{n-1})$.
 - ▶ 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(\mathbf{p})` que calcula las coordenadas de textura (para un punto \mathbf{p} devuelve el par $(u, v) = f(\mathbf{p})$ con las coords. de textura de \mathbf{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:

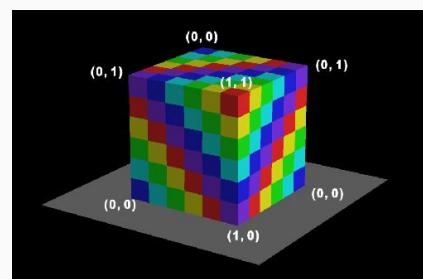


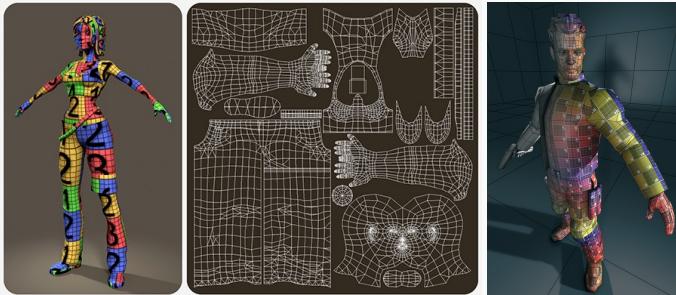
Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 131 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 132 de 208.

Ejemplo de uso de herramientas CAD.

En objetos complejos es necesario el uso de herramientas CAD:



Imagenes de Sean Dixon (izquierda, centro) y Mayan Escalante (derecha).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 133 de 208.

Tipos de asignación procedural

Hay dos opciones:

- ▶ **Asignación procedural a vértices:** se invoca `CoordText(vi)` 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*.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 134 de 208.

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.
- ▶ **Coordinadas paramétricas:** se pueden usar si la malla approxima 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 135 de 208.

Otras opciones para asignación procedural

Otras opciones (no lineales) son estas dos:

- ▶ **Coordinadas polares** (proyección en una esfera): el punto \mathbf{p} se expresa en coordenadas polares como una terna (α, β, r) , los valores u y v se obtienen de α y β .
- ▶ **Coordinadas cilíndricas** (proyección en un cilindro): el punto \mathbf{p} se expresa en coordenadas cilíndricas como una terna (α, y, r) , los valores u y v se obtienen de α e y .

Es muy complicado usarlas con asignación a vértices (α 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*).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 136 de 208.

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)

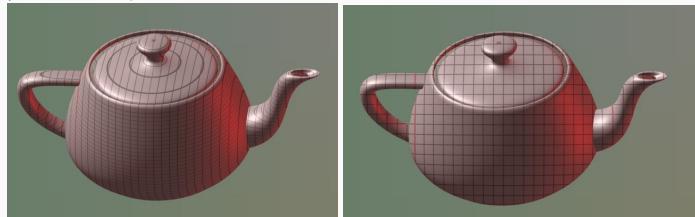
$$u = x = \mathbf{p} \cdot \mathbf{x} = (x, y, z) \cdot (1, 0, 0)$$

$$v = y = \mathbf{p} \cdot \mathbf{y} = (x, y, z) \cdot (0, 1, 0)$$

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 137 de 208.

Ejemplo de proyección paralela a Z.

Las coordenadas de \mathbf{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).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 138 de 208.

Coordenadas paramétricas.

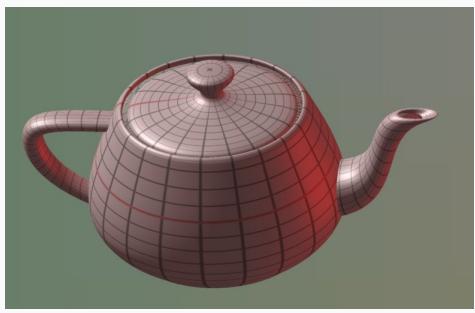
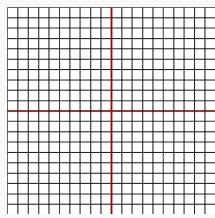
Una **superficie paramétrica** es una variedad plana de dos dimensiones (que puede ser abierta o cerrada), para la cual existe una función \mathbf{g} (con dominio en $[0, 1] \times [0, 1]$) tal que, si \mathbf{p} es un punto de la superficie, entonces existen (s, t) tales que $\mathbf{p} = \mathbf{g}(s, t)$:

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 139 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 140 de 208.

Coordenadas esféricas

Se basa en usar las coordenadas polares (longitud, latitud y radio) del punto \mathbf{p} :

- ▶ Equivale a una proyección radial en una esfera.
- ▶ Las coordenadas (α, β, 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 α en el rango $[-\pi, \pi]$ y β 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 141 de 208.

Ejemplo de coordenadas esféricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

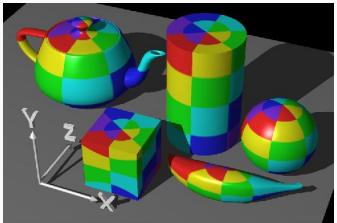
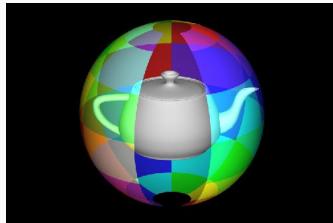


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 142 de 208.

Coordenadas cilíndricas

Se usan las coordenadas polares (ángulo y altura) del punto \mathbf{p} :

- ▶ Equivale a una proyección radial en un cilindro (cuyo eje es usualmente un eje vertical central al objeto).
- ▶ Las coordenadas (α, 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 α 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 143 de 208.

Ejemplo de coordenadas cilíndricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

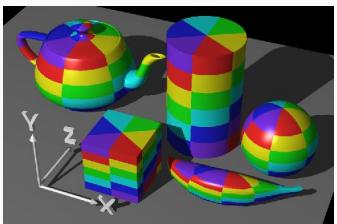


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 144 de 208.

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

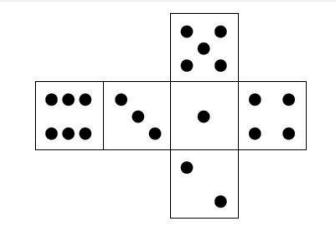
GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 145 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 146 de 208.

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)

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 147 de 208.

Problemas: coordenadas de textura (1/3, cont.)

Problema 3.10. (continuación)

Responde a estas cuestiones:

- Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
- 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 148 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 149 de 208.

Problemas: coordenadas de textura (3/3)

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 150 de 208.

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 Gouraud 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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 152 de 208.

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 p de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- ▶ Se usa la normal al polígono n_p .
- ▶ Se calcula el vector al observador v en p .

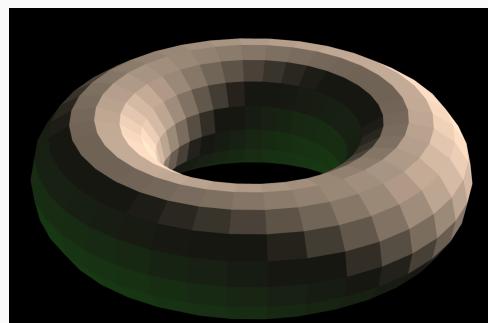
Las desventajas son:

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 153 de 208.

Resultados del sombreado plano.

Aquí vemos un objeto curvo aproximado con caras planas y visualizado con sombreado plano (MIL difuso).



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 154 de 208.

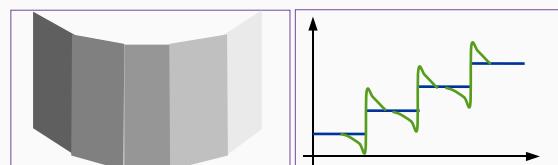
Resultados del sombreado plano

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:



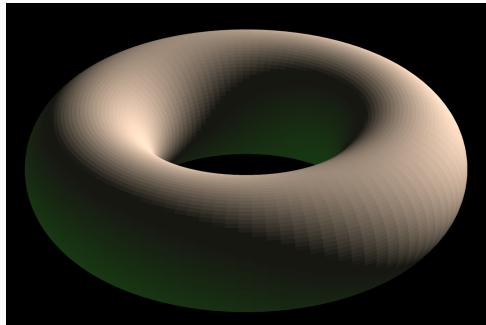
si no se quiere modelar un objeto formado realmente por caras planas, esta forma de visualizar produce resultados pobres. En algunos casos (objetos hechos de caras planas, iluminación puramente difusa) puede ser muy eficiente y realista.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 155 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 156 de 208.

Ejemplo de bandas Mach

En este objeto las bandas Mach son fácilmente apreciables:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 157 de 208.

Sombreado en los vértices

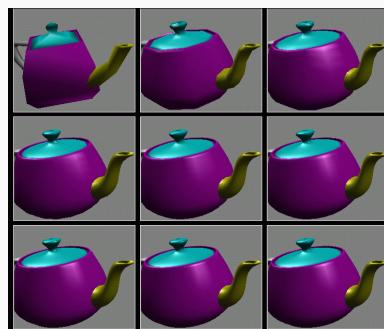
En esta modalidad (*vertex shading*), el MIL se evalua una vez en cada vértice del modelo.

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 158 de 208.

Pérdida de zonas brillantes

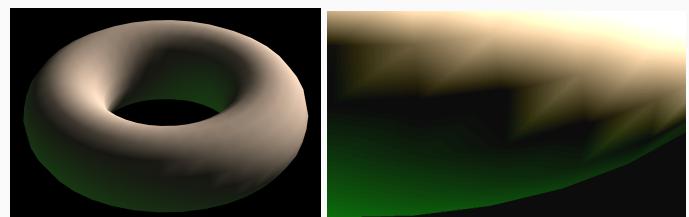
Los resultados mejoran, pero puede haber problemas de pérdida de zonas brillantes (componente pseudo-especular), en modelos con pocos polígonos que aproximan objetos curvos:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 159 de 208.

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)

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 160 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 161 de 208.

Ejemplo de sombreado

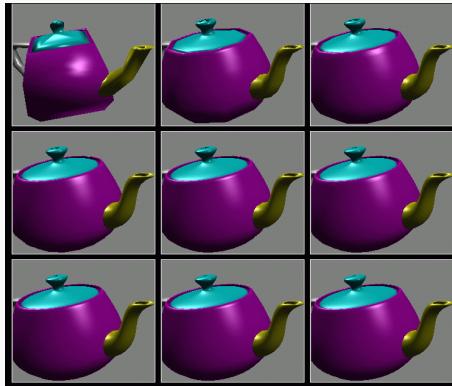
Esta imagen se ha creado con sombreado en los pixels:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 162 de 208.

Reproducción de zonas de brillo

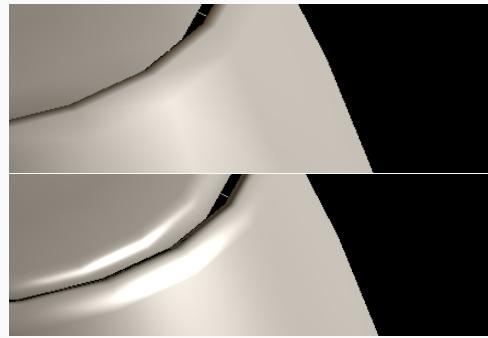
Con este sombreado se reproducen los brillos incluso a baja resolución:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 163 de 208.

Comparación de sombreado vértices y píxeles

Sombreado de vértices (arriba) y de píxeles (abajo), en iguales condiciones de iluminación, observador y atributos material:



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 164 de 208.

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas en el cauce programable.

- 3.1. Carga y configuración de texturas.
- 3.2. Shaders para iluminación y texturas
- 3.3. Implementación de la clase **Cauce**.

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.
Sección 3. Iluminación y texturas en el cauce programable
Subsección 3.1.
Carga y configuración de texturas..

Nombre de textura. Creación.

OpenGL puede gestionar más de una textura a la vez. Para diferenciarlas usa un valor entero (**GLuint**) único para cada una de ellas, que se denomina **nombre** (o **identificador**) de textura (*texture name*):

- ▶ Para crear o generar un nuevo nombre de textura único (distinto de cualquiera ya existente) usamos:

```
GLuint nombre_tex ;  
glGenTextures( 1, &nombre_tex ); // nombre_tex = nuevo nombre
```

- ▶ Para crear *n* nuevos nombres de textura (en un array de **GLuint**), hacemos:

```
GLuint arr_nombres_tex[n] ; // n es una constante entera (n > 0)  
glGenTextures( n, arr_nombres_tex ); // crea n nuevos nombres
```

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

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 167 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 168 de 208.

En el estado interno de OpenGL, por cada unidad de textura, hay en cada momento un nombre o identificador de una única *textura activa*:

- ▶ Cualquier operación de visualización de primitivas usará la textura asociada a dicho nombre.
- ▶ Cualquier operación de configuración de la funcionalidad de texturas se referirá a dicha textura activa.

Para cambiar la textura activa podemos hacer:

```
// activa textura con identificador nombre_tex :
glBindTexture( GL_TEXTURE_2D, nombre_tex );
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 169 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 170 de 208.

Especificación de los texels de la imagen de textura

La copia de los texels hacia la GPU se hace con **glTexImage2D**, y la generación de versiones a resolución reducida con **glGenerateMipMap**:

```
glTexImage2D
( GL_TEXTURE_2D, // GLenum target: tipo de textura.
 0,           // GLint level: nivel de mipmap.
GL_RGB,     // GLint internalformat: formato de destino en GPU.
ancho,      // GLsizei width: número de columnas de texels.
alto,       // GLsizei height: numero de filas de texels.
 0,           // GLint border: borde, no se usa, se pone a 0.
GL_RGB,     // GLenum format: formato origen en memoria apl.
GL_UNSIGNED_BYTE, // GLenum type: tipo valores.
imagen       // const void * data: puntero a texels en memoria apl.
);
// generar mipsmaps (versiones a resolución reducida)
glGenerateMipmap( GL_TEXTURE_2D );
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 171 de 208.

Selección de texels para texturas cercanas (magnificadas)

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, es la que se usa normalmente.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 172 de 208.

Selección de texels para texturas lejanas (minimizadas)

Por otro lado, OpenGL también permite especificar como se seleccionarán los texels en cada consulta cuando el pixel actual es más grande que los múltiples texels que se proyectan en él:

- ▶ En este caso, es acosejable usar los *mipmaps* (versiones de la textura a resoluciones inferiores a la original).
- ▶ Para seleccionar el texel con centro más cercano al centro del pixel:

```
glTexParameterI( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_NEAREST_MIPMAP_LINEAR );
```

- ▶ hacer interpolación bilineal entre los cuatro texels con centros más cercanos al centro del pixel:

```
glTexParameterI( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_LINEAR );
```

La opción inicial 1a, en las prácticas usaremos la 2a.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 173 de 208.

Selección de texels con coord. de textura fuera de rango

Por último, es posible seleccionar que se hace cuando se especifican coordenadas de textura fuera de rango (es decir, fuera de $[0, 1]$).

- ▶ En este caso, se puede elegir entre truncar las coordenadas al valor mínimo o máximo, repetir la textura (descartar la parte entera), o repetirla con espejo.
- ▶ Esto se puede hacer de forma independiente para la coordenada S o para la coordenada T.

Para repetir la textura en ambas coordenadas, hacemos:

```
glTexParameterI( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameterI( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
```

La opción inicial es repetir las texturas.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 174 de 208.

Subsección 3.2.

Shaders para iluminación y texturas.

Iluminación y texturas en el cauce programable

En OpenGL moderno se usa *sombreado de Phong* (sombreado en los pixels), por tanto:

- ▶ 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:
 - ▶ Lectura de parámetros del MIL:
 - ▶ Vector normal y el vector al observador
 - ▶ Posiciones y colores de las fuentes de luz.
 - ▶ Parámetros del material.
 - ▶ Color de la textura (si está habilitada).
 - ▶ Evaluación del MIL.

El MIL se evalua en el espacio de coordenadas de vista (ECC).

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 176 de 208.

Atributos de vértice (entradas vertex shader)

Recordamos los atributos de vértice (distintos por cada vértice) que son parte de las entradas al vertex shader:

```
// posición del vértice en coordenadas de objeto
layout( location = 0 ) in vec3 in_posicion_occ ;

// color del vértice
layout( location = 1 ) in vec3 in_color ;

// normal del vértice (en coordenadas de objeto)
layout( location = 2 ) in vec3 in_normal_occ ;

// coordenadas de textura del vértice
layout( location = 3 ) in vec2 in_coords_textura ;
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 177 de 208.

Parámetros *uniform* para iluminación y texturas

```
// Parámetros relativos al MIL y material actual
uniform bool u_eval_mil; // evaluar el MIL sí (true) o no (false)
uniform float u_mil_ka; // reflect. ambiental del MIL (ka)
uniform float u_mil_kd; // reflect. difusa del MIL (kd)
uniform float u_mil_ks; // reflect. pseudo-especular (ks)
uniform float u_mil_exp; // exponente para pseudo-especular (e)

// Parámetros relativos a texturas
uniform bool u_eval_text; // false -> no evaluar texturas, true -> sí
uniform int u_tipo_gct; // tipo gen.cc.tt. (0=desact., 1=obj., 2=cam.)
uniform vec4 u_coefs_s; // coeficientes para gen.cc.t (coordenada S)
uniform vec4 u_coefs_t; // coeficientes para gen.cc.t (coordenada T)

// Sampler de textura (objeto usado para consultar la textura)
uniform sampler2D u_tx; // ligado a la unidad de texturas 0

// Parámetros de las fuentes de luces actuales
uniform int u_num_luces; // núm. de luces
uniform vec4 u_pos_dir_luz_ec[max_num_luces]; // posic. o direcciones
uniform vec3 u_color_luz[max_num_luces]; // colores (Si)
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 178 de 208.

Variables *varying* para iluminación y texturas

Las variables *varying* relacionadas con iluminación y texturas incluyen: la posición del punto, la normal al punto, el vector hacia el observador (todos ellos en coordenadas de vista) y las coordenadas de textura.

```
out vec4 v_posic_ecc; // posición del punto (en coords de cámara)
out vec4 v_color; // color del vértice (interpolado a los pixels)
out vec3 v_normal_ecc; // normal (en coords. de cámara)
out vec2 v_coord_text; // coordenadas de textura
out vec3 v_vec_obs_ecc; // vector hacia el observador (en coords de cámara)
```

- ▶ Estas variables se calculan en el vertex shader, se interpolan, y se leen en el fragment shader
- ▶ La normal y el vector al observador no están normalizados (ya que son resultado de una interpolación que rompe la normalización)

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 179 de 208.

Código en el vertex shader

La función principal del vertex shader hace la transformación de coordenadas del vértice, pero también calcula las variables *varying*.

```
void main()
{
    // calcular posición y normal en coordenadas de mundo
    vec4 posic_wcc = u_mat_modelado * vec4( in_posicion_occ, 1.0 );
    vec3 normal_wcc = (u_mat_modelado_nor * vec4(in_normal_occ, 0)).xyz;

    // calcular variables varying
    v_posic_ecc = u_mat_vista * posic_wcc;
    v_normal_ecc = (u_mat_vista * vec4(normal_wcc, 0.0)).xyz;
    v_vec_obs_ecc = (-v_posic_ecc).xyz;
    v_color = vec4( in_color, 1 );
    v_coord_text = CoordsTextura();

    // calcular posición del vértice en coords. de recortado
    gl_Position = u_mat_proyeccion * v_posic_ecc;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 180 de 208.

Cálculo de coordenadas de textura

La función **CoordsTextura** se encarga de calcular las coordenadas de textura del vértice. Pueden ser las enviadas por la aplicación o generadas:

```
vec2 CoordsTextura() // calcula las coordenadas de textura
{
    if ( ! u_eval_text )           // si textura desactivada
        return vec2( 0.0, 0.0 );   // devolver cualquier cosa
    if ( u_tipo_gct == 0 )         // si text. activadas, gener. desact.
        return in_coords_textura.st; // devuelve coords. de tabla de cc.t.
    vec4 pos_ver ;                // posición a usar para generación
    if ( u_tipo_gct == 1 )         // si generación en coords. de objeto
        pos_ver = vec4( in_posicion_occ, 1.0 ); // dev. coords. obj.
    else                           // si generación en coords de cámara
        pos_ver = v_posic_ec ;     // dev. coords. de cámara

    return vec2( dot(pos_ver,u_coefs_s), dot(pos_ver,u_coefs_t) );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 181 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 182 de 208.

Código del fragment shader

La función principal del fragment shader calcula **out_color_fragmento**, haciendo consulta de texturas y evaluando el MIL, si procede:

```
void main()
{
    // calcular el color base del objeto (color_obj)
    vec4 color_obj ;
    if ( u_eval_text )           // si hay textura: consultar
        color_obj = texture( u_tx, v_coord_text );
    else                         // si no hay textura:
        color_obj = v_color ; // usar color interpolado

    // calcular el color del pixel (out_color_fragmento)
    if ( ! u_eval_mil ) // si está desactivada iluminación:
        out_color_fragmento = color_obj; // color pixel <- color objeto
    else // si está activada iluminación: evaluar MIL
        out_color_fragmento = vec4( EvalMIL( color_obj.rgb ), 1.0 );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 183 de 208.

Cálculo de vector hacia el observador y normal

```
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
}

// Calcula el vector normal, normalizado
vec3 Normal()
{
    vec3 n = u_usar_normales_tri ? NormalTriangulo()
                                    : normalize( v_normal_ec );
    return gl_FrontFacing ? n : -n ;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 184 de 208.

Vector hacia una fuente de luz

El vector hacia la i -ésima 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 ) ;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 185 de 208.

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 < u_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 = u_mil_kd*col*nl + u_mil_ks*pow(hn,mil_exp);
            s = s + c*u_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
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 186 de 208.

Subsección 3.3. Implementación de la clase Cauce..

Iluminación usando la clase Cauce

La clase **Cauce** incluye estos dos métodos:

El método **fijarEvalMIL** actualiza el uniform **eval_mil**:

```
void Cauce::fijarEvalMIL( const bool nue_eval_mil )
{
    glUseProgram( id_prog );
    glUniform1ui( loc_eval_mil, eval_mil ? 1 : 0 );
}
```

El método **fijarUsarNormalesTri** actualiza el parámetro uniform **u_usar_normales_tri**:

```
void Cauce::fijarUsarNormalesTri( const bool nue_usar_normales_tri )
{
    usar_normales_tri = nue_usar_normales_tri ;
    glUseProgram( id_prog );
    glUniform1ui( loc_usar_normales_tri, usar_normales_tri );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 188 de 208.

Parámetros de las fuentes

El método **fijarFuentesLuz** se invoca una vez por cuadro:

```
void Cauce::fijarFuentesLuz( const std::vector<Tupla3f> & color,
                             const std::vector<Tupla4f> & pos_dir_wc )
{
    const unsigned nl = color.size(); // número de fuentes
    std::vector<Tupla4f> 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 189 de 208.

Parámetros del MIL (material)

El método **fijarParamsMIL** fija los parámetros del material (reflectividades ambiente, difusa y especular, y el exponente)

```
void Cauce::fijarParamsMIL( const float mil_ka, const float mil_kd,
                           const float mil_ks, const float exp )
{
    glUseProgram( id_prog );
    glUniform1f( loc_mil_ka, mil_ka );
    glUniform1f( loc_mil_kd, mil_kd );
    glUniform1f( loc_mil_ks, mil_ks );
    glUniform1f( loc_mil_exp, exp );
}
```

se debe invocar cada vez que se quiera activar un nuevo material

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 190 de 208.

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 Cauce::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 );
        glUniform1ui( loc_eval_text, true );
    }
    else // desactivar
    {   glUniform1ui( loc_eval_text, false );
    }
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 191 de 208.

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 Cauce::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 );
    }
}
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 192 de 208.

Sección 4. Representación de materiales, texturas y fuentes..

- 4.1. Las clases **Textura** y **Material**
- 4.2. Fuentes de luz. La clase **ColecciónFuentes**
- 4.3. 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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 194 de 208.

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 las 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 2022-23.

Teoría. Tema 3. Visualización.

Sección 4. Representación de materiales, texturas y fuentes.

Subsección 4.1.

Las clases **Textura** y **Material**.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 195 de 208.

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( ContextoVis & cv ) ; // activar en cv y cv.cauce_act
    ~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 reflexión pseudo-especular
} ;
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 198 de 208.

Clase FuenteLuz

Informática Gráfica, curso 2022-23.
Teoría. Tema 3. Visualización.
Sección 4. Representación de materiales, texturas y fuentes.

Subsección 4.2. Fuentes de luz. La clase ColecciónFuentes.

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)
    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 ;
};
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 200 de 208.

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/-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
};
```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 201 de 208.

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.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 202 de 208.

Ejemplo de visualización

Si queremos visualizar un objeto (**objeto**) usando un puntero a un material (**material**), podemos guardar y después restaurar el material activo:

```
// (1) registrar material activo actualmente (suponemos que no es nullptr)
Material * material_previo = cv.material_act;

// (2) activar material actual:
material->activar( cv );

// (3) visualizar el objeto
objeto->visualizarGL( cv );

// (4) reactivar material activo al inicio
material_previo->activar( cv );
```

Antes de visualizar una escena, debemos de activar algún material por defecto y guardar el puntero en el contexto de visualización.

Informática Gráfica, curso 2022-23.

Teoría. Tema 3. Visualización.

Sección 4. Representación de materiales, texturas y fuentes.

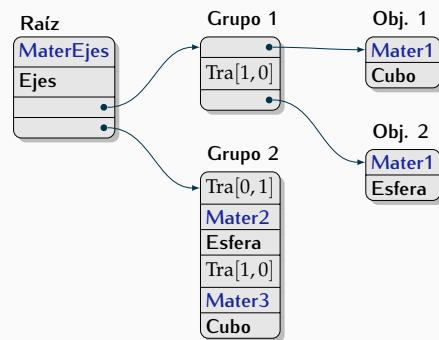
Subsección 4.3.

Materiales en el grafo 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).

Disponemos de las primitivas **Cubo**, **Esfera** y **Ejes**, y cuatro materiales posibles (se muestran en azul):



GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 205 de 208.

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 206 de 208.

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
} ;

```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 207 de 208.

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
        entradas[i].material->activar( cv ); // activar material
    break ;

```

- ▶ Al finalizar, reactivamos el material activo originalmente

```

if ( material_pre != cv.material_act ) // si ha cambiado
if ( material_pre != nullptr ) // y no es nulo:
    material_pre->activar( cv ); // activar el previo

```

GIM: Informática Gráfica- curso 22-23- creado el 14 de noviembre de 2022 – transparencia 208 de 208.

Fin de la presentación.

Informática Gráfica:

Teoría. Tema 4. Interacción. Animación.

Carlos Ureña

2022-23

Grado en Informática y Matemáticas

Grado en Informática y Administración y Dirección de Empresas

Dpt. Lenguajes y Sistemas Informáticos

ETSI Informática y de Telecomunicación

Universidad de Granada

Teoría. Tema 4. Interacción. Animación.

Índice.

1. Introducción
2. Eventos en GLFW
3. Posicionamiento
4. Control de cámaras
5. Selección
6. Animación

Informática Gráfica, curso 2022-23.
Teoría. Tema 4. Interacción. 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

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 4 de 81

Interactividad

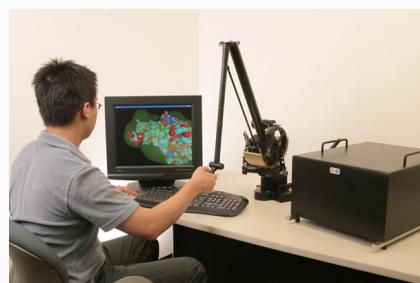
La incorporación de interactividad permite realizar aplicaciones que respondan ágilmente a las acciones de los usuarios y les ofrezcan retroalimentación sobre el efecto de dichas acciones.

La mayor parte de los sistemas gráficos son interactivos. Es esencial en:

- ▶ Videojuegos (*Videogames*) y Juego Serios (*Serious Games*).
- ▶ Sistemas de Diseño Asistido por Ordenador (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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 5 de 81

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 6 de 81

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 video.

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 mas 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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 13 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 14 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 15 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 16 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 19 de 81.

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*).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 20 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 21 de 81.

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_MovimientoRaton )
```

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 22 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 23 de 81.

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 dirección = 0.0 < yoffset ? +1 : -1 ;  
    ScrollVertical( dirección ); // función que procesa scroll (+1 o -1)  
}
```

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 24 de 81.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 25 de 81.

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 todos, en otro caso no hace nada.
- ▶ **ActualizarEscena**: se invoca continuamente, actualiza el modelo al siguiente estado de la animación.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 26 de 81.

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 están desactivadas:
  glfwWaitEvents(); // esperar que se produzcan eventos y procesarlos
terminar = terminar || glfwWindowShouldClose( ventana_glfw );
}
```

- ▶ **animacion**: tiene el estado de las animaciones.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 27 de 81.

Informática Gráfica, curso 2022-23.
Teoría. Tema 4. Interacción. Animación.

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:

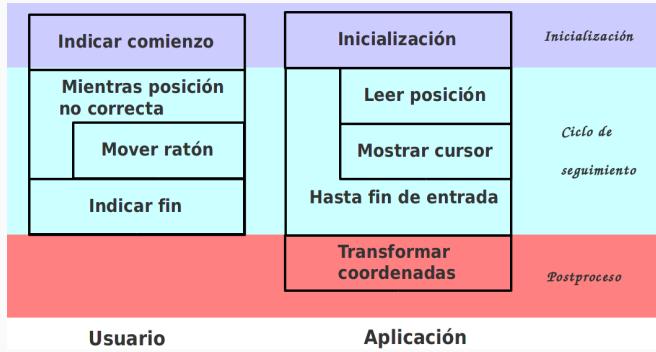
- ▶ **Inicialización**: inicialización de estado.
- ▶ **Ciclo de seguimiento**: hasta que el usuario no indica que está satisfecho, se ejecuta un bucle en el cual: (a) se procesan los eventos de entrada y (b) se actualiza la información visual de realimentación y la posición seleccionada.
- ▶ **Postproceso**: se transforma la posición final.



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 29 de 81.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 30 de 81.

Correspondencia entre operaciones del usuario y del sistema



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 31 de 81.

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 $M = O^{-1}D^{-1}$).

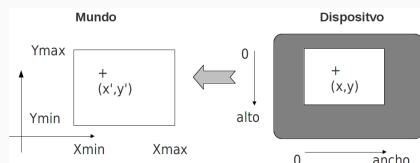
En visualización de modelos 3D en pantalla, hay varias estrategias

- ▶ **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 cursos 2D ligados:** se usan tres vistas ortográficas perpendiculares.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 32 de 81.

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):



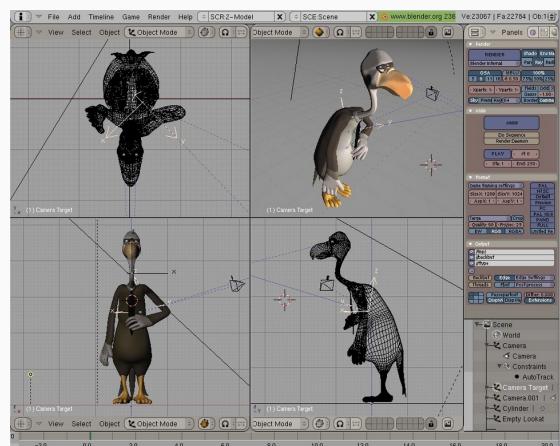
$$x_w = l + (r - l) \left(\frac{x_d + 1/2}{n_x} \right) \quad y_w = t - (t - b) \left(\frac{y_d + 1/2}{n_y} \right)$$

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 33 de 81.

Posicionamiento en 3D: tres cursos 2D ligados

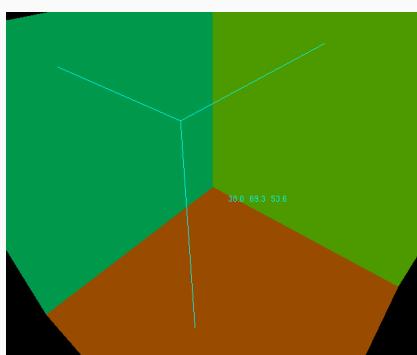
Se usan tres proyecciones ortográficas perpendiculares:



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 34 de 81.

Posicionamiento 3D: Cursor virtual 3D

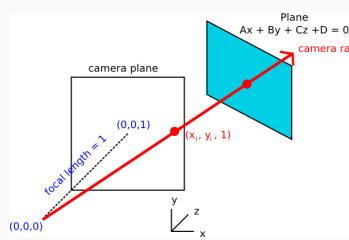
Se usa un cursor virtual en 3D. El desplazamiento se realiza en el plano X-Z o en el X-Y en función de los botones del ratón que estén pulsados.



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 35 de 81.

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



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 36 de 81.

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

- 4.1. Modelo y operaciones de cámaras
- 4.2. Modos de cámara. La cámara de 3 modos.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 37 de 81.

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:

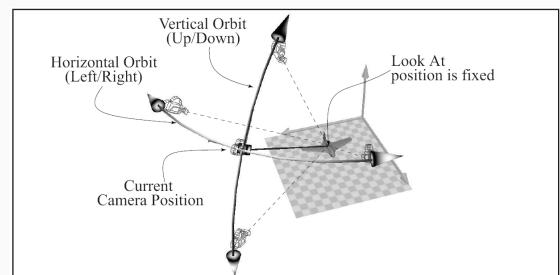


Figura obtenida de: K.Sung, P.Shirley, S.Baer **Essentials of Interactive Computer Graphics: Concepts and Implementation.**

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 39 de 81.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 40 de 81.

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 $[\vec{x}_{\text{ec}}, \vec{y}_{\text{ec}}, \vec{z}_{\text{ec}}, \dot{\theta}_{\text{ec}}]$, y dichas componentes están representadas en memoria por sus coordenadas homogéneas $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}, \mathbf{z}_{\text{ec}}$ y \mathbf{o}_{ec} en el marco de coordenadas del mundo \mathcal{W} :

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

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:

$$V = \begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 42 de 81.

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 :

$$\begin{aligned}\vec{z}_{ec} &= \vec{n} / \|\vec{n}\| & \vec{y}_{ec} &= \vec{z}_{ec} \times \vec{x}_{ec} \\ \vec{x}_{ec} &= (\vec{y}_w \times \vec{z}_{ec}) / \|\vec{y}_w \times \vec{z}_{ec}\| & \vec{o}_{ec} &= \vec{a}_t + \vec{n}\end{aligned}$$

(\vec{z}_{ec} es paralelo a \vec{n} , y el vector VUP siempre es \vec{y}_w).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 43 de 81.

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:

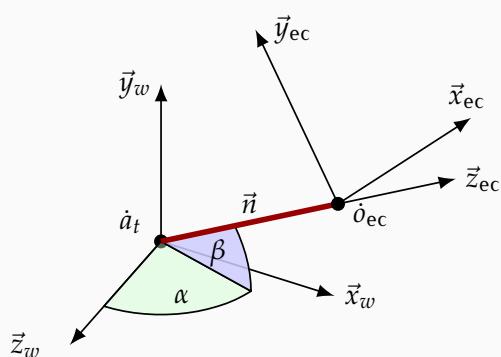
$$\begin{aligned}n_x &= r(\sin \alpha)(\cos \beta) & \alpha &= \text{atan2}(n_x, n_z) \\ n_y &= r(\sin \beta) & \beta &= \text{atan2}(n_y, r_h) \\ n_z &= r(\cos \alpha)(\cos \beta) & r &= \|\mathbf{n}\|\end{aligned}$$

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 $(-\pi, \pi]$.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 44 de 81.

Elementos del modelo de cámara

En esta figura se observan el marco de la cámara \mathcal{V} con origen en $\vec{o}_{ec} = \vec{a}_t + \vec{n}$, junto con el marco del mundo \mathcal{W} (trasladado a \vec{a}_t):



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 45 de 81.

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**.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 46 de 81.

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() ;
};
```

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 47 de 81.

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 desplRotaryX( 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 cámara 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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 48 de 81.

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,d_ini } ; // vector origen (esféricas)
    Tupla3f org_cartesianas = { 0.0,0.0,d_ini } ; // vector origen (cartesianas)
    Tupla3f eje[3] = { {1,0,0},{0,1,0},{0,0,1} } ; // ejes del MCV
};
```

Aquí **d_ini** es el radio inicial (en las prácticas es 3 unidades).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 50 de 81.

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 **o_{ec}** y el punto de atención **a_t** de forma solidaria:

► La operación **desplRotarXY(Δ_a, Δ_b)** supone:

1. **a_t** = **a_t** + Δ_x**x_{ec}** + Δ_y**y_{ec}**
2. **o_{ec}** = **a_t** + **n**

► La operación **moverZ(Δ_z)** supone:

1. **a_t** = **a_t** + Δ_z**z_{ec}**
2. **o_{ec}** = **a_t** + **n**

Las tuplas **s**, **n**, **x_{ec}**, **y_{ec}**, **z_{ec}** no cambian.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 51 de 81.

Cámara primera persona con rotaciones

En este caso se usan rotaciones entorno al origen de la cámara **o_{ec}**. Dichas rotaciones se implementan modificando los ángulos α y β que hay en **s**. El movimiento en Z es similar al anterior

► La operación **desplRotarXY(Δ_a, Δ_b)** supone:

1. **s** = **s** + (Δ_a, Δ_b, 0) (incrementa o decrementa α y β)
2. **n'** = **Cartesianas(s)** (es el nuevo valor de **n**).
3. **a_t** = **a_t** - (**n'** - **n**) (trasl. pto de atención a nueva pos.)
4. **n** = **n'**
5. actualizar **x_{ec}**, **y_{ec}** y **z_{ec}** (el origen **o_{ec}** no cambia)

► La operación **moverZ(Δ_z)** supone:

1. **a_t** = **a_t** + Δ_z**z_{ec}**
2. **o_{ec}** = **a_t** + **n** (los vectores **x_{ec}**, **y_{ec}** y **z_{ec}** no cambian)

Este modo es típico en videojuegos FPS (*First Person Shooter*), normalmente sin movimientos de rotación en vertical (Δ_b = 0)

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 52 de 81.

Cámara en modo orbital o examinar

En este caso se usan rotaciones entorno al punto de atención **a_t**. El movimiento en Z nos acerca o aleja a dicho punto (cambia el valor de **r** que hay en **s**):

► La operación **desplRotarXY(Δ_a, Δ_b)** supone:

1. **s** = **s** + (Δ_a, Δ_b, 0) (incrementa o decrementa α y β)
2. **n** = **Cartesianas(s)** (es el nuevo valor de **n**).
3. **o_{ec}** = **a_t** + **n**, actualizar **x_{ec}**, **y_{ec}**, **z_{ec}**

► La operación **moverZ(Δ_z)** supone:

1. $r = r_{min} + (r - r_{min})(1 + \epsilon)^{\Delta_z}$
2. **n** = **Cartesianas(s)**
3. **o_{ec}** = **a_t** + **n** (los vectores **x_{ec}**, **y_{ec}** y **z_{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 (Δ_z nunca debe estar en $(-1, 1)$).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 53 de 81.

Apuntar la cámara hacia un punto

Esta operación supone hacer que el punto de atención se fije a unas coordenadas de mundo **c** dadas, sin modificar el origen de cámara.

La operación **mirarHacia(c)** supone:

1. **n** = **n** + **a_t** - **c**
2. **s** = **Esfericas(n)**
3. **a_t** = **c**
4. actualizar **x_{ec}**, **y_{ec}** y **z_{ec}** (el origen **o_{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).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 54 de 81.

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 $(0, 1, 0)$.

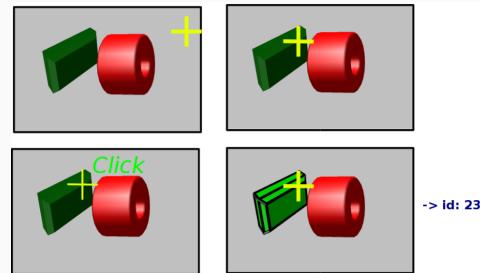
Escribe el pseudo-código de la citada función.

- 5.1. Introducción. Métodos de selección
- 5.2. Selección con *frame-buffer object* invisible.

Selección de objetos

La selección permite al usuario identificar componentes u objetos de la escena:

- ▶ Se necesita para identificar el objeto sobre el que actúan las operaciones de edición.
- ▶ Suele realizarse como posicionamiento seguido de búsqueda.



Identificadores de objetos

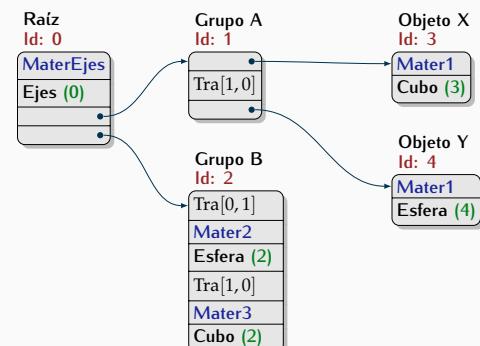
Para poder realizar la selección los componentes de la escena deben tener asociados identificadores numéricos (enteros), a distintos niveles:

- ▶ **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

Todos los objetos (instancias de **Objeto3D**) tienen asociado un valor entero: -1 significa que heredan el identificador del padre, 0 que no es seleccionable, > 0 que es seleccionable con ese identificador.



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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 61 de 81.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 62 de 81.

Modo de selección de OpenGL.

Se usa una funcionalidad de OpenGL, específica para este fin, requiere:

- ▶ 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.

En nuestro caso, no usaremos esta funcionalidad, al no estar disponible en OpenGL 3.3.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 63 de 81.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 64 de 81.

Selección con un *framebuffer* invisible

Se usa algún **frame buffer object** (array de colores de pixels en la memoria de la GPU), hay dos opciones:

- ▶ Usar uno de los dos framebuffers que se suelen existir en OpenGL para la técnica de *double buffering*.
- ▶ Usar un *framebuffer object* específicamente creado para esto. No depende de la existencia de *double buffering* y del acceso a sus buffers.

Nosotros usamos la segunda opción. Se debe:

- ▶ Crear y activar un *Frame Buffer Objects* (FBOs) del tamaño y características que queramos.
- ▶ Visualizar la escena sobre el FBO usando identificadores en lugar de colores.
- ▶ Leer identificador de objeto en el pixel.

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 \mathbf{o} , y como vector de dirección la tupla \mathbf{d} (la suponemos normalizada).
- ▶ Las coordenadas del mundo de los vértices del triángulo son $\mathbf{v}_0, \mathbf{v}_1$ y \mathbf{v}_2 .
- ▶ 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)

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 65 de 81.

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 $c \equiv 1 - a - b$ se les llama *coordenadas baricéntricas* de \mathbf{p}_t en el triángulo, se usan en ray-tracing).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 66 de 81.

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}_{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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 67 de 81.

Subsección 5.2.

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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 69 de 81.

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**.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 70 de 81.

Codificación y cambio de color

Debemos cambiar el color del cauce usando un identificador:

- ▶ Los identificadores son enteros sin signo (tipo **unsigned** de C/C++, de 4 bytes), con valores entre 0 y $2^{24} - 1$ (el byte más significativo es 0, se usan los tres menos significativos).
- ▶ Cada byte del identificador (entre 0 y 255) se convierte a un **float** (en [0..1]) que se usa para cambiar el color actual del cauce.

```
void FijarColorIdent( Cauce & cauce, 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

    cauce.fijarColor( float(byteR)/255.0f, float(byteG)/255.0f,
                      float(byteB)/255.0f );
}
```

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 71 de 81.

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 framebuffer
    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] ) ;
}
```

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 72 de 81.

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)

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 73 de 81.

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.

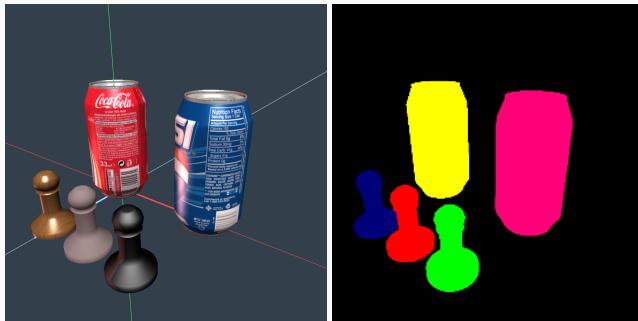
```
class Framebuffer
{
public:
    // crear un FBO de este tamaño
    Framebuffer( const int pancho, const int palto );
    // activar, recreando el FBO si hay cambio de tamaño
    void activar( const int pancho, const int palto );
    void desactivar(); // desactivar (activa el 0)
    ~Framebuffer(); // 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
};
```

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 74 de 81.

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.



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 75 de 81.

Informática Gráfica, curso 2022-23.
Teoría. Tema 4. Interacción. Animación.

Sección 6.
Animación.

Animación

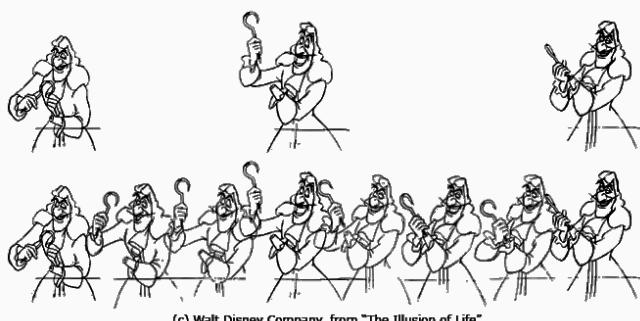
Generar animaciones implica:

- ▶ Regenerar la imagen periódicamente (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"

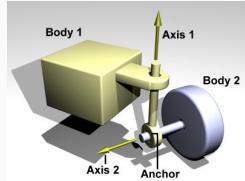
GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 77 de 81.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 78 de 81.

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:

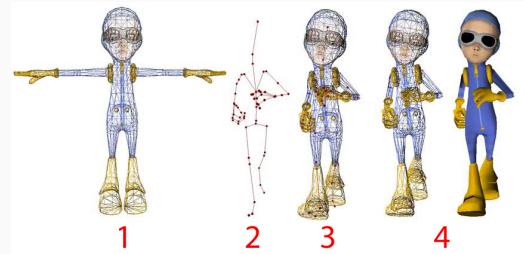
- ▶ ODE: Open Dynamic Engine.
- ▶ Newton: Newton Physics Engine
- ▶ Bullet: Physics Library



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 79 de 81.

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.



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 80 de 81.

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.



GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 81 de 81.

Carlos Ureña

2022-23

Grado en Informática y Matemáticas

Grado en Informática y Administración y Dirección de Empresas

Dpt. Lenguajes y Sistemas Informáticos

ETSI Informática y de Telecomunicación

Universidad de Granada

1. Técnicas realistas en rasterización
2. Ray tracing

Informática Gráfica, curso 2022-23.
Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.

Sección 1.
Técnicas realistas en rasterización.

Informática Gráfica, curso 2022-23.
Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.
Sección 1. Técnicas realistas en rasterización

Subsección 1.1.
Mipmaps..

- 1.1. Mipmaps.
- 1.2. Perturbación de la normal
- 1.3. Sombras arrojadas
- 1.4. Superficies transparentes. Refracción.
- 1.5. Superficies especulares

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 un serie de $n + 1$ texturas (bitmaps) obtenida a partir de una imagen o textura de $2^n \times 2^n$ texels.

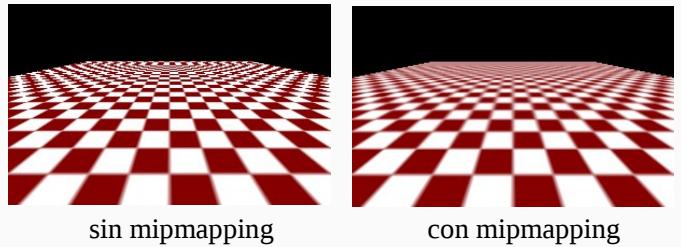
- ▶ La primera imagen (M_0) coincide con la original
- ▶ La i -ésima imagen (M_i) tiene como resolución $2^{n-i} \times 2^{n-i}$ texels.
- ▶ Cada texel de la imagen $i + 1$ (M_{i+1}) se obtiene a partir de cuatro texels de la imagen número i , promediándolos:

$$M_{i+1}[j, k] = \frac{1}{4} (M_i[2j, 2k] + M_i[2j + 1, 2k] + M_i[2j, 2k + 1] + M_i[2j + 1, 2k + 1])$$

Durante el sombreado, en cada punto \mathbf{p} a sombrear es necesario saber qué versión de la textura 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 $\log(d)$

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.



http://www.flipcode.com/archives/Advanced_OpenGL_Texture_Mapping.shtml

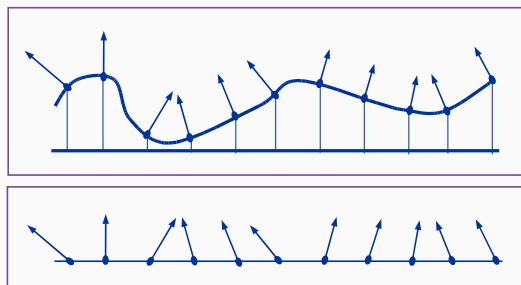
Rugosidades a pequeña escala

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 texture 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 esté codificada como una texture 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).

El procedimiento de perturbación de la normal usa como parámetros las derivadas parciales de f_h (d_u y d_v):

$$d_u = \frac{\partial f_h(u, v)}{\partial u} \quad d_v = \frac{\partial f_h(u, v)}{\partial v}$$

- ▶ 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

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:

- ▶ Δ 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:

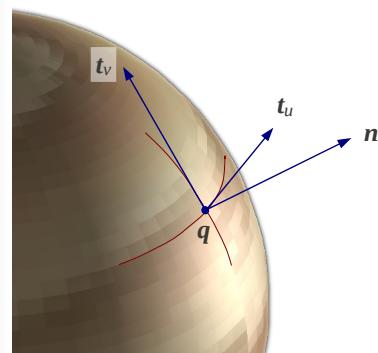
$$q = f_p(u, v)$$

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

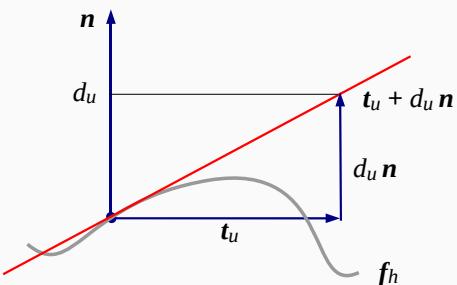
Estos vectores son tangentes a la superficie del objeto, ya que la normal original n es colineal con $t_u \times t_v$. Existen varias alternativas para obtenerlos:

- ▶ 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:

$$t'_u = t_u + d_u n \quad t'_v = t_v + d_v n$$

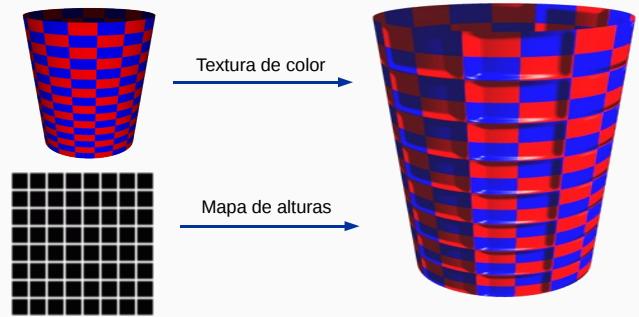


la normal modificada \mathbf{n}' es perpendicular a estos dos vectores, por tanto se calcula usando su producto vectorial (y normalizando)

$$\mathbf{n}' = \frac{\mathbf{n}''}{\|\mathbf{n}''\|} \quad \text{donde: } \mathbf{n}'' = \mathbf{t}_u' \times \mathbf{t}_v'$$

Ejemplo de texturas + perturbación de la normal (1)

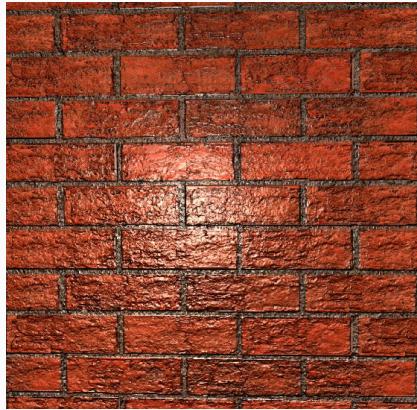
Imágenes de Fredo Durand y Barb Curtler:
<http://groups.csail.mit.edu/graphics/classes/6.837>



GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 19 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 20 de 94.

Ejemplo de texturas + perturbación de la normal (2)



Informática Gráfica, curso 2022-23.
 Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.
 Sección 1. Técnicas realistas en rasterización

Subsección 1.3.
 Sombras arrojadas.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 21 de 94.

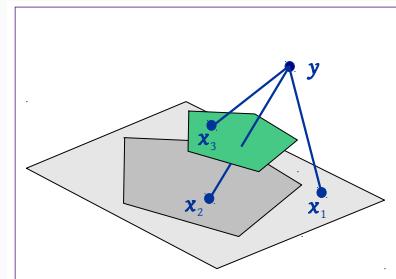
Sombras arrojadas y el MIL

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 :



$$V(\mathbf{x}_1, \mathbf{y}) = 1 \quad V(\mathbf{x}_2, \mathbf{y}) = 0 \quad V(\mathbf{x}_3, \mathbf{y}) = 1$$

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 23 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 24 de 94.

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

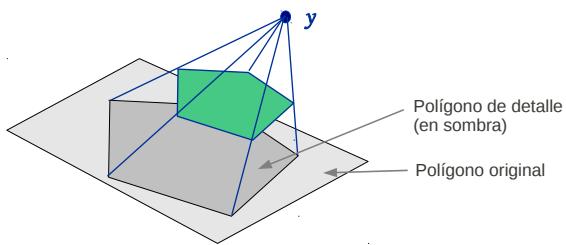
Supondremos escenas formadas por poliedros opacos delimitados por caras planas o polígonos planos individuales.

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 25 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 26 de 94.

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

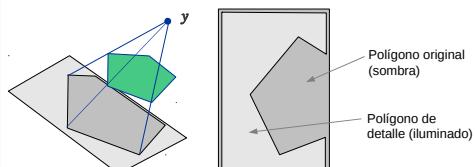
Otros algoritmos de sombras arrojadas (más eficientes) están basados en algoritmos de eliminación de partes ocultas ya existentes. Un ejemplo es el algoritmo de Weiler-Atherton-Greenberg (1978) para sombras arrojadas:

- ▶ 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.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 27 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 28 de 94.

Algoritmo de Weiler-Atherton-Greenberg (2)

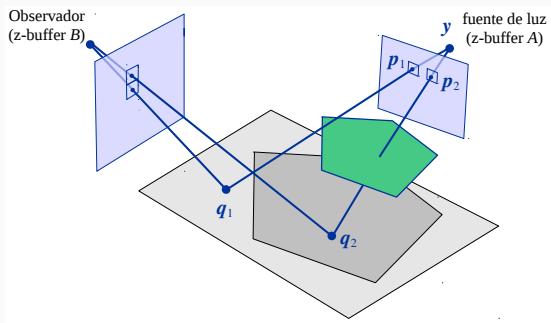


En general, los algoritmos con precisión de objetos para sombras son:

- ▶ 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:

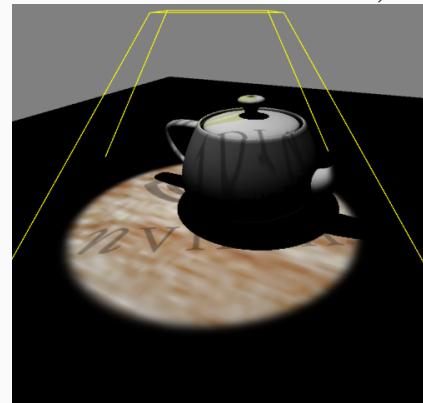


GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 29 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 30 de 94.

- ▶ 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 q_i desde el observador en un pixel, se debe calcular el color con el que se ve q_i , y por tanto es necesario comprobar si es visible desde la fuente, para ello:
 - ▶ se calcula p_i (la proyección de q_i en el plano de visión asociado a la fuente de luz)
 - ▶ se accede al pixel del Z-buffer A correspondiente a p_i , que contiene una distancia d
 - ▶ si $d < \|q_i - p_i\|$, entonces q_i no está iluminado (este es el caso de la figura), en otro caso q_i sí está iluminado.

(se proyecta una textura desde la fuente en los objetos):

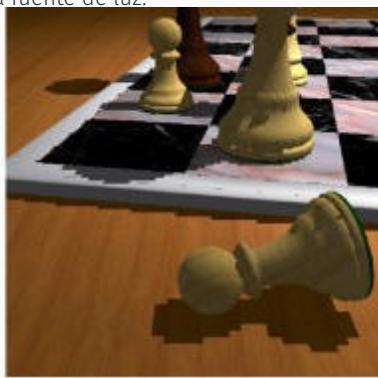


GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 31 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 32 de 94.

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:



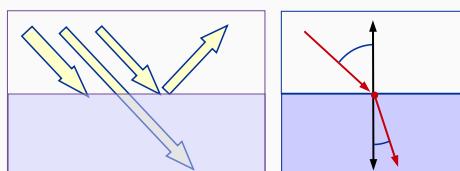
GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 33 de 94.

Informática Gráfica, curso 2022-23.
Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.
Sección 1. Técnicas realistas en rasterización

Subsección 1.4.
Superficies transparentes. Refracción..

Materiales transparentes

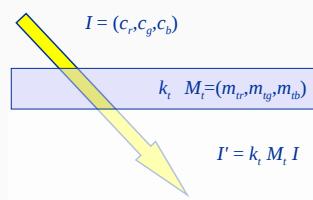
Hay objetos sólidos o líquidos que permiten pasar (además de reflejar o absorber) algunos fotones de la luz que los alcanza. Su estructura atómica permite a los rayos de luz viajar en línea recta.



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 $M_t = (m_{tr}, m_{tg}, m_{tb})$

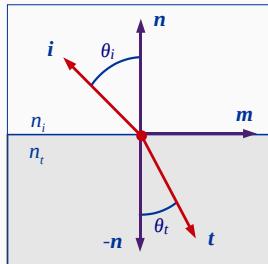
Por tanto, estos materiales están caracterizados por k_t y M_t

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 35 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 36 de 94.

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:

$$n_i \sin \theta_i = n_t \sin \theta_t$$



$$\begin{aligned} \mathbf{t} &= (r \cos \theta_i - \cos \theta_t) \mathbf{n} - r \mathbf{i} \\ \cos \theta_i &= \mathbf{i} \cdot \mathbf{n} \\ \cos \theta_t &= \sqrt{1 - r^2 [1 - (\mathbf{i} \cdot \mathbf{n})^2]} \\ r &= n_i / n_t \end{aligned}$$

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 $\theta_i > \theta_{\max}$.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 37 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 38 de 94.

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 es lo que 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 $\mathbf{t} = -\mathbf{i}$
- ▶ 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)

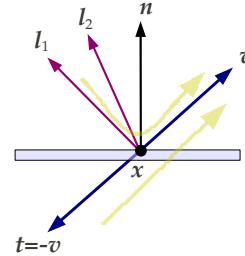
```

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:    $k_t = \text{fracción de luz refractada de } Q$ 
6:    $M_t = \text{color transparente de } Q$ 
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:



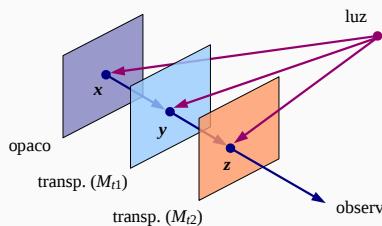
Si $k_t > 0$, al MIL debe sumársele la luz refractada proveniente del otro lado del polígono, en la dirección de v

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 39 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 40 de 94.

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:

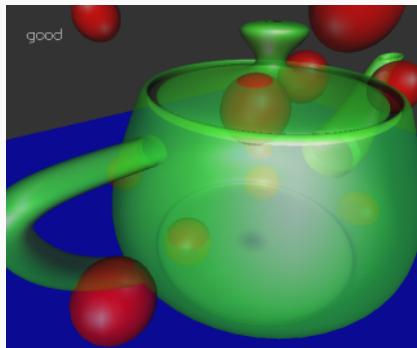
```

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:    $k_t = \text{fracción de luz refractada de } Q$ 
6:    $M_t = \text{color transparente de } Q$ 
7:   for cada pixel  $(x, y)$  ocupado por Q do
8:     if Q es visible en  $(x, y)$  then
9:        $I_m = \text{resultado de evaluar MIL}$ 
10:       $I[x, y] = I_m + k_t M_t I[x, y]$ 

```

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 41 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 42 de 94.



GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 43 de 94.

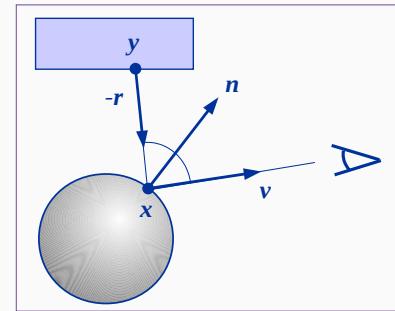
Reflexión especular de la luz

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).

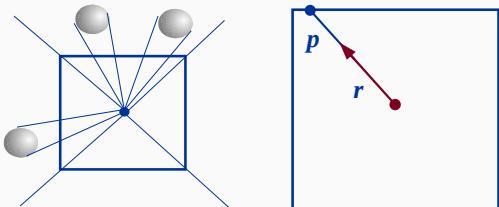


GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 45 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 46 de 94.

Mapas de entorno tipo caja (box map)

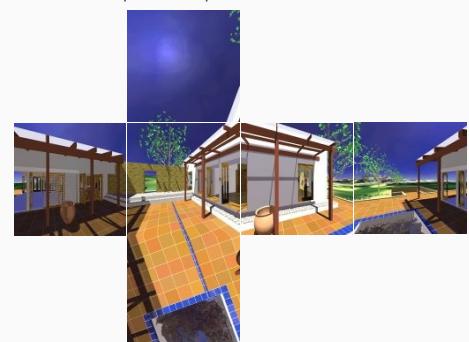
En esta técnica, el entorno se proyecta en las 6 caras de un cubo centrado en el objeto reflectante, obteniéndose 6 texturas.



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



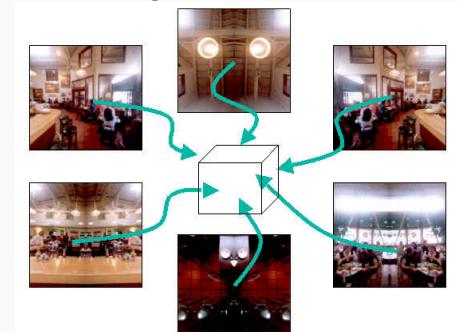
http://developer.nvidia.com/object/cube_map_ogl_tutorial.html

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 47 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 48 de 94.

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

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.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 49 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 50 de 94.

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:



GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 51 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 52 de 94.

Mapa de entorno

Ejemplo de combinación con texturas y perturbación de la normal. Además, la tetera se muestra en el entorno que refleja:



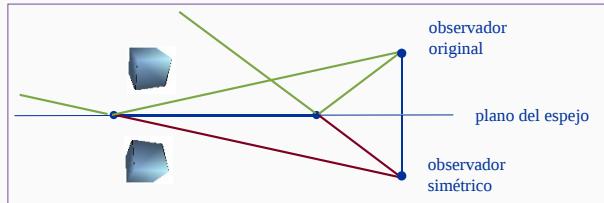
Ejemplo en cine de animación



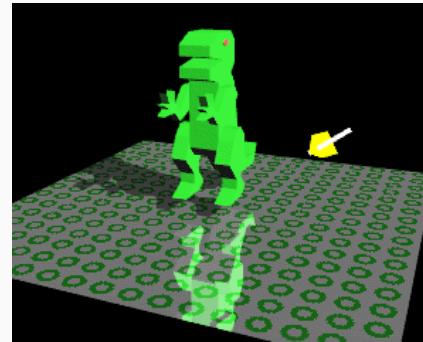
GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 53 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 54 de 94.

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.



2.1. El algoritmo de Ray-Tracing

- 2.2. Intersecciones rayo-objeto y rayo-escena
- 2.3. Problemas: Cálculo de intersecciones
- 2.4. Ejemplos

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*)
- ▶ Describo 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.

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.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 61 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 62 de 94.

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:

$$\begin{aligned} L_{\text{dir}}(\mathbf{p}, \mathbf{v}, \mathbf{l}) &\equiv k_a(\mathbf{p}) \cdot C(\mathbf{p}) \\ &+ k_d(\mathbf{p}) \cdot C(\mathbf{p}) \cdot \max(0, \mathbf{n} \cdot \mathbf{l}) \\ &+ k_s(\mathbf{p}) \cdot d_i \cdot [\max(0, \mathbf{r} \cdot \mathbf{v})]^e \end{aligned}$$

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:

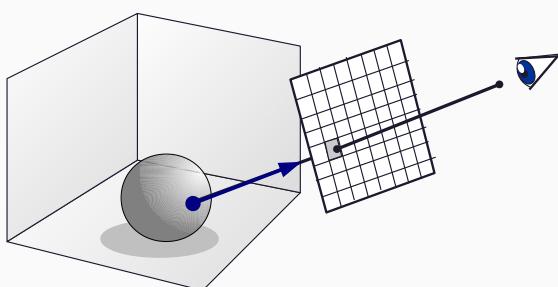
- ▶ \mathbf{r} ≡ vector reflejado en \mathbf{p} (depende de \mathbf{v} y \mathbf{n}).
- ▶ \mathbf{n} ≡ normal en \mathbf{p} (depende de \mathbf{p} y O).
- ▶ k_a, k_d, k_s ≡ parámetros del material en \mathbf{p} .
- ▶ $C(\mathbf{p})$ ≡ color del objeto en \mathbf{p}

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 63 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 64 de 94.

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:



GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 65 de 94.

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_{in} :

$$L_{\text{ind}}(\mathbf{p}, \mathbf{v}) \equiv A_G(\mathbf{p}) + k_{ps}(\mathbf{p})M_{ps}(\mathbf{p})L_{\text{in}}(\mathbf{p}, \mathbf{r}) + k_t(\mathbf{p})M_T(\mathbf{p})L_{\text{in}}(\mathbf{p}, \mathbf{t})$$

- ▶ $A_G(\mathbf{p})$ ≡ radiancia ambiente global (suple ilum. indirecta).
- ▶ $k_t(\mathbf{p})$ ≡ fracción de luz refractada en \mathbf{p} .
- ▶ $k_{ps}(\mathbf{p})$ ≡ fracc. de luz refl. de forma especular perfecta en \mathbf{p} .
- ▶ $M_{ps}(\mathbf{p})$ 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} ≡ vector reflejado en \mathbf{p} (depende de \mathbf{v} y \mathbf{n}).
- ▶ \mathbf{t} ≡ vector refractado en \mathbf{p} (depende de \mathbf{v} y \mathbf{n}).

El procedimiento principal de Ray-Tracing

El pseudocódigo del algoritmo, que recorre todos los pixels, puede quedar así:

```

1:  $\mathbf{o}$  := posición del observador, en coords. del mundo
2: for cada pixel  $(i, j)$  de la imagen do
3:    $\mathbf{q}$  := punto central (en WCC) del pixel  $(i, j)$ 
4:    $\mathbf{u}$  := vector desde  $\mathbf{o}$  hasta  $\mathbf{q}$  normalizado
5:    $rad := \text{RAYTRACING}(\mathbf{o}, \mathbf{u}, 1)$ 
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.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 66 de 94.

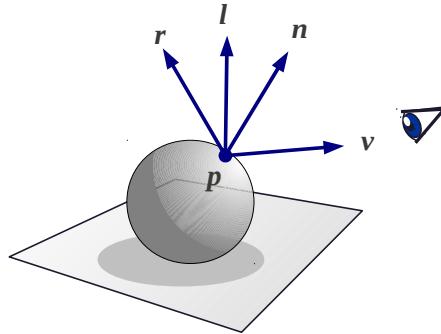
Esta función calcula la radiancia incidente sobre el punto \mathbf{o} , proveniente de la dirección \mathbf{u} (como una terna RGB). El entero n es el nivel de profundidad en las llamadas recursivas.

```

1: function RAYTRACING( punto  $\mathbf{o}$ , vector  $\mathbf{u}$ , 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  $\mathbf{o}$  en la dir.  $\mathbf{u}$  // (por ray-casting)
5:   if no existe ningún objeto visible then
6:     return radiancia de fondo correspondiente a  $\mathbf{u}$ 
7:    $\mathbf{p} :=$  punto de  $O$  intersecado
8:   return EVALUAMILREC(  $O$ ,  $\mathbf{p}$ ,  $-\mathbf{u}$ ,  $n$  )

```

Una vez se conoce el punto \mathbf{p} , se obtienen la normal \mathbf{n} , y los parámetros del MIL, que es evaluado:



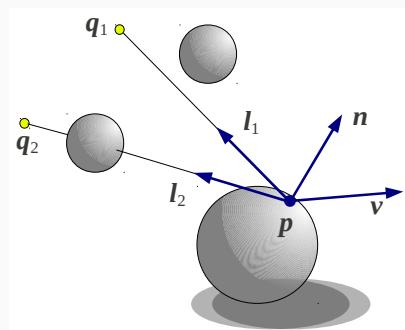
podemos considerar objetos curvos (esferas, cilindros, conos, etc..)

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 67 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 68 de 94.

Sombras arrojadas

Este método permite incorporar sombras arrojadas, usando Ray-Casting para visibilidad. Se comprueba si un segmento de recta desde \mathbf{p} hasta (o hacia) la fuente interseca algún objeto de la escena, es decir, se evalua $V(\mathbf{p}, \mathbf{q}_i)$

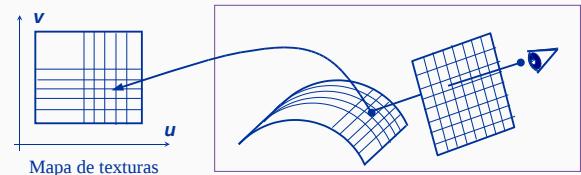


GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 69 de 94.

Detalles de las superficies

El objeto en el que está \mathbf{p} puede tener asociadas texturas (o mapas de pert. de la normal)

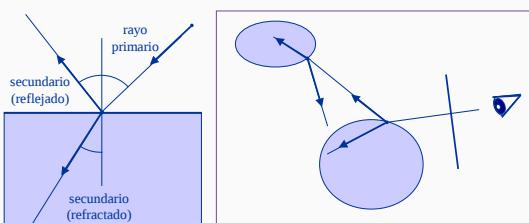
- ▶ A partir de \mathbf{p} se obtienen las coordenadas (u, v) en el espacio de la textura
- ▶ A partir de (u, v) se consulta la textura o texturas asociadas al objeto.



GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 70 de 94.

Raios 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 evalúa el MIL (EVALUAMILREC) llama recursivamente a la función de RAYTRACING,

```

1: function EVALUAMILREC(  $O$ ,  $\mathbf{p}$ ,  $\mathbf{v}$ ,  $n$  )
2:   Obtener paráms. del material de  $O$  en  $\mathbf{p}$  ( $\mathbf{n}, C, k_a, k_d, k_s, k_{ps}, M_{PS}, M_T, k_t$ )
3:   Obtener parámetros de fuentes de luz ( $n_L, \mathbf{l}_i, S_i$ )
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  $\mathbf{p}$  then // que sea visible
7:        $rad := rad + S_i \cdot DIRECTA(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$  // ilum. directa
8:     if  $k_t > 0$  and  $n < max$  then
9:        $\mathbf{t} :=$  vector refractado respecto de  $\mathbf{v}$ 
10:       $rad := rad + k_t \cdot M_T \cdot RAYTRACING(\mathbf{p}, \mathbf{t}, n+1)$  // componente refractada
11:      if  $k_{ps} > 0$  and  $n < max$  then
12:         $\mathbf{r} :=$  vector reflejado respecto de  $\mathbf{v}$ 
13:         $rad := rad + k_{ps} \cdot M_{PS} \cdot RAYTRACING(\mathbf{p}, \mathbf{r}, n+1)$  // componente reflejada
14:   return rad

```

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 71 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 72 de 94.

Esta función evalua L_{dir} (las componentes ambiental, difusa y especular correspondientes a una fuente de luz).

```

1: function DIRECTA( p, v, l )
2:   rad :=  $k_a \cdot C$ 
3:   if  $k_d > 0$  then
4:     rad := rad +  $k_d \cdot C \cdot \max(0, n \cdot l)$ 
5:   if  $k_s > 0$  then
6:     rad := rad +  $k_s \cdot d_i \cdot [\max(0, r \cdot v)]^e$ 
7:   return rad

```

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 73 de 94.

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 semirecta 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 $O(\log n_o)$.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 75 de 94.

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.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 76 de 94.

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 ∂O es un subconjunto de una superficie cuádratica, 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

$$F(\mathbf{p}) = s \cdot \left(\min_{\mathbf{q} \in \partial O} \|\mathbf{p} - \mathbf{q}\| \right) \text{ donde: } s \equiv \begin{cases} -1 & \text{si } \mathbf{p} \in O \\ +1 & \text{si } \mathbf{p} \notin O \end{cases}$$

Si no se dispone de SDF existen alternativas (Bisección, Newton, cotas de distancia, etc...).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 77 de 94.

Indexación espacial para métodos directos.

Para calcular las intersecciones rayo-escena podemos usar los algoritmos directos de intersección rayo-objeto para los 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 $O(\log n_t)$.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 78 de 94.

Un método iterativo, llamado **Sphere Tracing**, usa las SDFs F_i de los objetos. El algoritmo se basa en avanzar un punto a lo largo del rayo. A cada paso se avanza la mínima distancia del punto a los objetos de la escena, hasta que diverge o converge a un punto de intersección:

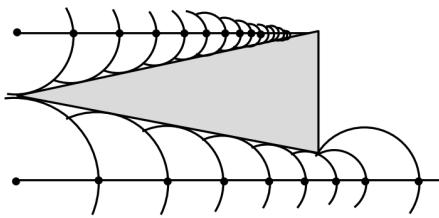


Figura obtenida del artículo *Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces*, de John C. Hart (1995). [PDF aquí](#).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 79 de 94.

Informática Gráfica, curso 2022-23.
Teoría. Tema 5. Realismo en Rasterización. Ray-tracing.
Sección 2. Ray tracing

Subsección 2.3.
Problemas: Cálculo de intersecciones.

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 INTERSECCIONESDF( 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](#)

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 80 de 94.

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).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 82 de 94.

Problema: intersección rayo-disco (2/2)

Problema: intersección rayo-esfera

Problema 5.2.

Diseña un algoritmo para calcular la primera intersección entre un rayo (con origen en **o** y vector **d**, normalizado) y una esfera de radio unidad y centro en el origen, si hay alguna.

Ten en cuenta que un punto cualquiera **p** está en esfera si y solo si el módulo de **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).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 83 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 84 de 94.

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).

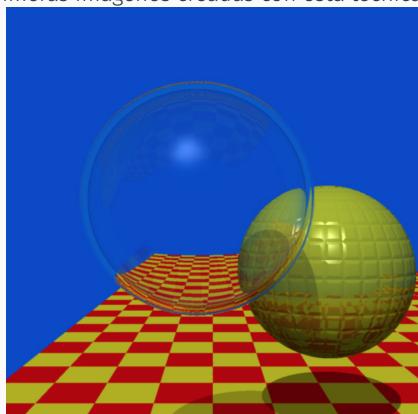
Usando esa definición diseña el algoritmo para calcular la intersección rayo-cilindro.

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).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 85 de 94.

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](#)

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 87 de 94.

Ejemplos: reflexión y refracción

Incluye: texturas, sombras arrojadas, reflexiones, *bump-mapping*.

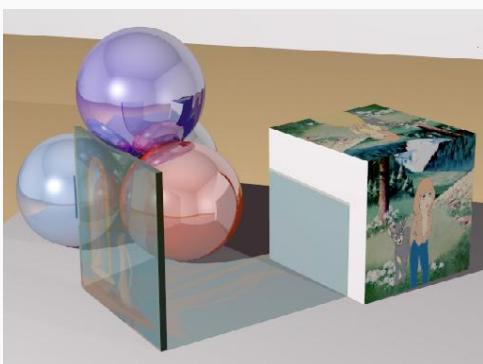


Universidad de Cornell (1998) [Computer Graphics programme: Reflection and Transparency](#).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 88 de 94.

Ejemplos: reflexiones especulares, sombras arrojadas.

Incluye: reflexiones, texturas, sombras arrojadas con texturas.



Ejemplos: escenas complejas, indexación espacial.

Escena compleja: múltiples objetos complejos. Indexación espacial.



Autor: Gilles Tran [Sitio web Oyonale.](#) ([Public Domain](#))

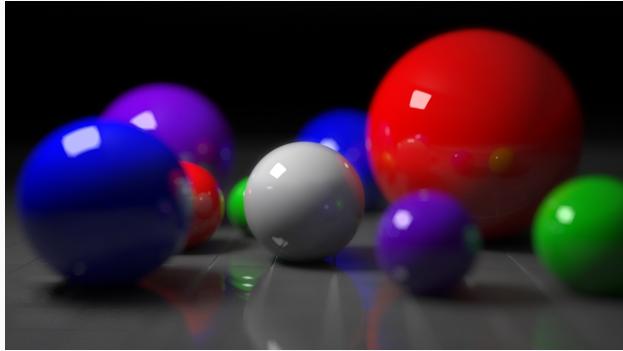
Universidad de Cornell (1998) [Computer Graphics programme: Reflection and Transparency](#).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 89 de 94.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 90 de 94.

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):

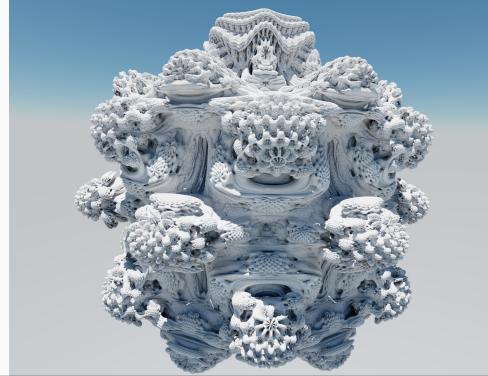


By Mimigu at [English Wikipedia: Ray Tracing \(Graphics\)](#). CC BY 3.0.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 91 de 94.

Ejemplos: Path-tracing, fractales.

Se usa *Path-tracing* (Kajiya, 1986, *Global Illumination*), e intersecciones con un fractal (con métodos numéricos iterativos).

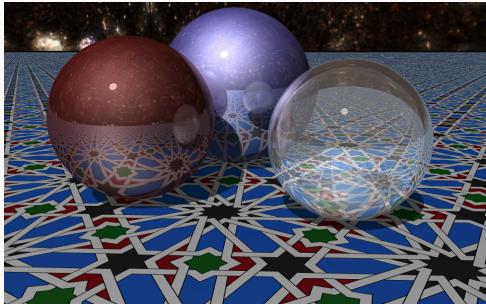


Por: Mikael Hvidtfeldt Christensen [Blog sobre arte generativo.](#)
<https://www.flickr.com/photos/syntopia/>.

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 92 de 94.

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)

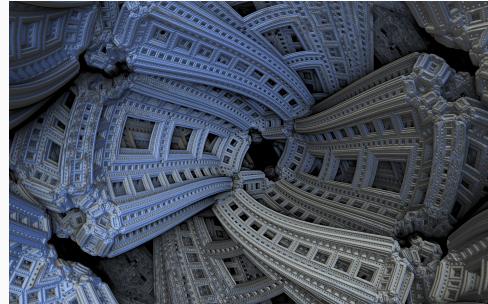


Animación disponible on-line en Shadertoy: <https://www.shadertoy.com/view/4sdfzn>

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 93 de 94.

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](#).

GIM: Informática Gráfica- curso 22-23- creado el 4 de diciembre de 2022 – transparencia 94 de 94.

Fin de la presentación.

Problema 1.1.

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

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

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

(el enunciado continua en la siguiente transparencia)

Problema 1.1. (continuación)

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

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

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

// Ejercicio 1

```
// A) GL_LINE_LOOP

// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos;

std::vector<Tupla2d> posiciones;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n)});
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_DOUBLE, 2,
posiciones.size(), posiciones.data());
    }
} else{
    glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa
}
```

// B) GL_LINES

```
// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos;

std::vector<Tupla2d> posiciones;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n)});
    posiciones.push_back({cos(2*M_PI*(i+1)/n),
sin(2*M_PI*(i+1)/n)})
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_DOUBLE, 2,
posiciones.size(), posiciones.data());
    }
} else{
    glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa
}
```

Problema 1.2.

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

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

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

// Ejercicio 1.2

```
// A) NO INDEXADA (3n vertices)

// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos;

std::vector<Tupla3f> posiciones;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n), 0.0});
    posiciones.push_back({cos(2*M_PI*(i+1)/n),
sin(2*M_PI*(i+1)/n), 0.0});
    posiciones.push_back({0.0, 0.0, 0.0});
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_FLOAT, 3,
posiciones.size(), posiciones.data());
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO ya
creado, simplemente se activa
```

// B) INDEXADA (3n indice y n+1 vertices)

```
// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos,
nombre_vbo_ind;

std::vector<Tupla3f> posiciones;
std::vector<Tupla3u> indices;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n), 0.0});
}
posiciones.push_back({0.0, 0.0, 0.0});

for(int i=0; i<n; i++){
    indices.push_back({i, (i+1)%n, n});
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_FLOAT, 3,
posiciones.size(), posiciones.data());
    }
    if(indices.size() > 0){
        nombre_vbo_ind = CrearVBOInd( indices );
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO ya
creado, simplemente se activa
}
```

Problema 1.3.

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

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

(el enunciado continua en la siguiente transparencia)

Problema 1.3. (continuación)

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

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

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

```
// Ejercicio 1.3
```

```
    // A) GL_LINE_LOOP
```

```
    // Poligono de n lados
```

```
    int n=5;
```

```
    int nombre_vao = 0;
```

```
    int nombre_vbo_posiciones, ind_pos,  
    ind_colores;
```

```
    std::vector<Tupla2d> posiciones;
```

```
    for(int i=0; i<n; i++){
```

```
        posiciones.push_back({cos(2*M_PI*i/n),  
        sin(2*M_PI*i/n)});
```

```
    }
```

```
    if(nombre_vao == 0){
```

```
    //A partir de ahora debemos crear un nuevo  
    VAO donde le vamos a volver a pasar los  
    datos necesarios para formar  
    // un polígono relleno con triángulos
```

```
    // Poligono de n lados
```

```
    std::vector<Tupla3u> indices;
```

```
    int nombre_vao_2, nombre_vbo_ind;
```

```
    posiciones.push_back({0.0, 0.0, 0.0});
```

```
    for(int i=0; i<n; i++){  
        indices.push_back({i, (i+1)%n, n});  
    }
```

```
    if(nombre_vao_2 == 0){
```

<pre> nombre_vao = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_DOUBLE, 2, posiciones.size(), posiciones.data()); } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } //Con esto le decimos que el color por defecto es el negro para los vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,0.0,1.0); glDrawArrays(GL_LINE_LOOP, 0, posiciones.size()); glBindVertexArray(0); //Hasta aquí hemos dibujado las aristas del polígono. </pre>	<pre> nombre_vao_2 = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } else{ glBindVertexArray(nombre_vao_2); // VAO ya creado, simplemente se activa } glVertexAttrib3f(ind_colores, 1.0,0.0,0.0,1.0); glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0); glBindVertexArray(0); //Ahora hemos dibujado el relleno del polígono(mostrando los triángulos en vez de las aristas). </pre>
---	--

Problema 1.4.

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

<pre> // Ejercicio 1.4 // A) GL_LINE_LOOP // Poligono de n lados int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla2d> posiciones; for(int i=0; i<n; i++){ posiciones.push_back({cos(2*M_PI*i/n), sin(2*M_PI*i/n)}); } posiciones.push_back({0.0, 0.0, 0.0}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; for(int i=0; i<n; i++){ indices.push_back({i, (i+1)%n, n}); } if(nombre_vao == 0){ nombre_vao = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } </pre>	<pre> //Con esto le decimos que el color por defecto es el negro para los vertices del VAO activo ahora mismo //El Último 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,0.0); // Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices glDrawArrays(GL_LINE_LOOP, 0, (posiciones.size()-1)); glVertexAttrib3f(ind_colores, 1.0,0.0,0.0); glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0); glBindVertexArray(0); </pre>
---	--

Problema 1.5.

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

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

<pre>// Ejercicio 1.5 // A) GL_LINE_LOOP // Poligono de n lados int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla3d> posiciones; std::vector<Tupla3f> colores; for(int i=0; i<n; i++){ posiciones.push_back({cos(2*M_PI*i/n), sin(2*M_PI*i/n), 0.0}); colores.push_back({rand()/float(RAND_MAX), rand()/float(RAND_MAX), rand()/float(RAND_MAX)}); posiciones.push_back({0.0, 0.0, 0.0}); colores.push_back({rand()/float(RAND_MAX), rand()/float(RAND_MAX), rand()/float(RAND_MAX)}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; for(int i=0; i<n; i++){ indices.push_back({i, (i+1)%n, n}); } if(nombre_vao == 0){ nombre_vao = crearVAO();</pre>	<pre>if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } if(colores.size()>0){ nombre_vbo_col = CrearVBOAtrib(ind_colores, GL_FLOAT, 3, colores.size(), colores.data()); } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } //El color por defecto es el negro para los vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,0.0); glDisableVertexAttribArray(ind_colores); //Quito la tabla de colores para pintar las aristas // Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices glDrawArrays(GL_LINE_LOOP, 0, (posiciones.size()-1)); glEnableVertexAttribArray(ind_colores); //Como vamos a pintar el relleno, habilito la tabla de colores glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0); glBindVertexArray(0);</pre>
---	--

Problema 1.6.

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

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

// Ejercicio 1.6

```
struct VStruct{  
    Tupla3f pos;  
    Tupla3f col;  
};  
  
int crearVBOAOS(std::vector<VStruct> v){  
    int nombre_vbo;  
    int tamano = v.size()*sizeof(VStruct);  
    glGenBuffers(1, &nombre_vbo);  
    glBindBuffer(GL_ARRAY_BUFFER,  
    nombre_vbo);  
  
    glBufferData(GL_ARRAY_BUFFER,tamani  
o,v.data(), GL_STATIC_DRAW);  
  
    glVertexAttribPointer(ind_posiciones, 3,  
    GL_FLOAT, GL_FALSE,sizeof(VStruct), 0);  
  
    glVertexAttribPointer(ind_colores, 3,  
    GL_FLOAT, GL_FALSE, sizeof(VStruct),  
    3*sizeof(float));  
  
    glEnableVertexAttribArray(ind_posiciones);  
    glEnableVertexAttribArray(ind_colores);  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
    return nombre_vbo;
```

// Poligono de n lados

```
std::vector<Tupla3u> indices;  
int nombre_vbo_ind;  
  
for(int i=0; i<n; i++){  
    indices.push_back(i, (i+1)%n, n);  
}  
  
if(nombre_vao == 0){  
    nombre_vao = crearVAO();  
  
    if(vectorAOS.size() > 0) {  
        nombre_vbo_aos =  
crearVBOAOS(vectorAOS);  
    }  
    if(indices.size() > 0){  
        nombre_vbo_ind = CrearVBOInd(  
indices );  
    }  
}  
else{  
    glBindVertexArray(nombre_vao); // VAO  
ya creado, simplemente se activa  
}  
  
//Con esto le decimos que el color por  
defecto es el negro para los vértices del  
VAO activo ahora mismo
```

```

};

// A) GL_LINE_LOOP

// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_aos, ind_pos,
ind_colores;

std::vector<VStruct> vectorAOS;

for(int i=0; i<n; i++){
    VStruct v;
    v.pos = {cos(2*M_PI*i/n),
    sin(2*M_PI*i/n), 0.0};
    v.col = {rand()/float(RAND_MAX),
    rand()/float(RAND_MAX),
    rand()/float(RAND_MAX)};
    vectorAOS.push_back(v);

}

VStruct v;
v.pos = {0.0, 0.0, 0.0};
v.col = {rand()/float(RAND_MAX),
rand()/float(RAND_MAX),
rand()/float(RAND_MAX)};
vectorAOS.push_back(v);

```

//El ultimo 1 se refiere a la transparencia
`glVertexAttrib3f(ind_colores, 0.0,0.0,0.0);`
`glDisableVertexAttribArray(ind_colores);`
//Quito la tabla de colores para pintar las aristas

// Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices
`glDrawArrays(GL_LINE_LOOP, 0,`
`(vectorAOS.size()-1));`

`glEnableVertexAttribArray(ind_colores);`
//Como vamos a pintar el relleno, habilito la tabla de colores
`glDrawElements(GL_TRIANGLES,`
`indices.size()*3, GL_UNSIGNED_INT, 0);`
`glBindVertexArray(0);`

Problema 1.7.

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

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

// Ejercicio 1.7

```
float r = float(ancho_actual)/alto_actual;

float x0, x1, y0, y1;
float z0 = -1.0, z1 = 1.0;

if (r >= 1){
    x0 = -r, x1 = r;
    y0 = -1.0, y1 = 1.0;
}
else{
    x0 = -1.0, x1 = 1.0;
    y0 = -1/r, y1 = 1/r;
}

float sx = 2 / (x1-x0), sy = 2 / (y1-y0), sz = 2/(z1-z0);
float cx = (x0+x1) / 2, cy = (y0+y1) / 2, cz = (z0+z1) / 2;

GLfloat matriz_proyeccion[16] = {
    sx, 0, 0, -cx*sx,
    0, sy, 0, -cy*sy,
    0, 0, sz, -cz*sz,
    0, 0, 0, 1
};
glUniformMatrix4fv( loc_mat_proyeccion, 1, GL_TRUE, matriz_proyeccion );
///////////////////////////////
```

Sea $C = [\hat{e}_x, \hat{e}_y, \hat{e}_z]$ un marco de referencia cartesiano. Sean \vec{a}, \vec{b} dos vectores cualesquiera. Si sus coordenadas en el marco C son $\vec{a} = ((a_x, a_y, a_z, 0))^T$ y $\vec{b} = ((b_x, b_y, b_z, 0))^T$, entonces:

$$\vec{a} = a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z \quad \vec{b} = b_x \hat{e}_x + b_y \hat{e}_y + b_z \hat{e}_z$$

$$\begin{aligned}\vec{a} \cdot \vec{b} &= (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_x \hat{e}_x + b_y \hat{e}_y + b_z \hat{e}_z) = (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_x \hat{e}_x) + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_y \hat{e}_y) + \\ &\quad + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_z \hat{e}_z) = a_x b_x (\hat{e}_x \cdot \hat{e}_x) + a_x b_y (\hat{e}_x \cdot \hat{e}_y) + a_x b_z (\hat{e}_x \cdot \hat{e}_z) + a_y b_x (\hat{e}_y \cdot \hat{e}_x) + a_y b_y (\hat{e}_y \cdot \hat{e}_y) + \\ &\quad + a_y b_z (\hat{e}_y \cdot \hat{e}_z) + a_z b_x (\hat{e}_z \cdot \hat{e}_x) + a_z b_y (\hat{e}_z \cdot \hat{e}_y) + a_z b_z (\hat{e}_z \cdot \hat{e}_z) = [a_x b_x + a_y b_y + a_z b_z]\end{aligned}$$

$$\begin{aligned}\vec{a} \times \vec{b} &= (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_x \hat{e}_x + b_y \hat{e}_y + b_z \hat{e}_z) = (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_x \hat{e}_x) + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_y \hat{e}_y) + \\ &\quad + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_z \hat{e}_z) = a_x b_x (\hat{e}_x \times \hat{e}_x) + a_x b_y (\hat{e}_x \times \hat{e}_y) + a_x b_z (\hat{e}_x \times \hat{e}_z) + \\ &\quad + a_y b_x (\hat{e}_y \times \hat{e}_x) + a_y b_y (\hat{e}_y \times \hat{e}_y) + a_y b_z (\hat{e}_y \times \hat{e}_z) + a_z b_x (\hat{e}_z \times \hat{e}_x) + a_z b_y (\hat{e}_z \times \hat{e}_y) + a_z b_z (\hat{e}_z \times \hat{e}_z) = \\ &= [(a_y b_z - a_z b_y) \hat{e}_x + (a_z b_x - a_x b_z) \hat{e}_y + (a_x b_y - a_y b_x) \hat{e}_z].\end{aligned}$$

- ④ Dada un vector \vec{v} , $\vec{v} \times \vec{v} = 0$ pues $\vec{v} \times \vec{v} = -\vec{v} \times \vec{v}$ y el único vector que cumple ser su mismo opuesto es el $\vec{0}$.

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y) \cdot \hat{e}_x + (a_z b_x - a_x b_z) \cdot \hat{e}_y + (a_x b_y - a_y b_x) \cdot \hat{e}_z$$

Para ver que son perpendiculares veamos que el producto escalar es 0:

$$\begin{aligned}1) (\vec{a} \times \vec{b}) \cdot \vec{a} &= (a_y b_z - a_z b_y) \cdot a_x + (a_z b_x - a_x b_z) \cdot a_y + (a_x b_y - a_y b_x) \cdot a_z = \\ &= a_x a_y b_z - a_x a_z b_y + a_y a_z b_x - a_x a_y b_z + a_x a_z b_y - a_y a_z b_x = 0. \checkmark\end{aligned}$$

$$\begin{aligned}2) (\vec{a} \times \vec{b}) \cdot \vec{b} &= (a_y b_z - a_z b_y) \cdot b_x + (a_z b_x - a_x b_z) \cdot b_y + (a_x b_y - a_y b_x) \cdot b_z = \\ &= a_x b_x b_z - a_z b_x b_y + a_z b_x b_y - a_x b_y b_z + a_x b_y b_z - a_y b_x b_z = 0! \checkmark\end{aligned}$$

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).

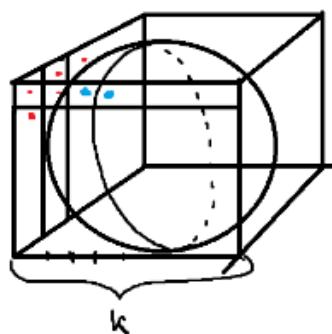
(continua en la siguiente transparencia)

Problema 2.1. (continuación)

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

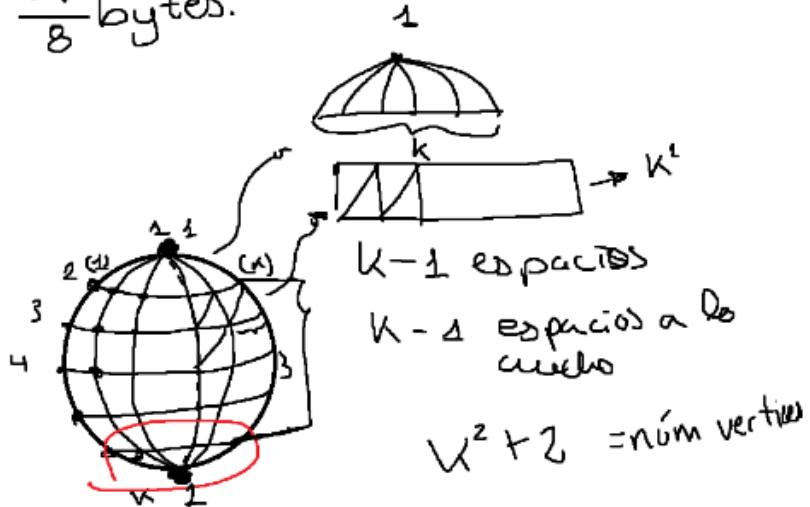
- ▶ Expresa el tamaño de ambas representaciones en bytes como una función de k .
- ▶ Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- ▶ Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

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



Si metemos la esfera en un cubo de lado k (k celdas)
Tenemos en total k^3 celdas

Por tanto, el modelo ocupa k^3 bits ó
 $\frac{k^3}{8}$ bytes.



Tapas son $k-1$ triángulos

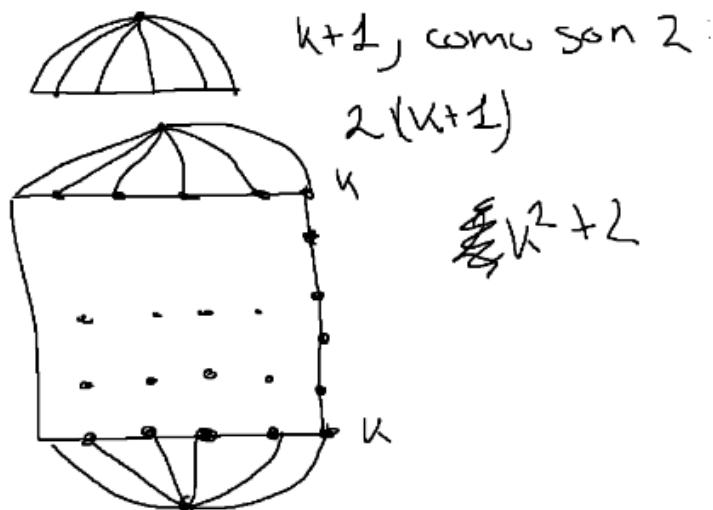
$$2(k-1) + 2(k-3)^2$$

$$2k^2 + 18 - 8k + 2k - 2 = 2k^2 - 4k + 16 - 16$$



trian

$3 \cdot N_t = \text{Num indices}$

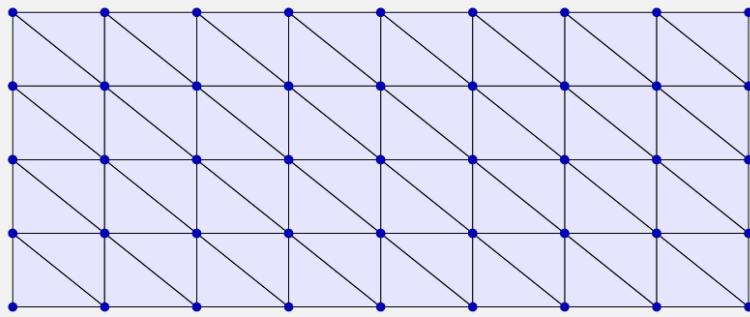


Resumiendo:

$$\begin{aligned} k^2 + 2 \text{ vértices} &\rightarrow (k^2 + 2) \cdot 3 \cdot 4 \text{ bytes} \\ 3 \cdot \underbrace{(2k^2 - 4k + 16)}_{n \text{ triángulos}} \cdot 4 &\quad \underbrace{n \text{ bytes para}}_{\text{vértices}} \\ &\quad \underbrace{n \text{ bytes de}}_{\text{tabla triángulos}} \\ \text{num. de enteros para} & \\ \text{rep. todos los triángulos} & \end{aligned}$$

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)

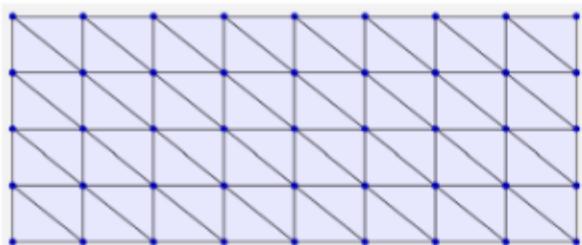
GIM: Informática Gráfica- curso 22-23- creado el 17 de septiembre de 2022 – transparencia 63 de 184

Problema 2.2. (continuación)

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

- 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 .
- Escribe el tamaño en KB suponiendo que $m = n = 128$.
- 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.

2.2.



Hay $(n+1)(m+1)$ vértices

Hay $2 \cdot n \cdot m$ triángulos B es flat

Para vértices $(n+1)(m+1) \cdot 2 \cdot \frac{1}{4}$

Pura triángulos

$$3 \cdot 2 \cdot n \cdot m \cdot 4 = 24n \cdot m$$

c)

$$\lim_{n \rightarrow \infty} \frac{sn + nm + sm + 8}{24nm} = \frac{0}{24} - \frac{L}{3}$$

\uparrow
 $n = \text{cras}$

Hay 3 veces más triángulos que vértices.

$$\lim_{n \rightarrow \infty} M_n = \frac{8 \text{ dy}}{24 \text{ mm}} + \frac{8 \text{ dy}}{24 \text{ mm}} + \frac{8 \text{ dy}}{24 \text{ mm}} + \frac{8 \text{ dy}}{24 \text{ mm}}$$

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).

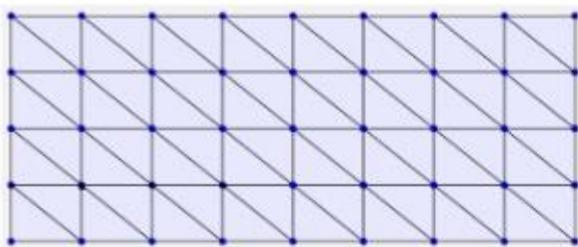
Responde a estas cuestiones:

(continua en la siguiente transparencia)

Problema 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.

2.3.



Hay m tiras, cada tira necesita
 $2n$ vértices

En total guardas $2nm$ vértices

Tienes $8 \cdot m$ Bytes de puntos

Tienes $2nm \cdot 2 \cdot 4$ Bytes para
vértices $\overset{2}{\text{Bytes}}$ $\overset{\text{Byte flat}}{\text{Bytes}}$ $\overset{\text{Bytes del}}{\text{2.2}}$

$$\lim_{n \rightarrow \infty, m \rightarrow \infty} \frac{32nm + 8n + 8m + 8}{16nm + 8m} =$$

= $\boxed{2}$ → las mallas indexadas
ocupan el doble.
(en 2D) ó

2 comp. por vértices)

c) $\lim_{m \rightarrow \infty, n \rightarrow \infty} \overbrace{\frac{(m-1)(n-1)}{2nm}}^{\text{nº vértices en 2.2}} = \frac{1}{2}$

La malla indexada tardará la mitad
que tiras de triángulos

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:

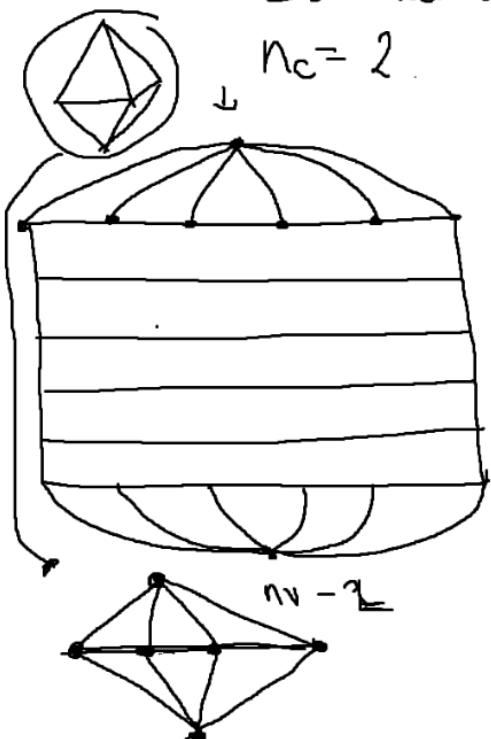
$$n_A = 3(n_V - 2)$$

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

(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).

2. 4.

Si tenemos n_V vértices



$$3 \cdot n_C = 2 n_A$$

fórmula Euler

$$n_V - n_A + n_C = 2$$

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 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.

```
void calcularAristas(){
    //Ejercicio 2.5 1) NO COHERENCIA
    std::vector<Tupla2i> ari;
    std::map<Tupla2i, Tupla2i> ari_mp;
    for(int i = 0; i < triangulos.size(); i++){
        for(int j=0; j < 3; j++){
            if(!ari_mp.find({triangulos[i][j],
                triangulos[i][(j+1)%3]}) &&
                !ari_mp.find(triangulos[i][(j+1)%3],
                {triangulos[i][j]}))
                ari_mp.insert({{triangulos[i][j],
                    triangulos[i][(j+1)%3]}});
        }
    }
    //Este codigo calcularía en nlog(n)
}
```

```
void calcularAristas(){
    //Ejercicio 2.5 2) COHERENCIA
    std::vector<Tupla2i> ari;
    std::map<Tupla2i, Tupla2i> ari_mp;
    for(int i = 0; i < triangulos.size(); i++){
        for(int j=0; j < 3; j++){
            if(triangulos[i][j]<triangulos[i][(j+1)%3]){
                ari.push_back({{triangulos[i][j],triangulos[i][(j+1)%3]}});
            }
        }
    }
    //Este codigo calcularía en n
}
```

Problema 2.6.

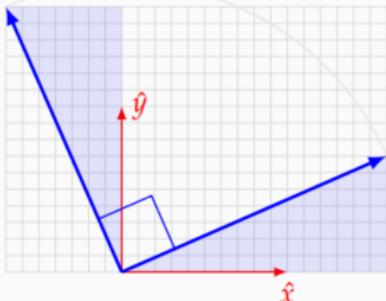
Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función que acepta un puntero a una **MallaInd** y devuelve un número real (asumir que se dispone del tipo **Tupla3f** y de los operadores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

```
// Ejercicio 2.6
void calculaArea(){
    double area = 0.0;
    for(int i=0; i < triangulos.size(); i++){
        Tupla3f v1 = (vertices[triangulos[i][0]] - vertices[triangulos[i][1]]);
        Tupla3f v2 = (vertices[triangulos[i][0]] - vertices[triangulos[i][2]]);
        Tupla3f cros = v1.cross(v2); // Producto vectorial, su módulo será el cuadrado de su
área
        area += sqrt(cros.lengthSq());
    }
    area = area/2;
}
```

Problema 2.7.

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

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

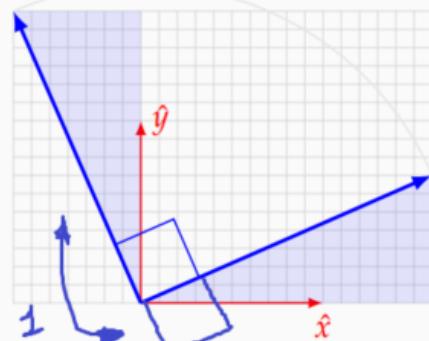


$$\vec{u} = a\hat{x} + b\hat{y}$$

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

$$\begin{aligned}\vec{u} \cdot P(\vec{u}) &= (a\hat{x} \cdot (-b\hat{x})) + a\hat{x} \cdot a\hat{y} + (b\hat{y} \cdot b\hat{x}) + b\hat{y} \cdot a\hat{y} \\ &\Rightarrow -ab + ba \Rightarrow 0\end{aligned}$$

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



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

Problema 2.12.

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

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

$$\vec{a} = (a_x \hat{x}, a_y \hat{y});$$

$$\vec{b} = (b_x \hat{x}, b_y \hat{y});$$

$$\text{Rot}[B]\vec{a} = \left[\underbrace{[\cos\theta \cdot a_x \cdot \hat{x} - \sin\theta \cdot a_y \cdot \hat{y}]}_{a_x}, \underbrace{[\sin\theta \cdot a_x \cdot \hat{x} + \cos\theta \cdot a_y \cdot \hat{y}]}_{a_y} \right]$$

$$\text{Rot}[B]\vec{b} = \left[\underbrace{[\cos\theta \cdot b_x \cdot \hat{x} - \sin\theta \cdot b_y \cdot \hat{y}]}_{b_x}, \underbrace{[\sin\theta \cdot b_x \cdot \hat{x} + \cos\theta \cdot b_y \cdot \hat{y}]}_{b_y} \right]$$

$$R_\theta(\vec{a}) \cdot R(\vec{b}) = \begin{aligned} & \cos^2\theta \cdot a_x \cdot b_x \cdot \hat{x} - \cos\theta \sin\theta \cdot a_x \cdot \hat{x} \cdot b_y \cdot \hat{y} \\ & - \sin\theta \cdot a_y \cdot \hat{y} \cdot \cos\theta \cdot b_x \cdot \hat{x} + \sin^2\theta \cdot a_y \cdot \hat{y} \cdot b_y \cdot \hat{y} \end{aligned}$$

$$\begin{aligned} & \Rightarrow \cos^2\theta a_x b_x + \sin^2\theta a_y b_y \\ & \Rightarrow 1 - \sin^2\theta a_x b_x + \sin^2\theta a_y b_y \\ & a_x b_x - \sin^2\theta (a_x b_x + a_y b_y); \end{aligned} \quad \left. \begin{aligned} & 2^\circ \text{ parte} \\ & a_y b_y + \sin^2\theta (a_y b_y + a_x b_x) \end{aligned} \right\}$$

$$R_\theta(\vec{a}) \cdot R(\vec{b}) = a_x b_x + a_y b_y$$

Problema 2.13.

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

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

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

P

$$\text{Rot}[\theta] \vec{a} = \left[\underbrace{[\cos \theta \cdot a_x \cdot \hat{x} - \sin \theta \cdot a_y \cdot \hat{y}]}_{a_x}, \underbrace{[\sin \theta \cdot a_x \cdot \hat{x} + \cos \theta \cdot a_y \cdot \hat{y}]}_{a_y} \right]$$
$$\| \text{Rot}[\theta] \vec{a} \| = \sqrt{\underbrace{\cos^2 \theta a_x^2}_{\frac{1}{1}} + \underbrace{\sin^2 \theta a_y^2}_{\frac{1}{1}} + \cancel{\hat{x}^2} + \underbrace{\sin^2 \theta a_x^2}_{\frac{1}{1}} + \underbrace{\cos^2 \theta a_y^2}_{\frac{1}{1}} - \cancel{\hat{x}^2}}$$
$$\underbrace{\cos^2 \theta a_x^2 + \sin^2 \theta a_x^2 + \sin^2 \theta a_y^2 + \cos^2 \theta a_y^2}_{a_x^2 + a_y^2}$$
$$a_x^2 + a_y^2$$

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 θ , se cumple:

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

$$2.14 \quad R_\theta(\vec{a} \times \vec{b}) = R_\theta(\vec{a}) \times R_\theta(\vec{b})$$

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

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

$$(a_y b_z - a_z b_y) \hat{x}$$

$$\cos \theta (a_z b_x - a_x b_z) - \sin \theta (a_x b_y - a_y b_x) = y$$

$$\sin \theta (a_z b_x - a_x b_z) + \cos \theta (a_x b_y - a_y b_x) = z$$

$$R_\theta(\vec{a}) = \begin{pmatrix} a_x & x \\ \cos \theta a_y - \sin \theta a_z & y \\ \sin \theta a_y + \cos \theta a_z & z \end{pmatrix}$$

$$R_\theta(\vec{b}) = \begin{pmatrix} b_x & x \\ \cos \theta b_y - \sin \theta b_z & y \\ \sin \theta b_y + \cos \theta b_z & z \end{pmatrix}$$

$$\begin{pmatrix} (\cos \theta a_y - \sin \theta a_z)(\sin \theta b_y + \cos \theta b_z) - (\sin \theta a_y + \cos \theta a_z)b_x \\ \sin \theta a_y + \cos \theta a_z \cdot b_x - a_x(\sin \theta b_y + \cos \theta b_z) \\ a_x(\cos \theta b_y - \sin \theta b_z) - b_x(\cos \theta a_y - \sin \theta a_z) \end{pmatrix}$$

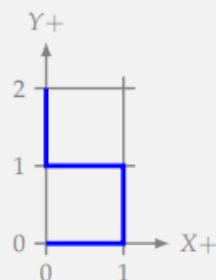
$$\cos^2 \theta a_y b_y + \cos \theta a_y b_z - \sin \theta a_z b_y - \sin \theta a_y b_z - \sin \theta a_x b_z - \sin \theta a_y b_x - \cos \theta a_z b_x$$

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 **opengl3-minimo** para esto.



```
void gancho(){
```

```
// A) GL_LINE_LOOP
```

```
// Polígono de n lados
```

```
int nombre_vao = 0;
```

```
int nombre_vbo_posiciones, ind_pos, ind_colores;
```

```
std::vector<Tupla2d> posiciones;
```

```

posiciones.push_back({0,0});
posiciones.push_back({1,0});
posiciones.push_back({1,1});
posiciones.push_back({0,1});
posiciones.push_back({0,2});

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_DOUBLE, 2, posiciones.size(),
posiciones.data());
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa
}

//Con esto le decimos que el color por defecto es el negro para los vertices del VAO activo
ahora mismo
//El último 1 se refiere a la transparencia
glVertexAttrib3f(ind_colores, 0.0,0.0,1.0,1.0);

glDrawArrays(GL_LINES, 0, posiciones.size());

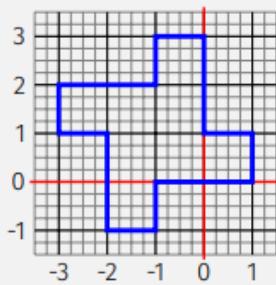
glBindVertexArray(0);

//Hasta aquí hemos dibujado las aristas del polígono.
}

```

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 **compMM** con **MAT_Traslacion** y **MAT_Rotacion**.

// Ejercicio 2.16

<pre>void gancho_trasladado(float x, float y){ compMM(MAT_Translacion({x,y, 0.0})); gancho(); compMM(MAT_Translacion({-x,-y, 0.0})); } }</pre>	<pre>void gancho_x4(){ compMM(MAT_Translacion({-1,0})); //Esta es la traslación de todos los ganchos gancho_trasladado(1, 0); compMM(MAT_Rotacion(90,{0,0,1})); gancho_trasladado(1, 0); compMM(MAT_Rotacion(90,{0,0,1})); gancho_trasladado(1, 0); compMM(MAT_Rotacion(90,{0,0,1})); gancho_trasladado(1, 0); }</pre>
---	--

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 distintos \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**)

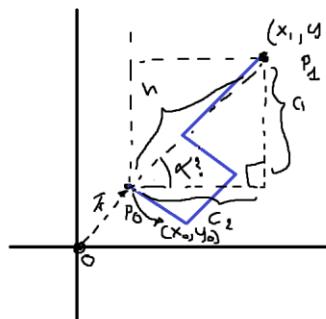
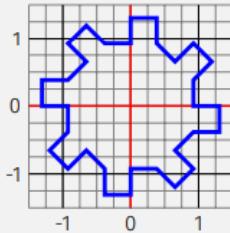
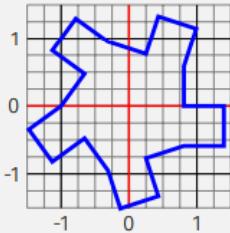
Escribe una solución (a) acumulando matrices de rotación, traslación y escalado en la matriz *modelview* de OpenGL. Escribe otra solución (b) en la cual la matriz *modelview* se calcula directamente sin necesidad de usar funciones trigonométricas (como lo son el arcotangente, el seno, coseno, arcoseno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

// Ejercicio 2.17

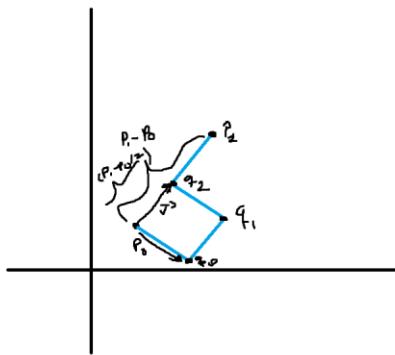
```
void ganchoEntrePuntos(Tupla3f p0, Tupla3f p1){  
  
void ganchoMultiple(int n){  
  
    std::vector<Tupla3f> posiciones;  
  
    for(int i=0; i<n; i++){  
        posiciones.push_back({cos(2*M_PI*i/n), sin(2*M_PI*i/n), 0.0});  
    }  
  
    for(int i=0; i<n; i++){  
        ganchoEntrePuntos(posiciones[i], posiciones[(i+1)%n]);  
    }  
}
```

Problema 2.18.

Usa 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).



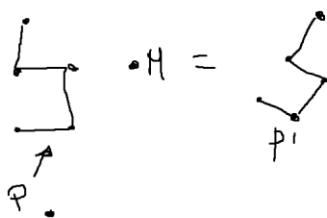
$$\begin{aligned} h^2 &= C_2^2 + C_1^2 \\ h &= \sin \alpha \cdot C_1 \\ \sin \alpha &= \frac{h}{C_1} \\ \alpha &= \arcsin\left(\frac{h}{C_1}\right) \end{aligned}$$



$$\sqrt{(P_1 - P_0) \cdot \text{sq-length}} \rightarrow \text{escalado}$$

$(P_0 - O)$ → traducción

$$C_2 = x_1 - x_0 \quad C_1 = y_1 - y_0$$



$$\begin{matrix} P_0 \\ \vdots \\ P_n \end{matrix} \xrightarrow{H} \begin{matrix} P'_1 \\ \vdots \\ P'_n \end{matrix}$$

$$(P_1 - P_0)/2 = \vec{V} = a\vec{x} + b\vec{y}$$

$$\vec{V}_{q_0} = b\vec{x} - a\vec{y}$$

$$P_0 + \vec{V}_{q_0} = q_0 \quad P_0 + \vec{V}_{q_0} + \vec{V} = q_1$$

$$P_0 + \vec{V} = q_2 \quad P_0 + 2\vec{V} = P_1$$

$$\{P_0, q_1, q_2, q_0, P_1\} = P' \text{ Juntos org para matriz } M$$

$$V_i(a_i, b_i) = a_i \hat{x} + b_i \hat{y}$$

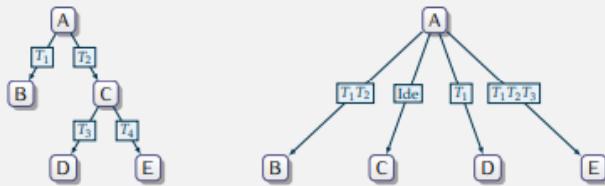
$$V'_i(a_i, b_i) = a_i \underbrace{\hat{x}}_{\downarrow} + b_i \underbrace{\hat{y}}_{\downarrow}$$

$$c_0 \hat{x} + c_1 \hat{y} \quad d_0 \hat{x} + d_1 \hat{y}$$

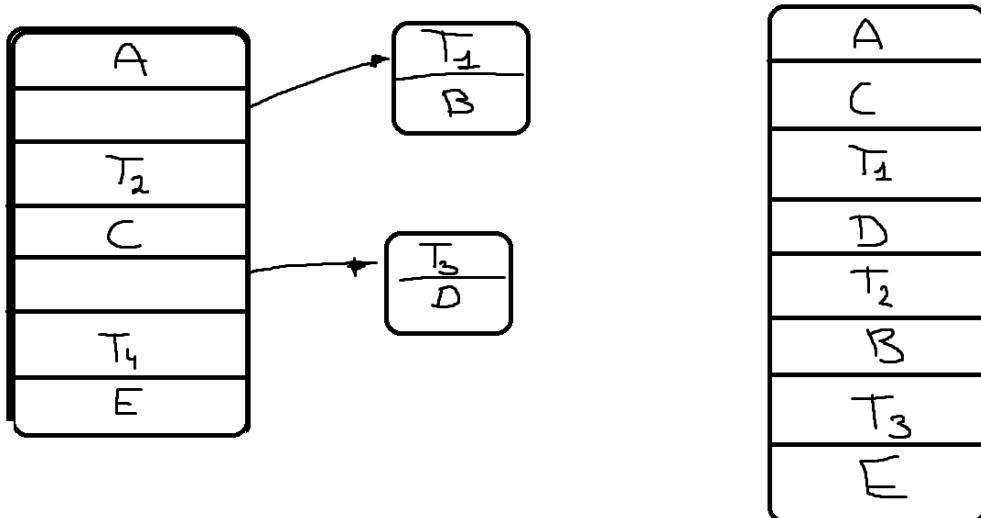
$$\begin{pmatrix} e_i \\ f_i \end{pmatrix} = \begin{pmatrix} c_0 & d_0 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} a_i \\ b_i \end{pmatrix}$$

Problema 2.19.

Dados los dos siguientes grafos de escena sencillos:

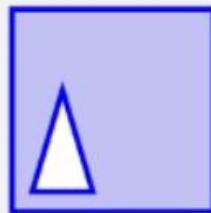


Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones: T_1, T_2 y T_3 .



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 **opengl3-minimo**)

// Ejercicio 2.20

<pre>void FiguraSimple(){ int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla3d> posiciones; posiciones.push_back({0,0,0}); posiciones.push_back({1,0,0}); posiciones.push_back({1,1,0}); posiciones.push_back({0,1,0}); posiciones.push_back({0.25,0.25,0}); posiciones.push_back({0.75,0.25,0}); posiciones.push_back({0.5,0.5,0}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; indices.push_back({0,1,2}); indices.push_back({2,3,0}); indices.push_back({4,5,6});</pre>	<pre>//El color por defecto es el negro para los //vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,1.0); // Asi no cogemos el centro, para hacer //las lineas, como es la funcion DrawArrays //no cogera la tabla de indices glDrawArrays(GL_LINE_LOOP, 0, 4); glVertexAttrib3f(ind_colores, 0.0,0.0,0.25); glDrawElements(GL_TRIANGLES, 2*3, GL_FLOAT, indices.data()); glVertexAttrib3f(ind_colores, 0.0,0.0,1.0); glDrawArrays(GL_LINE_LOOP, 4, 3); glVertexAttrib3f(ind_colores, 1.0,1.0,1.0); glDrawElements(GL_TRIANGLES, 3, GL_FLOAT, indices.data() + 6 * sizeof(float)); glBindVertexArray(0); }</pre>
---	---

```

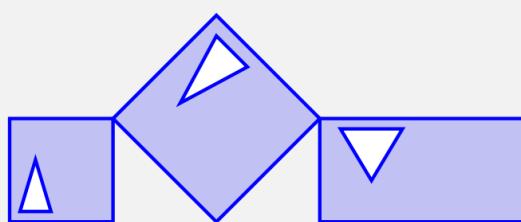
if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
        CrearVBOAtrib(ind_pos, GL_FLOAT, 3,
        posiciones.size(), posiciones.data());
    }
    if(indices.size() > 0){
        nombre_vbo_ind = CrearVBOInd(
        indices );
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO
ya creado, simplemente se activa
}

```

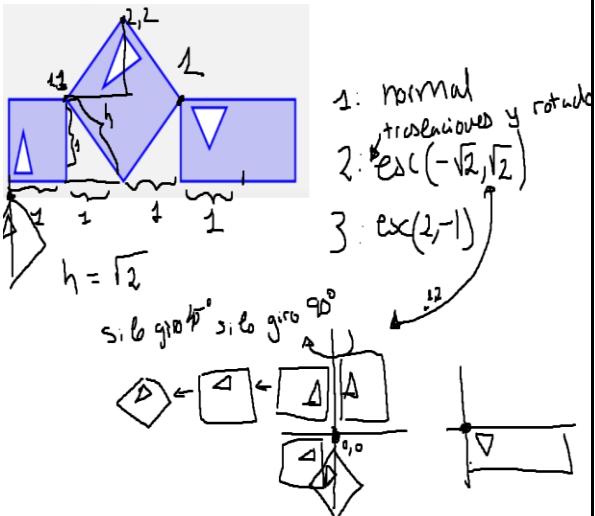
Problema 2.21.

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 (**pushMM** y **popMM**), junto con **MAT_Traslacion** y **MAT_Escalado**:



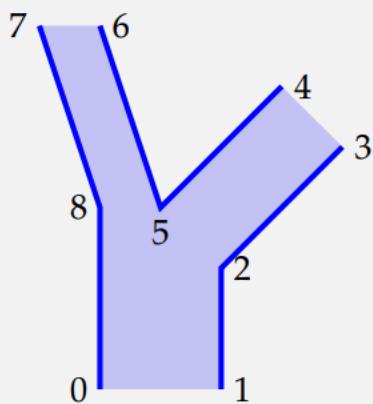
// Ejercicio 2.21

```
void FiguraCompleja(){
    figuraSimple();
    pushMM();
    compMM(MAT_Translacion({2,2,0}));
    compMM(MAT_Rotacion(135,{0,0,1}));
    compMM(MAT_Escalado(-sqrt(2),
    sqrt(2),1));
    figuraSimple();
    popMM();
    pushMM();
    compMM(MAT_Translacion({3,1,0}));
    compMM(MAT_Escalado(2,-1,1));
    figuraSimple();
    popMM();
}
```



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)

CIM - Informática Gráfica - curso 22-23 - creado el 17 de septiembre de 2022 - transparencia 167 de 197

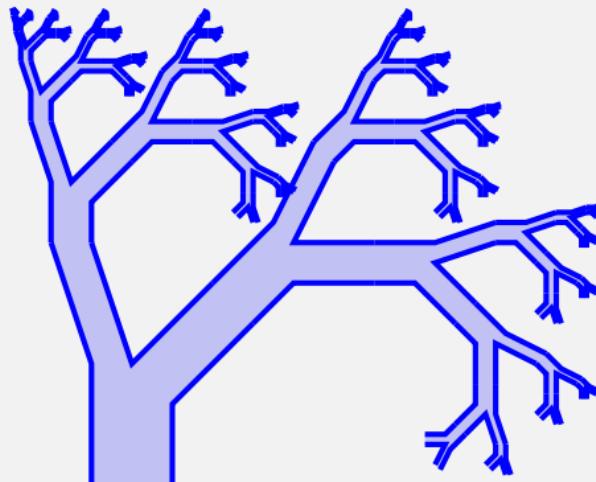
// Ejercicio 2.22

```
if(nombre_vao == 0){
```

<pre> void Tronco(){ int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla3d> posiciones; posiciones.push_back({0,0,0}); posiciones.push_back({1,0,0}); posiciones.push_back({1,1,0}); posiciones.push_back({2,2,0}); posiciones.push_back({1.5,2.5,0}); posiciones.push_back({0.5,1.5,0}); posiciones.push_back({0,0,3,0,0}); posiciones.push_back({-0.5,3,0,0}); posiciones.push_back({0,0,1.5,0}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; indices.push_back({0,8,1}); indices.push_back({8,5,2}); indices.push_back({1,2,8}); indices.push_back({7,6,8}); indices.push_back({8,6,5}); indices.push_back({4,3,2}); indices.push_back({4,5,2}); } </pre>	<pre> nombre_vao = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } //El color por defecto es el negro para los vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,1.0); // Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices glDrawArrays(GL_LINES, 1, 3); glDrawArrays(GL_LINES, 4, 3); glDrawArrays(GL_LINES, 7, 3); glVertexAttrib3f(ind_colores, 0.0,0.0,0.25); glDrawElements(GL_TRIANGLES, indices.size()*3, GL_FLOAT, indices.data()); glBindVertexArray(0); } </pre>
--	--

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)



GIM: Informática Gráfica- curso 22-23- creado el 17 de septiembre de 2022 – transparencia 168 de 184.

```
void Arbol(int n, int exp){  
    Tronco();  
    if(n!=0){  
        pushMM();  
        MAT_Translacion({-  
            0.5*pow(0.5,exp),3*pow(0.5,exp),0});  
        MAT_Escalado({0.5,0.5,0.5});  
        Arbol(n-1,exp+1);  
        popMM();  
        pushMM();  
  
        MAT_Translacion({1.5*pow(sqrt(0.5),exp),2.5  
            *pow(sqrt(0.5),exp),0});  
        MAT_Rotacion(45,{0,0,1});  
  
        MAT_Escalado(sqrt(0.5),sqrt(0.5),sqrt(0.5));  
        Arbol(n-1,exp+1);  
        popMM();  
    }  
}
```

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 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**.

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 (α y β) 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 (ϕ) 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 ángulos 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.

```
/*
ObjetoModificable::ObjetoModificable(Tupla
3f traslacion, float angulo_rotacion, Tupla3f
rotacion, Tupla3f escalado, Objeto3D &
ObjetoModificable){

    agregar(MAT_Traslacion(traslacion));
    agregar(MAT_Rotacion(angulo_rotacion,
rotacion));
    agregar(MAT_Escalado(escalado(0),
escalado(1), escalado(2)));

    agregar(& ObjetoModificable);
}
*/
AndroidCabeza::AndroidCabeza(){

    int indice = agregar(MAT_Rotacion((0.0),
{0.0, 1.0, 0.0}));

    agregar(new ObjetoModificable({1, 2.5,
0}, 135, {1,0,0}, {0.1,0.1,0.1}, * new
SemiEsfera(20,20, {0,0,0})));
    agregar(MAT_Traslacion({0, 2.2, 0}));
    agregar(MAT_Rotacion(180, {0,0,1}));


}
```

```
class AndroidCabeza : public
NodoGrafoEscena
{ public:

    Matriz4f * rotacionCabeza;
    AndroidCabeza() ; // constructor
    unsigned leerNumParametros() const;
    void actualizarEstadoParametro( const
unsigned iParam, const float tSec );
} ;

class AndroidCuerpo : public
NodoGrafoEscena
{ public:

    AndroidCuerpo() ; // constructor
} ;

class AndroidExtremidad : public
NodoGrafoEscena
{ public:

    Matriz4f * rotacionExtremidad;
    int identificador;
```

<pre> agregar(new SemiEsfera(20, 20, {0, 1, 0})); rotacionCabeza = leerPtrMatriz(indice); } unsigned AndroidCabeza::leerNumParametros() const{ return 1; } void AndroidCabeza::actualizarEstadoParametro (const unsigned iParam, const float tSec) { assert(iParam < leerNumParametros()); switch (iParam) { case 0: * rotacionCabeza = MAT_Rotacion(sin(tSec)*40, {0,1,0}); break; default: * rotacionCabeza = MAT_Rotacion(sin(tSec)*40, {0,1,0}); break; } } AndroidCuerpo::AndroidCuerpo(){ //agregar(MAT_Traslacion({0, 0, 0})); agregar(MAT_Escalado(1, 2, 1)); agregar(new Cilindro(20,20, {0, 1, 0})); } AndroidTubo::AndroidTubo(){ agregar(new ObjetoModificable({0, 3, 0}, 180, {1,0,0}, {1,1,1}, * new SemiEsfera(20,20, {0,1,0}))); agregar(new ObjetoModificable({0, 0, 0}, 0, {1,0,0}, {1,1,1}, * new SemiEsfera(20,20, {0,1,0}))); agregar(MAT_Escalado(1,3,1)); agregar(new Cilindro(20,20, {0, 1, 0})); } </pre>	<pre> class ObjetoModificable : public NodoGrafoEscena { public: ObjetoModificable(Tupla3f traslacion, float angulo_rotacion, Tupla3f rotacion, Tupla3f escalado, Objeto3D & ObjetoModificable) ; // constructor } ; AndroidExtremidad(Tupla3f traslacion, Tupla3f escalado, int id) ; // constructor unsigned leerNumParametros() const; void actualizarEstadoParametro(const unsigned iParam, const float tSec); } ; //** class AndroidTubo : public NodoGrafoEscena { public: AndroidTubo() ; // constructor } ; class NodoAndroid : public NodoGrafoEscena { public: AndroidCabeza * cabeza; AndroidCuerpo * cuerpo; AndroidExtremidad *brazo1, * brazo2, * pierna1, * pierna2; NodoAndroid() ; // constructor unsigned leerNumParametros() const; void actualizarEstadoParametro(const unsigned iParam, const float tSec); } ; NodoAndroid::actualizarEstadoParametro(const unsigned iParam, const float tSec) { assert(iParam < leerNumParametros()); switch (iParam) { case 0: </pre>
--	---

```

AndroidExtremidad::AndroidExtremidad(Tu
pla3f traslacion, Tupla3f escalado, int id){

    identificador = id;

    agregar(MAT_Traslacion(traslacion));
    agregar(MAT_Escalado(escalado[0],
    escalado[1], escalado[2]));

    agregar(MAT_Traslacion({0,2,0}));
    int indice = agregar(MAT_Rotacion((0.0),
    {0.0, 1.0, 0.0}));
    agregar(MAT_Traslacion({0,-2,0}));

    agregar(new AndroidTubo());

    rotacionExtremidad =
leerPtrMatriz(indice);
}

unsigned
AndroidExtremidad::leerNumParametros()
const{

    return 1;
}

void
AndroidExtremidad::actualizarEstadoParam
etro( const unsigned iParam, const float
tSec )
{
    assert(iParam < leerNumParametros());

    switch (iParam)
    {
        case 0:

            if(identificador == 0){
                * rotacionExtremidad =
MAT_Rotacion(sin(tSec)*40, {1,0,0});
            }
            if(identificador == 1){
                * rotacionExtremidad =
MAT_Rotacion(-1*sin(tSec)*40, {1,0,0});
            }

            break;

        default:
            * rotacionExtremidad =
MAT_Rotacion(sin(tSec)*40, {1,0,0});
    }
}

```

```

    brazo1-
>actualizarEstadoParametro(iParam, tSec);
    brazo2-
>actualizarEstadoParametro(iParam, tSec);
    cabeza-
>actualizarEstadoParametro(iParam, tSec);
    break;

    default:
        brazo1-
>actualizarEstadoParametro(iParam, tSec);
        brazo2-
>actualizarEstadoParametro(iParam, tSec);
        cabeza-
>actualizarEstadoParametro(iParam, tSec);

        break;
    }
}

```

```

        break;
    }

}

NodoAndroid::NodoAndroid(){

    cabeza = new AndroidCabeza();
    agregar(cabeza);

    cuerpo = new AndroidCuerpo();
    agregar(cuerpo);

    brazo1 = new AndroidExtremidad({-
1.4,1,0}, {0.3, 0.3, 0.3}, 0);
    brazo2 = new
AndroidExtremidad({1.4,1,0}, {0.3, 0.3, 0.3},
1);
    agregar(brazo1);
    agregar(brazo2);

    pierna1 = new AndroidExtremidad({-0.5, -0.5,0}, {0.3, 0.3, 0.3}, 2);
    pierna2 = new AndroidExtremidad({0.5, -0.5,0}, {0.3, 0.3, 0.3}, 3);
    agregar(pierna1);
    agregar(pierna2);

}

unsigned
NodoAndroid::leerNumParametros() const{

    return 1;
}

void

```

Problema 3.1.

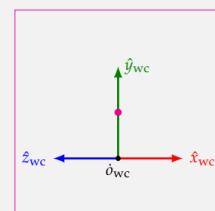
Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja, verde y azul), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

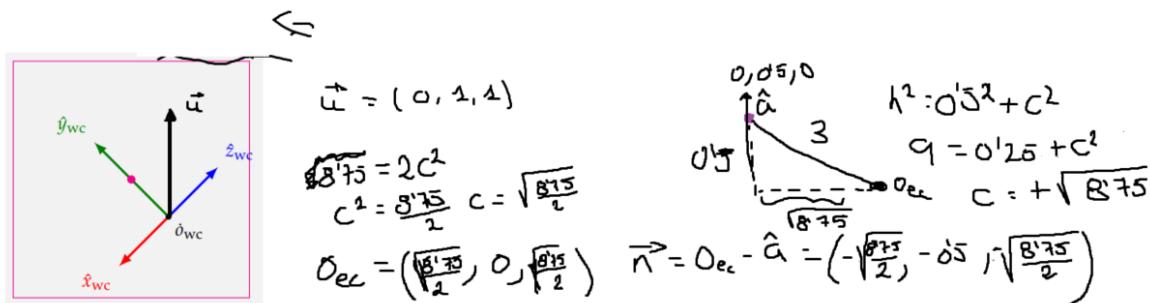
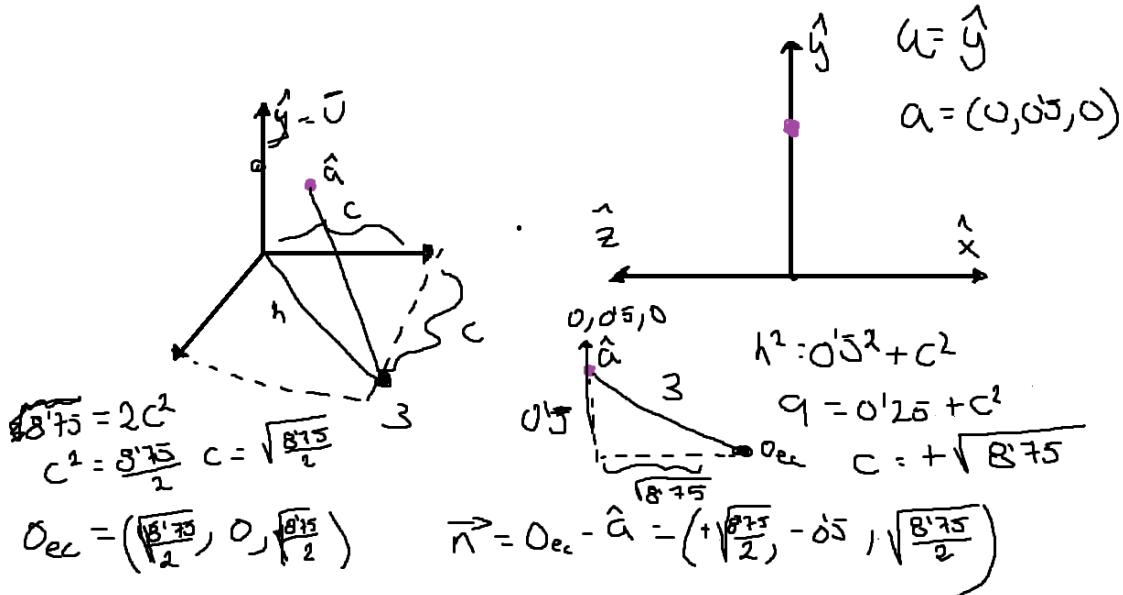
1. El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2. El punto de coordenadas $(0,0.5,0)$ (aparece como un disco de color morado en la figura) debe aparecer en el centro 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).

Problema 3.1. (continuación)

Escribe unos valores que podríamos usar para **a**, **u** y **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 3.3.

Escribe el código para calcular los vectores de coordenadas x_{ec} , y_{ec} , z_{ec} y o_{ec} que definen el marco de vista a partir de los vectores de coordenadas a , u y n (todos estos vectores de coordenadas son de tipo **Tupla3f**).

Problema 3.4.

Partiendo de los vectores de coordenadas x_{ec} , y_{ec} , z_{ec} y 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 V y luego asigna valor a $V(i, j)$ para cada fila i y columna j , ambas entre 0 y 3).

// Ejercicio 3.3

```
void CalcularMarcoCamara(Tupla3f a,
Tupla3f u, Tupla3f n){

    Tupla3f o_ec = a+n;
    Tupla3f z_ec = n.normalized();
    Tupla3f x_ec = (u.cross(n)).normalized();
```

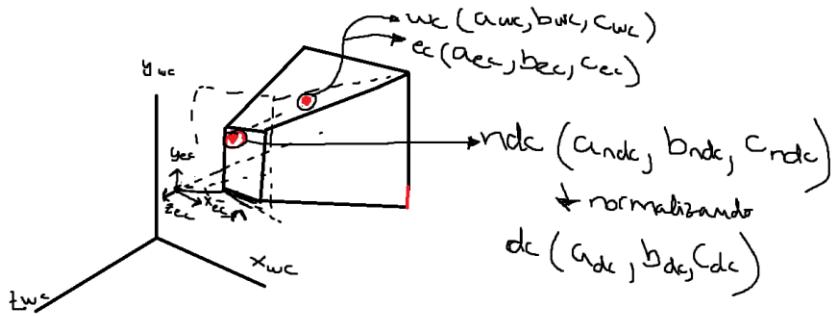
// Ejercicio 3.4

```
void CalcularMarcoCamaraXYZ(Tupla3f x_ec,
Tupla3f y_ec, Tupla3f z_ec, Tupla3f o_ec){

    Matriz4f * V;
    for(int i=0; i<2; i++){
        V[0][i] = x_ec[i];
```

```
Tupla3f y_ec =  
(z_ec.cross(x_ec)).normalized();  
}  
  
V[1][i] = y_ec[i];  
V[2][i] = z_ec[i];  
}  
V[0][3] = -x_ec.dot(o_ec);  
V[1][3] = -y_ec.dot(o_ec);  
V[2][3] = -z_ec.dot(o_ec);  
V[3][3] = 1;  
}
```

Explicación pasar a coordenadas normalizadas de proyección:



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 sitúa 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.

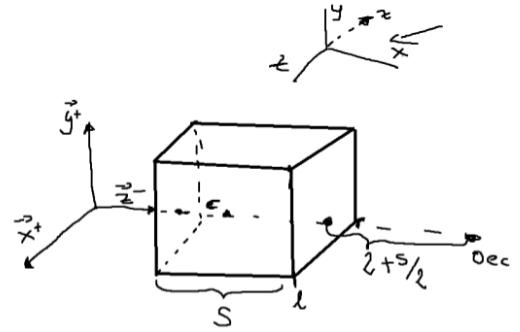
(continua en la siguiente página)

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.

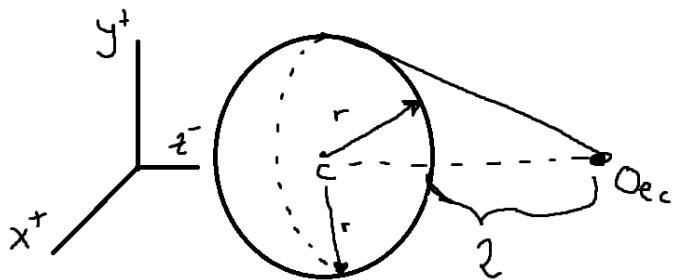
Con estos requerimientos, indica como calcular los valores l, r, t, b, n y f (para obtener la matriz Q de proyección), en función de s y (c_x, c_y, c_z) .



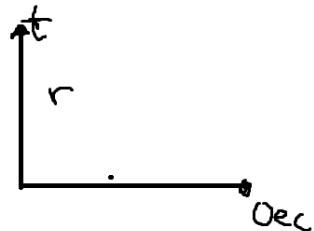
$$\begin{aligned}
 n &= 2 + \frac{s}{2} & f &= 2 + \frac{3s}{2} \\
 r &= -\frac{s}{2} & t &= \frac{s}{2} \\
 l &= +\frac{s}{2} & b &= -\frac{s}{2}
 \end{aligned}$$

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}).



$$\mathbf{o}_{es} = (c_x, c_y, c_z + 2 + r)$$



$$\begin{aligned} n &= 2 & f &= 2 + 2r \\ l &= +r & r &= -r \\ t &= +r & b &= -r \end{aligned}$$

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 $w = h$.

Este problema es parecido al del tema 1. Debemos modificar t/b o r/e según el ratio del *viewport*

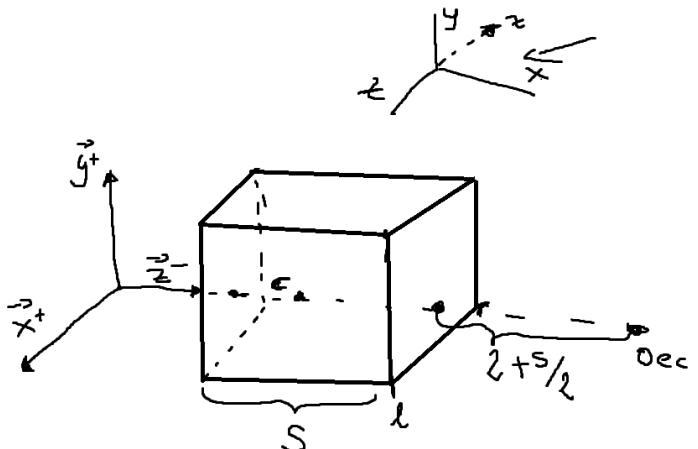
Si $w/h > 1$

$$r = r \cdot \frac{w}{h} \quad e = e \cdot \frac{w}{h}$$

Si $w/h < 1$

$$t = t \cdot \frac{h}{w} \quad b = b \cdot \frac{h}{w}$$

Así no deformaremos los objetos.

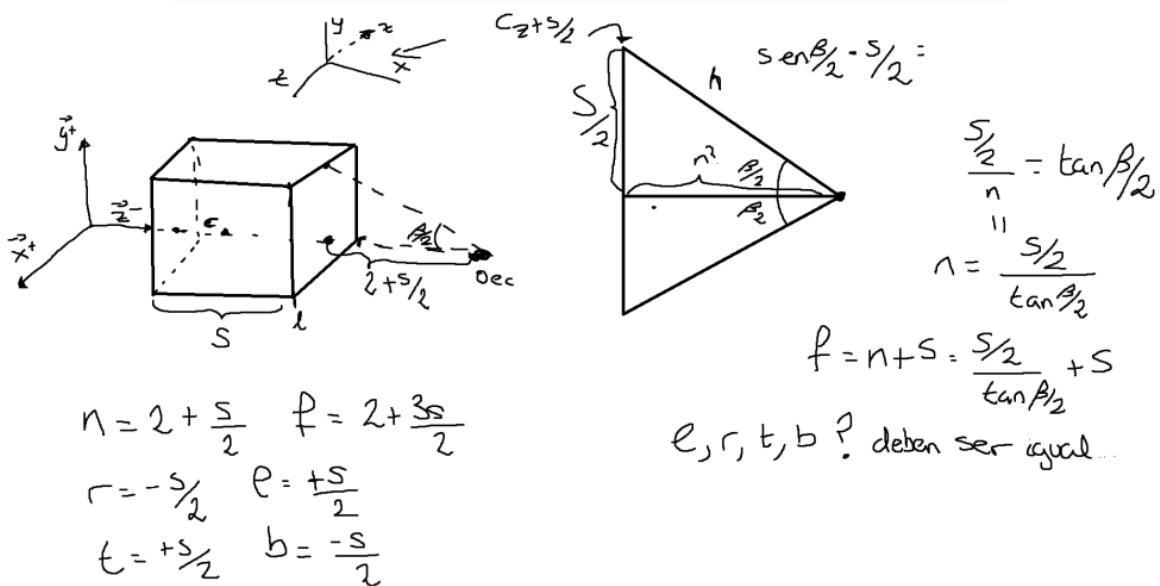


$$\begin{aligned} n &= 2 + \frac{s}{2} & f &= 2 + \frac{3s}{2} \\ r &= -\frac{s}{2} & e &= \frac{s}{2} \\ t &= +\frac{s}{2} & b &= -\frac{s}{2} \end{aligned}$$

Problema 3.8.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo β 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 β .

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 β , s y $\mathbf{c} = (c_x, c_y, c_z)$).



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 ($k_d = 1$ en todos los puntos, y k_a y k_s 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.

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 $p = (0, 2, 0)$. El observador está situado en $\mathbf{o} = (2, 0, 0)$. En estas condiciones:

- Describe razonadamente en qué punto de la superficie de la esfera el brillo será máximo si el material es puramente difuso ($k_d = 1$ en todos los puntos, y k_a y k_s 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.

• material puramente difuso.

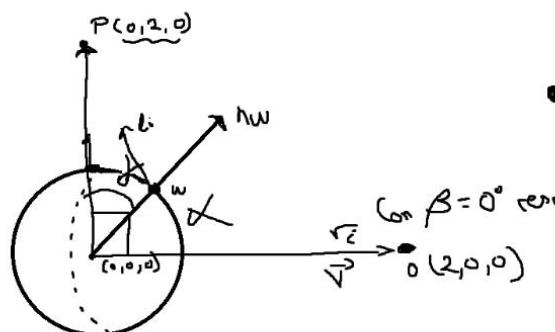
El máximo es cuando $n_p \cdot \mathbf{l}_i = \mathbf{n}_p$

$$\therefore \alpha = 0$$

Por tanto, el punto debe ser el $(0, 2, 0)$

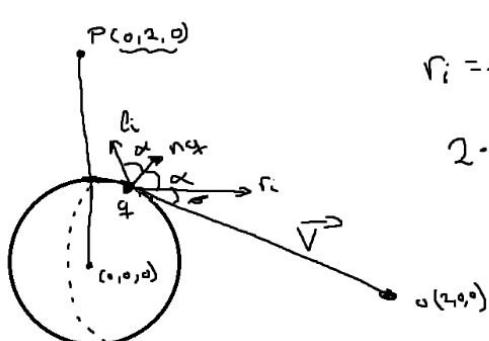
¿Es visible para el observador? No, porque la esfera no es el \mathbf{p}' es un anterior de la \mathbf{t}_g en el \mathbf{p} es paralela a \mathbf{o} .

(Para calcular de una esfera centrada en $(0, 0, 0)$ su punto de máximo brillo difuso respecto a una luminaria en $\mathbf{l}(l_x, l_y, l_z)$ se tendría que calcular $\frac{\mathbf{l} - (0, 0, 0)}{\|\mathbf{l} - (0, 0, 0)\|}$)



• Si n_p y \mathbf{l}_i coinciden, el brillo es máximo

con $\beta = 0^\circ$ respecto a \mathbf{v}



$$r_i = l_i (\mathbf{n}_p \cdot \mathbf{n}_p) \mathbf{n}_p - \mathbf{l}_i = \mathbf{v}$$

$$2 \cdot ((0, 2, 0) - \mathbf{p})(\mathbf{p} - (0, 0, 0)) \cdot ((0, 2, 0) - \mathbf{p}) = \\ = (2, 0, 0) - \mathbf{p}$$

$$h_i = \frac{(\mathbf{p}_{\text{fuente}} - \text{centro}) + (0 - \text{centro})}{\|\text{mod del vector}\|}$$

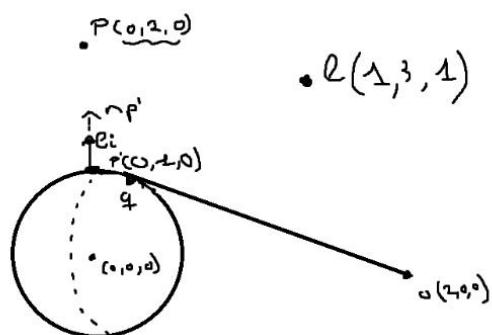
$$h_i = \frac{0, 2, 0 + 2, 0, 0}{\|(2, 2, 0)\|} = \frac{(2, 2, 0)}{\sqrt{2^2 + 2^2}} = \frac{(2, 2, 0)}{\sqrt{8}} = \frac{(2, 2, 0)}{2\sqrt{2}}$$

$$h_i = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0 \right) = \mathbf{p} \text{ (porque el centro es el } 0, 0, 0\text{)}$$

$$+ \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right) = \mathbf{p}$$

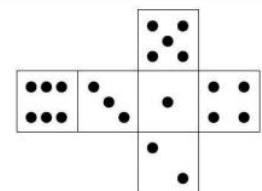
Aplicar el cálculo de h_i
suponiendo que el punto es el centro

Si \mathbf{l}_i sera visible. $\exists \alpha$: tg en la esfera de la recta desde el \mathbf{o} con vértice de \mathbf{p}



Problema 3.10.

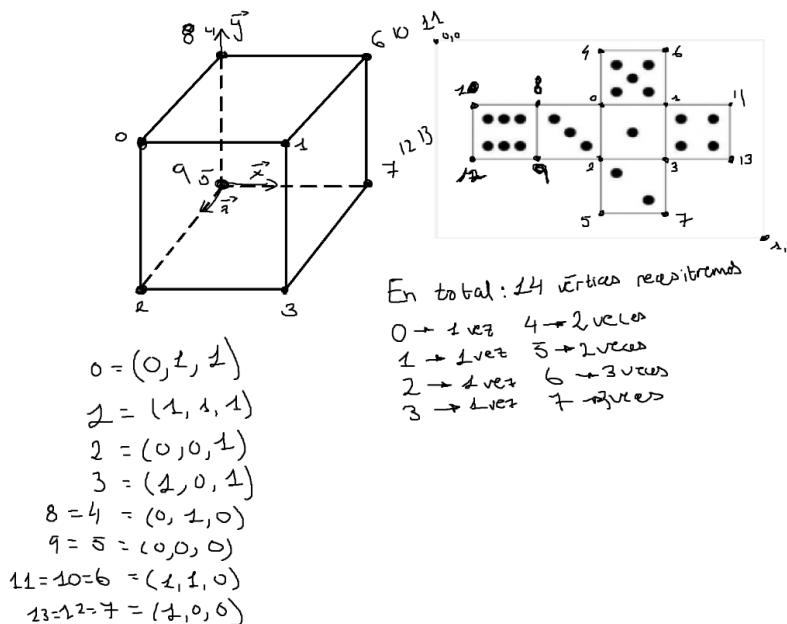
Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar un texture que incluya las caras de un dado. Para ello disponemos de una imagen de texture que tiene una relación de aspecto 4:3. La imagen aparece aquí:



Problema 3.10. (continuación)

Responde a estas cuestiones:

- Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
- Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de texture, 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 texture en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de texture.



Triángulos:
 $\{(0,6,4), (0,1,6), (0,2,1), (2,3,1)$
 $(1,11,3), (3,14,13), (2,7,5),$
 $(3,7,2), (2,8,9), (2,0,8)\}$
 $(4,8,10), (4,10,12)\}$

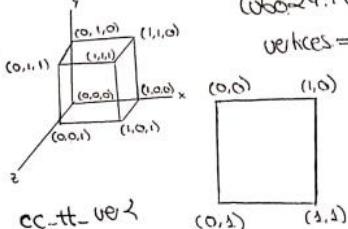
Cord - text =
 $(0.5, 0.33) = 0 \quad (0.75, 0.33) = 7$
 $(0.75, 0.33) = 1 \quad (0.25, 0.33) = 8$
 $(0.25, 0.66) = 2 \quad (0.25, 0.66) = 9$
 $(0.75, 0.66) = 3 \quad (0, 0.33) = 10$
 $(0, 0.33) = 4 \quad (1, 0.33) = 11$
 $(0.5, 0) = 5 \quad (0, 0.66) = 12$
 $(0.75, 0) = 6 \quad (1, 0.66) = 13$

Problema 3.11.

Considera de nuevo el cubo y la texture 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 texture 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 texture. Asimismo, escribe como sería la tabla de normales.

(45) (Cube 24)



cc_tt_ver2

$\{2,0,1\} // 0$

$\{1,1,1\} // 1$

$\{0,0,1\} // 2$

$\{1,0,0\} // 3$

$\{1,1,0\} // 4$

$\{0,1,0\} // 5$

$\{0,1,1\} // 6$

$\{0,0,0\} // 7$

$\{1,0,1\} // 8$

$\{0,0,1\} // 9$

$\{0,0,0\} // 10$

$\{1,0,0\} // 11$

$\{0,1,0\} // 12$

$\{1,1,0\} // 13$

$\{1,1,1\} // 14$

$\{1,0,1\} // 15$

$\{0,1,1\} // 16$

$\{0,0,1\} // 17$

$\{1,0,0\} // 18$

$\{0,0,0\} // 19$

$\{0,1,0\} // 20$

$\{1,1,0\} // 21$

$\{0,0,1\} // 22$

$\{1,0,1\} // 23$

8;

Cube24::Cube24() : MallaInd ("Cube 24") {

vertices = { 2,0,0,0 // 0

1,0,0,1 // 1

1,0,1,0 // 2

1,0,1,1 // 3

1,1,0,0 // 4

1,1,0,1 // 5

1,1,1,0 // 6

1,1,1,1 // 7

1,0,0,0 // 8

1,0,0,1 // 9

1,0,1,0 // 10

1,0,1,1 // 11

1,1,0,0 // 12

1,1,0,1 // 13

1,1,1,0 // 14

1,1,1,1 // 15

2,0,0,0 // 16

2,0,0,1 // 17

2,0,1,0 // 18

2,0,1,1 // 19

2,1,0,0 // 20

2,1,0,1 // 21

2,1,1,0 // 22

2,1,1,1 // 23

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.

```
void Camara3Modos::Problema41(float t){  
    //Se desplaza con un movimiento uniforme, por tanto el o_ec (oe_x + a*t, oe_y + b*t, oe_z + c*t)  
    //El punto de atención también cambia de la misma forma a = (a_x + q*t, a_y + r*t, a_z + l*t)  
    //Por tanto para cada segundo t, tendremos un punto de atención, un origen y el vector VUP  
    //que nos dicen que será siempre (0,1,0). Con esta información podemos obtener fácilmente  
    //x_ec, y_ec, y z_ec con los que sabemos calcular la nueva matriz de proyección (hecho en un  
    problema anterior)  
    Tupla3f o_ec = {oe_x + a*t, oe_y + b*t, oe_z + c*t};  
    Tupla3f a = {a_x + q*t, a_y + r*t, a_z + l*t};  
    Tupla3f n = o_ec - a;  
    Tupla3f z_ec = n.normalized();  
    Tupla3f x_ec = (u.cross(n)).normalized();  
    Tupla3f y_ec = (z_ec.cross(x_ec)).normalized();  
  
    //Sabiendo x_ec, y_ec y z_ec podemos construir nuestra nueva matriz de vista para cada  
    segundo t  
    Matriz4f * V;  
  
    for(int i=0; i<2; i++){  
        V[0][i] = x_ec[i];  
        V[1][i] = y_ec[i];  
        V[2][i] = z_ec[i];  
    }  
  
    V[0][3] = -x_ec.dot(o_ec);  
    V[1][3] = -y_ec.dot(o_ec);  
    V[2][3] = -z_ec.dot(o_ec);  
    V[3][3] = 1;  
  
    cauce.fijarMatrizVista(V);  
    //Alternativamente el método a emplear sería este:  
    glUniformMatrix4fv( loc_mat_vista, 1, GL_FALSE, *V );  
}
```

$$\begin{aligned}
 o_x + d_x \cdot t &= V_{12}^x \cdot a + V_{2z}^x \cdot b \\
 o_y + d_y \cdot t &= V_{12}^y \cdot a + V_{2z}^y \cdot b \\
 o_z + d_z \cdot t &= V_{12}^z \cdot a + V_{2z}^z \cdot b
 \end{aligned}
 \quad \left\{
 \begin{aligned}
 o_y + \frac{dy \cdot V_{12}^x \cdot a + dy \cdot V_{2z}^x \cdot b + dy \cdot o_x}{dx} &= V_{12}^y \cdot a + V_{2z}^y \cdot b \\
 o_z + \frac{dz \cdot V_{12}^x \cdot a + dz \cdot V_{2z}^x \cdot b + dz \cdot o_x}{dx} &= V_{12}^z \cdot a + V_{2z}^z \cdot b
 \end{aligned}
 \right.$$

$$t = \frac{V_{12}^x \cdot a + V_{2z}^x \cdot b + o_x}{dx}$$

$$dy \cdot V_{12}^x \cdot a + dy \cdot V_{2z}^x \cdot b + dz \cdot o_x = V_{12}^y \cdot a \cdot dx + V_{2z}^y \cdot b \cdot dx + o_y \cdot dx$$

$$o_x = \frac{V_{12}^y \cdot b \cdot dx + o_y \cdot dx - dy \cdot V_{2z}^x \cdot b - dy \cdot o_x}{dy \cdot V_{12}^x - V_{12}^y \cdot dx}$$

* Despejamos b en la segunda:

$$dz \cdot V_{12}^x \cdot a + dz \cdot V_{2z}^x \cdot b + dz \cdot o_x = o_x \cdot o_x + dz \cdot V_{12}^z \cdot a + dz \cdot V_{2z}^z \cdot b$$

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 \mathbf{o} , y como vector de dirección la tupla \mathbf{d} (la suponemos normalizada).
- ▶ Las coordenadas del mundo de los vértices del triángulo son $\mathbf{v}_0, \mathbf{v}_1$ y \mathbf{v}_2 .
- ▶ 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).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 65 de 81

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 $c \equiv 1 - a - b$ se les llama *coordenadas baricéntricas* de \mathbf{p}_t en el triángulo, se usan en ray-tracing).

//Ejercicio 4.2

```

void calcularInterseccion(){
    Tupla3f o, d, v1, v2, v0;
    //o+t*d= (v1-v2)*a + (v0-v2)*b tal que 0<=a+b<=1
    /*
    Habría que resolver el siguiente sistema de ecuaciones (3 incógnitas y 3 ecuaciones, además tenemos en cuenta
    la restricción 0<=a+b<=1):
    ox+dx*t = (v1-v2)_x*a + (v0-v2)_x*b
    oy+dy*t = (v1-v2)_y*a + (v0-v2)_y*b
    oz+dz*t = (v1-v2)_z*a + (v0-v2)_z*b
    */
    if(d.dot((v1-v2).cross((v0-v2)))!=0){

        // Resuelvo el sistema y saco a,b y t
        t = blabla;
        a = blabla;
        b = blabla;

        if(0<=a+b<=1 && t>0)
            return true;
    }
}

```

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 $\mathbf{x}_{ec}, \mathbf{y}_{ec}$ y \mathbf{o}_{ec} con los versores y la tupla $\mathbf{\omega}_{ec}$ 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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 67 de 81

// Ejercicio 4.3

```
void calcularRayo(int x_dc, int y_dc, int w, int h, float t, float b, float n, float f, float r, float l){  
    // Asumimos que xl = yb = 0 ya que solo afectan al posicionamiento de la ventana  
    // respecto a la pantalla al completo  
    // int xl=0, yb = 0;  
    // float x_ndc = (2*(x_dc-xl))/w, y_ndc= (2*(y_dc-yb))/h, z_ndc = 0;  
  
    // float x_cc = x_ndc * (r-l), y_cc = y_ndc * (t-b);  
    // Calculamos la matriz Q con t,b,r,l,n,f  
    // Matriz4f Q = MAT_Perspectiva(l,r,b,t,n,f);  
    // Tupla3f t = {x_cc,y_cc,0};  
    // Tupla3f t_ec = MAT_Inversa(Q)*t;  
    // Para construir la matriz V debemos tener en cuenta los vectores x_ec, y_ec, z_ec y  
    o_ec que nos dan  
    // este fue un ejercicio ya realizado anteriormente.  
    // Tupla3f t_wc = MAT_Inversa(V)*t_ec;  
  
    // El punto o será el o_ec en coordenadas de mundo (que es como nos la dan), y el  
    vector d es el resultante  
    // de hacer t_wc - o_ec  
}
```

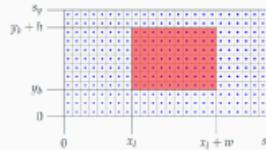
$$(x_{dc}, y_{dc}, z_{dc}, 1)^T = D(x_{ndc}, y_{ndc}, z_{ndc}, 1)^T$$

$$(x_{dc}, y_{dc}, z_{dc})^T \cdot (x_{ndc}, y_{ndc}, z_{ndc})^{-1} = D$$

D

$$D = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x_f + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y_b + \frac{h}{2} \\ 0 & 0 & \frac{z_f - z_{ndc}}{2} & \frac{z_f + z_{ndc}}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Suponemos que la ventana tiene s_x columnas y s_y filas, y que el gestor de ventanas acepta coordenadas de píxeles enteras no negativas.



$$\text{se deben cumplir estas desigualdades: } \begin{cases} 0 \leq x_l < x_f + w \leq s_x \\ 0 \leq y_b < y_b + h \leq s_y \end{cases}$$

Vamos a usarlo
 $x_l = y_b = 0$ ya que no interfiere
 $\Rightarrow z_{ndc} = \emptyset$ (supone que grande la prof)

$$x_{dc} - x_l = \frac{(x_{ndc} + 1)w}{2}$$

$$\frac{2(x_{dc} - x_l) - l}{w} = x_{ndc}$$

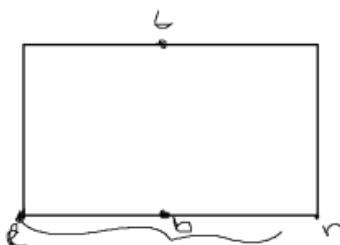
$$\frac{2(y_{dc} - y_b)}{w} = y_{ndc}$$

$$(x_{ndc}, y_{ndc}, 1) = Q \begin{pmatrix} x_{ec} \\ y_{ec} \\ z_{ec} \\ 1 \end{pmatrix}$$

$$x_{ec} = (f - b) \cdot x_{ndc}$$

$$y_{ec} = (r - l) \cdot y_{ndc}$$

Teniendo f, r, n, f, b, l podemos obtener Q



Teniendo $Q, (x_{ec}, y_{ec}, 0)$ y

$$\text{Sabiendo } (x_{ec}, y_{ec}, 0) = Q (x_{ndc}, y_{ndc}, z_{ndc})$$

$$Q^{-1}(x_{ec}, y_{ec}, 0) = (x_{ndc}, y_{ndc}, z_{ndc})$$

Sabiendo las eje coordinadas

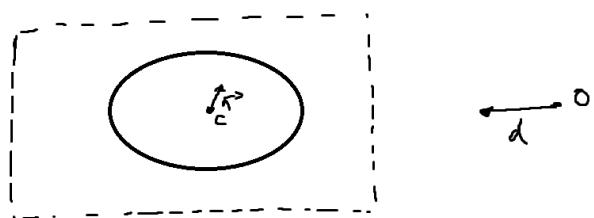
podemos obtener los $x_{ndc}, y_{ndc}, z_{ndc}$
que corresponden ya sea de tener

$x_{ec}, y_{ec}, 0$ transformarlos en matriz
de vista y (no olvidar que
 $z_{ndc} = x_{ec} \cdot y_{ec}$)

$$\text{Al hacer } V^{-1}(x_{ec}, y_{ec}, 0) = (x_{ndc}, y_{ndc}, z_{ndc})$$

Ese es el punto al que proyectamos y
a su vez el vector que va desde del a ese punto
es el \vec{x}

5.1.



Hay que calcular intersección del plano dado por c y \vec{n}
y el rayo \vec{d} y o ($o + t \cdot \vec{d}$)

Ver si ese punto l está a una distancia
menor que r de c ($\|l - c\| \leq r$)
(Siempre y cuando $t \geq 0$)

*Cómo calcular el punto del plano

Supongamos un punto l del plano, el vector
 $\vec{l} - \vec{c}$ debe ser perpendicular a \vec{n}

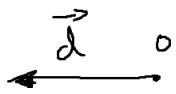
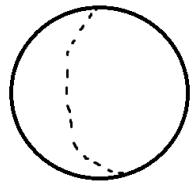
$(l - c) \cdot \vec{n} = 0$ Si ese punto también lo podemos
calcular como $l = o + t \cdot \vec{d}$

$$(o + t \cdot \vec{d}) - c \cdot \vec{n} = 0$$

$$(o_x + t \cdot d_x - c_x) \cdot n_x + (o_y + t \cdot d_y - c_y) \cdot n_y + (o_z + t \cdot d_z - c_z) \cdot n_z = 0$$

Sólo t es incógnita.

5.2.



$$\| \vec{p} \| = 1$$

$$\| \vec{p} - \vec{r} \| \leq \| \vec{p} \|$$

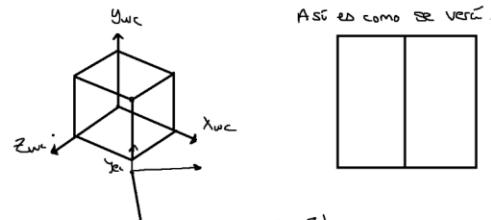
$$P = t \cdot \vec{d} + o \quad (t \cdot \vec{d} + o) \cdot (t \cdot \vec{d} + o) - 1 = 0$$

$$(td_x + o_x)(td_x + o_x) + (td_y + o_y)(td_y + o_y) + (td_z + o_z)(td_z + o_z) - 1 = 0$$

Calcular t y debe ser $t > 0$.

- B) Dadas las siguientes configuraciones, dibujar como se vería un cubo unitario (vértices en $(0,0,0)$ y en $(1,1,1)$). Se supone que hay un plano de recorte delantero (PLANO FRONTAL), que coincide con el plano de proyección. El PRP está en el eje z a una distancia d. Mostrar también los ejes. El sistema es coordenado derecho.

VRP	VPN	VUP
$(0,0,2)$	$(1,0,1)$	$(0,1,0)$
$(0,10,0)$	$(0,2,0)$	$(0,0,-1)$
$(-1,-1,-1)$	$(-1,-1,-1)$	$(0,1,0)$
$(-1,0,-1)$	$(-1,0,-1)$	$(1,1,0)$
$(2,0,0)$	$(-1,0,-1)$	$(0,-1,0)$



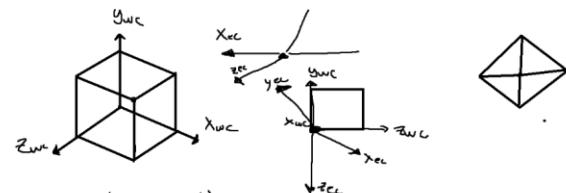
$$VUPXVPN = x_{ec} = \begin{pmatrix} x & y & z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$\Rightarrow x + 0y + 0z - 1z = 1x - 1z$$

$$VPN \times x_{ec} = \begin{pmatrix} x & y & z \\ 0 & 0 & 1 \\ 1 & 0 & -1 \end{pmatrix} =$$

$$= 0x + 1y + 0z - 0z + 1y - 0x =$$

$$2y$$



$$z_{ec} = (-1, -1, 1)$$

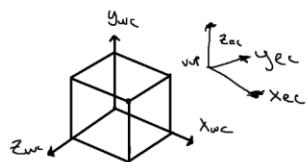
$$VPN \begin{vmatrix} x & y & z \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{vmatrix} = -x + 0y + 0z + z - 0y + 0x =$$

$$= -x + z = (-1, 0, 1)$$

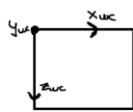
$$VPN \begin{vmatrix} x & y & z \\ -1 & -1 & 1 \\ -1 & 0 & 1 \end{vmatrix} = -x + y + 0z - z + y_1 0x$$

$$= -x + 2y - z$$

$$y_{ec} = (-1, 2, -1)$$



$$VPN \begin{vmatrix} x & y & z \\ 0 & 0 & -1 \\ 0 & 2 & 0 \end{vmatrix} = 0x + 0y + 0z - 0z - 0y + 2x$$



|| (0,0,0) | (-1,0,-1) | (0,-1,0) ||

C) Implementar en seudocódigo la rotación con respecto a un eje arbitrario, definido por dos puntos.

```
Tupla4f rotar(float grados, Tupla4f e, Tupla4f punto){
    Matriz4f *R[4][4];
    float s = sin(grados*M_PI/180), c = cos(grados*M_PI/180);
    float a[3][3];
    for(int i=0; i < 3; i++){
        for(int j=0; j < 3; j++){
            a[i][j] = (1-c)*e[i]*e[j];
        }
    }
}
```

```

    }

R[0][0] = a[0][0]+c; R[0][1] = a[0][1]-s*e[2]; R[0][2] = a[0][2]+s*e[1]; R[0][3] = 0;
R[1][0] = a[1][0]+s*e[2]; R[1][1] = a[1][1]+c; R[1][2] = a[1][2]-s*e[0]; R[1][3] = 0;
R[2][0] = a[2][0]-s*e[1]; R[2][1] = a[2][1]+s*e[0]; R[2][2] = a[2][2]-c; R[2][3] = 0;
for(int i=0; i < 3 ; i++)
    R[3][i] = 0;
R[3][3] = 1;

return R*punto;
}

```

Diferencia entre geometria y topologia

Geometria se basa en la posicion de los vertices.

Topologia se basa en la relación entre aristas y triangulos.

Geometria almacena vertices

Topologia almacena triangulos y aristas.

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).

¿Por qué no varía el objeto al ensanchar la pantalla?

Cambiamos la matriz de proyección y deformando la imagen, si no la cambiaríamos se ensancharían las dos cosas y se distorsionaría todo

- K) Al cambiar la ventana de una aplicación su tamaño pasa a ser AnchoxAlto. Dada la distancia d1 al plano delantero y d2 al plano trasero, indicar como quedaría la llamada a glFrustum ()

La función **MAT_Perspectiva** permite generar una matriz de proyección perspectiva de forma más fácil en ocasiones:

```
MAT_Perspectiva( GLdouble β, GLdouble a, // calcula Q a partir de β,a,n,f
GLdouble n, GLdouble f ) ;
```

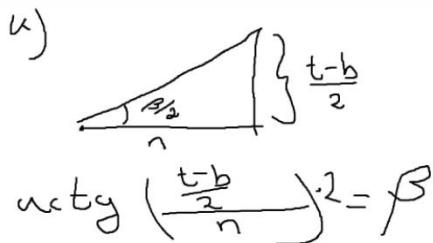
esta función equivale a **MAT_Frustum** centrado con:

$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

donde:

β ≡ 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 ≡ relación de aspecto (aspect ratio) de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).



- L) ¿Qué diferencia hay entre modo inmediato y modo retenido en OpenGL.

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

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

M) Se desea obtener una relación 1:1 en un espacio 2D sin deformación en OpenGL. ¿Cuales deberían ser los valores de `glOrtho`?

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.

L y R, t y b deben estar relacionados de manera de que uno sea el negativo del otro.

O) Hemos importado un modelo PLY y queremos comprobar que es cerrado. Escribir el seudocódigo (indicar las estructuras de datos que se usan)
Tienes que comprobar que no existe ninguna arista que solo está anexa a un triángulo.

- R) Calcular la matriz de transformación resultante de aplicar las siguientes instrucciones OpenGL:

```
glLoadIdentity();
glRotatef(30,0,1,0);
glScalef(1.0,0.5,2.0);
glTranslatef(30,10,20);
```

$$\text{Esc} [s_x, s_y, s_z] = \quad \text{Tra} [d_x, d_y, d_z] = \quad \text{Ciz}_{xy}[a] =$$

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rot}_x[\alpha] = \quad \text{Rot}_y[\alpha] = \quad \text{Rot}_z[\alpha] =$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Primero multiplicas por la de rotacion, luego la de escalado a la derecha y luego la de translación a la derecha.

- S) Indicar que hace cada una de estas primitivas de OpenGL. Poner un ejemplo dibujado, numerando los vértices. (GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_POLYGON, GL_QUADS, GL_QUAD_STRIP, GL_TRIANGLES, GL_TRIANGLE, GL_TRIANGLE_FAN)

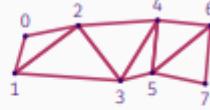
Puntos GL_POINTS	Segmentos GL_LINES	Polilínea abierta GL_LINE_STRIP	Polilínea cerrada GL_LINE_LOOP

Tira de triángulos

GL_TRIANGLE_STRIP

Polygonos:

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

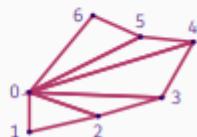


Abanico de triángulos

GL_TRIANGLE_FAN

Polygonos:

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

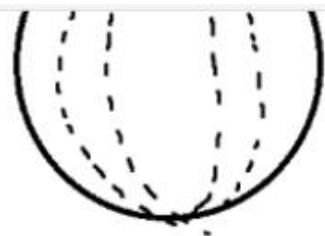


Polígonos de más de tres vértices

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

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

W) Se han generado los puntos que representan a una esfera (m puntos para la curva generatriz, n divisiones). Se han guardado en un vector, por meridianos. Implementar un procedimiento que almacene las caras.



$V = [v_0, v_1, \dots, v_m, v_{m+1}, \dots, v_{n+m}]$

$\underbrace{v_0, v_1, \dots, v_m}_{\text{1er merid}}$ en totale ring
 n meridianen

E-riangles $\{(v_{k+i}, v_{k+j}, v_{(k+1) \cdot j + i})\}$

Método Inicializar MallaRevol:

```

void MallaRevol::inicializar
(
    const std::vector<Tupla3f> & perfil, // tabla de vértices del perfil original
    const unsigned num_copias // número de copias del perfil
)
{
    // COMPLETAR: Práctica 2: completar: creación de la malla.....

    float x,y;
    vertices.clear();
    for(int i=0; i <= num_copias-1; i++){
        for(int j=0; j <= perfil.size()-1; j++){
            Tupla3f p = perfil.at(j);
            Tupla3f q;

            q =
MAT_Rotacion(360*i/(num_copias-1),
{0.0,1.0, 0.0})*p;
            vertices.push_back(q);
        }
    }

    // Añadimos los triangulos
    triangulos.clear();
    float k;
    for(int i = 0; i <= num_copias - 2; i++){
        for(int j=0; j <= perfil.size() - 2; j++){
            k = i*perfil.size() + j;
            triangulos.push_back({k,
k+perfil.size(), k+perfil.size()+1});
            triangulos.push_back({k,
k+perfil.size()+1, k+1});

        }
    }
}

```

```

// Practica 4
//-----// COMPLETAR: Práctica 4: cálculo normales y coordenadas de textura
///////////////////////////////
// Calculamos las normales de las aristas
std::vector<Tupla3f> normales_m;
for (unsigned int i = 0; i < perfil.size()-1; i++){
    float v_1 = (perfil[i+1] - perfil[i])(0);
    float v_2 = (perfil[i+1] - perfil[i])(1);
    Tupla3f m_i({v_2, -v_1, 0.0}); // giro de un vector -90°
    if (m_i.lengthSq() != 0) // por si hubiera dos puntos seguidos iguales
        m_i = m_i.normalized();
    normales_m.push_back(m_i);
}

// Calculamos las normales de los vertices
std::vector<Tupla3f> normales_n;
normales_n.push_back(normales_m[0]);
for (unsigned int i = 1; i < perfil.size()-1; i++){
    normales_n.push_back( (normales_m[i-1] + normales_m[i]).normalized());
    normales_n.push_back(
normales_m[perfil.size()-2] );

    // Calculamos los vectores d y t, junto con sumas_parciales (vector auxiliar)
    std::vector<float> d;
    std::vector<float> t;
    std::vector<float> sumas_parciales;
    float suma_total;

    for (unsigned int i = 0; i < perfil.size()-1; i++)
        d.push_back( sqrt( (perfil[i+1] - perfil[i]).lengthSq() ) );

```

```

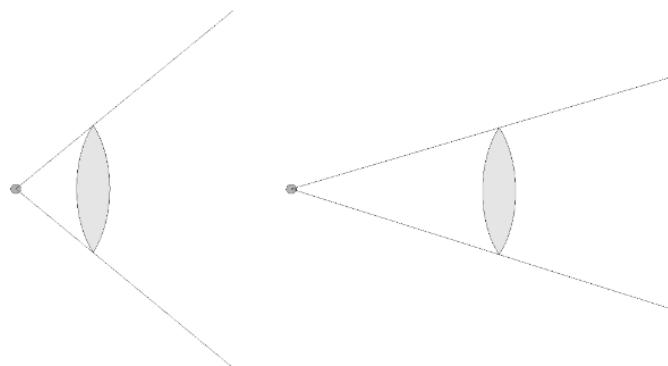
sumas_parciales.push_back(0.0);
for (unsigned int i = 1; i < perfil.size(); i++)

sumas_parciales.push_back(sumas_parciales[i-1] + d[i-1]);

suma_total =
sumas_parciales[perfil.size()-1];
t.push_back(0.0);
for (unsigned int i = 1; i < perfil.size(); i++)
    t.push_back( sumas_parciales[i] /
suma_total );
///////////////////////////////
for(int i=0; i <= num_copias-1; i++){
    for(int j=0; j <= perfil.size()-1; j++){
        nor_ver.push_back(
MAT_Rotacion(2.0*180.0*i / (num_copias-1), {0.0, 1.0, 0.0}) * normales_n[j] );
        cc_tt_ver.push_back({float(i) /
(num_copias-1), 1-t[j]}); // i y j están
cambiados
    }
}
}

```

3. Las lentes de las cámaras con zoom permiten cambiar la zona visible, desde ángulos más grandes (gran angular) a más pequeños (tele). ¿Cómo se podría conseguir el mismo efecto con los parámetros del glFrustum? Explicarlo y poner ejemplos de valores. (2)



Tenemos los parámetros l,r,b,t,n,f:

La función **MAT_Perspectiva** permite generar una matriz de proyección perspectiva de forma más fácil en ocasiones:

```
MAT_Perspectiva( GLdouble β, GLdouble a, // calcula Q a partir de β,a,n,f
                  GLdouble n, GLdouble f ) ;
```

esta función equivale a **MAT_Frustum** centrado con:

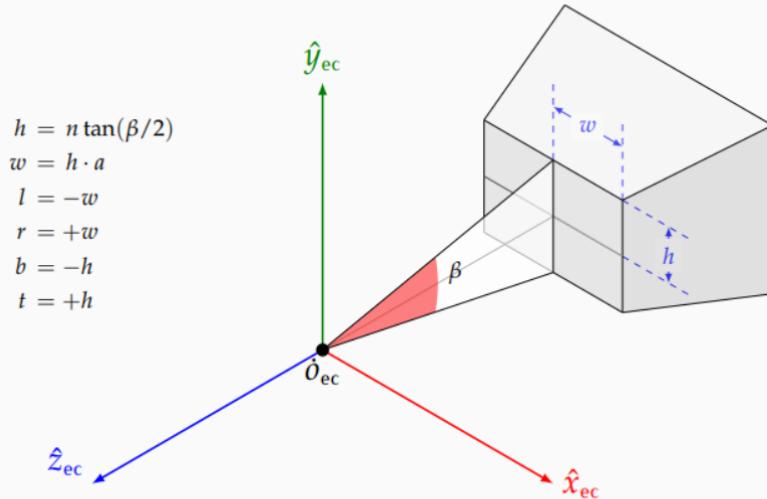
$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

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

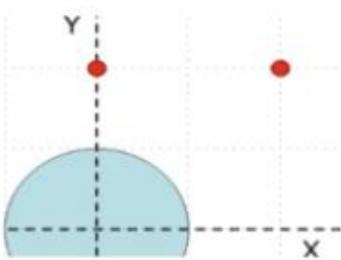
$a \equiv$ **relación de aspecto (aspect ratio)** de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).

El significado de los parámetros se aprecia en esta figura:



Esta perspectiva es *centrada*, ya que $r = -l$ y $t = -b$

- b) Supongamos que tenemos una esfera difusa de material blanco de radio unidad centrada en el origen, una fuente de luz puntual en $(0, 2, 0)$ de color azul $(0, 0, 1)$ y otra fuente de luz puntual en $(2, 2, 0)$ de color rojo $(1, 0, 0)$. ¿Dónde tendremos el máximo valor de iluminación difusa en la esfera para las dos luces? ¿En qué zona de la esfera se producirá una mezcla de color tal que las intensidades difusas producidas por las dos luces sean semejantes. (2)



Ej 3. 2019

$$\frac{n_p \cdot l_b}{P} = \frac{n_{\bar{p}} \cdot l_r}{P}$$

$0,2,0-p$ $2,2,0-p$

$$P \cdot (-P_x, 2-P_y, -P_z) = P \cdot (2-P_x, 2-P_y, -P_z)$$

$$-P_x^2 + 2P_y - P_y^2 - P_z^2 = 2P_x - P_x^2 + 3P_y - 2P_y^2 - P_z^2$$

$$-2p_x + p_y^2 = 0$$

$$p_y^2 = 2p_x \Rightarrow p_y = \sqrt{2p_x}$$

$$P_x^2 + P_y^2 = 1$$

$$P_x^2 + 2P_x - 1 = 0$$

$$P_y = \sqrt{2(-1 + \sqrt{2})} = \underline{\underline{10.9101}}$$

Colapsar Aristas para minimización de caras

```

void colapsarArista(){
    //Tenemos el punto inicial pi y final pf de la arista y la lista de triangulos
    v = pf-pi;
    nuevo_punto = pi + v/2;
    for(todos los triangulos)
        IF triangulo contiene o el punto final o el inicial, lo quitamos y lo formamos con el nuevo
        punto
    }

void calcularVerticesMasRepetidosConAristaCompartida(){
    //1. Primero debemos calcular las aristas que no forman parte del contorno para que no se
    pierda
    // la forma de la malla

    for(cada_arista)
        comprobar que la arista aparezca en 2 o mas triangulos

    IF no aparece en otro triangulo
        quito el triangulo de la lista de candidatos a colapsar

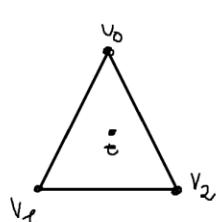
    for(triangulos_candidatos)
        colapsar_algunas de las aristas
}

```

//-----

una ocupado por sus vecinos.

3. Describir como se pueden seleccionar los vértices de una malla en un programa que visualiza mallas de triángulos.

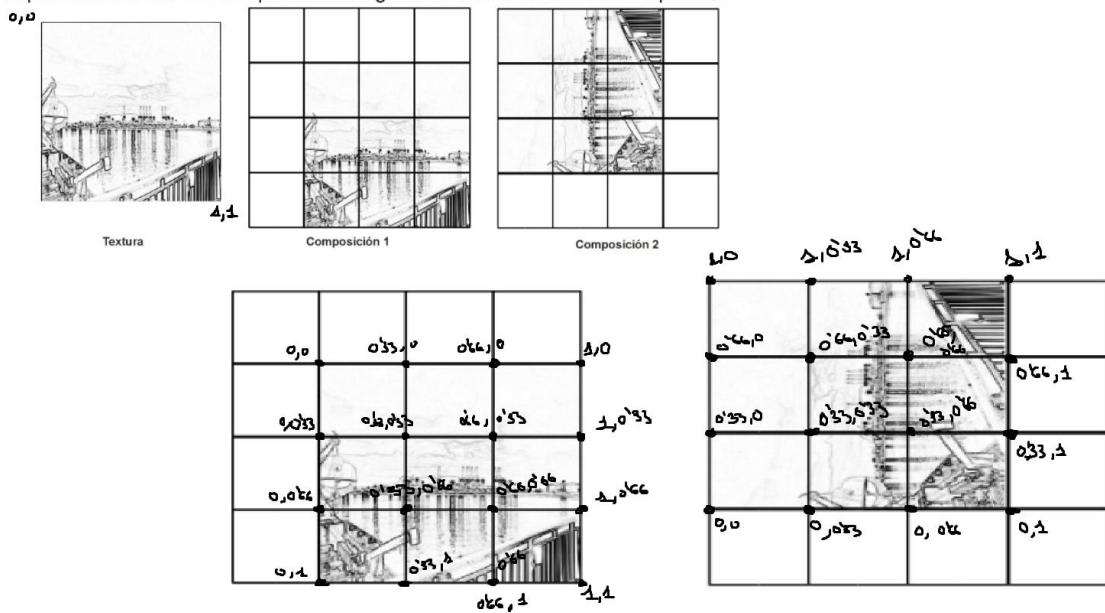


En el 4.3 sabemos reflejar el rayo.
En el problema 4.2 sabemos
calcular la intersección entre un
rayo y un triángulo.

Una vez tenemos el pto t:

1. Calcular $\|\vec{t} - \vec{v}_0\|$
2. Calcular $\|\vec{t} - \vec{v}_1\|$
3. Calcular $\|\vec{t} - \vec{v}_2\|$
4. El vértice asociado al módulo menor
es el que entendemos como clic

6. Se ha creado una retícula de 4x4 cuadrados. Indicar las coordenadas de textura de las esquinas de cada uno de ellos para obtener las dos composiciones siguientes con la textura de la izquierda.



5. Indicar cual será la posición de la luz en los casos siguientes:

- | | | | |
|----|---|----|---|
| a) | <code>GLfloat Position[4]={1,1,1,0};</code> | b) | <code>GLfloat Position[4]={1,1,1,0};</code> |
| - | <code>glMatrixMode(GL_MODELVIEW);</code> | - | <code>glMatrixMode(GL_MODELVIEW);</code> |
| - | <code>glPushMatrix();</code> | - | <code>glPushMatrix();</code> |
| - | <code>glLoadIdentity();</code> | - | <code>glLoadIdentity();</code> |
| - | <code>glLightfv(GL_LIGHT0, GL_POSITION, Position);</code> | - | <code>glRotate(45,0,1,0);</code> |
| - | <code>glPopMatrix();</code> | - | <code>glLightfv(GL_LIGHT0, GL_POSITION, Position);</code> |
| - | <code>glPopMatrix();</code> | - | <code>glPopMatrix();</code> |

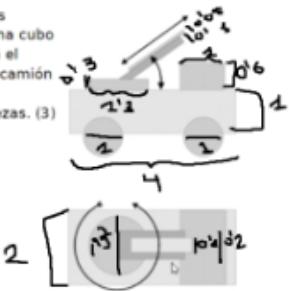
- a) La luz va a estar en la posición indicada $(1,1,1,0)$
 b) Hay que rotar la posición 45° en el eje y en $(\sqrt{2}, 1, 0, 0)$

$$\text{Nuevo_pos} = \text{MAT_Rot}(45, (0, 1, 0)) * \text{pos}$$

$$\begin{pmatrix} \cos(45) & 0 & \sin(45) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(45) & 0 & \cos(45) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} =$$

$$\begin{aligned} x' &= \cos 45 \cdot 1 + \sin 45 \cdot 1 = \frac{\sqrt{2}}{2} \\ y' &= 1 \\ z' &= -\sin 45 \cdot 1 + \cos 45 \cdot 1 = 0 \end{aligned}$$

1. Generar el grafo de escena incluyendo las transformaciones, tal que partiendo de una cubo unidad y un cilindro unidad centrados en el origen permite obtener un modelo de un camión con escalera. Hacer un dibujo del posicionamiento y dimensiones de las piezas. (3)



Escalera(α)

Translado ($t = 1, 0, 0$)

Esc ($1/2, 0.05, 0.1$)

Cubo

Base Escalera

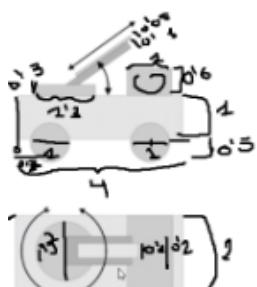
Trans ($1.5, 1, 0, 0$)
Esc ($1.5, 0.1, 0.2$)

Cubo

Cilindro Esc
Esc ($-1/3, 0.3, 1/3$)
Cilindro

Esc Comp (γ, α, Tr)
Rot ($\gamma, 0, 1, 0$)
Cilindro Esc
Trans ($0, 0.3, 0$)
Base Esc (α, Tr)

B3 Esc Anim (α, Tr)
Rot ($\alpha, 0, 1$)
Base Esc
Trans ($0, 0.1, 0$)
Escalera (Tr)



Cabin
Trans ($3, 1.5, 0$)
Esc ($1, 0.6, 2$)
Cubo

Chasis
Trans ($2, 0.15, 0$)
Esc ($1, 4, 2$)
Cubo

Rueda
Trans ($0.5, 0, 0$)
Rot ($90, 4, 0, 0$)
Esc ($0, 2, 0$)
Cilindro

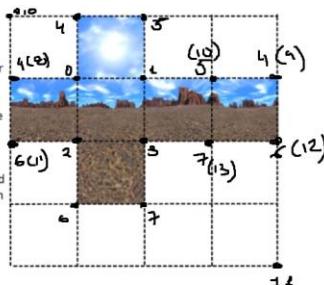
Ruedas Delanteras
Trans ($3.3, 0, 0$)
Rueda

Base (γ, α, Tr)
Rueda
Ruedas Delanteras
Chasis
Cabin
Translado ($0, 1.5, 0$)
Esc Comp (γ, α, Tr)

Distintas formas para modelar un sólido definido por fronteras como revolución, etc.

Revolution, barrido el del eje Y o X o Z, aristas aladas.

2. Sean seis imágenes que se han agrupado como una sola y se quieren usar para un skybox (caja que engloba la escena). Considerar un cubo creado como lista de vértices y caras (8 vértices y 12 caras inicialmente). Indica como se han de asignar las coordenadas de textura a las caras de dicho cubo (nota: hay que ser coherente con las imágenes del skybox y recordad que en OpenGL las imágenes están normalizadas con el origen en la posición superior izquierda). (2)



Caras: $(0,1,3), (0,2,3), (9,12,10)$
 $(4,0,4), (4,5,1), (10,13,12)$
 $(2,3,6), (6,7,3)$
 $(8,11,0), (8,2,0)$
 $(-1,10,3), (10,3,13)$

0	una vez	4	3 veces
1	una vez	5	2 veces
2	una vez	6	3 veces
3	una vez	7	2 veces

$P_4 \rightarrow 8,9$
 $P_5 \rightarrow 10$
 $P_6 \rightarrow 11,12$
 $P_7 \rightarrow 13$

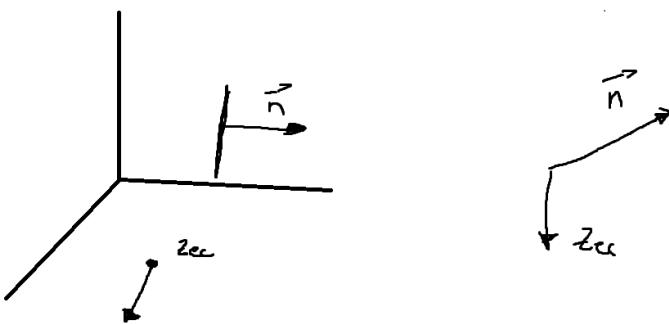
Texturas:

$(0^{\circ}25, 0^{\circ}25)$	$/ / \circ$	$(6^{\circ}75, 0^{\circ})$	$(0^{\circ}75, 0^{\circ}25)$
$(0^{\circ}5, 0^{\circ}25)$		$(5^{\circ}25, 0^{\circ}25)$	$(0^{\circ}10, 0^{\circ}25)$
$(5^{\circ}25, 0^{\circ}5)$		$(5^{\circ}25, 0^{\circ}75)$	$(4, 0^{\circ}25)$
$(0^{\circ}5, 0^{\circ}5)$		$(0, 0^{\circ}25)$	$(4, 0^{\circ})$
$(0^{\circ}25, 0^{\circ})$		$(4, 0^{\circ}25)$	$(0^{\circ}75, 0^{\circ}25)$

6. Explique los pasos que se siguen en OpenGL para utilizar una imagen como textura (1)

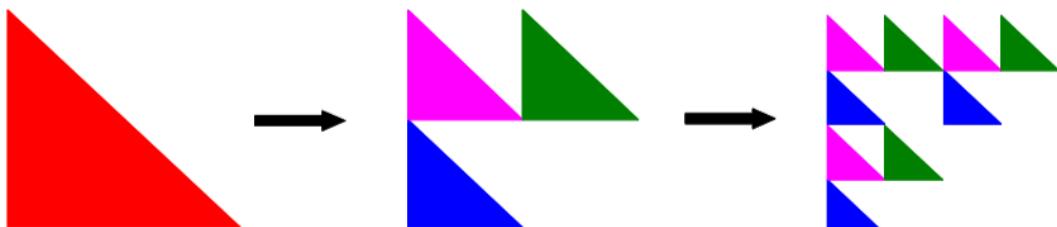
La cargas la manda a la gpu,(matriz de texels) la normaliza, y luego la interpola y asignas a cada uno de los vértices.

2. Supongamos un objeto descrito con listas de vértices y caras (`vector<vertex3f> Vertices; vector<vertex3i> Triangulos`) y un punto donde se sitúa el observador. Implementar en pseudocódigo el algoritmo de eliminación de las caras traseras (back face culling) que permite seleccionar las caras visibles a un observador. ¿Qué listas ha de devolver el algoritmo? (2)



1. Sacamos con la lista de vértices y de triángulos una lista de normales asociados a esos triángulos
2. Si un vector de esa lista y Zer forman un ángulo mayor o igual que 90° y menor o igual que 270° lo borramos
3. Desolvemos la lista con los triángulos que no cumplen la condición anterior.

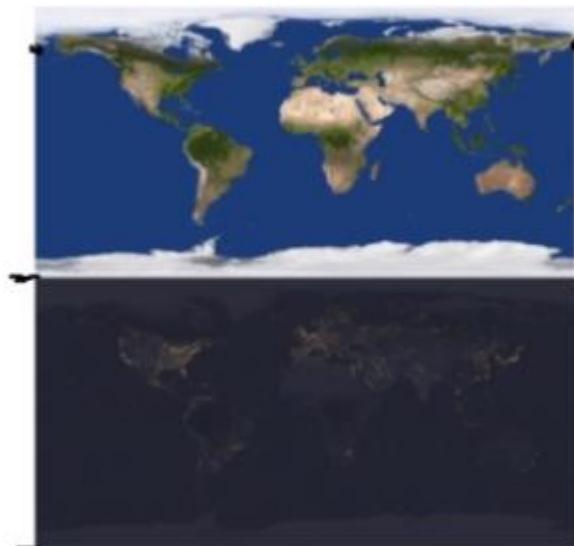
1) Programar usando C++ y OpenGL un procedimiento que de forma recursiva permita generar el siguiente patrón. Se indicará el número de niveles de la recursión mediante un parámetro. (2.5 pt)



```
void calculaTriangulo(unsigned n, unsigned exp){
    if(n==1)
        dibujaTriangulo();
    else{
        MAT_Escalado(0.5,0.5);
        calculaTriangulo(n-1, exp+1);
        MAT_Translacion(0,0.5*exp);
        calculaTriangulo(n-1,exp+1);
        MAT_Translacion(0.5*exp,0);
        calculaTriangulo(n-1,exp+1);
    }
}
```

}

3) Dada la siguiente imagen que representa simultáneamente el planeta tierra de día y de noche, escriba mediante código C++ la asignación de texturas para una hora h del día a la malla poligonal que representa una esfera `Esfera::Esfera(int nmeridianos, int nparalelos)` creada con la siguiente llamada `Esfera=new Esfera(24,24)`, suponiendo que el eje de rotación de la Tierra permanece en todo momento perpendicular al plano orbital con respecto al sol. ((2.5 pt)



```
void calculaTexturaPlaneta(){
    for(int i=0; i < 24; i++){
        for(int j=0; j<24; j++){
            if(i<12){
                coord_tex.push_back({i*1/23,j*(0.5/23)});
            }
            else{
                coord_tex.push_back({i*1/23,0.5+j*(0.5/23)});
            }
        }
    }
}
```

Ejercicio 1. Considera una malla de n triángulos almacenada en memoria con un vector caras (con n entradas), de forma que $caras[i][j]$ es un entero, en concreto el índice del vértice número j de la cara número i (con $0 \leq i \leq n$ y $0 \leq j < 3$).

1. Con esta definición, escribe el código de una función con esta declaración:

```
bool comparten_vertice(int c1, int c2);
```

que devuelve true cuando las caras número $c1$ y $c2$ comparten un vértice (devuelve false si esto no es así).

```
bool comparten_vertice(int c1, int c2){  
    const int NUMERO_VERTICES = 3;  
    bool comparten_vertice = false;  
  
    for (int j = 0; j < NUMERO_VERTICES && !comparten_vertice; ++j){  
        for (int k = 0; k < NUMERO_VERTICES && !comparten_vertice; ++k){  
            if (caras[c1][j] == caras[c2][k])  
                comparten_vertice = true;  
        }  
    }  
  
    return comparten_vertice;  
}
```

2. Escribe el código de otra función:

```
bool comparten_aristas(int c1, int c2);
```

que devuelve true cuando las caras número $c1$ y $c2$ comparten una arista (devuelve false si esto no es así).

```
bool comparten_arista(int c1, int c2){  
    const int NUMERO_VERTICES = 3;  
    bool comparten_arista = false;  
    bool comparten_vertice = comparten_vertice(c1, c2);  
    int vertice_a;  
    int vertice_b;  
  
    vertice_a = -1;  
    vertice_b = -1;  
  
    if (comparten_vertice){  
        for (int j = 0; j < NUMERO_VERTICES && !comparten_arista; ++j){  
            for (int k = 0; k < NUMERO_VERTICES && !comparten_arista; ++k){  
                if ( caras[c1][j] == caras[c2][k] && vertice_a == vertice_b )  
                    vertice_a = caras[c1][j];  
                else if ( caras[c1][j] == caras[c2][k] && vertice_a != vertice_b )  
                    comparten_arista = true;  
            }  
        }  
    }  
  
    return comparten_arista;  
}
```

2. Considera una esfera representada como una malla de n triángulos, que está almacenada en memoria con un vector cara, un vector vertice y un vector normal , de forma que cara[i][j] es un entero, en concreto el índice del vértice número j de la cara número i en el vector vertice, normal[i] es el vector normal al vértice i, y vertice[i] contiene el vértice i. Los vectores y los puntos se almacenan como una estructura con tres campos (x,y,z). Describe usando pseudocódigo una función para transformar la esfera en un elipsoide. Indica como se podría realizar una animación del proceso.

```

void esferaEclipsoide(){
    //El vector cara se mantiene igual. Los que van a cambiar son vertices y el normal
    //Para hacer el elipsoide vamos a escalarlo en el eje x

    for Vertice v en Vertices:
        v = MAT_Escalado(2,1,1)*v;

    vector listaNormalesTri;
    vector listaNormalesVer;
    for triangulo in Triangulos:
        listaNormalesTri.push_back(calculaNormal(triangulo));
        for vertices in triangulos:
            listaNormalesVer[ver]+=listaNormalesTri(triangulo);

    for normal in listaNormalesVer:
        normal = normal.normalizada;

}

```

```

void Mallalnd::calcularNormales()
{
    // COMPLETAR: en la práctica 4: calculo de las normales de la malla
    // se debe invocar en primer lugar 'calcularNormalesTriangulos'

    /////////////////////////////////
    calcularNormalesTriangulos();

    nor_ver = std::vector<Tupla3f>(vertices.size(), Tupla3f(0.0, 0.0, 0.0));
    for (unsigned int i = 0; i < triangulos.size(); i++){
        for (unsigned int j = 0; j < 3; j++){
            unsigned int indice_vertice = triangulos[i](j);

            nor_ver[indice_vertice] = nor_ver[indice_vertice] + nor_tri[i];
        }
    }

    for (unsigned int i = 0; i < nor_ver.size(); i++)
        if (nor_ver[i].lengthSq() != 0.0)
            nor_ver[i] = nor_ver[i].normalized();
    ///////////////////////////////
}

```

```

void Mallalnd::calcularNormalesTriangulos()
{
    // si ya está creada la tabla de normales de triángulos, no es necesario volver a crearla
    const unsigned nt = triangulos.size() ;
    assert( 1 <= nt );
    if ( 0 < nor_tri.size() )
    {
        assert( nt == nor_tri.size() );
        return ;
    }

    // COMPLETAR: Práctica 4: creación de la tabla de normales de triángulos
    // ....
    Tupla3f vector_a, vector_b;
    Tupla3f vector_mc;

    Tupla3f vector_normal;

    Tupla3f vector_nulo = {0.0,0.0,0.0};

    for(int n=0; n<triangulos.size(); n++){

        vector_a = vertices.at(triangulos.at(n)[1]) - vertices.at(triangulos.at(n)[0]);
        vector_b = vertices.at(triangulos.at(n)[2]) - vertices.at(triangulos.at(n)[0]);

        vector_mc = vector_a.cross(vector_b);

        if(vector_mc.lengthSq() != 0.0){

            vector_normal = vector_mc.normalized();

            nor_tri.push_back(vector_normal);

        }else{

            nor_tri.push_back(vector_nulo);
        }
    }
}

```

PREGUNTAS-FRECUENTES-EXAMENES-IN...



BrokenQuagga



Informática Gráfica



3º Grado en Ingeniería Informática



**Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada**

Te regalamos



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

1

Abre tu Cuenta
Online
sin comisiones
ni condiciones

2

Haz una compra
igual o superior
a 15€ con tu
nueva tarjeta

3

BBVA
te devuelve
un máximo de
15€

Te regalamos



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve un máximo de 15€

PREGUNTAS FRECUENTES EXAMENES INFORMÁTICA GRÁFICA

Explique, lo mas detallado posible, las distintas formas de realizar un pick.

Hay 3 formas de realizarlo:

- **Identificación por color:** A cada objeto que se quiere identificar se le asigna un identificador, un numero natural. Este número es convertido a un color. Se activa la eliminación de partes ocultas. Cuando se dibuja el objeto, se usa el color que tiene asociado. Al mover el cursor y pulsar para realizar la selección, se guardan las coordenadas x e y del pixel seleccionado. Se lee el pixel del buffer en la posición x e y. Se convierte al color seleccionado.

En el caso de representar el color con el modelo RGB y 24 bits, se tiene la posibilidad de identificar $2^{24}-1$ objetos diferentes. El blanco indica que no se selecciona nada.

Para pasar del identificador al color, se usan máscaras de bits para obtener cada parte. Para pasar de color a ID se hacen los pasos inversos

- **Lanzado de un rayo:** La idea básica es que cuando pulso un botón del ratón, voy a seleccionar el objeto más cercano que está en la posición del cursor. Para ello, obtenemos la posición x e y del cursor en coordenadas del dispositivo y las convertimos a coordenadas de vista. Hacemos pasar una línea recta por el centro de proyección y la nueva posición. Esto es, obtenemos la ecuación de una recta. Calculamos la intersección con los objetos. Si hay intersección, se añade a la lista guardando el ID del objeto y la profundidad. Por último, ordenamos en profundidad y devolvemos el ID de la intersección más cercana.
- **Por ventana:** Este método consiste en aprovechar la etapa de discretización del cauce visual. Esto es, cuando pasamos de fórmulas a píxeles. La idea consiste en que cuando se marca la posición con el ratón, se obtiene una posición x e y. Alrededor de dicha posición se crea una pequeña ventana. Una vez identificados los píxeles que conforman la ventana, lo único que hay que hacer es dibujar cada objeto al cual se le asigna un identificador. Si al convertir el objeto en píxeles, coincide con alguno o varios de la ventana, entonces hay selección. Se guarda el identificador del objeto y la profundidad. Finalmente, podemos hacer una ordenación por profundidad y quedamos con el identificador del más cercano.

Describa el flujo de transformaciones que se realizan en OpenGL desde que proporcionamos las coordenadas 3D de un modelo hasta que tenemos una imagen en pantalla. Indique el propósito de cada etapa y el resultado obtenido tras cada una de las transformaciones.

1. Transformación del modelo: Situarlo en escena, cambiarlo de tamaño y crear modelos compuestos de otros más simples.
2. Transformación de vista: Poner al observador en la posición deseada.
3. Transformación de perspectiva: Pasar de un mundo 3D a una imagen 2D.
4. Rasterización: Calcular para cada pixel su color, teniendo en cuenta la primitiva que se muestra, su color, material, texturas, luces, etc.
5. Transformación del dispositivo: Adaptar la imagen 2D a la zona de dibujado.

Describa las formas de interacción de la luz como partícula según la superficie de los objetos y el proceso que sigue para realizar un suavizado de Gouraud.

Si la luz interactúa con una superficie pulida opaca, es una reflexión especular o regular. Si la luz interactúa con una superficie rugosa opaca, es una reflexión difusa.

El suavizado de Gouraud se realiza mediante las normales de los vértices para calcular la intensidad de la luz. Luego se interpolan esas intensidades para encontrar los valores en los píxeles en los que proyecta el polígono en pantalla.

¿Qué es la transformación de vista?

La transformación de vista es una transformación que permite cambiar de sistema de coordenadas. Esta transformación permite simular el posicionamiento de la cámara en cualquier posición y orientación aplicando transformaciones geométricas (traslaciones, rotaciones, etc.).

Indique qué parámetros de la cámara están implicados en la transformación de vista y cómo se usan para obtener la transformación de la vista.

VRP: Posición donde está la cámara.

VPN: Hacia donde mira la cámara.

VUP: Indica la orientación hacia arriba.

¿Qué matriz de OpenGL almacena la transformación de vista?

GL_MODELVIEW

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Te regalamos

15€

- 1 Abre tu Cuenta Online sin comisiones ni condiciones
- 2 Haz una compra igual o superior a 15€ con tu nueva tarjeta
- 3 BBVA te devuelve un máximo de 15€



Cree un ejemplo de transformación de vista incluyendo las llamadas de OpenGL.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0,0,-10);
glRotatef(37,1,0,0);
glRotatef(45,0,1,10);
```

Enumere y explique las propiedades de la transformación de perspectiva.

Acortamiento perspectivo: objetos más lejanos producen una proyección más pequeña.

Puntos de fuga: cualquier par de líneas paralelas convergen en un punto llamado punto de fuga.

Inversión de vista: los puntos que están detrás del centro proyección se proyectan invertidos.

Distorsión topológica: cualquier elemento geométrico que tenga una parte delante y otra detrás del centro proyección produce dos proyecciones semiinfinitas.

Queremos realizar acercarnos a un objeto para ver sus detalles. Explicar cómo se podría hacer en una proyección de perspectiva.

La solución más sencilla consiste en acercarse al objeto. El problema está en que si no se cambia el plano delantero habrá un momento en el que se alcanza el objeto y lo recortará. Si colocamos la cámara en una posición donde los planos de corte no recorten el objeto, se puede hacer un zoom simplemente cambiando el tamaño de la ventana de proyección.

Indica los pasos que hay que realizar en OpenGL y los elementos que intervienen y por tanto han de estar definidos para conseguir que una escena se vea iluminada.

Definir los vectores normales de cada cara: Es necesario definir un vector normal (perpendicular a la superficie apuntando hacia fuera de la parte visible) por cada uno de los vértices de nuestra representación.

Situar las luces: Para iluminar una escena será necesario situar las luces. OpenGL maneja dos tipos de iluminación:

- Luz ambiental: ilumina toda la escena por igual, ya que esta no proviene de una dirección predeterminada.

- Luz difusa: viene de una dirección específica, y depende de su ángulo de incidencia para iluminar una superficie en mayor o menor medida.

Definiendo materiales: OpenGL permite controlar la forma en que la luz se refleja sobre nuestros objetos, que es lo que se conoce como definición de materiales.

Te regalamos



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve un máximo de 15€

Dado un cubo definido por sus vértices, vector<`_vertex3f`> Vertices, y sus triángulos, vector<`_vertex3ui`> Triangles, indique lo siguiente si queremos mostrar el cubo texturado:

a) La estructura de datos para guardar las coordenadas de textura

Sería un vector para vértices en dos dimensiones. Su tamaño sería igual al número de vértices:

```
Vector<_vertex2d> Vertices_texcoordinates;
Vertices_texcoordinates.resize(vertices.size());
```

b) La función que dibujaría el objeto texturado

```
glBegin(GL_TRIANGLES);
for (unsigned int i=0; i<Triangles.size();i++)
{
    glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._0]);
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._0]);
    glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._1]);
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._1]);
    glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._2]);
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._2]);
}
glEnd();
```

c) Un ejemplo de coordenadas de textura para cada vértice, si la textura se aplica a todas las caras sin repetirla (esto es, cada cuadrado NO muestra la textura completa).

Si no tenemos en cuenta el problema de los puntos repetidos, bastaría con desplegar el cubo sobre la textura y asignar los valores de las coordenadas de textura correspondientes. Por ejemplo:

```
Vertices_texcoordinates[0]=_vertex2f(0,0.5);
Vertices_texcoordinates[1]=_vertex2f(0.25,0.5);
Vertices_texcoordinates[2]=_vertex2f(0,0.75);
Vertices_texcoordinates[3]=_vertex2f(0.25,0.75);
Vertices_texcoordinates[4]=_vertex2f(0.75,0.5);
Vertices_texcoordinates[5]=_vertex2f(0.75,0.5);
Vertices_texcoordinates[6]=_vertex2f(0.5,0.75);
Vertices_texcoordinates[7]=_vertex2f(0.5,0.5);
```

Explica el funcionamiento Z-Buffer.

El algoritmo del Z-buffer es del tipo espacio-imagen. Cada vez que se va a renderizar un píxel, comprueba que no se haya dibujado antes en esa posición un pixel que esté más cerca respecto a la cámara (posición en eje Z). Este algoritmo funciona bien para cualquier tipo de objetos: cóncavos, convexos, abiertos y cerrados. Cuando dibujamos un objeto, la profundidad de sus pixeles se guardan en este buffer. Si utilizamos dichos pixeles en pantalla para dibujar otro objeto, se producirá la comparación de las profundidades de dichos pixeles. Si la profundidad del último píxel es mayor que la nueva (está más lejos) el pixel nuevo no se pinta, mientras que si está más cerca (la profundidad es menor), se dibuja el pixel y se guarda la nueva profundidad en el zbuffer.

Para activarlo, hay que hacer una llamada a
`glEnable(GL_DEPTH_TEST)`

Esta llamada le dice a OpenGL que active el test de profundidad. Además, cada vez que se redibuja la escena, aparte de borrar el buffer de color, hay que borrar el buffer de profundidad. Esto se hace con la llamada
`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`.

Por último, pero no menos importante, al inicializar OpenGL se le tiene que definir el buffer de profundidad en el modo de visualización.

**¿Qué es una partícula en Informática Gráfica? Describe su ciclo de vida.
¿Puedes poner algún ejemplo?**

Las partículas son elementos lógicos a las que se debe otorgar propiedades gráficas para que sean visibles. Por ejemplo, cada partícula se puede sustituir por un objeto geométrico, gracias a lo cual es posible mostrar cualquier forma o dibujo. Con la adición del material correspondiente se pueden utilizar sistemas de partículas para mostrar humo, niebla o fuego.

¿Cómo se calculan las normales a un vértice?

Podemos definir el vector normal de un vértice como el vector normal de un plano tangente al objeto en dicho vértice, pero esto es muy difícil de calcular. Por ello, realizamos un aproximación muy fiable: la suma de las normales de los triángulos adyacentes a dicho vértice. Para ello, calculamos las normales de todos los triángulos como el producto vectorial de dos de sus aristas, teniendo en cuenta que las normales tienen dirección hacia el exterior del objeto. Las normalizamos con la siguiente fórmula:

$$\frac{n_c}{\|N_c\|}$$

A continuación, sumamos para cada cara, el valor de su normal a la normal de cada uno de sus vértices. Una vez acabado, normalizamos las normales de vértices.

$$\frac{n_v}{\|N_v\|}$$

Describa las transformaciones de vista que se pueden aplicar a una cámara.

Las transformaciones de vista son `glFrustum` y `glOrtho`. La declaración de estas transformaciones es:

- `glFrustum (left, right, bottom, top, near, far);`
- `glOrtho (left, right, bottom, top, near, far);`

La función `glFrustum` describe una matriz de perspectiva que produce una proyección en perspectiva.

La función `glOrtho` describe una matriz de perspectiva que produce una proyección paralela.

En ambas, la matriz actual se multiplica por esta matriz y el resultado reemplaza a la matriz actual.

Describe brevemente para que sirven, en un programa OpenGL/glut, cada una de estas cuatro funciones:

- `glutDisplayFunc(funcion);` Esta función se llamará cada vez que se dibuje la ventana.
- `glutReshapeFunc(funcion);` Función de control del cambio de tamaño de la ventana de visualización.
- `glutKeyboardFunc(funcion);` Función de control de eventos con el teclado.
- `glutSpecialFunc(funcion);` Función de control de eventos con el teclado para cuando se ha pulsado una tecla especial.

Explique el sistema de colores que se usa en OpenGL.

El sistema de color empleado en OpenGL es el sistema RGB. RGB significa los colores rojo, verde y azul: los colores primarios aditivos. A cada uno de estos colores se le asigna un valor, en OpenGL generalmente un valor entre 0 y 1. El valor 1 significa la mayor cantidad posible de ese color, y 0 significa ninguna cantidad de ese color. Podemos mezclar estos tres colores para obtener una gama completa de colores.

Sobre las proyecciones. Indicar si es verdadero V o falso F.

- La proyección de perspectiva acorta los objetos más lejanos (V).
- Dos líneas paralelas en el modelo sólo fogan si no son paralelas al plano de proyección (F).
- El vector Z del sistema cartesiano del observador (punto de mira punto del observador) y el vector de inclinación pueden tener cualquier orientación (V).
- El vector de inclinación siempre coincide con el eje Y del observador (F)
- La ventana debe estar centrada para que se pueda realizar la proyección (F)
- En algunos casos es obligatorio poner el plano delantero detrás del plano trasero (F)
- Si un objeto tiene todos sus vértices detrás del centro de proyección, no se podrá proyectar correctamente (V)
- En una proyección paralela el zoom se puede implementar moviendo los planos de corte (V)

Preguntas-Teoria-Examen-Resuelta...



Zukii



Informática Gráfica



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

Te regalamos

15€



1

Abre tu Cuenta
Online
sin comisiones
ni condiciones

2

Haz una compra
igual o superior
a 15€ con tu
nueva tarjeta

3

BBVA
te devuelve
un máximo de
15€

Te regalamos



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

1
Abre tu Cuenta Online sin comisiones ni condiciones

2
Haz una compra igual o superior a 15€ con tu nueva tarjeta

3
BBVA te devuelve un máximo de 15€



@Zukii on Wuolah

Algunas preguntas de IG Teoría Resueltas:

(Para el examen de teoría, suelen ser algunas preguntas de teoría y un grafo de escena)

Pregunta 1:

Parámetros de la cámara y como se usan en la transformación de vista Matriz y ejemplo en OpenGL

Transformación de vista: transformación que permite cambiar de sistemas de coordenadas, desde el Sistema de Coordenadas Mundial al de Vista. Permite simular el posicionamiento de la cámara en cualquier posición y orientación. A partir de los parámetros se define el Sist. Coordenadas de Vista. El SCV se alinea con el SCM aplicando transformaciones geométricas: 1 translación y 3 rotaciones.

Los parámetros implicados son:

- Punto en el plano de proyección o Punto del Observador (PO), en coordenadas mundiales 3D.
- Punto de Mira (MP), punto al que apunta la cámara, en coordenadas mundiales 3D.
- Normal al plano de proyección (PM-PO = Z)
- Vector de inclinación de la cámara (VI), en coordenadas mundiales 3D.
- Centro de Proyección (CP) o posición de la cámara, en coordenadas mundiales 3D.
- Planos de recorte frontal (F) y posterior o trasero (T), distancia sobre el eje z del sistema de coordenadas del observador.
- Ventana en el plano de proyección ($Wxmin, Wxmax, Wymin, Wymax$), en coordenadas mundiales 2D

Proceso en la transformación de vista:

1. Se posicionan el PO y el PM y se traza el vector que los une, es decir, el vector normal Z del que se obtiene el plano de proyección (PP)
2. Del plano de proyección se obtiene el vector de inclinación de la cámara (VI)
3. Se posiciona el Centro de Proyección (CP) y con la dimensión de la imagen se trazan varias líneas al Plano de Proyección (PP) creando la ventana en el plano de proyección.
4. Con dichas líneas trazadas, se colocan de forma paralela al PP los planos de recorte frontal (F) y el plano de recorte posterior o trasero (T)

El OpenGL los parámetros son:

- VRP: posición donde se encuentra la cámara equivale al origen del SCV (vista), y es un punto dado en el SCM (mundial)

- *VPN: hacia donde mira la cámara, equivale al eje Z del SCV, y es un vector dado en el SCM.*
- *VUP: indica la orientación hacia arriba (up). Es un vector dado en el SCM.*

La matriz de OpenGL que almacena dicha transformación es la GL_MODELVIEW.

Ejemplo de uso:

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(0,0,-15);  
glRotatef(35,1,0,0);  
glRotatef(45,0,1,0);
```

Propiedades de la transformación en perspectiva:

1. *Acortamiento perspectivo: los objetos que más lejanos se encuentre producirán una proyección más pequeña*
2. *Puntos de fuga: cualquier par de líneas paralelas convergen en un punto de corte llamado punto de fuga*
3. *Inversión de vista: los puntos que se encuentren detrás del centro de proyección de proyectarán invertidos.*
4. *Distorsión topológica: cualquier elemento geométrico que tenga una parte delante y otra detrás del centro de proyección produce dos proyecciones semiinfinitas.*

Pregunta 2:

Métodos de Selección

Tres formas para realizar la selección:

1. *Identificación por color*

A cada objeto identificable se le asigna un identificador (número natural) que se convierte a color. Al dibujar dicho objeto se usa dicho color asociado almacenados en el framebuffer. Al seleccionar un pixel del objeto con coordenadas x e se inspecciona en el framebuffer dicha posición y se identifica el color. Para pasar del identificador al color se usan máscaras de bits para obtener cada parte.

2. *Intersección rayo escena*

El usuario indica con el ratón el objeto más cercano en la posición del cursor. Se obtiene la posición x e y del cursor y se convierten a coordenadas de vista. Se traza un vector desde el CP y el punto indicado y se obtiene la ecuación de dicha recta con

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Te regalamos



1

Abre tu Cuenta
Online
sin comisiones
ni condiciones

2

Haz una compra
igual o superior
a 15€ con tu
nueva tarjeta

3

BBVA
te devuelve
un máximo de
15€



la que se calcula la intersección con los objetos intersectables. Si se ha producido una intersección se añade a la lista guardando el identificador del objeto y su profundidad. Por último, se ordena dicha lista por profundidad y devolvemos el identificador del objeto más cercano (el primero de la lista, el de menor profundidad)

3. Subvolumen de visión

Se marca una posición con el ratón obteniendo la posición x e y. Alrededor de dicha posición se crea una ventana de unos pocos píxeles y se identifican dichos píxeles. Se dibuja cada objeto con su correspondiente identificador. Si al convertir los objetos en píxeles coincide con algunos de la ventana se ha producido una selección. Se guarda el identificador del objeto y la profundidad y se hace una ordenación por profundidad para quedarnos con el identificador del objeto más cercano.

Pregunta 3:

Algoritmo zbuffer. Explícalo

Método de eliminación de caras o partes ocultas. Devuelve para cada pixel el color del objeto más cercano al observador, puede emplearse cualquier objeto, no solo por el que está definido por caras planas.

El algoritmo necesita de una memoria en donde se va almacenando los valores de profundidad para cada pixel en una matriz (llamada zbuffer). La matriz se inicializa inicialmente a infinito.

Se itera sobre cada polígono de la esfera sobre todos los píxeles que la componen, de los que se calculará la distancia en z hacia la cámara y se actualiza la matriz según el siguiente criterio:

1. *Si el valor en z para cada pixel es menor que el valor del z buffer (es decir, el objeto en cuestión se encuentra más cerca de la cámara) se reemplaza el valor de ese pixel en el zbuffer por el valor del pixel del polígono actual*
2. *Si el valor en el z buffer es mayor (es decir, el objeto se encuentra más lejos que otro objeto ya visto) no se modifica la matriz.*

Te regalamos

15€

1 / 6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

Abre tu Cuenta
Online
sin comisiones
ni condiciones

2

3

@Zukii on Wuolah



Pregunta 4:

¿Cómo hacer zoom en una escena en perspectiva cambiando los parámetros?

Dos formas

Ventaja: solución más sencilla

- Se mueve el CP (centro de proyección) y se cambia el radio de visión, si hacemos el radio más grande los objetos se verán más pequeños, si se hace el radio más pequeño los objetos se verán más grandes.

Problema: si no se cambia el plano delantero habrá un momento en el que se alcance el objeto y se recorte.

- Cambiando el tamaño de la imagen, como por ejemplo la vista del orto con el factor, pues ese factor también se puede poner en la vista en perspectiva Dicho factor hace el plano más grande o pequeño. Con la vista en perspectiva podemos poner los planos más lejos (los que van desde el CP al punto de corte) o multiplicando el plano por un factor y hacerlo más grande.

Ventaja: esto se consigue si se coloca la cámara en una posición donde los planos de corte no recorten el objeto

Si se fija el tamaño tendríamos que cambiar el ángulo de visión sin tocar el tamaño de la imagen, la distancia que hay desde el centro de proyección al plano

Pregunta 5:

Modelos de Iluminación en OpenGL

El modelo de iluminación de OpenGL tiene 3 componentes:

1. Componente difusa
 - Modela la reflexión de objetos que no son brillantes
 - Depende de la orientación del objeto, de su superficie y de la posición de la fuente de luz.
 - La orientación del objeto se modela con el vector normal.
 - La posición de la luz se modela con el vector entre la posición de la luz y un punto del objeto.
 - La reflexión depende del ángulo que forman ambos vectores. Si ambos vectores están normalizados, el ángulo se calcula con su producto escalar.
 - Hay que tener en cuenta la constante de reflexividad difusa del objeto y la componente difusa de la luz.
 - El máximo punto de luz no varía respecto al espectador.
2. Componente especular
 - Modela la reflexión de objetos que son brillantes
 - Depende de la orientación del objeto, de su superficie y de la posición del observador.
 - La orientación del objeto se modela con el vector normal.
 - La posición de la luz se modela con el vector entre la posición de luz y un punto del objeto.
 - Dado el rayo de luz que incide con un ángulo alfa con respecto a la normal, se produce un rayo reflejado con el mismo ángulo alfa de salida. Además, tenemos el vector que se forma entre la posición del observador y el punto de incidencia del rayo.
 - La reflexión depende del ángulo que forma el vector del rayo reflejado y el vector del observador. Si los vectores están normalizados el ángulo se calcula como el producto escalar.
 - Hay que tener en cuenta la constante de reflexividad especular del objeto y de la componente especular de la luz
 - El máximo punto de luz varía respecto al espectador.
3. Componente ambiental
 - Modela la inter reflexión de la luz en los múltiples objetos de la escena. Implica que la luz reflejada viene de todas las direcciones.
 - Se crea para hacer que la parte de los objetos a los que no le llega luz directamente no aparezca de color negro.
 - La reflexión depende de una constante de reflexividad ambiental del objeto y de la componente ambiental de la luz.

Y te recomiendo practicar los grafos de escena porque probablemente ese sea tu examen de teoría

@Zukii on Wuolah

PD: Si necesitas las prácticas, están subidas en mi perfil de wuolah jejeje.

Espero que os sirva de cara a los próximos exámenes y recordad que en mi perfil tenéis para muchas asignaturas, exámenes y prácticas resueltas, apuntes, etc. Suerte ^^



respuestasexameneoriagrupoB.pdf



PruebaAlien



Informática Gráfica



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada

Te regalamos

15€



1

Abre tu Cuenta
Online
sin comisiones
ni condiciones

2

Haz una compra
igual o superior
a 15€ con tu
nueva tarjeta

3

BBVA
te devuelve
un máximo de
15€

Te regalamos



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

BBVA está adherido al Fondo de Garantía de Depósitos de Entidades de Crédito de España. La cantidad máxima garantizada es de 100.000 euros por la totalidad de los depósitos constituidos en BBVA por persona.

1

Abre tu Cuenta Online sin comisiones ni condiciones

2

Haz una compra igual o superior a 15€ con tu nueva tarjeta

3

BBVA te devuelve un máximo de 15€

1. ¿Qué es la transformación de vista?

La transformación de vista es una transformación que permite cambiar de sistema de coordenadas, desde el SCM al SCV. Esta transformación permite simular el posicionamiento de la cámara en cualquier posición y orientación. A partir de los parámetros se define el SCV. El SCV se alinea con el SCM aplicando transformaciones geométricas: 1 traslación y 3 rotaciones.

¿Qué parámetros de la cámara están implicados?

Los parámetros que definen la TV son:

VRP: Posición donde está la cámara. Origen del SCV. Es un punto dado en el SCM

VPN: Hacia donde mira la cámara. Es el eje z del SCV. Es un vector dado en el SCM

VUP: Indica la orientación hacia arriba. Es un vector dado en el SCM

¿Qué matriz de OpenGL almacena dicha transformación?

La GL_MODELVIEW

Cree un ejemplo incluyendo las llamadas de OpenGL.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0,0,-10);
glRotatef(37,1,0,0);
glRotatef(45,0,1,10);
```

2. Enumere y explique las propiedades de la transformación de perspectiva

- Acortamiento perspectivo: objetos más lejanos producen una proyección más pequeña
- Puntos de fuga: cualquier par de líneas paralelas convergen en un punto llamado punto de fuga
- Inversión de vista: los puntos que están detrás del centro proyección se proyectan invertidos
- Distorsión topológica: cualquier elemento geométrico que tenga una parte delante y otra detrás del centro proyección produce dos proyecciones semiinfinitas.

3. Queremos realizar acercarnos a un objeto para ver sus detalles. Explicar cómo se podría hacer en una proyección de perspectiva, ventajas e inconvenientes.

a) La solución más sencilla consiste en acercarse al objeto. El problema está en que si no se cambia el plano delantero habrá un momento en el que se alcanza el objeto y lo recortará

b) Si colocamos la cámara en una posición donde los planos de corte no recorten el objeto, se puede hacer un zoom simplemente cambiando el tamaño de la ventana de proyección.

4. Dado un cubo definido por sus vértices, vector<_vertex3f> Vertices, y triángulos, vector<_vertex3ui> Triangles, indique lo siguiente si queremos mostrar el cubo texturado:

1) La estructura de datos para guardar las coordenadas de textura

Sería un vector para dos flotantes. Su tamaño sería igual al número de vértices

Vector<_vertex2d> Vertices_texcoordinates;

2) La función que dibujaría el objeto texturado

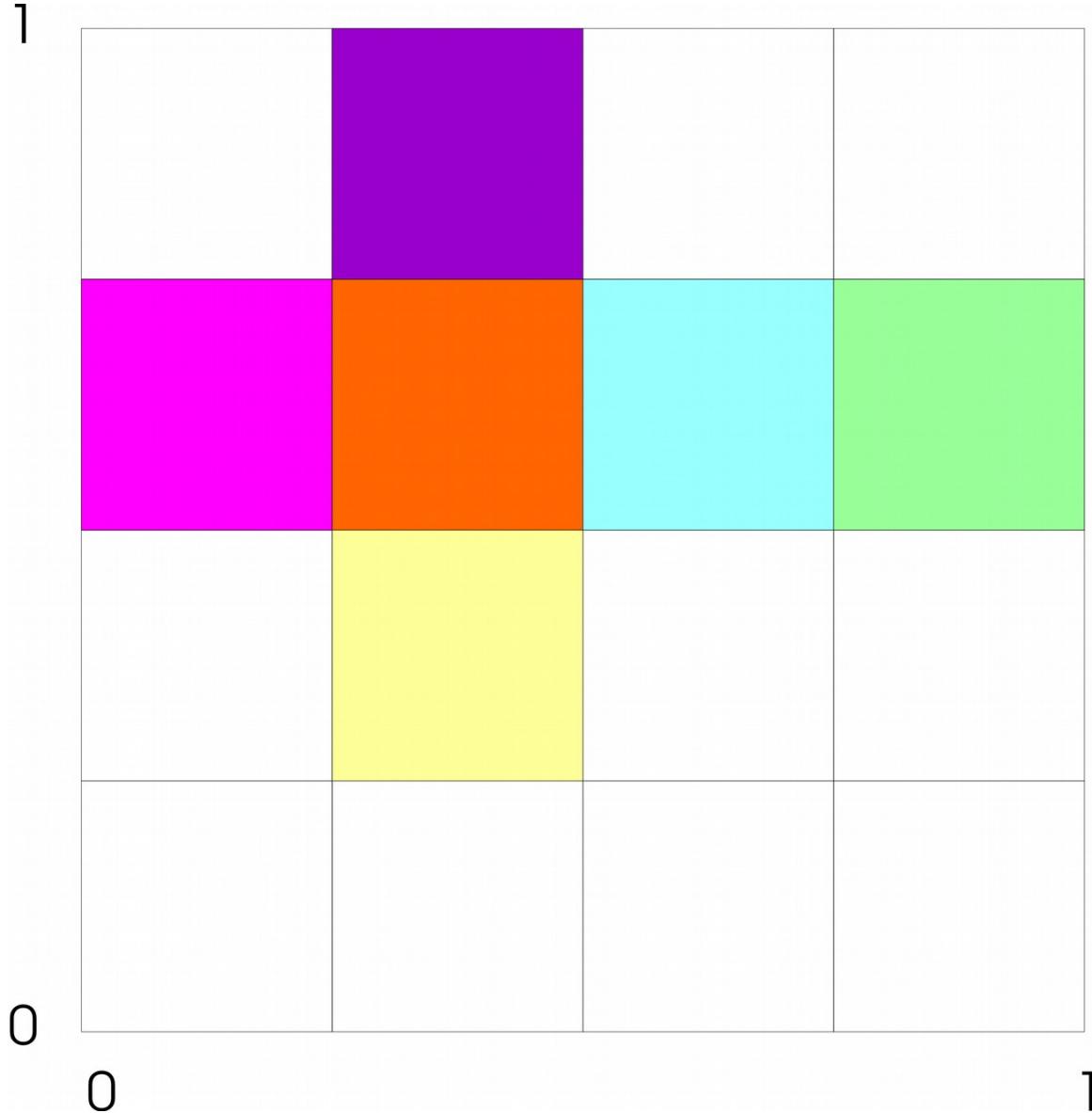
```
glBegin(GL_TRIANGLES);  
for(unsigned int i=0; i<Triangles.size();i++){  
    glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._0]);  
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._0]);  
    glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._1]);  
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._1]);  
    glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._2]);  
    glVertex3fv((GLfloat *) &Vertices[Triangles[i]._2]);  
}  
glEnd();
```

3) Un ejemplo de coordenadas de textura para cada vértice, si la textura se aplica a todas las caras sin repetirla (esto es, cada cuadrado NO muestra la textura completa)

Si no tenemos en cuenta el problema de los puntos repetidos, bastaría con desplegar el cubo sobre la textura y asignar los valores de las coordenadas de textura correspondientes. Por ejemplo:

```
Vertices_texcoordinates[0]=_vertex2f(0,0.5);  
Vertices_texcoordinates[1]=_vertex2f(0.25,0.5);  
Vertices_texcoordinates[2]=_vertex2f(0,0.75);  
Vertices_texcoordinates[3]=_vertex2f(0.25,0.75);  
Vertices_texcoordinates[4]=_vertex2f(0.75,0.5);  
Vertices_texcoordinates[5]=_vertex2f(0.75,0.5);
```

```
Vertices_texcoordinates[6]=_vertex2f(0.5,0.75);  
Vertices_texcoordinates[7]=_vertex2f(0.5,0.5);
```



IGejerciciosresueltos.pdf



patriciacorhid



Informática Gráfica



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



**Que no te escriban poemas de amor
cuando terminen la carrera** ►►►►►►►►

☺
(a nosotros por
suerte nos pasa)

WUOLAH



(a nosotros por suerte nos pasa)

IG: Soluciones Relación General

Tema 1

① `glViewport(x, y, ancho, alto);` (Pag 111)

coordenadas de la esquina inferior izquierda (en pixels)

- Si $ancho \leq alto$:

`glViewport(0, $\frac{alto-ancho}{2}$, ancho, ancho);`



- Si $alto > ancho$:

`glViewport($\frac{ancho-alto}{2}$, 0, alto, alto);`



- Caso general:

$$m = \min\{ancho, alto\};$$

`glViewport($\frac{ancho-m}{2}$, $\frac{alto-m}{2}$, m, m);`

③ `void dibujarPoligono(int n) {`

`std::vector<Tupla3f> vertices;`

`for (int i=0; i<n; i++) {`

`float alpha = i * 2.0 * M_PI / n;`

`vertices.push_back({0.75 * cos(alpha), 0.75 * sin(alpha), 0});`

`}` // Desactivar el test de profundidad

`glDisable(GL_DEPTH_TEST);`

`glColor3f(0, 1, 1);`

 // Registrar tabla:

 RAM



`glBindBuffer(GL_ARRAY_BUFFER, 0);` // Define array de vértices

`glBindBuffer(GL_ARRAY_BUFFER, 0);` // Se hace con tablas de tipo GL_ARRAY_BUFFER

`glEnableClientState(GL_VERTEX_ARRAY);` // Dibuja lo último registrado

`glDrawArrays(GL_POLYGON, 0, vertices.size());` // Dibuja lo último registrado

 // Relleno:

`glColor3f(0, 0.2, 0.8);`

 // Índice donde empieza a pintar

 // Contorno:

`glLineWidth(2);`

`glColor3f(0, 0.2, 0.8);`

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirme
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

// Registrar tabla:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);  
glVertexAttribPointer(3, GL_FLOAT, 0, vertices.data());  
glEnableClientState(GL_VERTEX_ARRAY);
```

} No hace falta volver a registrar la tabla

// Dibujar:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
glDrawArrays(GL_POLYGON, 0, vertices.size());
```

// Restaura variables:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  
glLineWidth(1);  
glEnable(GL_DEPTH_TEST);  
glBindVertexArray(0);
```

Con Begin - End:

```
void dibujarPoligono(int n) {
```

// Crear vértices + desactivar profundidad igual que en la otra función

// Relleno

```
glColor3f(0, 1, 1);  
glBegin(GL_POLYGON);  
for (int i = 0; i < n; i++)  
    glVertex3fv(vertices[i]);  
glEnd();
```

// Contorno

```
glLineWidth(2);  
glColor3f(0, 0.2, 0.8);  
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
glBegin(GL_POLYGON);  
for (int i = 0; i < n; i++)  
    glVertex3fv(vertices[i]);
```

```
glEnd();
```

// Restaura variables + glBindVertexArray(0);

}

Que no te escriban poemas de amor cuando terminen la carrera

(a nosotros por suerte nos pasa)



Ayer a las 20:20

Oh Wuolah wuolitah
Tu que eres tan bonita

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

No si antes decirte
Lo mucho que te voy a recordar



Envía un mensaje...



WUOLAH



③ Con VAO:

```
void dibujarPoligono (int n)
// Crear vértices + desactivar profundidad igual que en la otra función
glColor3f(0, 1, 1);
// VAO
GLuint id_vao;
glCreateVertexArrays(1, &id_vao); // Crear 1 VAO
glBindVertexArray(id_vao); // Activar VAO
glBufferData(GL_ARRAY_BUFFER, tam, vertices.data(), GL_STATIC_DRAW);
// Crear VBO unsigned long tam = sizeof(Tupla3f)*(vertices.size());
GLuint id_vbo;
glCreateBuffers(1, &id_vbo); // Crear 1 VBO
glBindBuffer(GL_ARRAY_BUFFER, id_vbo); // Activar VBO
glBufferData(GL_ARRAY_BUFFER, tam, vertices.data(), GL_STATIC_DRAW);
// RDM → GPU
// No dejar activado el VBO
// Dibujar relleno:
glDrawArrays(GL_POLYGON, 0, vertices.size());
// Dibujar contorno:
glLineWidth(2);
glColor3f(0, 0.2, 0.8);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glDrawArrays(GL_POLYGON, 0, vertices.size());
glBindBuffer(GL_ARRAY_BUFFER, 0); // No dejar activado el VBO
glBindVertexArray(0);
// Restaurar variables + glBindVertexArray(0);
```

{

④) llenar limpia TODA la ventana. (Pag 107)

Con glClearColor (r, g, b, opacidad) cambiamos el color con el que limpiamos (el que se queda tras borrar).

Opacidad $\in [0, 1]$

⑤) glRect: dibuja un rectángulo.

void glRectf(GLfloat x₁, GLfloat y₁, GLfloat x₂, GLfloat y₂);

un vértice

el vértice opuesto

GL-COLOR-BUFFER-BIT | GL-DEPTH-BUFFER-BIT;

glClearColor (0.4, 0.4, 0.4); glClear (GL-COLOR-BUFFER-BIT | GL-DEPTH-BUFFER-BIT);
int m = min {ancho, alto}; glColor3f (1, 1, 1);
glRectf (ancho - m, alto - m, ancho + m, alto + m);





(a nosotros por suerte nos pasa)

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

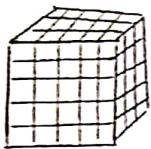
Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

Tema 2

⑩. Enumeración espacial:



Hay K cuadraditos en cada fila. Luego hay en total de K^3 celdas. Como cada una ocupa un bit, se requieren $\frac{K^3}{8}$ bytes.

• Modelo de fronteras:



Hay K meridianos y K paralelos, como en cada meridiano hay K vértices y hay $K \Rightarrow$ hay K^2 vértices. A su vez, cada vértice son 3 componentes que ocupan 4 bytes cada una. Luego son $3 \cdot 4 \cdot K^2$ bytes para los vértices.

Por otro lado, puedo identificar cada cuadradito por uno de sus vértices, luego hay K^2 cuadraditos ($2K^2$ triángulos). Para cada triángulo, guardo 3 enteros de 4 bytes cada uno, que son 3 \cdot 4 \cdot $2K^2$ bytes para triángulos.

En total se almacenan $12K^2 + 24K^2 = 36K^2$ bytes.

⑭ Caso 1:

Por cada entrada de la tabla triángulos hay 3 posibles aristas. Para comprobar si hemos añadido ya la arista, usamos una matriz triangular (pares), donde cada fila corresponde a un vértice y contiene los vértices con mayor índice que son adyacentes a este y para los que ya hemos almacenado la arista. Esto tiene complejidad $O(n)$.

```
void generarAri() {
    vector<vector<int>> pares(n - 1);
```

```
    int mayor, menor;
```

```
    for (int i = 0; i < tri.size(); i++) {
```

```
        for (int j = 0; j < 3; j++) {
```

```
            menor = min(tri[i][j], tri[i][((j + 1) % 3)]);
```

```
            mayor = max(tri[i][j], tri[i][((j + 1) % 3)]);
```

```
i & (find(pares[menor].begin(), pares[menor].end(), mayor)) == pares[menor].end() {
```

```
                ari.push_back({menor, mayor}); // añadimos la arista
```

```
                pares[menor].push_back(mayor);
```

```
            }
```

s: La arista no
estaba añadida

ari.push_back({menor, mayor}); // añadimos la arista

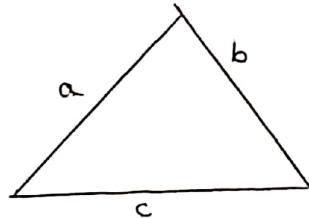
pares[menor].push_back(mayor);

Caso 2:

Cada arista aparece 2 veces, en una nos da un como (a, b) y en otra como (b, a) . Solo metemos una vez, cuando la 1^a componente es menor que la 2^a.

void generarArC() {

```
int a, b;
for (int i=0; i<tri.size(); i++) {
    for (int j=0; j<3; j++) {
        a = tri[i][j];
        b = tri[i][((j+1)%3)];
        if (a < b)
            ar.push_back({a, b});
    }
}
```



(15) Fórmula de Herón:

Área = $\sqrt{(s-a)(s-b)(s-c)s}$, donde $s = \frac{a+b+c}{2}$

• Propiedad del producto vectorial:

Área triángulo = $\frac{\|\vec{a} \times \vec{b}\|}{2}$ (Pag 162 del Tema 1)

double Area (const Malla& malla)

double a = 0; //área

Tupla3F p, q, r, u, v;

for (int i=0; i<malla.triangulos.size(); i++) {

//Vértices del triángulo

p = malla.vertices[malla.triangulos[i][0]];

q = malla.vertices[malla.triangulos[i][1]];

r = malla.vertices[malla.triangulos[i][2]];

//Dos aristas del triángulo

u = q - p;

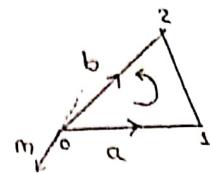
v = r - p;

a += sqrt(u.cross(v).lengthSq());

a /= 2.0;

return a;

(17) void calcularNormalesTriangulos ()



Tipo3f a, b, m;

for (int i = 0; i < triangulos.size(); i++) {

a = vertices[Triangulos[i][0]] - vertices[Triangulos[i][1]];

b = vertices[Triangulos[i][2]] - vertices[Triangulos[i][1]];

m = a.cross(b);

if (m[X] != 0 or m[Y] != 0 or m[Z] != 0)

m = m.normalized();

nor_tri.push_back(m);

}

void calcularNormales () {

calcularNormalesTriangulos();

for (int i = 0; i < vertices.size(); i++)

nor_ver.push_back(<0.0, 0.0, 0.0>);

for (int i = 0; i < triangulos.size(); i++) // Para cada triángulo, sumar su normal a los vértices que lo componen.

for (int j = 0; j < 3; j++)

nor_ver[Triangulos[i][j]] = nor_ver[Triangulos[i][j]] + nor_tri[i];

for (int i = 0; i < nor_ver.size(); i++)

if (nor_ver[i][X] != 0 or nor_ver[i][Y] != 0 or nor_ver[i][Z] != 0)

nor_ver[i] = nor_ver[i].normalized();

}

el nuevo vértice sería:

(x, y, z) + N(x, y, z). E, donde $N(x, y, z)$ es su normal.

$(x, y, z) + N(x, y, z) \cdot E$, donde $N(x, y, z)$ es su normal.

(18) Con el test de profundidad desactivado, si pintamos las aristas después se nos quedan todas las aristas deante del relleno, no sirve. Con el test de profundidad, como las aristas no tienen la misma z que los triángulos, por pequeños errores de cálculo algunas aristas pueden quedar ocultas por los de color. Para solucionarlo, desplazaremos los vértices de los triángulos. Para que quede por la malla en dirección de su normal, para que quede por encima del relleno. Se escribe lo siguiente enriba del código:

glEnable(GL_POLYGON_OFFSET_FILL); *Fill para que afecte al relleno, Line para las líneas.*
 glPolygonOffset(1.0, 1.0);

Los valores positivos del primer parámetro meten el dibujo hacia adentro, y los negativos lo sacan para afuera.

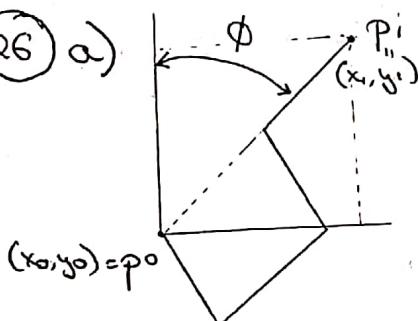
(Y ya podemos añadir aquí código como en el ejercicio ③ para dibujar la figura).

24 void gancho()

```
glBegin(GL_UNESTRIPE);
    glVertex2f(0, 0);
    glVertex2f(1, 0);
    glVertex2f(1, 1);
    glVertex2f(0, 1);
    glVertex2f(0, 2);
    glEnd();
```

{

26 a)



Componemos la matriz Modelview para trasladar el gancho y que aparezca como en la figura.
 Como las rotaciones se hacen en sentido antihorario, rotamos con ángulo -phi.

```
void gancho_2p_a(float x0, float y0, float x1, float y1)
{
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
    float phi = atan2(x1 - x0, y1 - y0); // ángulo phi
    float mod = sqrt((x0 - x1) * 2 + (y0 - y1) * 2); // ||p1 - p0||

    float esc = mod / 2.0;
    glTranslatef(x0, y0, 0);
    glRotatef(-phi * 180 / M_PI, 0, 0, 1);
    glScalef(esc, esc, esc);
    gancho();
}
```

{



(a nosotros por suerte nos pasa)

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

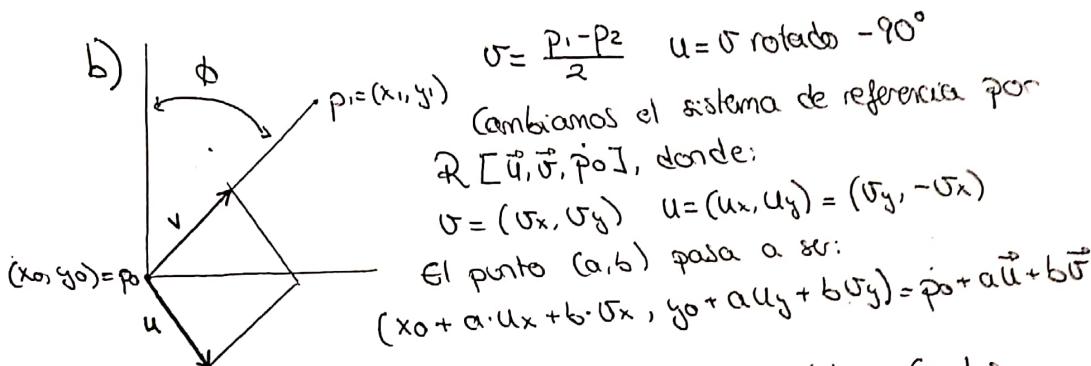
Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

②6) a) NOTA 1: atan2($\Delta x, \Delta y$) hace la arctangente bien, ya que tiene en cuenta los signos de Δx y Δy para devolver el ángulo (en radianes) en el cuadrante correcto.

NOTA 2: El escalado y la rotación pueden intercambiarse por ser el escalado uniforme. Si no, el escalado iría primero.

NOTA 3: El factor de esc = $\frac{\|p_1 - p_0\|}{2}$ porque la longitud del gancho (2 unidades) la lleva a $\frac{\|p_1 - p_0\|}{2}$, luego cada unidad de escala $\frac{\|p_1 - p_0\|}{2}$



②7) Para pintar una figura con n ganchos. Los vértices (puntos iniciales y finales de éstos) serán las raíces n-ésimas de la unidad.

Como el primer gancho de la figura no empieza en el (1,0,0), como que empieza a mitad, metemos un desfase al ángulo de la mitad del ángulo que usaremos:

void ensenaje(int n){

float alpha, beta;

for (int i=0; i<n; i++) {

alpha = (float)(i-0.5)*2.0*M_PI/n;

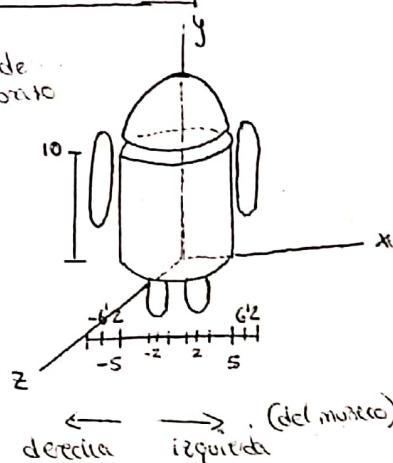
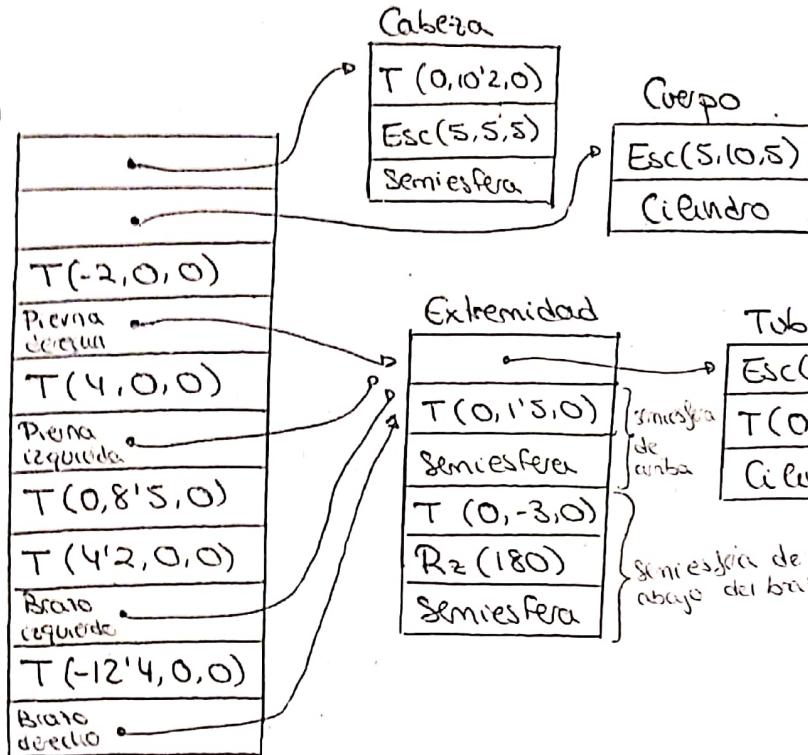
beta = (float)(i+0.5)*2.0*M_PI/n;

ganchos - 2p - b(cos(alpha), sin(alpha), cos(beta), sin(beta));

}

en vez de ser i, (i+1), con el
desfase queda (i-0.5), (i+0.5).

(29) a)



b) void Tubo()

```
glPushMatrix();
glScalef(1, 3, 1);
glTranslatef(0, -0'5, 0);
Cilindro();
glPopMatrix();
```

```
} void Extremidad() {
```

```
glPushMatrix();
Tubo();
glTranslatef(0, 1'5, 0);
Semiesfera();
glTranslatef(0, -3, 0);
glRotatef(180, 0, 0, 1);
Semiesfera();
glPopMatrix();
```

```
} void Cabeza() {
```

```
glPushMatrix();
glTranslatef(0, 10'2, 0);
glScalef(5, 5, 5);
Semiesfera();
glPopMatrix();
```

```
}
```

(ejer 29)

```
void Extremidad(float alpha) {
glPushMatrix();
glTranslatef(0, 1'5, 0);
glRotatef(alpha, 0, 0, 1);
glTranslatef(0, -1'5, 0);
: }
```

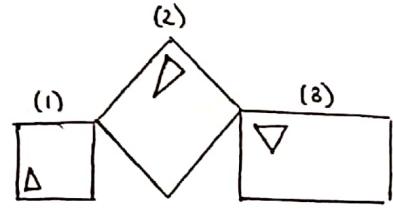
(ejer 30)

```
void Cabeza(float phi) {
glPushMatrix();
glRotate(phi, 0, 1, 0);
: }
```

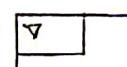
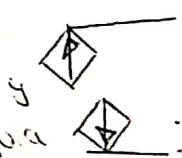
(29) b) void Cuerpo() {
 glPushMatrix();
 glScalef(5, 10, 5);
 Cilindro();
 glPopMatrix();
}

(ejer 30)

void Android() { //void Android effekt alpha, kai beta, fka. fka. fka.
 Cabeza(); //Cabeza (phi)
 Cuerpo();
 glTranslatef(-2, 0, 0);
 Extremidad(); //Extremidad (0)
 glTranslatef(4, 0, 0);
 Extremidad(); //Extremidad (0)
 glTranslatef(0, 8.5, 0);
 glTranslatef(4.2, 0, 0);
 Extremidad(); //Extremidad (alpha)
 glTranslatef(-12.4, 0, 0);
 Extremidad(); //Extremidad (beta)
}



(31) void figura_completa() {
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 glPushMatrix();
 glScalef(0.5, 0.5, 0.5); //Rescala todo la figura
 figura_simple(); (1)
 glPushMatrix();
 glTranslatef(2, 2, 0); //Reflejo respecto eje y
 glScalef(1, -1, 1); //Reflejo respecto eje y
 glRotatef(45, 0, 0, 1); //Rota la figura
 glScalef(sqrt(2), sqrt(2), sqrt(2));
 figura_simple(); (2)
 figura_simple(); (3)
 glPopMatrix();
 glPopMatrix();
 glTranslatef(3, 1, 0);
 glScalef(2, -1, 1); //Escala y reflejo a la vert ~
 figura_simple(); (3)
 glPopMatrix();
}



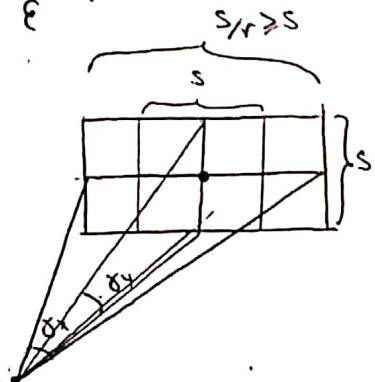
(32) void figura-compleja-rec (int altura){
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 rec(altura);
 }

void rec (int altura){
 if (altura == 0) // Si soy de los chiquitos, me dibujo
 figura-simple();

else {
 figura-simple();
 glTranslatef(1, 0, 0);
 glScalef(0.5, 0.5, 0.5);
 rec(altura - 1);
 glTranslatef(0, 1, 0);
 rec(altura - 1);
 }

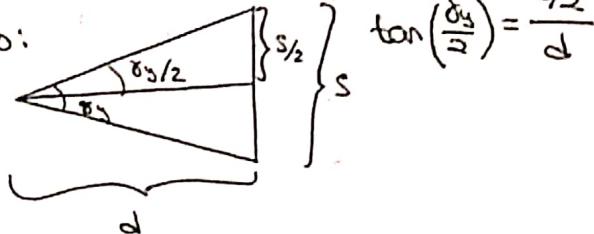
// Si no:
 Dibujo el fractal de altura - 1, lo
 desplazo para arriba y dibujo
 debajo otro igual.
 deabajo izq, lo traslado y "me
 escalo" (red), lo traslado y "me
 dibujo" (dibujo al grande).

(41)



$$\text{si: ancho} > \text{alto} \Rightarrow r = \frac{\text{alto}}{\text{ancho}} \leq 1 \quad y \quad \delta_m = \delta_y$$

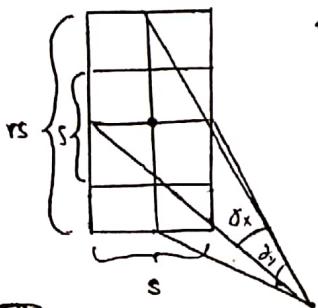
por tanto:



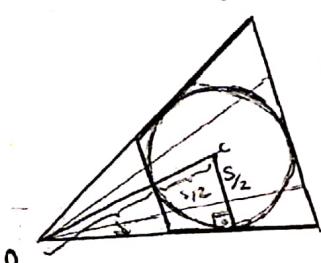
$$\text{si alto} < \text{ancho} \Rightarrow r > 1: \quad y \quad \delta_m = \delta_x$$

La manera de sacar d es análoga

$$\tan\left(\frac{\delta_m}{2}\right) = \frac{s/2}{d} \Rightarrow d = \frac{s}{2 \tan\left(\frac{\delta_m}{2}\right)}$$



(42)



Como en el (41), lo único que necesitas
 para aplicar eso es calcular la d de
 forma diferente.

$$\sin\left(\frac{\delta_m}{2}\right) = \frac{s/2}{d+s/2} \Rightarrow d = \frac{s/2}{\sin\left(\frac{\delta_m}{2}\right)} - \frac{s}{2}$$

Mismo código que en el (41) y solo cambio el d.

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

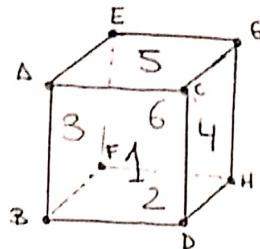
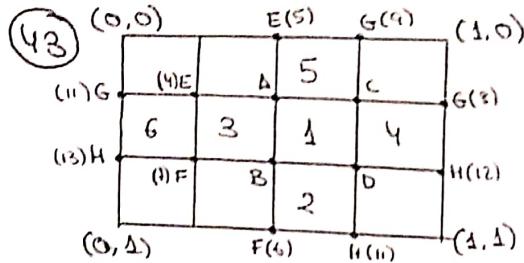
No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita



- Recorrer los triángulos en sentido antihorario
- Recorrer los triángulos en sentido horario

a) El modelo tendrá como mínimo 14 vértices.
(El punto A tiene una única consideración de textura,
pero el F tiene 2, por eso hay que duplicarlo).

b) Dado:: Dado ()

vertices = {
 <0,0,1> //A 0
 <0,0,0> //B 1
 <1,1,1> //C 2
 <1,0,1> //D 3
 <0,1,0> //E 4
 <0,1,1> //F 5
 <0,0,0> //F 6
 <0,0,0> //F 7
 <1,1,0> //G 8
 <1,1,0> //G 9
 <1,1,0> //G 10
 <1,0,0> //H 11
 <1,0,0> //H 12
 <1,0,0> //H 13
 }

triangulos = {
 <0,1,2> //Cara 1
 <1,3,2>
 <3,1,6> //Cara 2
 <6,11,3>
 <0,4,7> //Cara 3
 <0,7,1>
 <2,3,12> //Cara 4
 <2,12,8>
 <0,2,5> //Cara 5
 <5,2,9>
 <4,10,7> //Cara 6
 <10,13,7>
 }

cc_tt_ver = {
 <0,5,1>, //A
 <0,5,2>, //B
 <0,7,1>, //C
 <0,7,2>, //D
 <0,2,5>, //E
 <0,5,0>, //F
 <0,5,2>, //G
 <0,2,5>, //H
 <1,1>, //A
 <0,7,5,0>, //G
 <0,1,2>, //G
 <0,7,5,2>, //H
 <2,2>, //H
 <0,2,3>, //H
 }

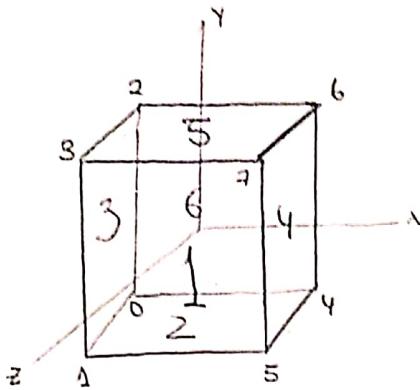
NodoDado:: NodoDado ()

```
Textura * text = new Textura("dce.jpg");
Material * mat = new Material(1, 1, 1, 1);
agregar(mat);
agregar(new Dado);
}
```

(44)

			10	14		
		2	5	15		
22	18	2	3	19	23	7
6	3	1	17	21	5	4
20	16	0				4
			9	13		
			2	12		
			8			

Con el cubo de
centro $(0,0,0)$ y
lado 2



Para que funcione necesito triplicar todos los vértices, teniendo 24 en total (porque necesito que las normales estén bien calculadas).

Dado 24:: Dado 24(): mallaInd ("Dado24")

vertices = {
 $\langle -1, -1, -1 \rangle // 0, 8, 16$
 $\langle -1, -1, +1 \rangle // 1, 9, 17$
 $\langle -1, +1, -1 \rangle // 2, 10, 18$
 $\langle -1, +1, +1 \rangle // 3, 11, 19$
 $\langle 1, -1, -1 \rangle // 4, 12, 20$
 $\langle 1, -1, +1 \rangle // 5, 13, 21$
 $\langle 1, +1, -1 \rangle // 6, 14, 22$
 $\langle 1, +1, +1 \rangle // 7, 15, 23$
 \vdots
}

f;

cc-tt-ver = {
 $\langle 1/4, 2/3 \rangle // 0$
 $\langle 1/2, 2/3 \rangle // 1$
 $\langle 1/4, 1/3 \rangle // 2$
 $\langle 1/2, 1/3 \rangle // 3$
 $\langle 1, 2/3 \rangle // 4$
 $\langle 3/4, 2/3 \rangle // 5$
 $\langle 1, 1/3 \rangle // 6$
 $\langle 3/4, 1/3 \rangle // 7$
 $\langle 1/2, 1/4 \rangle // 8$
 $\langle 1/2, 2/3 \rangle // 9$
 $\langle 1/2, 0 \rangle // 10$
 $\langle 1/2, 1/3 \rangle // 11$
 $\langle 3/4, 1/4 \rangle // 12$
 $\langle 3/4, 2/3 \rangle // 13$
 $\langle 3/4, 0 \rangle // 14$
 $\langle 3/4, 1/3 \rangle // 15$
}

$\langle 1/4, 2/3 \rangle // 16$
 $\langle 1/2, 2/3 \rangle // 17$
 $\langle 1/4, 1/3 \rangle // 18$
 $\langle 1/2, 1/3 \rangle // 19$
 $\langle 0, 2/3 \rangle // 20$
 $\langle 3/4, 2/3 \rangle // 21$
 $\langle 0, 1/3 \rangle // 22$
 $\langle 3/4, 1/3 \rangle // 23$

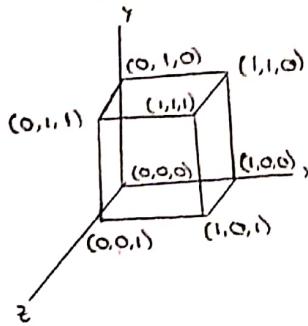
f;

NORMALES DE LOS VÉRTICES:

$0, 1, 2, 3: \langle -1, 0, 0 \rangle$
 $4, 5, 6, 7: \langle 1, 0, 0 \rangle$
 $8, 9, 12, 13: \langle 0, -1, 0 \rangle$
 $10, 11, 14, 15: \langle 0, 1, 0 \rangle$
 $16, 18, 20, 22: \langle 0, 0, -1 \rangle$
 $17, 19, 21, 23: \langle 0, 0, 1 \rangle$

f

(45) (Cubo 24)

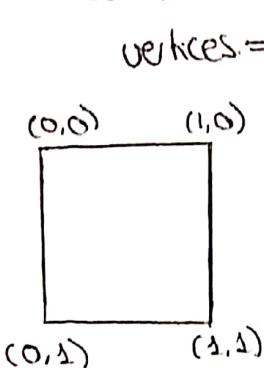


cc-tt-ver2

$\{0,1\}$ //0
 $\{1,1\}$ //1
 $\{0,0\}$ //2
 $\{1,0\}$ //3
 $\{0,1\}$ //4
 $\{1,1\}$ //5
 $\{0,0\}$ //6
 $\{1,0\}$ //7
 $\{0,1\}$ //8
 $\{1,0\}$ //9
 $\{0,0\}$ //10
 $\{1,1\}$ //11
 $\{0,1\}$ //12
 $\{1,1\}$ //13
 $\{0,0\}$ //14
 $\{1,1\}$ //15
 $\{0,0\}$ //16
 $\{1,0\}$ //17
 $\{0,1\}$ //18
 $\{1,1\}$ //19
 $\{0,0\}$ //20
 $\{1,1\}$ //21
 $\{0,0\}$ //22
 $\{1,0\}$ //23

8;

(Cubo 24): MallaInd ("Cubo 24")



vertices = {

{0,0,0}	//0
{0,0,1}	//1
{0,1,0}	//2
{0,1,1}	//3
{1,0,0}	//4
{1,0,1}	//5
{1,1,0}	//6
{1,1,1}	//7

{0,0,0}	//8
{0,0,1}	//9
{0,1,0}	//10
{0,1,1}	//11
{1,0,0}	//12
{1,0,1}	//13
{1,1,0}	//14
{1,1,1}	//15
{0,0,0}	//16
{0,0,1}	//17
{0,1,0}	//18
{0,1,1}	//19
{1,0,0}	//20
{1,0,1}	//21
{1,1,0}	//22
{1,1,1}	//23

(46) a) La reflectividad del material en P (pag 96) da la iluminación en cada punto viene dada por:

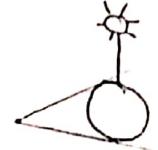
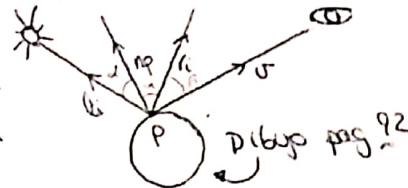
$$C_i = M_s(p) + M_d(p) \max(0, n \cdot l_i) + M_g(p) \cdot d_i [\max(0, n \cdot v)]^e$$

$$\text{En este caso, } C_i = \max(0, n \cdot l_i) \times (1, 1, 1)$$

Como $n \cdot l_i = \|n\| \|l_i\| \cos(\alpha)$ donde α es el ángulo que forman, el punto con máxima iluminación es aquel tal que $d_i = 0$. ($n = l_i$)

Por tanto, el punto con máxima iluminación es el $(0, 1, 0)$, el polo norte de la esfera.

Ese punto no es visible por el observador, que está en el punto $(2, 0, 0)$, solo puede ver hasta el cono tangente a la esfera.



b) Modelo de Blinn-Phong (pag 93) da reflectividad pseudo-especular del material se obtiene:

$$f_{rs}(p, v, l_i) = M_s(p) d_i [n_p \cdot u_i]^e$$

El punto de máxima iluminación d_i es 1 si $n_p \cdot l_i > 0 \Rightarrow$ el punto de máxima iluminación está en el ~~hemisferio~~ hemisferio norte.

Este es el caso. $M_s(p)$ es $(1, 1, 1)$ en nuestro caso. Para maximizar $f_{rs}(p, v, l_i)$ hay que maximizar $n_p \cdot u_i$, y eso es máximo cuando el ángulo que forman es 0, es decir, cuando $u_i = n_p$.

$$u_i = \frac{l_i + v}{\|l_i + v\|} \text{ bisectriz de } l_i \text{ y } v.$$

$$l_i = \overrightarrow{p \cdot q_i} = \frac{(-p_x, 2-p_y, -p_z)}{\|(-p_x, 2-p_y, -p_z)\|}$$

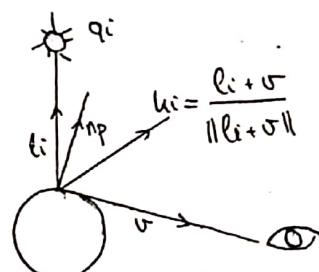
$$v_i = \overrightarrow{p \cdot o} = \frac{(2-p_x, -p_y, -p_z)}{\|(2-p_x, -p_y, -p_z)\|}$$

$$\text{Tomo } p = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right), n_p = \overrightarrow{0p} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right)$$

$$l_i = \left(-\frac{\sqrt{2}}{2}, 2 - \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right) / 1.6345 \quad \Rightarrow \quad u_i = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \Rightarrow u_i = n_p$$

$$v_i = \left(2 - \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right) / 1.6345$$

Por tanto p es un punto de máxima iluminación.



(a nosotros por suerte nos pasa)

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolitah
Tu que eres tan bonita

(46) b) PARA SABER SI UN PUNTO ES VISIBLE:

Trazaremos la recta que pasa por el punto y por el observador, y la intersectaremos con la esfera.

De las soluciones del sistema, solo es observable la solución más cercana al observador.

do recto que une $P = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$ com $(2,0)$ é:

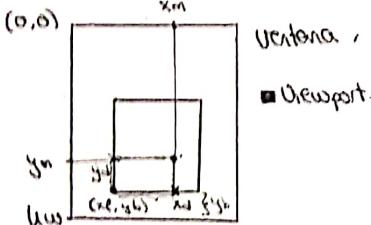
$$y = \frac{i\sqrt{5}}{4-\sqrt{5}}(x-2), \text{ luego:}$$

$$\begin{cases} x^2 + y^2 = 1 \\ y = \frac{-\sqrt{2}}{4-\sqrt{2}}(x-2) \end{cases}$$

TGMA 4

(47) Coordenadas de dispositivo:

Son aquellas relaciones al viewer



al viewport.
Pasamos las coordenadas en función de la ventana (la ventana superior es el $(0,0)$ y la y crece para abajo) a coordenadas relativas al viewport ((x_l, y_l) , es el $(0,0)$) y la y crece para arriba) → relativas a la esquina inferior izq.

$$x_d = x_m - x_l$$

$$\begin{aligned} x_d &= x_m - x_l \\ y_d &= y_w - y_m - y_b \quad (\text{en pixels}) \end{aligned}$$

Las preferencias de dispositivos: la variedad

• Consideradas normalizadas de acuerdo a los de dispositivos

Normalizamos los coordenados de cada 1"

$$x_{ndc} = x_d / \omega$$

$$y_{\text{rect}} = y_d / h$$

• Consideradas de mundo:

de mundo: El viewport es la cara delantera del view-frustum, que está en el mundo. Como las coordenadas normalizadas de dispositivo son ~~esta~~ la proporción donde está el punto si el ancho y alto del dispositivo.

$\rightarrow \text{Xndc}(\tau - t)$
populer arcto latif

$$y_w = b + \underbrace{y_{ndc}}_{\text{proporción alto total}} (t - b)$$

Sumamos (l, b) porque no mediamos respecto al $(0, 0)$.

TEMAS 5

48 (Pág 62)

El rayo parte de O y tiene vector director d , luego la ec. de la semirecta que lo describe es: $p(t) = O + t d$ con $t > 0$.

Si el rayo es paralelo al plano que contiene al triángulo \Rightarrow no hay intersección.

\Rightarrow no hay intersección.

Si no es paralelo \Rightarrow interseca con el plano, hay que ver:

Si la intersección cae dentro o fuera del triángulo.

Todo punto del triángulo tiene las coordenadas barycentricas:

$$q = a(\vec{v}_1 - \vec{v}_0) + b(\vec{v}_2 - \vec{v}_0) + \vec{v}_0 \quad a \geq 0, b \geq 0, a+b \leq 1$$

La intersección con el plano se da si $\exists t > 0$ tal que:

$$(*) O + t d = \vec{v}_0 + a(\vec{v}_1 - \vec{v}_0) + b(\vec{v}_2 - \vec{v}_0)$$

Tomamos el sistema de referencia $R[\vec{v}_1 - \vec{v}_0, \vec{v}_2 - \vec{v}_0, \vec{n}, \vec{v}_0]$, donde \vec{n} es la normal del triángulo. En este sistema de referencia, el triángulo está en el plano $z=0$.

A partir de ahora trabajamos con coordenadas respecto a dicho sistema de referencia.

$O R = (0_x, 0_y, 0_z) \quad d_R = (d_x, d_y, d_z)$. Luego, (*) quedaría:

$$O R + t d_R = (a, b, 0). \quad \text{El sistema a resolver es:}$$

$$\begin{cases} 0x + t d_x = a \\ 0y + t d_y = b \\ 0z + t d_z = 0 \end{cases} \Rightarrow$$

1) Si $d_z = 0 \Rightarrow$ el rayo es paralelo al plano (o está contenido), luego no hay intersección.

2) Si $d_z \neq 0$:

$$t = \frac{-0z}{dz}, \quad a = 0x + \frac{-0z}{dz} d_x, \quad b = 0y - \frac{0z}{dz} d_y.$$

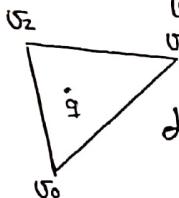
El rayo interseca con el triángulo si:

$$-a + b \leq 1$$

$$-a \geq 0, b \geq 0$$

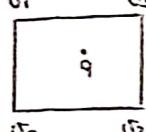
$$-t > 0$$

Pasamos $(a, b, 0)$ al sistema de coordenadas del mundo y calculamos su distancia con O . (coordenadas de q y t pedidos)



(48) AMPLIACIÓN:

Si en vez de un triángulo tengo un paralelogramo,
los puntos del cuadrado son:



$$q = a(v_1 - v_0) + b(v_2 - v_0) + v_0 \text{ con } a \geq 0, b \geq 0, \max\{a, b\} \leq 1$$

(porque en mi sistema de referencia $\|v_2 - v_0\| = \|v_1 - v_0\| = 1$)

(49) Sea $q = a(v_1 - v_0) + b(v_2 - v_0) + v_0$ el punto de intersección dentro del triángulo, entonces:

$$q = (1-a-b)v_0 + av_1 + bv_2$$

Luego su normal va a ser:

$$n_q = \frac{(1-a-b)n_0 + an_1 + bn_2}{\|(1-a-b)n_0 + an_1 + bn_2\|}$$

Del mismo modo, sus coordenadas de texture serán:

$$cc_tt_q = (1-a-b)cc_tt_0 + acc_tt_1 + bcc_tt_2$$

O los del color:

$$c_q = (1-a-b)c_0 + ac_1 + bc_2$$

(50) Superficie de la esfera: $\lambda p |F(p)| = 0$ con

$$F(p) = \|p - c\|^2 - r^2, \text{ con } p = o + td, t > 0.$$

$F(p) = \|o + td - c\|^2 - r^2$, con $p = o + td, t > 0$

$$F(p) = \begin{cases} < 0 & \text{si } p \text{ en el interior de la esfera} \\ = 0 & \text{" " en la esfera} \\ > 0 & \text{" " el exterior de la esfera} \end{cases}$$

La intersección del rayo con la esfera es la solución

$$\text{de } \|o + td - c\|^2 - r^2 = 0 \Rightarrow$$

$$\text{de } \|o + td - c\|^2 - r^2 = 0 \Rightarrow (ox + tdx - cx)^2 + (oy + tdy - cy)^2 + (oz + tdz - cz)^2 - r^2 = 0$$

$$\Rightarrow (ox + tdx - cx)^2 + (oy + tdy - cy)^2 + (oz + tdz - cz)^2 = r^2$$

Vemos si $\exists t > 0$ que sea posible

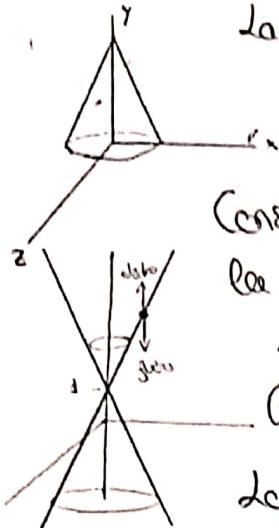
Vemos si hay 2 soluciones: tomamos la más pequeña

- Si hay 2 soluciones: toma la más pequeña

- Si hay 1 solución: esa es la intersección

- Si no hay solución: no hay intersección

(51) • Cono: Habrá como máximo 2 intersecciones.



La ecuación del cono es:

$$\left\{ \begin{array}{l} x^2 + z^2 = (y-1)^2 \\ \text{cono} \end{array} \right. \quad 0 \leq y \leq 1 \quad \cup \quad \left\{ \begin{array}{l} x^2 + z^2 \leq 1, y=0 \\ \text{base} \end{array} \right.$$

Consideramos el cono infinito $x^2 + z^2 = (y-1)^2$, y

la función $F(x, y, z) = x^2 + z^2 - (y-1)^2$

$$F(x, y, z) = \begin{cases} < 0 & \text{si } (x, y, z) \text{ fuera del cono infinito} \\ = 0 & " " \quad \text{en el cono infinito} \\ > 0 & " " \quad \text{dentro del cono infinito} \end{cases}$$

(signo constante)

La intersección del rayo $0+td$ con el cono infinito es la solución de la ecuación:

$$(0x + tdx)^2 + (0z + tdz)^2 - (0y + tdy - 1)^2 = 0$$

tales que $t > 0$.

De esas soluciones, nos quedamos con aquellas tales que $0y + tdy \in [0, 1]$, que son las que intersectan con nuestro cono original.

* Si salen 2 soluciones distintas de este tipo, estas son nuestras intersecciones.

* En otro caso, calcularemos los puntos de corte con la tapa.

- Si $dy = 0$, nunca corta la tapa (paralelo a ésta).

- Si $dy \neq 0$: Tomo $t_0 = \frac{-0y}{dy}$ (despejo de $0y + tdy = 0$)

Si $t_0 < 0$, no corta la tapa

Si otro caso, habrá intersección con la tapa

$$\text{Si } (0x + t_0 dx)^2 + (0z + t_0 dz)^2 \leq 1. \quad (x^2 + z^2 \leq 1)$$

Si $(0x + t_0 dx)^2 + (0z + t_0 dz)^2 > 1$

De las soluciones obtenidas, la intersección del rayo

con el objeto será la más cercana al observador.

(la que tiene el t menor)

Que no te escriban poemas de amor
cuando terminen la carrera ➡➡➡➡➡



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

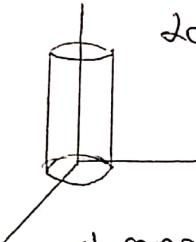
Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

51

• Cilindro: Habrá como máximo 2 intersecciones



La ecuación de nuestro cilindro es:

$$\begin{cases} x^2 + z^2 = 1, 0 \leq y \leq 1 \\ x^2 + z^2 \leq 1, y = 0 \end{cases}$$
$$x^2 + z^2 \leq 1, y = 1$$

Consideramos el cilindro infinito $x^2 + z^2 = 1$ y
el campo escalar $F(p) = \begin{cases} < 0 & \text{si } p \text{ está dentro del cilindro} \\ = 0 & \text{si } p \text{ está en el cilindro} \\ > 0 & \text{si } p \text{ " fuera del cilindro} \end{cases}$

con $F(x, y, z) = x^2 + z^2 - 1$.
La intersección del rayo $O + t\vec{d}$ con el cono infinito es la
solución de la ecuación:

$$(Ox + t\vec{d}x)^2 + (Oz + t\vec{d}z)^2 - 1 = 0 \quad (\text{tales que } t > 0)$$

De estas soluciones nos quedamos con las que
son las que intersectan con
 $Oy + t\vec{d}y \in [0, 1]$, que son las que intersectan con
nuestro cilindro original.

* Si salen dos soluciones distintas de este tipo, estas
son nuestras intersecciones.

* En otro caso, calculamos los puntos de corte con
las tapas.

- Si $\vec{d}y = 0$: nunca corta las tapas (paralelo a estas)

- Si $\vec{d}y \neq 0$:

Tapa inferior: Está en el plano $y=0$. Despejo de $Oy + t\vec{d}y = 0$.

Tomo $t_0 = \frac{-Oy}{\vec{d}y}$. Si $t_0 < 0$, no corta la tapa.

En otro caso, habrá intersección con la tapa si:

$$(Ox + t\vec{d}x)^2 + (Oz + t\vec{d}z)^2 \leq 1$$

($Ox + t\vec{d}x)^2 + (Oz + t\vec{d}z)^2 \leq 1$ Despejo de $Oy + t\vec{d}y = 1$:

Tapa superior: Está en el plano $y=1$. Despejo de $Oy + t\vec{d}y = 1$.

Tomo $t_0 = \frac{1-Oy}{\vec{d}y}$. Si $t_0 < 0$, no corta la tapa.

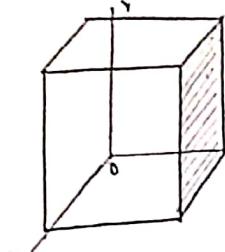
En otro caso, habrá intersección con la tapa si:

$$(Ox + t\vec{d}x)^2 + (Oz + t\vec{d}z)^2 \leq 1$$

De las soluciones obtenidas, la intersección del rayo
con el objeto será la que tenga la t menor.

• **Cubo:** Consideramos el cubo de lado dos y centro

$(0,0,0)$. Habrá como máximo dos intersecciones con el cubo, mientras que no las encontramos, seguimos buscando.



Para cada cara del cubo, hay que ver si el rayo

interseca con esta.

Tomemos por ejemplo la cara π , situada en el plano $x=1$.

Buscamos las soluciones de $0x + tdx = 1$, que será la intersección del rayo con el plano. Si $dx=0$, paralelo,

no cortará ni el plano $x=1$ ni el $x=-1$.

Si $dx \neq 0$, tomo $t_0 = \frac{1-0x}{dx}$. Si $t_0 < 0$, no hay

intersección con el plano.

Si $t_0 > 0$, compruebo si cae dentro de la cara del

cubo, es decir que:

$$-1 < 0y + t_0 dy < 1 \quad y \quad -1 < 0z + t_0 dz < 1.$$

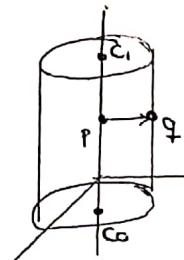
(Este es igual para todas las caras, repetir hasta encontrar 2 intersecciones, que es el máximo que puede haber, aunque puede haber menos).

De las soluciones encontradas, nos quedamos con aquella con t menor.

⑤ La idea es normalizar el objeto y aplicarle esas mismas transformaciones al rayo. Usamos los ejercicios anteriores para calcular la intersección y a esa solución le aplicamos las transformaciones inversas.

(Lo hacemos con un cambio de sistema de referencia)

53) • Cilindro:



sea q el punto donde quiero calcular la normal.

- Si q no está en las tapas:

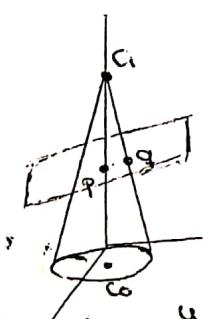
Tomo el plano paralelo a las tapas que pasa por q . Tomo el punto p : corte de ese plano con el eje del cilindro. La normal es el vector \vec{pq} normalizado.

NOTA: El eje del cilindro es la recta que pasa por uno de sus centros, (x_0, y_0, z_0) y tiene de vector director la resta de ambos centros, $\sigma = (x_0, y_0, z_0) - (x_1, y_1, z_1)$.

NOTA: Sean v_0, v_1, v_2 puntos de un plano, $n = (v_1 - v_0) \times (v_2 - v_0)$ vector normal al plano de coordenadas (n_1, n_2, n_3) . La ecuación del plano es: $n_1 x + n_2 y + n_3 z + D = 0$, sustituyendo en un punto del plano, obtengo el D .

- Si q está en la tapa:
 - si está en la tapa de arriba, la normal es $c_1 - c_0 (\vec{c_0 c_1})$ normalizado.
 - si está en la tapa de abajo, la normal es $c_0 - c_1 (\vec{c_1 c_0})$ normalizado.

• Cono:



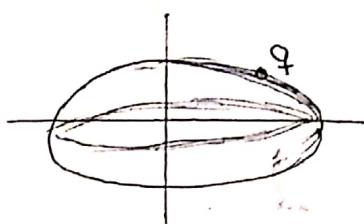
- Si q está en la tapa:
 - su normal es $c_0 - c_1 (\vec{c_1 c_0})$ normalizado.

- Si q no está en la tapa:

Calculo el plano perpendicular a $\vec{q c_1}$ que pasa por q (el que tiene a $\vec{q c_1}$ como normal), y su intersección con el eje del cono, p .

La normal será el vector \vec{pq} normalizado.

• Elipsoide:



* Esfera:

La normal es el vector que une el centro con el punto q normalizado.

* Elipsoide: Tomamos q punto en la elipsoide.

Sea A la matriz que transforma la esfera en el elipsoide.

Como las traslaciones no afectan a la normal

podemos suponer que A es de 3×3 .

Sea n la normal en la esfera de $A^{-1}q$, entonces

la normal de q es $n_q = (A^T)^{-1} \cdot n$ (Tema 2, pag 78)

normalizada.

54) (Tema 3, pag 22 y 23)

Rasterización:

para cada primitiva $P \neq \emptyset$
encontrar el conjunto S de píxeles $\sim P$

El algoritmo tiene complejidad $O(n \cdot p)$.

Raytracing:

para cada pixel $q \sim P$
para cada primitiva $\sim P \} \max(n, n) = n$

calcular T , conjunto de primitivas $\sim P$.

para cada primitiva $\sim P$.

Complejidad $O(n \cdot p)$, pero con indexación especial,
~~supuestamente~~ para cálculo de T en cada pixel,
~~supuestamente~~ conseguimos $O(p \log(n))$.