

Informática Gráfica: Teoría. Tema 1. Introducción.

Carlos Ureña

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

2018-19

Teoría. Tema 1. Introducción.

Índice.

- 1 Introducción
- 2 El proceso de visualización
- 3 La librería OpenGL. Un programa sencillo.
- 4 Programación básica del cauce gráfico
- 5 Apéndice: puntos, vectores, marcos, coordenadas y matrices.

Sección 1

Introducción

1.1. Concepto y metodologías

1.2. Aplicaciones.

Subsección 1.1

Concepto y metodologías

El término *Informática Gráfica*

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

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

En este curso introductorio nos centraremos esencialmente en:

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

Áreas científicas implicadas

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

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

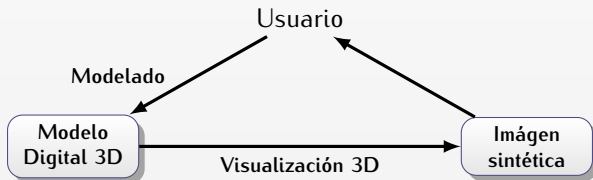
en aplicaciones específicas, se usan otros campos de la Informática en particular o la Ciencia en general (p.ej. para desarrollo de videojuegos se usan también técnicas de Inteligencia Artificial).

Informática Gráfica 3D interactiva

Los elementos esenciales de una aplicación gráfica son:

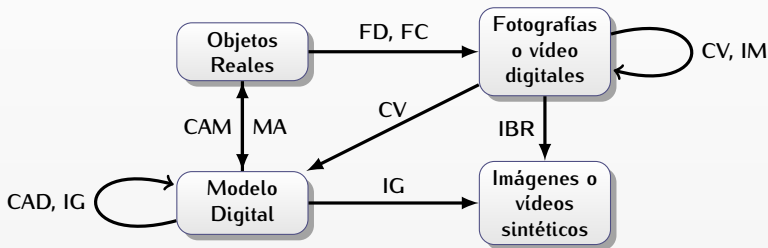
- ▶ **Modelos digitales** de objetos reales, ficticios o de datos
- ▶ **Imágenes o vídeos digitales** que se usan para visualizar dichos objetos.

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



Informática Gráfica y Computación Visual

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



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

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

Subsección 1.2

Aplicaciones.

Aplicaciones

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

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

Videojuegos



Fotograma del videojuego *Watch Dogs* de Ubisoft.

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

Realidad Aumentada (AR)



Imagen:

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

Películas y animaciones generadas por ordenador

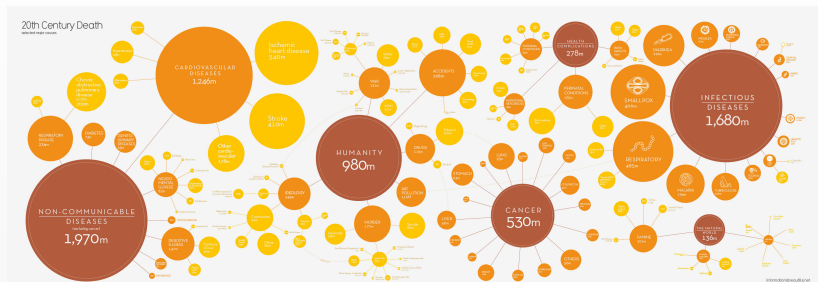


Fotograma del tráiler cinematográfico del videojuego *Watch Dogs*. Imagen creada por DigiC Pictures para Ubisoft, usando Arnold de Solid Angle.

Img:  <http://www.fxguide.com/featured/the-state-of-rendering-part-2/#arnold>.

Vídeo:  <http://www.youtube.com/watch?v=xLLHYBlyBb8>

Visualización de datos



Frecuencia de causas de muerte en el siglo XX:

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

Visualización en Medicina



Obtenido del sitio web *MIT Technology Review*

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

Cirujía asistida con Realidad Aumentada

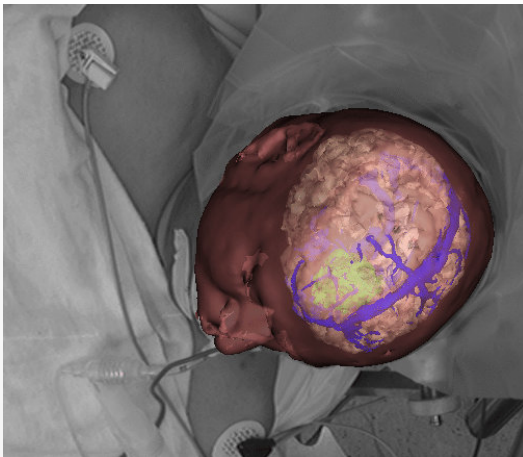
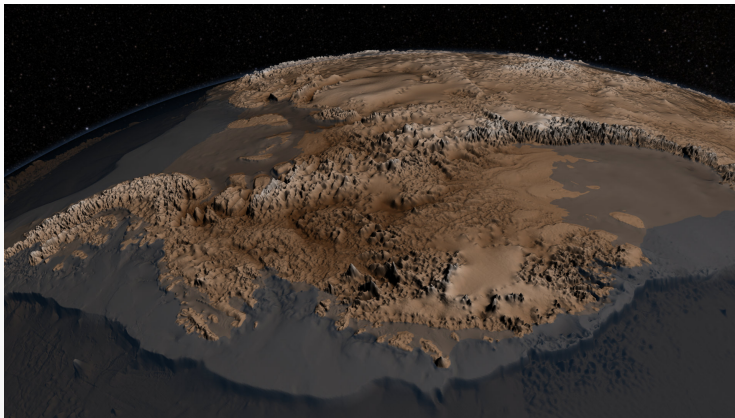


Imagen creada por Christopher Brown, Universidad de Rochester:

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

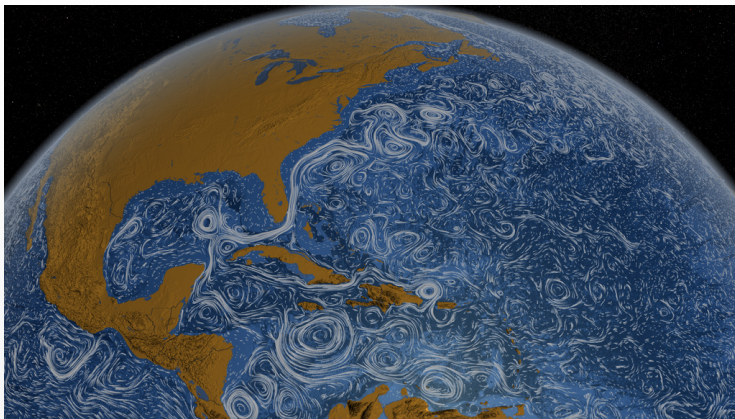
Visualización científica (geología)



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

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

Visualización científica (climatología)



Visualización de las corrientes oceánicas (NASA):

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

Simuladores y entrenamiento



Simulador de conducción de Mercedes-Benz:

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

Patrimonio histórico



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

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

Sección 2

El proceso de visualización

- 2.1. Introducción
- 2.2. Rasterización y ray-tracing.
- 2.3. El cauce gráfico en rasterización.

Subsección 2.1

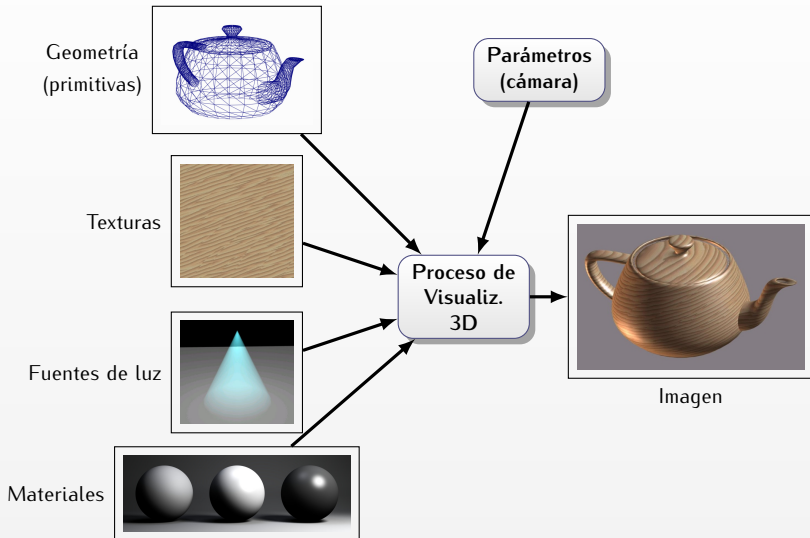
Introducción

El proceso de visualización 3D: entradas.

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

El proceso de visualización 3D: esquema.



Subsección 2.2

Rasterización y ray-tracing.

Visualización basada en *rasterización*

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

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

- ▶ Las primitivas constituyen 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 del orden del número de primitivas por el número de pixels (tiempo en $O(n \cdot p)$)

Visualización basada en *Ray-Tracing*

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

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

- ▶ Cuando se trata de visualización 3D, la implementación de este 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.

Comparativa

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.

► 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.

Subsección 2.3

El cauce gráfico en rasterización.

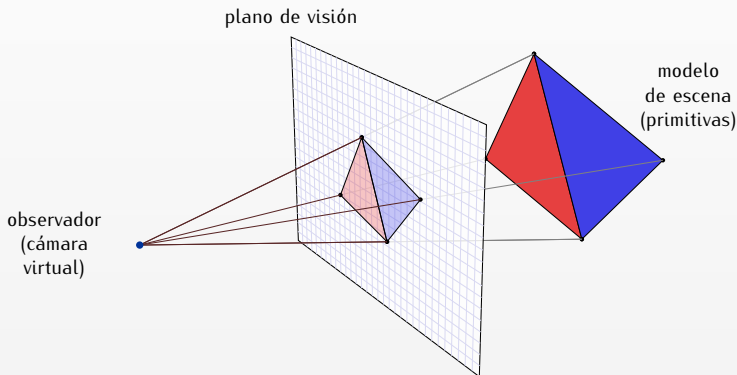
El cauce gráfico en rasterización

El término **cauce gráfico** hace referencia al conjunto de etapas de cálculo que se usan para rasterizar y visualizar cada primitiva:

- 1 Cada **primitiva** es **transformada** en diversos pasos hasta encontrar su proyección en el plano de la imagen.
 - ▶ este proceso depende de la geometría de la escena, y de la posición, orientación y características de la **cámara virtual** usada para obtener la imagen
- 2 La proyección es **rasterizada** (discretizada), y se encuentran los pixels que cubre.
 - ▶ en la visualización 3D, esto incluye calcular como se tapan las primitivas (polígonos) entre ellas.
- 3 **Sombreado**: en cada pixel cubierto se calcula el color que se le debe asignar:
 - ▶ para esto, se tiene en cuenta la primitiva: su **color**, el tipo de **material** que pretende imitar, las **texturas**, las **fuentes de luz** y otros parámetros de visualización.

Transformación y proyección

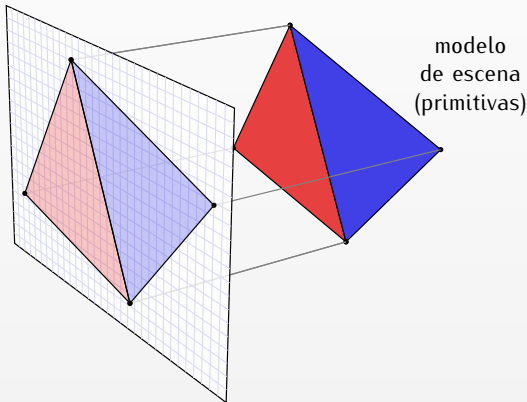
Cada primitiva se sitúan en su lugar en el espacio, y se encuentra su proyección en un plano imaginario (**plano de visión**, *viewplane*) situado entre el **observador** y la escena (las primitivas):



Proyección paralela

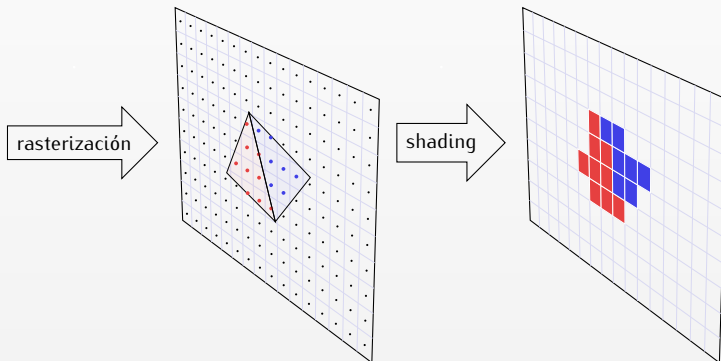
La proyección puede ser **perspectiva** (como en la transparencia anterior), o **paralela**, como aparece aquí:

plano de visión



Rasterización y sombreado

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



Sección 3

La librería OpenGL. Un programa sencillo.

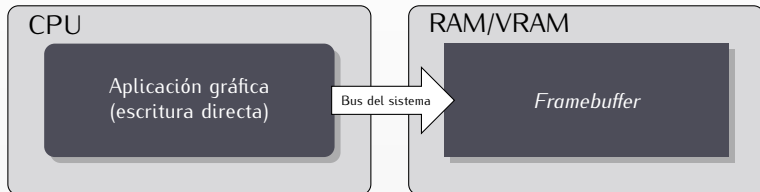
- 3.1. Aplicaciones y bibliotecas gráficas con GPUs.
- 3.2. La API OpenGL.
- 3.3. Programación y eventos en GLFW
- 3.4. Tipos de primitivas.
- 3.5. Visualización de primitivas.

Subsección 3.1

Aplicaciones y bibliotecas gráficas con GPUs.

Aplicaciones de escritura directa

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

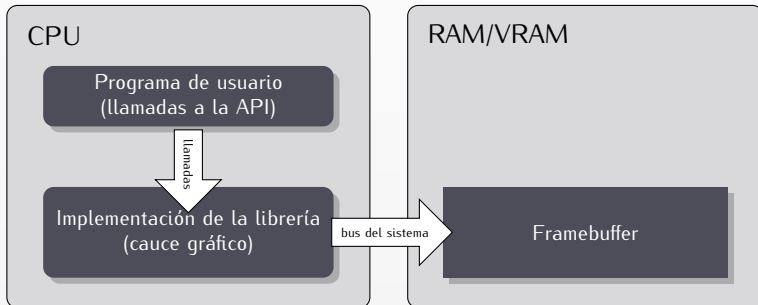


Desventajas

- ▶ 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.
- ▶ La aplicación gráfica no puede coexistir con otras, por ejemplo el gestor de ventanas.

Uso de APIs gráficas

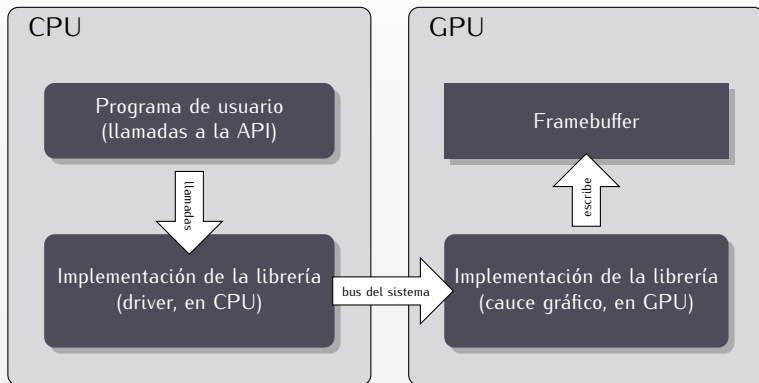
El uso de implementaciones de APIs gráficas estandarizadas y portables (junto con gestores de ventanas y eventos) proporciona portabilidad y posibilidad de acceso simultáneo de varias aplicaciones



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

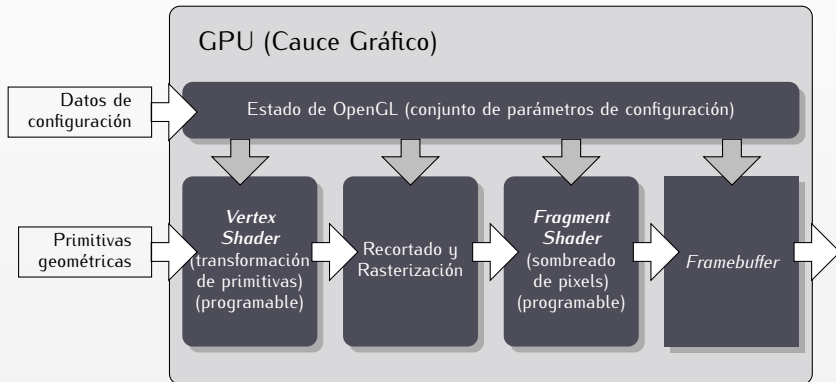
Uso de APIs y hardware gráfico (GPUs)

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



El cauce programable en GPUs

La GPU admite básicamente primitivas y datos de configuración de estado como entrada, y produce una señal de video para el monitor. El cauce gráfico consta de varias etapas (algunas de ellas programables):



APIs gráficas más usadas

- ▶ **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** (1995): parecida a OpenGL, diseñada por Microsoft para las plataformas Windows y Xbox, hay implementaciones de los fabricantes de GPUs. La versión 12 (2014) incluye funcionalidad avanzada de bajo nivel con baja sobrecarga.
- ▶ **OpenGL ES** (2003): subconjunto de OpenGL, orientado a dispositivos móviles.
- ▶ **WebGL** (2011): basada en OpenGL ES, diseñada para programas Javascript en navegadores.
- ▶ **Metal** (2014): parecida a OpenGL y DirectX 12, pero diseñada e implementada exclusivamente por Apple para macOS, iOS y tvOS.
- ▶ **Vulkan** (2016): sucesora de OpenGL, inspirada en DirectX 12 y Metal también diseñado por *Khronos group*,

Subsección 3.2

La API OpenGL.

La API OpenGL



- ▶ **OpenGL** es la **especificación** de un conjunto de funciones útil para visualización 2D/3D basada en rasterización (un documento con: funciones, sus parámetros y comportamiento).
- ▶ Permite la rasterización de primitivas de bajo nivel (polígonos), 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.
- ▶ La principal alternativa de igual nivel es **Direct X** (Microsoft), solo para Windows.

Características de OpenGL

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

Historia de OpenGL.

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

Bibliotecas complementarias: GLU y GLFW

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

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

GLFW es una librería auxiliar de OpenGL, es portable, sirve para:

- ▶ Gestión de ventanas.
- ▶ Gestión de eventos de entrada.

Los nombres de las funciones de GLU comienzan con `glu` y las de GLFW con `glfw`.

Subsección 3.3

Programación y eventos en GLFW

Eventos y sus tipos

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

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

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

Funciones gestoras de eventos (*callbacks*)

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

- ▶ Tras invocar a una de estas funciones, se dice que el correspondiente evento ya ha sido **procesado o gestionado**.
- ▶ En el programa se puede establecer que tipos de eventos se deben procesar y que funciones deben hacerlo.
- ▶ Para cada tipo de evento, la función que lo gestione debe aceptar unos determinados parámetros.
- ▶ Los parámetros permiten indicar a la función alguna información adicional relativa al evento, a modo de 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

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 (como mínimo, la de redibujado)
 - ▶ Configuración inicial de OpenGL, si es necesario.
- ▶ Función de visualización de un frame o cuadro.
- ▶ **Bucle principal** (gestiona eventos y visualiza frames)

Bucle de gestión de eventos

A partir de la inicialización, una aplicación típica OpenGL/GLFW emplea todo su tiempo en el **bucle principal** o **bucle de gestión de eventos**:

- ▶ 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 del bucle se dan estos dos pasos:
 - 1 Se espera hasta que ocurre algún evento.
 - 2 Se extrae el siguiente evento de la cola: si hay designada una función gestora para ese tipo de evento, se ejecuta dicha función.
 - 3 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)

En GLFW, el programador debe implementar explícitamente este bucle.

Estructura del programa.

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

```
void FGE_Redibujado ( )
{ ....}
void FGE_CambioTamano( int nuevoAncho, int nuevoAlto )
{ .... }
// otros callbacks:
// .....
void Inicializa_GLFW( int argc, char * argv[] )
{ .... }
void Inicializa_OpenGL( )
{ ..... }
int main( int argc, char *argv[] )
{
    Inicializa_GLFW(argc,argv) ; // crea un 'rendering context'
    Inicializa_OpenGL() ;       // usa el 'rendering context'
    // bucle principal
    .....
    glfwTerminate() ;           // cerrar la ventana
}
```

Bucle principal en GLFW

El bucle principal, o bucle de gestión de eventos, puede tener esta estructura

```
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 ) ;
}
```

En las funciones gestoras de eventos, cuando corresponda, se ponen a **true** las variables **redibujar_ventana** o **terminar_programa** (ambas son variables lógicas globales).

Código de inicialización de GLFW

Un ejemplo de código de inicialización sencillo para la librería GLFW sería el siguiente:

```
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 glfw_window), activar el rendering context
    glfw_window = glfwCreateWindow( ventana_tam_x, ventana_tam_y,
                                   "Practicas IG (18-19)", nullptr, nullptr );
    glfwMakeContextCurrent( glfw_window );
    // definir las funciones gestoras de eventos...
    glfwSetWindowSizeCallback ( glfw_window, FGE_CambioTamano );
    glfwSetKeyCallback         ( glfw_window, FGE_PulsarTeclaEspecial );
    glfwSetCharCallback        ( glfw_window, FGE_PulsarTeclaCaracter );
    glfwSetMouseButtonCallback ( glfw_window, FGE_ClickRaton );
    glfwSetCursorPosCallback   ( glfw_window, FGE_RatonMovido );
    glfwSetScrollCallback      ( glfw_window, FGE_Scroll );
}
```

Subsección 3.4

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 atributos adicionales a sus coordenadas.

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

- ▶ Por tanto, además de la secuencia de vértices, es necesario tener información acerca de que tipo de primitiva representa dicha secuencia.
- ▶ Cada primitiva puede tener sus propios atributos, adicionales a los de cada uno de sus vértices.

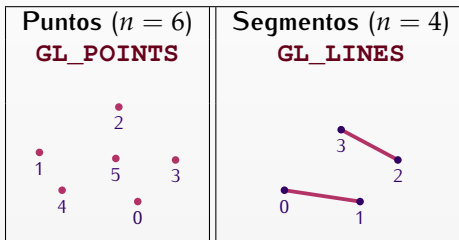
Tipos de primitivas

Una secuencia ordenada de n coordenadas de vértices puede codificar

- ▶ n **puntos** aislados (n arbitrario).
- ▶ uno o varios segmentos de recta, en concreto:
 - ▶ $n/2$ **segmentos independientes** (n par).
 - ▶ $n - 1$ segmentos formando una **polilínea abierta** ($n \geq 2$).
 - ▶ n segmentos formando una **polilínea cerrada** ($n \geq 3$).
- ▶ uno o varios polígonos, en concreto:
 - ▶ $n/3$ **triángulos** (n múltiplo de 3).
 - ▶ $n/4$ **cuadriláteros** (n múltiplo de 4).
 - ▶ un **polígono** con n lados ($n \geq 3$)
 - ▶ $n - 2$ triángulos compartiendo aristas (**tira de triángulos**), cada triángulo comparte dos vértices con el anterior ($n \geq 3$).
 - ▶ $n - 1$ triángulos compartiendo un vértice (**abanico de triángulos**) todos los triángulos comparten el primer vértice, y cada triángulo comparte dos vértices con el anterior ($n \geq 3$).
 - ▶ $(n - 2)/2$ cuadriláteros compartiendo aristas (**tira de cuadriláteros**), cada cuadrilátero comparte dos vértices con el anterior ($n \geq 4$, n par).

Primitivas de tipo puntos y segmentos.

Una secuencia de n coordenadas de vértices ($\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}$) puede visualizarse como un conjunto de puntos o un conjunto de segmentos



(en OpenGL se definen varias constantes enteras para identificar los distintos tipos de primitivas, en estos casos se usan las constantes **GL_POINTS** y **GL_LINES**)

Polígonos delanteros y traseros. Cribado.

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

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

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

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

Esta clasificación tiene utilidad especialmente en visualización 3D.

Modo de polígonos.

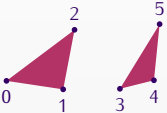
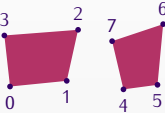
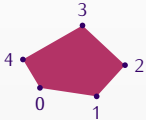
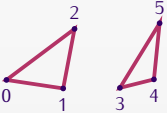
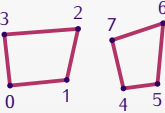
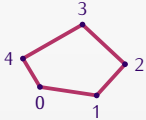
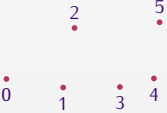


En el caso de las primitivas de polígonos, OpenGL puede visualizarlos de varias formas, según el valor de un parámetro de configuración en el estado de OpenGL, que se llama el **modo de polígonos**, y que permite seleccionar una de estas opciones:

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

El modo de polígonos se puede cambiar en cualquier momento con la función **glPolygonMode**, para todos los polígonos o bien de forma distinta para los delanteros y los traseros. Inicialmente, el modo activo es el modo relleno para todos los polígonos.

Primitivas tipo polígonos (no adyacentes)

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

Primitivas/ Modos	Triángulos GL_TRIANGLES	Cuadriláteros GL_QUADS	Polígono GL_POLYGON
Modo relleno GL_FILL			
Modo líneas GL_LINE			
Modo puntos GL_POINT			

Primitivas tipo polígonos (adyacentes)

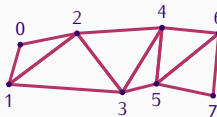
Cada polígono comparte vértices (los vemos solo el modo de líneas):

Tira de triángulos

GL_TRIANGLE_STRIP

Polígonos:

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

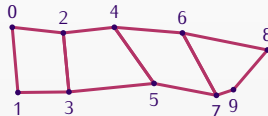


Tira de cuadriláteros

GL_QUAD_STRIP

Polígonos:

(0, 1, 3, 2), (2, 3, 5, 4),
(4, 5, 7, 6), (6, 7, 9, 8), ...

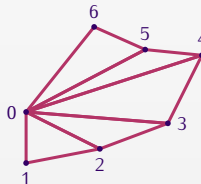


Abanico de triángulos

GL_TRIANGLE_FAN

Polígonos:

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



Polígonos de más de tres vértices

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

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

Subsección 3.5

Visualización de primitivas.

Envío de primitivas en modo inmediato

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

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

Envío de primitivas en modo diferido

El modo inmediato es muy ineficiente en tiempo por requerir el envío de todos los vértices por el bus del sistema a la GPU en cada visualización, aunque no cambien. Por eso actualmente se usa el **modo diferido**:

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

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

Formas de envío de la secuencia (modo inmediato)

En modo inmediato, las coordenadas de los vértices se pueden enviar de dos formas:

- ▶ Usando una llamada a **glVertex** por cada vértice (entre **glBegin** y **glEnd**).
 - ▶ El método es lento pues **requiere una llamada por vértice**.
 - ▶ Funcionalidad declarada **obsoleta** en OpenGL 3.0 y **eliminada** de OpenGL 3.1.
- ▶ Usando una **única** llamada a **glDrawArrays** para todos los vértices.
 - ▶ Requiere **almacenar la secuencia de coordenadas** en un array en la memoria RAM.
 - ▶ OpenGL recibe la dirección del array y lee todas las coordenadas
 - ▶ Por tanto, las implementaciones **pueden hacerlo de forma más eficiente en tiempo** que con **glBegin/glEnd**
- ▶ Usando una llamada a **glDrawElements** (secuencia indexada)

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

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

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

Ejemplo de glBegin/glEnd

Suponemos que el array en memoria (en la memoria del *cliente*, en terminología de OpenGL) con las coordenadas se declara como sigue:

```
const int num_verts = n ;           // número total de vértices en el array
GLfloat vertices[num_verts*3] =    // array con las coords. de verts.
{
    x0, y0, z0,                     // coordenadas del 1er vértice
    x1, y1, z1,                     // coordenadas del 2o vértice
    ...
    xn-1, yn-1, zn-1             // coords. n-ésimo vértice
} ;
```

en el ámbito de estas declaraciones, el envío del array se puede hacer así:

```
glBegin( tipo_prim ); // aquí tipo_prim es una de las constantes de tipo (p.ej. GL_TRIANGLES)
for( int i = 0 ; i < num_verts ; i++ )
    glVertex3f( vertices[3*i+0], vertices[3*i+1], vertices[3*i+2] );
glEnd();
```

Variantes de `glVertex`

Existen otras funciones en la familia de `glVertex`

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

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

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

Array de coordenadas de vértices.

La función **glDrawArrays** visualiza una secuencia completa de vértices (una o varias primitivas), usando una sola llamada:

- Todas las coordenadas (de tipo **float** o **double**) deben estar en memoria usando un espaciado constante entre ellas (nulo en los ejemplos).

3D

x_0	y_0	z_0	x_1	y_1	z_1	\dots	x_{n-1}	y_{n-1}	z_{n-1}
-------	-------	-------	-------	-------	-------	---------	-----------	-----------	-----------

2D

x_0	y_0	x_1	y_1	\dots	x_{n-1}	y_{n-1}
-------	-------	-------	-------	---------	-----------	-----------

- La dirección de memoria y la primera coordenada (x_0) es un valor (un puntero) conocido de tipo **float *** (o **double ***)
- Para especificar dicha dirección y la estructura del array se debe usar **glVertexPointer**.
- Las coordenadas se pueden almacenar en un vector de tipo **std::vector<GLfloat>**.

Uso de `glDrawArrays`

El array en memoria (en la memoria del *cliente*, en terminología de OpenGL) puede declararse como sigue:

```
const int num_verts = n ;           // número total de vértices en el array
GLfloat vertices[num_verts*3] =    // array con las coords. de verts.
{
    x0, y0, z0,                     // coordenadas del 1er vértice
    x1, y1, z1,                     // coordenadas del 2o vértice
    ...
    xn-1, yn-1, zn-1             // coords. n-ésimo vértice
} ;
```

en el ámbito de estas declaraciones, el envío del array se puede hacer así:

```
glEnableClientState( GL_VERTEX_ARRAY );           // habilitar array de vértices
glVertexPointer( 3, GL_FLOAT, 0, vertices );       // establecer dirección y estructura
glDrawArrays( tipo_pri, 0, num_verts );           // visualizar primitiva tipo tipo_pri
glDisableClientState( GL_VERTEX_ARRAY );          // deshabilitar array de vért.
```

Arrays de vértices en 2D

La función **glDrawArrays** puede usarse con tuplas de 2 coordenadas para dibujo 2D (la tercera coordenada, la Z, se pone a 0)

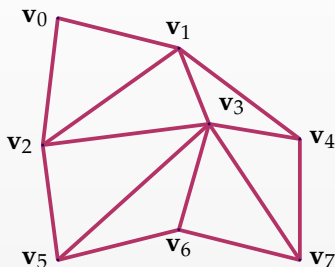
```
const int num_verts = n ;           // número total de vértices en el array
GLfloat vertices[num_verts*2] =    // array con las coords. de verts.
{
    x0, y0,
    x1, y1,
    ...
    xn-1, yn-1
} ;
```

el envío se haría de forma muy similar:

```
glEnableClientState( GL_VERTEX_ARRAY );           // habilitar array de vértices
glVertexPointer( 2, GL_FLOAT, 0, vertices );       // establecer dirección y estructura
glDrawArrays( tipo_pri, 0, num_verts );           // visualizar primitiva tipo tipo_pri
glDisableClientState( GL_VERTEX_ARRAY );          // deshabilitar array de vért
```


Problema de vértices replicados

Muchas veces necesitamos usar un vértice para varias primitivas:



Si usamos **GL_TRIANGLES**, la secuencia de vértices es

$v_0, v_2, v_1, v_1, v_2, v_3,$
 $v_1, v_3, v_4, v_2, v_5, v_3,$
 $v_3, v_5, v_6, v_3, v_6, v_7,$
 v_3, v_7, v_4

Supone **emplear más memoria y/o tiempo para visualizar del necesario**. En este ejemplo necesitamos una secuencia de 21 coordenadas de vértices, de las cuales solo hay 8 distintas (p.ej., el vértice v_3 aparece repetido 6 veces)

Uso de `glDrawElements`

Para evitar tener que repetir coordenadas, se puede usar

`glDrawElements` en lugar de **`glDrawArrays`**:

- ▶ Se usa un array de coordenadas de vértices, con la misma estructura que **`glDrawArrays`**, y que se especifica igualmente con **`glVertexPointer`**
- ▶ Ahora, la secuencia de vértices que forman las primitivas no coincide con la secuencia de coordenadas en memoria.
- ▶ Se usa un array de índices (*elements*) para especificar que vértices y en que orden se usan para construir la secuencia de coordenadas.
- ▶ Cada entrada es un índice (entero sin signo) que representa el número de vértice en el array de vértices (comenzando en cero).
- ▶ La función **`glDrawElements`** se usa para conseguir esto, y admite como parámetro la dirección, el tipo y el tamaño de la tabla de índices.

Arrays de índices y vértices

Para el ejemplo anterior, se requieren estas tablas:

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	\dots	x_7	y_7	z_7
-------	-------	-------	-------	-------	-------	-------	-------	-------	---------	-------	-------	-------

0	2	1	1	2	3	1	3	4	2	5	3	3	5	6	3	6	7
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- ▶ El primer índice (de tipo **unsigned int**) tiene está en una dirección de memoria (un puntero de tipo **unsigned int ***) conocida
- ▶ Los índices están siempre contiguos en memoria. Pueden ser **unsigned char**, **unsigned short**, o **unsigned int** (en los ejemplos usaremos **unsigned int**).

Uso de `glDrawElements`

Los arrays se pueden declarar como sigue:

```
const unsigned
    num_verts    = n ,           // número total de vértices en el array
    num_indices   = m ;           // número total de índices

GLfloat vertices[num_verts*3] =    // array con las coords. de verts.
{
    x0, y0, z0, x1, y1, z1, ...
    xn-1, yn-1, zn-1
} ;

unsigned indices[num_indices] =    // array con los índices
{
    i0, i1, i2, i3, ...,
    im-2, im-1
} ;
```

en el ámbito de estas declaraciones, el envío del array se haría con:

```
glEnableClientState( GL_VERTEX_ARRAY ); // habilitar array de vértices
glVertexPointer( 3, GL_FLOAT, 0, vertices ); // establecer dirección y estructura
// visualizar primitiva tipo tipo_pri, recorriendo los vértices en el orden de los índices:
glDrawElements( tipo_pri, num_indices, GL_UNSIGNED_INT, indices );
glDisableClientState( GL_VERTEX_ARRAY ); // deshabilitar array
```

Cambio del modo de visualización de polígonos

Se hace usando la llamada:

```
glPolygonMode ( GL_FRONT_AND_BACK, modo )
```

- ▶ *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 nuevo modo se aplica hasta que se cambie (forma parte del estado de OpenGL).
- ▶ El valor inicial es **GL_FILL**.
- ▶ En OpenGL 2.1 o anteriores (no en posteriores), 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.

Asignación de colores a vértices.

OpenGL siempre tiene un **color actual**. Es una terna RGB que se puede cambiar con **glColor**. Al inicio tiene un valor por defecto.

Cada vértice que procesa OpenGL tiene **siempre** asociado un color:

- ▶ Si se visualiza con **glBegin/glEnd**, se usa el color actual (puede cambiarse antes de cada vértice).
- ▶ Si se visualiza con **glDrawArrays** o **glDrawElements**:
 - ▶ Si hay array de colores habilitado: se usan los colores del array de colores (posiblemente distintos)
 - ▶ Si no hay array de colores habilitado: se usa el color actual (el mismo para todos los vértices).

Envío de colores con begin/end

Con **glBegin/glEnd**, es posible cambiar el color actual:

- Una sola vez, antes del primer vértice (todos los vértices son del mismo color)

```
glColor3f( 1.0, 0.0, 0.0 );           // color actual = rojo
glBegin( GL_TRIANGLES );
    glVertex3f( 0.0, 0.9, 0.0 ); // enviar primer vértice, con color rojo
    glVertex3f( -0.9, -0.9, 0.0 ); // enviar segundo vértice, con color rojo
    glVertex3f( +0.9, -0.9, 0.0 ); // enviar tercer vértice, con color rojo
glEnd();
```

- Antes de cada vértice, entre **glBegin** y **glEnd** (permite asignar colores distintos a cada vértice)

```
glBegin( GL_TRIANGLES );
    glColor3f ( 1.0, 0.0, 0.0 ); // color actual = rojo
    glVertex3f( 0.0, 0.9, 0.0 ); // enviar primer vértice, con color rojo
    glColor3f ( 0.0, 1.0, 0.0 ); // color actual = verde
    glVertex3f( -0.9, -0.9, 0.0 ); // enviar segundo vértice, con color verde
    glColor3f ( 0.0, 0.0, 1.0 ); // color actual = azul
    glVertex3f( +0.9, -0.9, 0.0 ); // enviar segundo vértice, con color azul
glEnd();
```

Envío de colores en un array de colores

Cuando se visualiza con `glDrawArrays` o con `glDrawElements`, es posible *habilitar* un **array de colores**. Contiene un color RGB para cada vértice.

```
// declaramos los vectores (usando vectores STL)
const float
    vertices[num_verts*3] = { 0.0,0.9,0.0,  -0.9,-0.9,0.0,  0.9,-0.9,0.0 } ,
    colores[num_verts*3]  = { 1.0,0.0,0.0,  0.0, 1.0,0.0,  0.0, 0.0,1.0 } ;

// especificar y habilitar puntero a vértices
glVertexPointer( 3, GL_FLOAT, 0, vertices );
glEnableClientState( GL_VERTEX_ARRAY );

// especificar y habilitar puntero a colores
glColorPointer( 3, GL_FLOAT, 0, colores );
glEnableClientState( GL_COLOR_ARRAY );

// dibujar
glDrawArrays( GL_TRIANGLES, 0, num_verts );

// deshabilitar punteros a vértices y colores
glDisableClientState( GL_VERTEX_ARRAY );
glDisableClientState( GL_COLOR_ARRAY );
```


Envío en modo diferido. VBOs

Para enviar las primitivas en modo diferido, debemos usar uno o varios bloques contiguos de memoria en la GPU:

- ▶ Cada uno de esos bloques se denomina VBO (*Vertex Buffer Object*).
- ▶ Cada VBO se gestiona desde el programa usando un identificador (o *name*), que es un valor (entero sin signo) único, distinto de cero, generado por OpenGL (lo llamamos el *identificador de VBO*)
- ▶ Un VBO puede contener una o varias tablas de datos de atributos de primitivas (una tabla es una lista de coordenadas de vértices, o de colores, o de otros tipos de atributos de primitivas), o bien una o varias tablas de índices.
- ▶ Durante la visualización, para usar una tabla en un VBO, debemos especificar el **identificador de VBO** y el **desplazamiento entero dentro del VBO**, en lugar de un simple puntero a RAM como hacemos en modo inmediato.

Ejemplo de creación de un VBO y su visualización

En este ejemplo vemos como se puede enviar y visualizar una secuencia de vértices usando un VBO. La transferencia a la GPU se hace únicamente antes de la primera visualización

```
static GLuint ident_vbo = 0 ; // variable permanente: identificador de VBO

if ( ident_vbo == 0 )          // si no hay un identificador válido, crear VBO:
{
    glGenBuffers( 1, &ident_vbo );          // crear VBO, obtener identificador
    glBindBuffer( GL_ARRAY_BUFFER, ident_vbo ); // activar el VBO
    glBufferData( GL_ARRAY_BUFFER,          // transferencia RAM -> GPU:
                  sizeof(float)*num_verts*3, // tamaño en bytes
                  vertices, GL_STATIC_DRAW ); // puntero a RAM, uso (hint)
}
else // si ya hay VBO:
    glBindBuffer( GL_ARRAY_BUFFER, ident_vbo ); // activar VBO usando su ident.

// dibujar igual que en modo inmediato (excepto que el puntero es relativo al VBO: es 0)
glEnableClientState( GL_VERTEX_ARRAY ); // usar tabla de vértices
glVertexPointer( 3, GL_FLOAT, 0, 0 ); // lugar y formato de las coordenadas
glDrawArrays( tipo_pri, 0, num_verts ); // visualizar
glDisableClientState( GL_VERTEX_ARRAY ); // desactivar puntero a vértices
glBindBuffer( GL_ARRAY_BUFFER, 0 ); // desactivar VBO (ident. == 0)
```

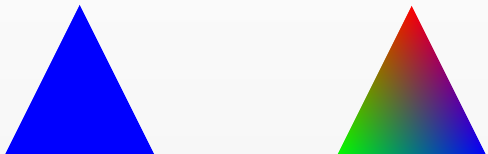
Uso de VBOs con atributos y secuencias indexadas

Lo más frecuente es usar los VBOs con diversos atributos de vértices, y usando secuencias indexadas

- ▶ Más adelante veremos ejemplos como crear varios VBOs para conseguir esto (ver: visualización de mallas en modo diferido, del tema 2 de teoría)
- ▶ Las tablas de coordenadas y las de atributos pueden almacenarse en uno o varios VBOs.
- ▶ La tabla de índices necesita un VBO distinto del anterior o anteriores.
- ▶ En la actualidad, esta es la forma habitual de visualizar objetos 3D complejos

Modo de sombreado

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



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

Eliminación de partes ocultas (EPO) con Z-buffer

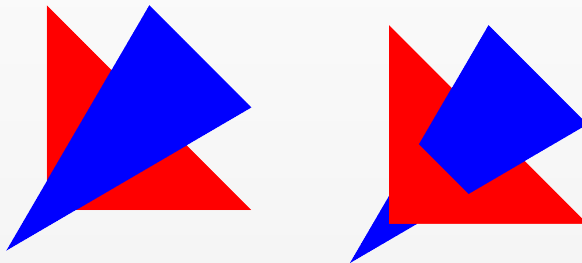
OpenGL usa las coordenadas Z de los vértices para calcular (por interpolación) la profundidad en Z en cada pixel de cada primitiva visualizada (Z es la dirección perpendicular a la pantalla).

Existe un buffer (llamado **Z-buffer**) donde se guarda la coordenada Z de lo que hay dibujado en cada pixel. Esto permite hacer el **test de profundidad** (*depth test*).

- ▶ Esto permite dibujar primitivas 3D con cálculo correcto de las 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.
- ▶ Cuando se desactiva, al dibujar una primitiva en un pixel siempre se sobrescribe lo que hubiese escrito antes en el pixel, sin considerar la Z.

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.

La función de redibujado

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

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

Ejemplo de función de redibujado

Un ejemplo sencillo para la función de redibujado es esta:

```
void VisualizarFrame ()
{
    // comprobar si ha habido error, restablecer variable de error
    CError ();
    // limpiar la ventana: limpiar colores y limpiar Z-buffer
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // envío de primitivas en modo inmediato:
    glEnableClientState( GL_VERTEX_ARRAY );
    glVertexPointer( ..... ); glDrawArrays( ..... );
    glVertexPointer( ..... ); glDrawArrays( ..... );
    .....

    // visualización de la imagen creada
    glfwSwapBuffers ();
    // comprobar si ha habido error en esta función
    CError ();
}
```


Detección de errores de OpenGL

Las funciones OpenGL pueden activar un código de error interno que debe ser comprobado para verificar si la aplicación está funcionando correctamente. Esto se simplifica con la macro **CError()**. Se declara como:

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

y se define así:

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

Atributos de las primitivas

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:
 - ▶ **Color** usado para visualizar puntos, líneas o polígonos rellenos (una terna RGBA).
 - ▶ **Ancho** (en pixels) de las líneas (real).
 - ▶ **Ancho** (en pixels) de los puntos (real).
 - ▶ Modo de dibujar los polígonos. Hay estas tres posibilidades:
 - ▶ **Rellenos** (con color actual).
 - ▶ **Alambre** se visualiza una polilínea recorriendo las aristas (*wireframe*)
 - ▶ **Puntos** se visualizan un punto en cada vértice.
- ▶ Otros atributos:
 - ▶ **Color** que será usado cuando se limpie la ventana (antes de dibujar) (RGBA).

Inicialización de OpenGL

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

```
void Inicializa_OpenGL( )
{
    // comprobar si el flag de error de OpenGL ya estaba activado (si estaba aborta)
    CError();
    // 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
    glColor3f( 0.7, 0.2, 0.4 );
    // establecer ancho de líneas o segmentos (en pixels)
    glLineWidth( 2.0 );
    // establecer diámetro de los puntos (en pixels)
    glPointSize( 3.0 );
    // establecer modo de visualización de prim.
    // (las tres posibilidades son: GL_POINT, GL_LINE, GL_FILL)
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    // habilitar eliminación de partes ocultas usando el Z-buffer
    glEnable( GL_DEPTH_TEST );
    // comprobar si ha habido algún error en esta función
    CError();
}
```

Definición del *viewport*

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

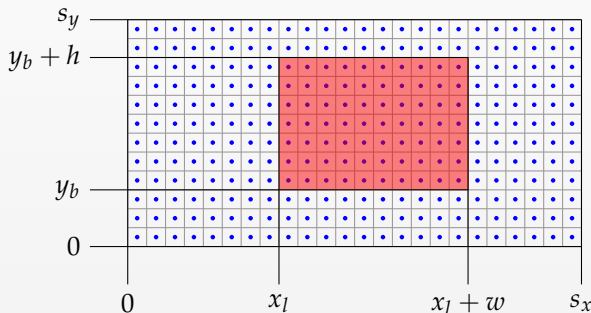
```
glViewport( izqui, abajo, ancho, alto ) ;
```

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

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

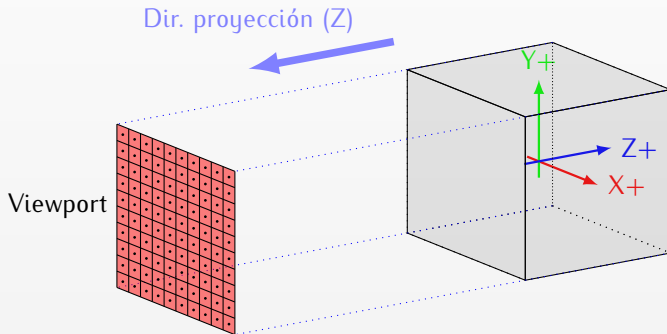
El viewport y la ventana como rejillas de pixels

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



Región visible y proyección sobre el viewport

Inicialmente, OpenGL usa una proyección paralela (al eje Z), y la región visible es el cubo de lado 2 con centro en el origen (ocupa el intervalo $[-1,1]$ en los tres ejes):



La función gestora del cambio de tamaño

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 );  
}
```

Documentación on-line sobre OpenGL y GLFW

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

Sección 4

Programación básica del cauce gráfico

- 4.1. Cauce programable
- 4.2. Ejemplo de shaders básicos.
- 4.3. Creación y ejecución de programas.
- 4.4. Funciones auxiliares.

Subsección 4.1

Cauce programable

Transformación y sombreado

Hay (entre otros) dos pasos importantes del cálculo de OpenGL que (usualmente) se ejecutan en la GPU o la librería gráfica:

1 Transformación:

En esta etapa se parte de la 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 situa la rasterización y el recortado de polígonos.

Vertex y Fragment Shaders

A los subprogramas que ejecutan los cálculos descritos antes se les denomina en general *shaders*, hay de dos tipos:

- 1 **Procesador de vértices (vertex shader):** subprograma encargado de la transformación de coordenadas.
 - ▶ Se ejecuta cada vez que se especifica una coordenada de un vértice nuevo (con **glVertex**, **glDrawArrays** u otras llamadas).
 - ▶ Produce como resultado las **coordenadas normalizadas del vértice en la ventana**.
 - ▶ Puede producir otros atributos (p.ej., el color asociado al vértice).
- 2 **Procesador de fragmentos (píxeles) (fragment shader):** subprograma encargado del sombreado.
 - ▶ Se ejecuta cada vez que se determina que una primitiva se proyecta en un pixel de la ventana.
 - ▶ Produce como resultado el **color del pixel**.

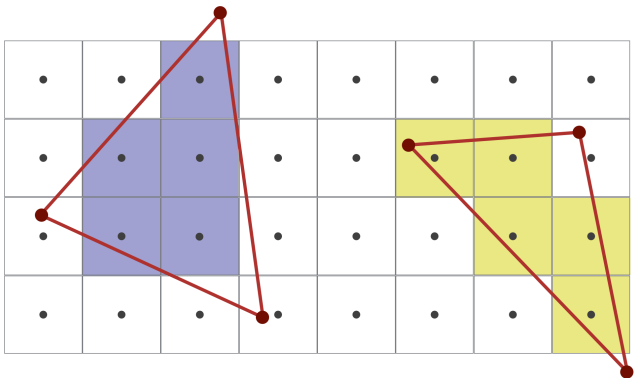
Programación de shaders

En OpenGL hay dos opciones para seleccionar los shaders que se usan durante la visualización:

- ▶ **Cauce de funcionalidad fija** (*fixed function pipeline*):
 - ▶ Se usan shaders predefinidos en OpenGL (fijos).
 - ▶ Solo disponible hasta OpenGL 3.0
- ▶ **Cauce programable** (*programmable pipeline*):
 - ▶ El programador de la aplicación especifica el código fuente de los shaders
 - ▶ Dicho código se escribe en el lenguaje de alto nivel llamado **GLSL**, parecido a C pero más simple.
 - ▶ 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.

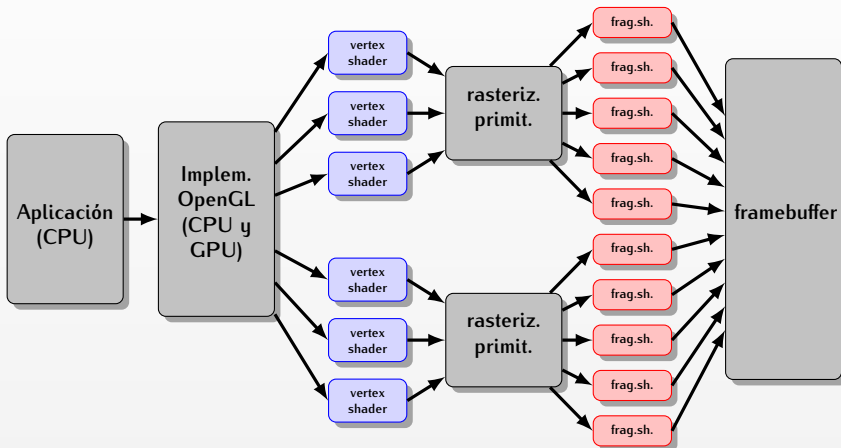
Rasterización de 2 triángulos

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



Cauce gráfico: DFD simplificado

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



Subsección 4.2

Ejemplo de shaders básicos.

Creación y uso de shaders

Un par formado por un *vertex shader* y un *fragment shader* forman un programa (*program*)

- ▶ Los dos shaders deben estar almacenados en memoria en variables de tipo **char** * (vectores de caracteres o cadenas, acabados en 0). Es conveniente almacenarlos en archivos en el sistema de archivos.
- ▶ 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 programa.
- ▶ Una aplicación puede generar uno o varios programas. En OpenGL 3.0 y anteriores, siempre está disponible, además, el programa del cauce fijo.
- ▶ En cada momento hay un programa activo, que se usa para visualizar, y que se puede cambiar en cualquier momento.

Vertex shader elemental:

El objetivo de este shader es producir el color y la posición de un vértice. Se puede almacenar en un archivo con extensión `.glsl`.

```
void main()
{
    // El objetivo es escribir estas variables:
    // - gl_Position posición del vértice en pantalla
    // - gl_FrontColor color asociado al vértice
    //
    // Se pueden leer (entre otras) las variables:
    // - gl_ModelViewProjection: matriz actual de transformación de coordenadas
    // - gl_Vertex: coordenadas del punto enviadas por la aplicación
    // - gl_Color: color actual especificado con 'glColor'

    gl_FrontColor = gl_Color ;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Las variables de entrada solo están disponibles en OpenGL 3.0 y anteriores. En versiones posteriores, es necesario usar parámetros de los shaders definidos por el programador (se verá más adelante).

Fragment shader elemental:

El objetivo de este shader es producir el color de un pixel donde se proyecta una primitiva:

```
void main()
{
    // El objetivo es escribir la variable:
    // - gl_FragColor: contiene el color a asignar al pixel.
    //
    // Se pueden leer (entre otras) la variable:
    // - gl_Color : color de la primitiva, obtenido de los colores de los vértices que la forman.

    gl_FragColor = gl_Color;
}
```

- Este par de shaders, combinados, permiten visualizar los ejemplos de este capítulo igual que con la funcionalidad fija.

Subsección 4.3

Creación y ejecución de programas.

Identificación y funcionalidad para shaders y programas

Para usar un programa en una aplicación, es necesario compilar sus dos shaders y enlazar el programa, desde la propia aplicación (en **tiempo de ejecución** de la misma):

- ▶ Cada shaders o programa se identifica en la aplicación con un valor entero (**GLuint**), que llamamos su **identificador**.
- ▶ Existen funciones para:
 - ▶ 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.
 - ▶ Activar un programa (**glUseProgram**).

(solo disponibles en la versión 2.0 y posteriores de OpenGL)

Compile shaders

Esta función crea un nuevo shader a partir de un archivo, y lo compila.

- ▶ Si no hay errores, devuelve identificador de shader
- ▶ El parámetro `tipoShader` puede valer **GL_FRAGMENT_SHADER** o bien **GL_VERTEX_SHADER**.

```
GLuint CompileShader( const char * nombreArchivo, GLenum tipoShader )
{
    // crear shader nuevo, obtener identificador (tipo GLuint)
    const GLuint idShader = glCreateShader( tipoShader );

    // leer archivo fuente de shader en memoria, asociar fuente al shader
    const GLchar * fuente = LeerArchivo( nombreArchivo );
    glShaderSource( idShader, 1, &fuente, NULL );
    delete [] fuente ; fuente = NULL ; // libera memoria del fuente

    // compilar y comprobar errores
    glCompileShader( idShader );
    VerErroresCompile( idShader ); // opcional, muy conveniente

    // devolver identificador de shader como resultado
    return idShader ;
}
```

Crear y enlazar un programa

Esta función crea un nuevo programa, compilando y enlazando sus dos shaders, a partir de los nombres de archivos.

```
GLuint CrearPrograma( const char * archFrag, const char * archVert )
{
    // crear y compilar shaders, crear el programa
    const GLuint
        idFragShader = CompileShader( archFrag, GL_FRAGMENT_SHADER ),
        idVertShader = CompileShader( archVert, GL_VERTEX_SHADER ),
        idProg
            = glCreateProgram();

    // asociar shaders al programa
    glAttachShader( idProg, idFragShader );
    glAttachShader( idProg, idVertShader );

    // enlazar programa y comprobar errores
    glLinkProgram( idProg );
    VerErroresEnlazar( idProg ); // opcional, muy conveniente

    // devolver identificador de programa
    return idProg ;
}
```

Inicialización y creación de shaders (1/2)

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> // incluir en lugar de GL/gl.h, antes de GL/glut.h
#include <GL/glut.h>
...
GLuint idProg ; // identificador de programa
....
void Inicializa_OpenGL()
{
    // leer punteros a funciones 2.0+ con GLEW
    GLenum codigoError = glewInit();
    if ( codigoError != GLEW_OK ) // comprobar posibles errores
    {
        std::cout << "Imposible inicializar 'GLEW', mensaje: "
                   << glewGetErrorString(codigoError) << std::endl ;
        exit(1);
    }
    ....
}
```


Inicialización y creación de shaders (2/2)

```
.....
```

```
// comprobar si OpenGL ver 2.0 + está soportado (usando GLEW)
```

```
if ( ! GLEW_VERSION_2_0 )
```

```
{   cout << "OpenGL 2.0 no soportado." << endl << flush ;
```

```
    exit(1);
```

```
}
```

```
// hacer el resto de inicializaciones (igual que antes)
```

```
.....
```

```
// compilar shaders, crear programa
```

```
idProg = CrearPrograma( "fragment-shader.glsl", "vertex-shader.glsl");
```

Uso del programa

En la función de redibujado (o las llamadas desde ella), podemos:

- ▶ Llamar a **glUseProgram** para activar un programa previamente creado
- ▶ Especificar 0 como identificador para usar el cauce fijo, si se desea usar (no disponible en OpenGL 4.0 y posteriores)

```
GLuint idProg ;  
.....  
void FGE_Redibujado()  
{  
    glUseProgram( 0 ) ; // activar programa de funcionalidad fija  
    // envío de primitivas con los shaders estándar de OpenGL  
    .....  
  
    glUseProgram( idProg ); // activar nuestro programa  
    // envío de primitivas con el shader creado por nosotros:  
    .....  
}
```

Subsección 4.4

Funciones auxiliares.

Verificar errores de compilación

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

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

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

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

Verificar errores de enlazado

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

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

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

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

Lectura de un archivo

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

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

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

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

Sección 5

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

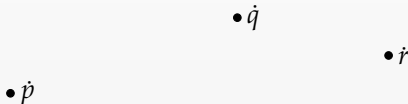
- 5.1. Puntos y vectores
- 5.2. Marcos de referencia y coordenadas
- 5.3. Coordenadas homogéneas
- 5.4. Operaciones entre vectores: producto escalar y vectorial
- 5.5. Transformaciones geométricas y afines
- 5.6. Matrices de transformación
- 5.7. Representación y operaciones con tuplas
- 5.8. Representación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

Subsección 5.1

Puntos y vectores

Puntos y vectores

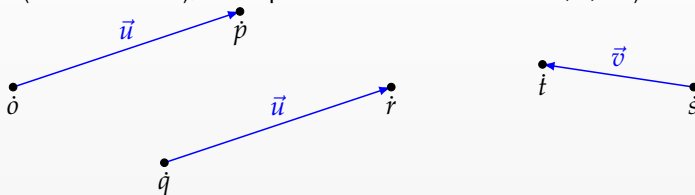
Los modelos 3D y 2D de objetos y figuras que vamos a representar se pueden construir cada uno de ellos en base a un conjunto abstracto (con estructura de **espacio afín**), cuyos elementos son puntos de un determinado espacio donde imaginamos el modelo. Cada uno de estos **puntos** o **localizaciones**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



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

Vectores

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



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

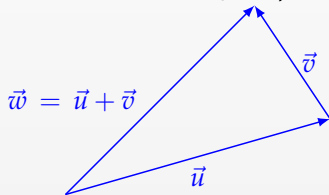
Resta de puntos, suma de vectores

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



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

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



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

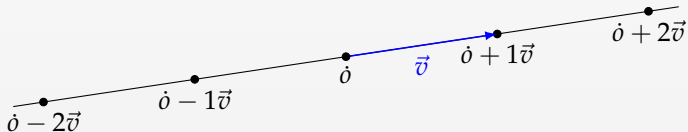
El **vector nulo** lo notamos como $\vec{0}$, y se define como $\vec{0} = \dot{p} - \dot{p}$ (para cualquier punto \dot{p}).

Producto de vectores y valores escalares

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



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

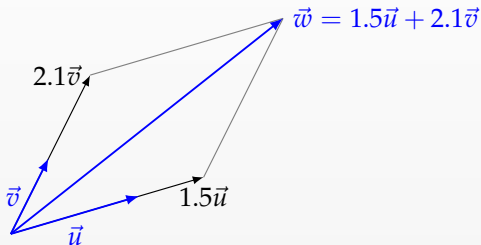


Subsección 5.2

Marcos de referencia y coordenadas

Bases de vectores

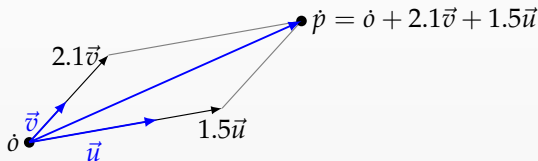
Usando dos vectores cualquiera \vec{u} y \vec{v} del plano (no paralelos ni nulos), podemos escribir cualquier otro vector \vec{w} como una combinación lineal de ellos:



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

Marcos de referencia y coordenadas

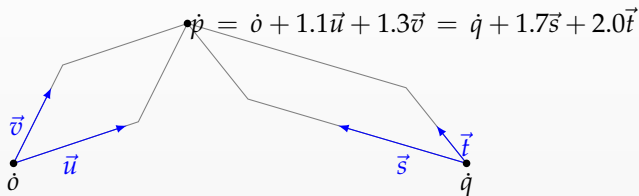
Si fijamos un punto \dot{o} (origen) y una base $\{\vec{u}, \vec{v}\}$, cualquier punto \dot{p} del plano se puede escribir como $\dot{p} = \dot{o} + a\vec{u} + b\vec{v}$:



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

Coordenadas y puntos

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



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

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

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

Subsección 5.3

Coordenadas homogéneas

Coordenadas homogéneas

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

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

Notación para tuplas de coordenadas

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

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

nótese que hemos usado el símbolo en negrita \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 explícitamente los puntos con sus coordenadas homogéneas, usando algún marco de referencia:

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

Coordenadas homogéneas de vectores

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

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

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

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

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

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

Operaciones usando coordenadas

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

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

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

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

Subsección 5.4

Operaciones entre vectores: producto escalar y vectorial

El marco de referencia especial

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

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

Producto escalar módulo de vectores

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

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

Hay muchas funciones que cumplen estas dos propiedades. Para concretar a cual de ellas no referimos, usamos el marco especial $\mathcal{E} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$. Se cumple:

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

El **módulo** (o norma) de un vector \vec{u} se nota con $\| \cdot \|$ y es un valor real que se define como:

$$\|\vec{u}\| = \sqrt{\vec{u} \cdot \vec{u}} \quad (\text{se cumple: } \|a\vec{u}\| = |a| \|\vec{u}\|)$$

Marcos cartesianos

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

- Sus vectores tienen longitud unidad:

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

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

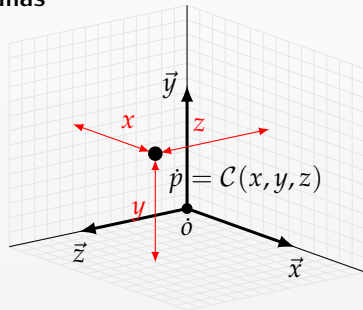
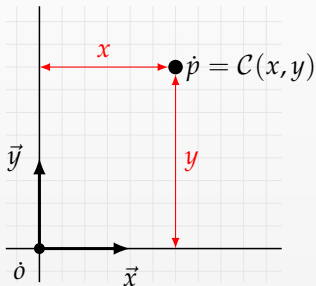
$$\vec{e}_x \cdot \vec{e}_y = \vec{e}_y \cdot \vec{e}_z = \vec{e}_z \cdot \vec{e}_x = 0$$

En estas condiciones, decimos que \mathcal{C} es un marco de referencia **cartesiano**.

(el marco de referencia \mathcal{E} es cartesiano por definición)

Marcos y coordenadas cartesianas

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



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

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 usamos las coordenadas de cualquier otro marco cartesiano distinto de \mathcal{C}).

- Como consecuencia, el módulo se puede obtener como:

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

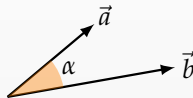
- El módulo de un vector coincide con su **longitud** en el espacio (ya que de los versores de \mathcal{E} dijimos que tenían longitud unidad por definición). El módulo, calculado así, es siempre el mismo en cualquier marco cartesiano.

Interpretación geométrica del producto escalar

Dados dos vectores \vec{a} y \vec{b} (ninguno nulo) llamamos α al ángulo que hay entre ellos. Se cumple:

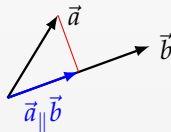
- El producto escalar es proporcional al coseno de α :

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



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

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



- Si $\|\vec{b}\| = 1$ entonces: $\vec{a}_{\parallel \vec{b}} = (\vec{a} \cdot \vec{b}) \vec{b}$
- Si $\|\vec{a}\| = 1$ y $\|\vec{b}\| = 1$ entonces: $\vec{a} \cdot \vec{b} = \cos \alpha$

Producto vectorial de vectores

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

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

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

$$\vec{x} \times \vec{y} = \vec{z} \qquad \vec{y} \times \vec{z} = \vec{x} \qquad \vec{z} \times \vec{x} = \vec{y}$$

Cálculo del producto vectorial

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

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

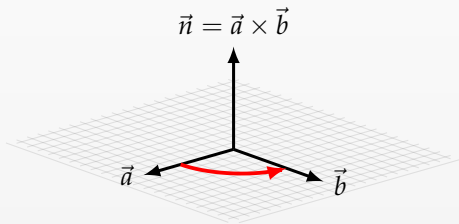
$$\mathcal{C} \begin{pmatrix} a_x \\ a_y \\ a_z \\ 0 \end{pmatrix} \times \mathcal{C} \begin{pmatrix} b_x \\ b_y \\ b_z \\ 0 \end{pmatrix} = \mathcal{C} \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \\ 0 \end{pmatrix}$$

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

Interpretación geométrica del producto vectorial (1)

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})



En los marcos de referencia a derechas, 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}

Interpretación geométrica del producto vectorial (2)

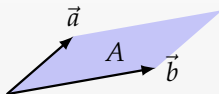
En un marco cartesiano

- ▶ 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, si $\|\vec{a}\| = 1$ y $\|\vec{b}\| = 1$, entonces:

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

Subsección 5.5

Transformaciones geométricas y afines

Transformación geométrica

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

Transformación geométrica

Una **transformación geométrica** T es una aplicación que asocia a cualquier punto \dot{p} de un espacio afín en otro punto \dot{q} del mismo u otro espacio afín, y escribimos

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

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

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

Transformación de coordenadas

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

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

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

$$q_0 = f_0(p_0, p_1, p_2)$$

$$q_1 = f_1(p_0, p_1, p_2)$$

$$q_2 = f_2(p_0, p_1, p_2)$$

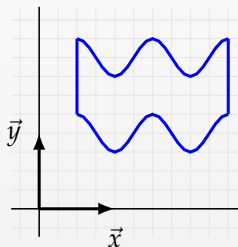
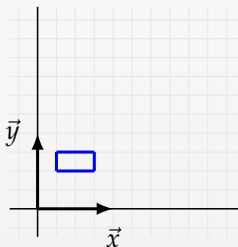
Lógicamente, para una única transformación T , las funciones f_0 , f_1 y f_2 **dependen del sistema de referencia \mathcal{R} en uso.**

Ejemplo de transformación en 2D

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

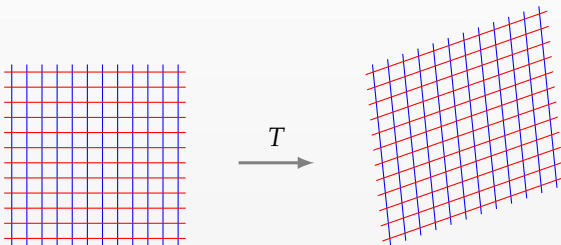
$$f_0(x, y) = 4x - 1 \quad f_1(x, y) = 2y + \frac{2 + \cos((8x - 2)\pi)}{4}$$

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



Definición de transformación afín

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



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

Propiedades de las transformaciones afines

Una transformación afín T cumple estas propiedades:

- Puesto que T conserva el paralelismo, se cumple:

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

- Luego podemos extender T a los vectores:

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

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

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

Subsección 5.6

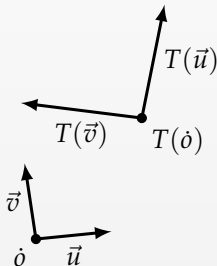
Matrices de transformación

Transformación de marcos

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

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

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



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

Transformación de coordenadas

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

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

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

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

Matriz de transformación de coordenadas (puntos)

Matricialmente:

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

o lo que es lo mismo:

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

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

Matriz de transformación de coordenadas (vectores)

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

- Aplicando un razonamiento similar al usado para los puntos, obtenemos 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**. Esa matriz **determina como se transforman las coordenadas tanto de puntos como de vectores**

- ▶ M permite obtener las coordenadas de los puntos transformados en términos de las coordenadas de los puntos originales (en ese marco)
- ▶ En 3D es una matriz 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

Descomposición de una matriz

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

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

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

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

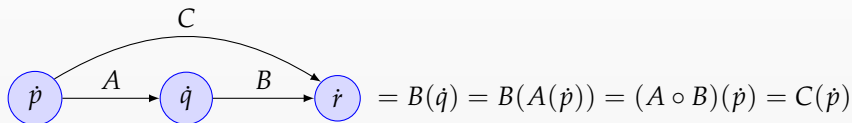
Ventajas del uso de coords. homogéneas

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

- ▶ 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 $C = A \circ B$:



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

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

la composición es **asociativa**

Transformación inversa

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

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

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. Si consideremos la transformación compuesta

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

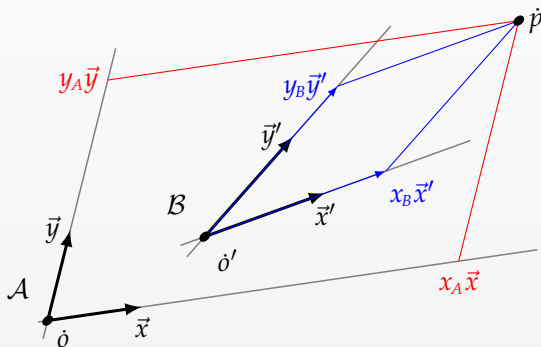
Entonces, la matriz M_C asociada a C será el producto de las matrices M_i asociadas a cada una de las T_i :

$$M_C = M_n M_{n-1} \dots M_2 M_1$$

Nótese que el producto es derecha a izquierda (primero 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 cualesquiera $\mathcal{A} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$ y $\mathcal{B} = [\vec{x}', \vec{y}', \vec{z}', \dot{o}']$,
y un punto $\dot{p} = \mathcal{B}(x_B, y_B, z_B, 1)^t = \mathcal{A}(x_A, y_A, z_A, 1)^t$



$$\dot{p} = \dot{o} + x_A \vec{x} + y_A \vec{y} = \dot{o}' + x_B \vec{x}' + y_B \vec{y}'$$

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:

$$\begin{aligned}\vec{x}' &= \mathcal{A}\mathbf{a} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}](a_x, a_y, a_z, 0)^t = a_x\vec{x} + a_y\vec{y} + a_z\vec{z} + 0\dot{o} \\ \vec{y}' &= \mathcal{A}\mathbf{b} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}](b_x, b_y, b_z, 0)^t = b_x\vec{x} + b_y\vec{y} + b_z\vec{z} + 0\dot{o} \\ \vec{z}' &= \mathcal{A}\mathbf{c} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}](c_x, c_y, c_z, 0)^t = c_x\vec{x} + c_y\vec{y} + c_z\vec{z} + 0\dot{o} \\ \dot{o}' &= \mathcal{A}\mathbf{d} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}](d_x, d_y, d_z, 1)^t = d_x\vec{x} + d_y\vec{y} + d_z\vec{z} + 1\dot{o}\end{aligned}$$

esto se puede expresar matricialmente:

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

Por tanto, la matriz 4x4 $M_{\mathcal{A},\mathcal{B}}$ transforma el sistema de referencia \mathcal{A} en el sistema de referencia \mathcal{B}

Descomposicion de $M_{A,B}$

La matriz $M_{A,B}$ que acabamos de considerar se puede descomponer en el producto de una matriz de desplazamiento $D_{A,B}$ y una matriz $R_{A,B}$ (sin desplazamientos):

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

- La matriz $R_{A,B}$ no tiene términos de desplazamiento.
- La matriz $D_{A,B}$ es la que produce un desplazamiento, de forma que el origen de \mathcal{A} (el punto \dot{o}) se lleva hasta el origen de \mathcal{B} (el punto \dot{o}'), (el vector de desplazamiento es $\dot{o}' - \dot{o} = \mathcal{A}(d_x, d_y, d_z, 0)^t$).

Transformación de coordenadas

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

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

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

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

de donde se deduce que

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

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

Interpretación dual de las matrices

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

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

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

- 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}):

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

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

Transformación inversa. Descomposición.

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

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

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

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

de cualquiera de estas dos igualdades se hace evidente que:

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

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

Descomposición de la inversa

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

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

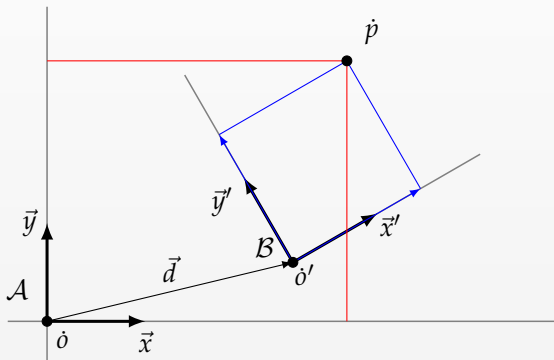
la matriz $D_{\mathcal{A},\mathcal{B}}^{-1}$ es un desplazamiento que lleva \dot{o}' a \dot{o} , es decir, un desplazamiento por el vector $\dot{o} - \dot{o}' = \mathcal{A}(-d_x, -d_y, -d_z, 0)^t$.

Matricialmente:

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

Relación entre marcos cartesianos

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



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

Transformación inversa entre marcos cartesianos

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

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

Matricialmente, por tanto:

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

es decir, la transformación inversa entre marcos cartesianos es muy fácil de construir directamente.

Subsección 5.7

Representacion y operaciones con tuplas

Representación en memoria de coordenadas.

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

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

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

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

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

Creación, consulta y modificación de tuplas.

Este código válido ilustra las distintas opciones:

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

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

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

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

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

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

Operaciones entre tuplas y escalares.

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

```
// declaraciones de tuplas y de valores escalares
Tupla3f  a,b,c ;
float    s,l ;

// operadores binarios y unarios de suma/resta/negación
a = b+c ;
a = b-c ;
a = -b  ;

// multiplicación y división por un escalar
a = 3.0f*b ;      //  $\vec{a} = 3\vec{b}$ 
a = b*4.56f ;     //  $\vec{a} = 4.56\vec{b}$ 
a = b/34.1f ;     //  $\vec{a} = (1/34.1)\vec{b}$ 

// 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  $\vec{a} = \vec{b} \times \vec{c}$  (solo para tuplas de 3 valores)
l = a.lengthSq()  ; //  $l = \|\vec{a}\|^2$  (calcular módulo al cuadrado)
a = b.normalized() ; //  $\vec{a} =$  copia normalizada de  $\vec{b}$  ( $\vec{b}$  no cambia)
```

Subsección 5.8

Representación y operaciones sobre matrices.

Representación de transf. en memoria.

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

```
#include <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 c4( 1.0,2.0,3.0,4.0 ) ; Tupla3f c3(1.0,1.0,3.0);
m1 = m2*c4 ; m1 = m2 * c3
// conversión a puntero a 16 flotantes (float *) (formato OpenGL)
float * pm = m ; const float * pcm = m ;
// escritura en la salida estándar (varias líneas)
cout << m << endl ;
```

Matrices más usuales

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

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

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

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

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

Fin de la presentación.