Informática Gráfica:
Teoría. Tema 1. Introducción.

Carlos Ureña

2020-21

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

Teoría. Tema 1. Introducción.

Índice.

Sección 1.
Introducción.

Subsección 1.1.
Concepto y metodologías.

El término *Informática Gráfica*

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

> *El campo de la Informática dedicado al estudio de los algoritmos, técnicas o metodogías 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)

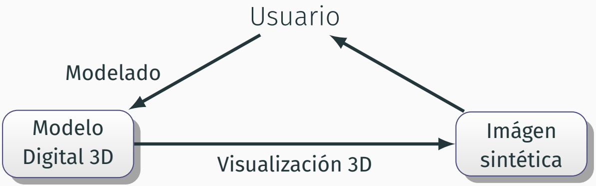
en aplicaciones específicas, se usan otros campos de la Informática en particular o la Ciencia en general (p.ej. para desarrollo de videojuegos se usan también técnicas de Inteligencia Artificial).

Informática Gráfica 3D interactiva

Los elementos esenciales de una aplicación gráfica son:

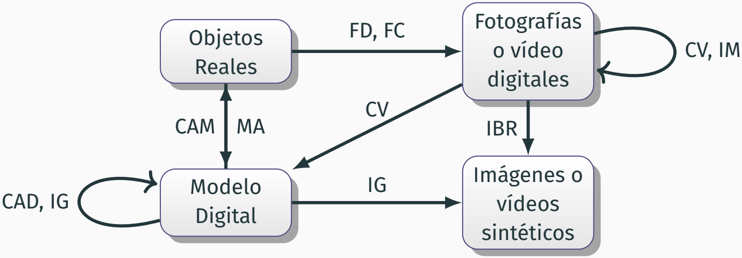
- **Modelos digitales** de objetos reales, ficticios o de datos
- **Imágenes o vídeos digitales** que se usan para visualizar dichos objetos.

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 técnologías:



FD	Fotografía Digital	FC	Fotografía Computacional
CV	Visión por Ordenador	IBR	*Rendering* Basado en Imág.
CAD	Diseño Asistido por Ord.	CAM	Fabric. Asistida por Ord.
MA	Adquisición de Modelos	IM	Tratamiento de Imágenes

Subsección 1.2. Aplicaciones..

Aplicaciones

Las aplicaciones son muy numerosas e invanden 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

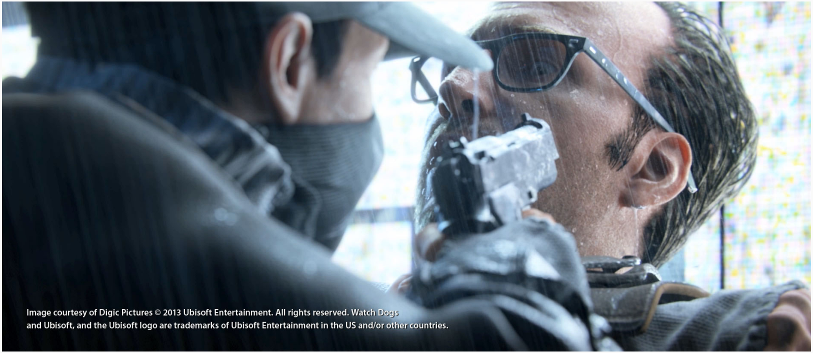
Fotograma del videojuego *Watch Dogs* de Ubisoft.
Vídeo: ☞ http://www.youtube.com/watch?v=kPYgXvgS6Ww

Realidad Aumentada (AR)

Imagen:

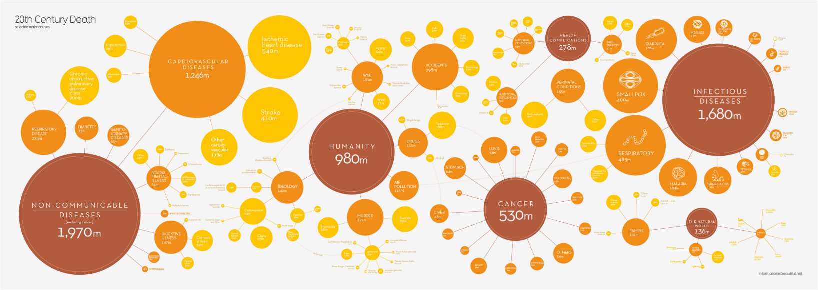
☞ http://technomarketer.typepad.com/technomarketer/2009/04/firstlook-augmented-reality.h

Películas y animaciones generadas por ordenador



Fotograma del tráiler cinemático del videojuego *Wacth 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

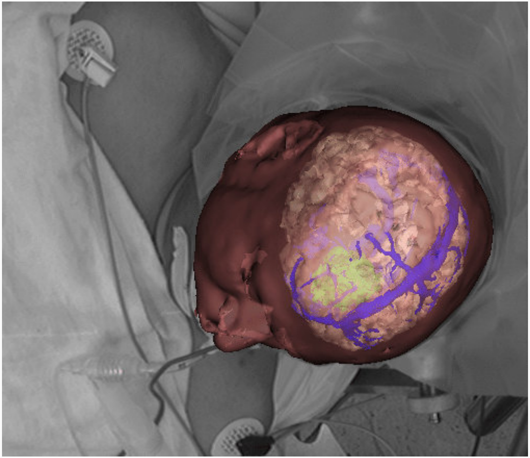
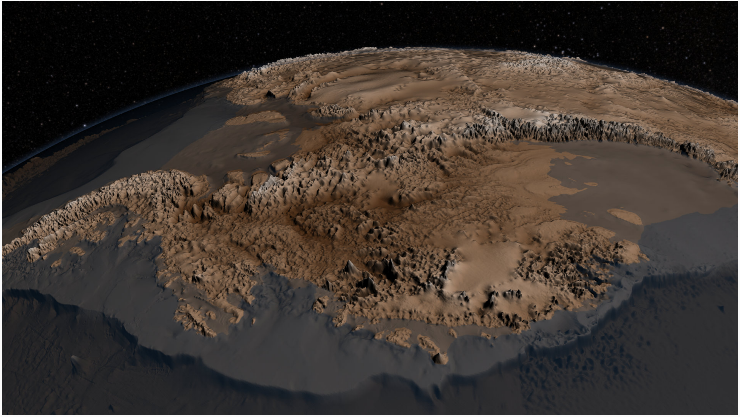


Imagen creada por Christopher Brown, Universidad de Rochester:

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

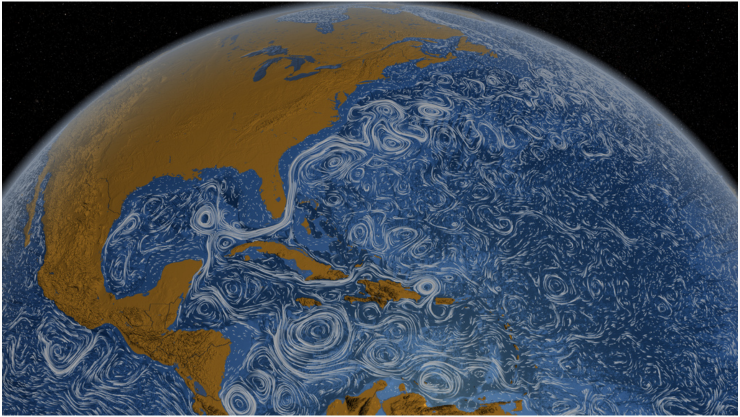
Visualización científica (geología)



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

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

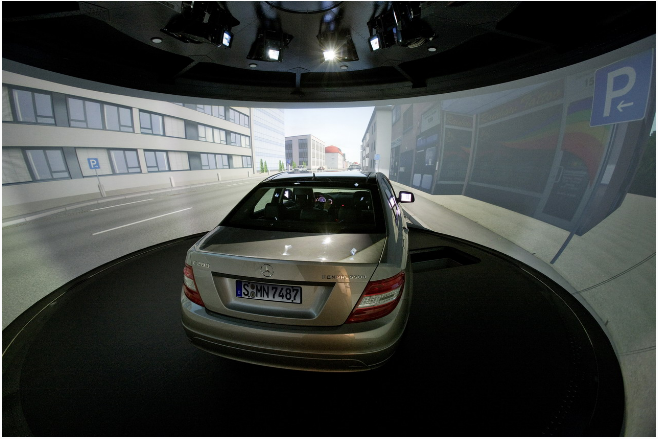
Visualización científica (climatología)



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 Standford.

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

Sección 2.
El proceso de visualización.

Subsección 2.1.
Programas gráficos: interactivos versus off-line.

Programas gráficos

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

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

Programas gráficos interactivos

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

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

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

Programas gráficos no interactivos

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

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

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

Subsección 2.2.
El proceso de visualización.

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

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

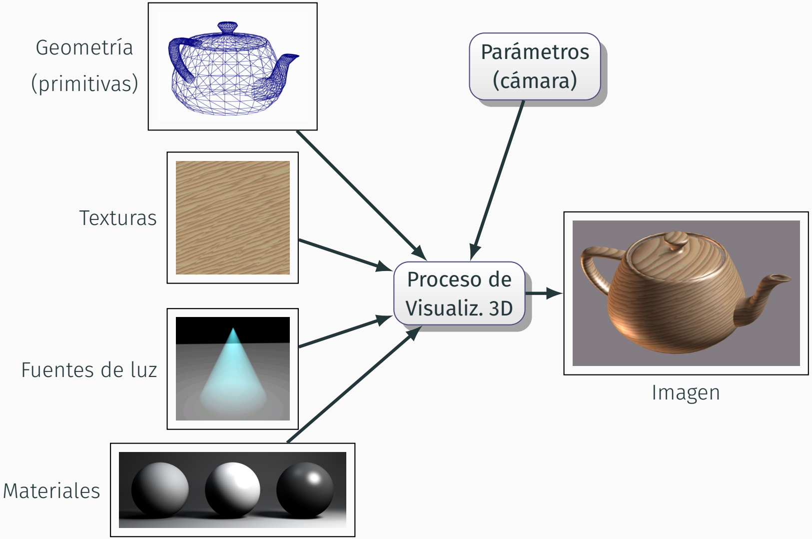
- **Modelo de escena:** estructura de datos en memoria que representa lo que se quiere ver, esta formado por varias partes:
  - **Modelo geométrico**: conjunto de **primitivas** (típicamente polígonos planos), que definen la forma de los objetos a visualizar
  - **Modelo de aspecto**: conjunto de parámetros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

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

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

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

El proceso de visualización 3D: esquema



Subsección 2.3.
Rasterización y ray-tracing..

Visualización basada en *rasterización*

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

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

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

Visualización basada en *Ray-Tracing*

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

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

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

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

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

Rasterización

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

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

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

Ray-tracing

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

Subsección 2.4.
El cauce gráfico en rasterización.

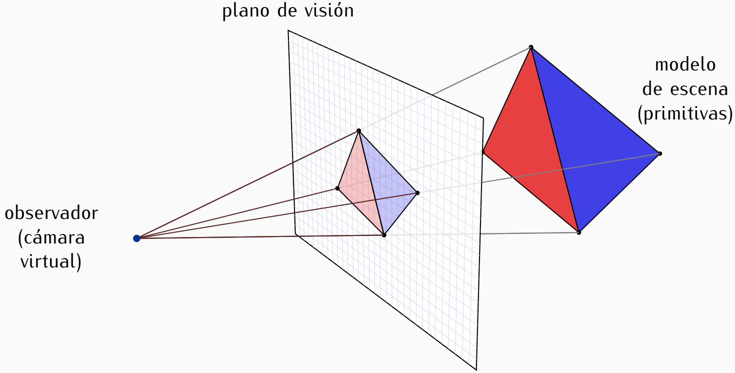
El cauce gráfico: entradas y salidas

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

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

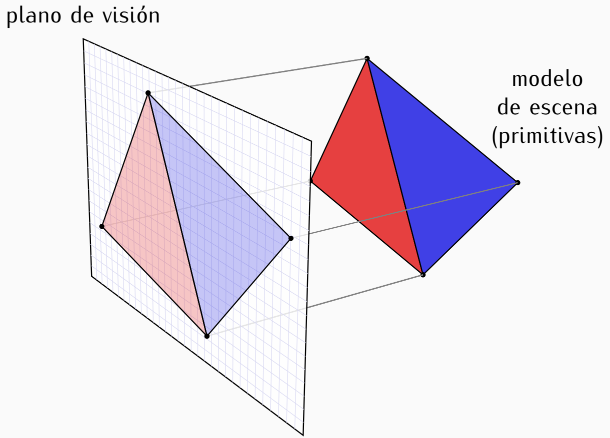
Transformación y proyección

Cada primitiva se situan 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):



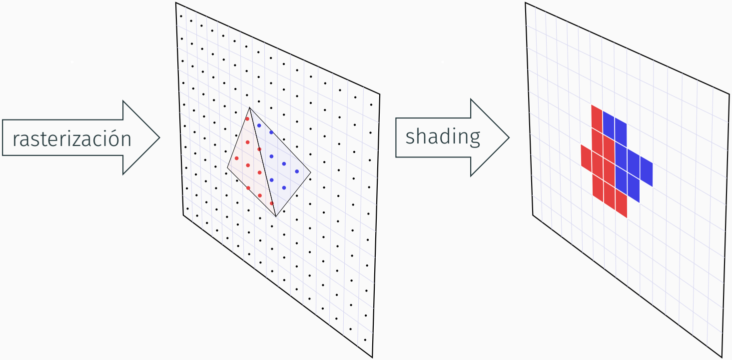
Proyección paralela

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



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.



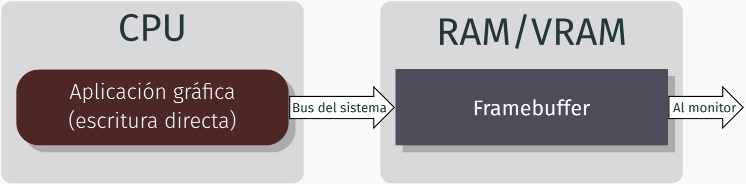
Sombreado: básico versus avanzado

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

Subsección 2.5.
Las APIs de rasterización.

Aplicaciones de escritura directa

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

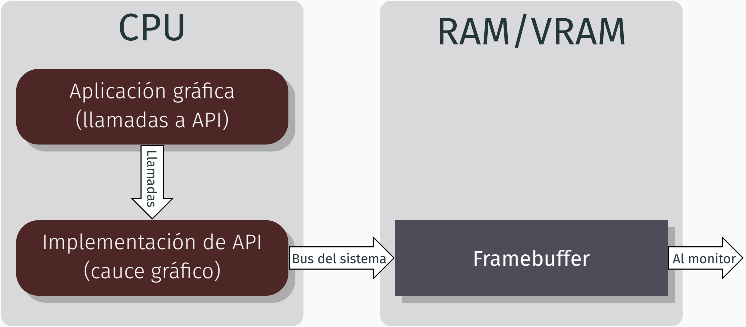


Desventajas

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

Uso de APIs gráficas

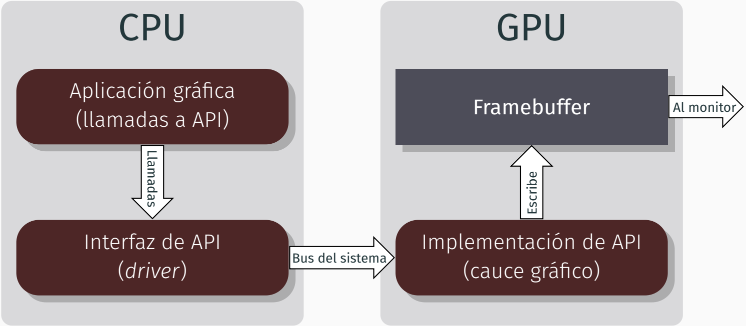
El uso de implementaciones de APIs gráficas portables (y gestores de ventanas) proporciona **portabilidad** y **acceso simultáneo**



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

Uso de APIs y hardware gráfico (GPUs)

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



APIs de rasterización en GPUs: las primeras APIs

Las dos primeras APIs existentes son estas:

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

Hay dos APIs adicionales, basadas en OpenGL

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

APIs de rasterización en GPUs: APIs modernas

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

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

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

Subsección 2.6.
El cauce gráfico en GPUs.

Etapas del cauce gráfico (1/2)

El cauce gráfico tiene estas etapas:

1. **Procesado de vértices:** parte de una secuencia de vértices (puntos del espacio) y produce una secuencia de primitivas (puntos, segmentos o triángulos). Tiene estas sub-etapas:

   1.1. **Transformación:** los vértices de cada **primitiva** son transformados en diversos pasos hasta encontrar su proyección en el plano de la imagen. Es realizado por un sub-programa llamado **Vertex Shader** (modificable por el programador, o *programable*).

   1.2. **Teselación y nivel de detalle:** transformaciones adicionales avanzadas, realizadas por varios programas, entre ellos el **geometry shader** (programable). No lo vamos a estudiar.

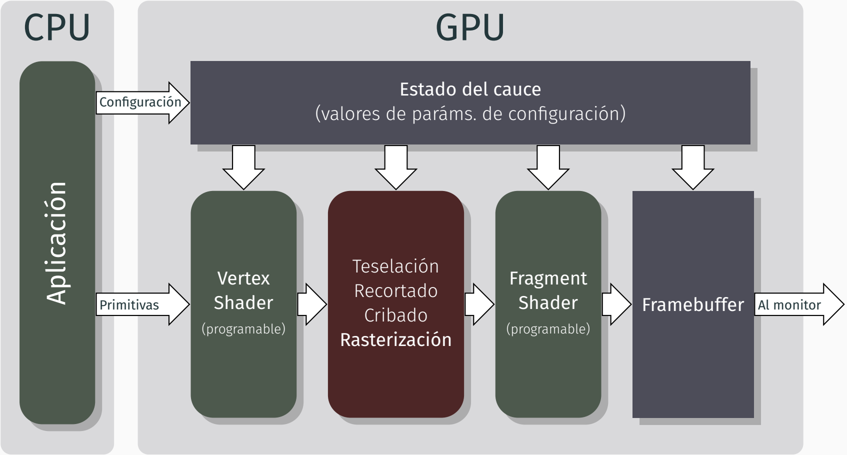
Etapas del cauce gráfico (2/2)

El cauce gráfico tiene estas etapas:

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

Esquema simplificado del cauce gráfico en una GPU

DFD simplificado de una aplicación gráfica y el cauce en GPU



Tipos de cauce gráfico: funcionalidad fija o programable

Respecto de la posibilidad de programar partes del cauce:

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

Evolución del cauce programable

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

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

Sección 3.
La librería OpenGL. Visualización..

Subsección 3.1.
La API OpenGL..

La API OpenGL

- **OpenGL** es la **especificación** de un conjunto de funciones útil para visualización 2D/3D basada en rasterización (un documento con: funciones, sus parámetros y comportamiento).
- Permite la rasterización de primitivas de bajo nivel (polígonos, lineas segmentos), de forma eficiente y portable.
- **OpenGL ES** (*OpenGL for Embedded Systems*): variante de OpenGL para dispositivos móviles y consolas.
- **GLSL** (*GL Shading Language*): lenguaje de programación de *shaders* que se usa con OpenGL.
- 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 escribian para hardware específico.
- 1988: Silicon Graphics inc. era lider 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 tambien 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*). Está 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 portable y *open source*, sirve para:

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

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

Subsección 3.2.
Programación y eventos en GLFW.

Eventos y sus tipos

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

- **Teclado**: pulsación o levantado una tecla, de tipo carácter o de otras teclas.
- **Ratón**: pulsación o levantado de botones del 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.

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

Estructura de un programa (1/2)

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

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

Estructura del programa. (2/2)

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

```
void VisualizarFrame( ) // se encarga de redibujar la imagen
   { .... }
void FGE_CambioTamano( GLFWwindow* ventana, int nuevoAncho, int nuevoAlto )
   { .... }
FGE_PulsarLevantarTecla( GLFWwindow* ventana, int tecla,  .... )
   { .... }
FGE_PulsarLevantarBotonRaton( GLFWwindow* ventana, int boton, .... )
   { .... }
void Inicializa_GLFW( int argc, char * argv[] )
   { .... }
void Inicializa_OpenGL( )
   { .... }
void BucleEventos_GLFW()
   { .... }
int main( int argc, char *argv[] )
{
   Inicializa_GLFW(argc,argv) ; // crea una ventana
   Inicializa_OpenGL() ;        // inicializa estado del cauce
   BucleEventos_GLFW() ;        // ejecuta el bucle (ver más abajo)
   glfwTerminate();             // cerrar la ventana
}
```

Bucle principal o de gestión de eventos

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

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

Código de inicialización de GLFW (1/2)

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

```
void Inicializa_GLFW( int argc, char * argv[] )
{
   // intentar inicializar, terminar si no se puede
   if ( ! glfwInit() )
   {  cout << "Imposible inicializar GLFW. Termino." << endl ;
      exit(1) ;
   }

   // especificar que función se llamará ante un error de GLFW
   glfwSetErrorCallback( ErrorGLFW );

   // crear la ventana (var. global ventana_glfw), activar el rendering context
   ventana_glfw = glfwCreateWindow( ventana_tam_x, ventana_tam_y,
                        "Practicas IG (19-20)", nullptr, nullptr );
   glfwMakeContextCurrent( ventana_glfw ); // necesario para OpenGL

   .....
}
```

Código de inicialización de GLFW (2/2)

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

```
void Inicializa_GLFW( int argc, char * argv[] )
{
   .....

  // definir cuales son las funciones gestoras de eventos...
  glfwSetWindowSizeCallback ( ventana_glfw, FGE_CambioTamano );
  glfwSetKeyCallback        ( ventana_glfw, FGE_PulsarLevantarTecla );
  glfwSetMouseButtonCallback( ventana_glfw, FGE_PulsarLevantarBotonRaton);
  glfwSetCursorPosCallback  ( ventana_glfw, FGE_MovimientoRaton );
  glfwSetScrollCallback     ( ventana_glfw, FGE_Scroll );
}
```

Bucle principal en GLFW (sin animaciones)

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

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

Subsección 3.3.
Tipos de primitivas..

Especificación de primitivas

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

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

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

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

Tipos de primitivas: puntos y segmentos

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

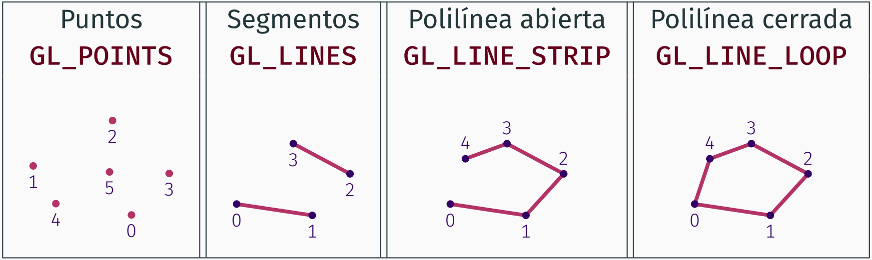
- n **puntos** aislados (n arbitrario).
- uno o varios segmentos de recta, en concreto:
  - $n/2$ **segmentos independientes** (n par).
  - $n-1$ segmentos formando una **polilínea abierta** ($n \geq 2$).
  - n segmentos formando una **polilínea cerrada** ($n \geq 3$).

Tipos de primitivas: polígonos

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

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

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



- En OpenGL se definen varias constantes de tipo `GLenum` (entero sin signo) para identificar los distintos tipos de primitivas
- En estos casos se usan las constantes `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP` o `GL_LINE_LOOP`)

Polígonos delanteros y traseros. Cribado.

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

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

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

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

Esta clasificación tiene utilidad especialmente en visualización 3D.

Modo de visualización de polígonos.

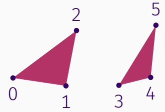
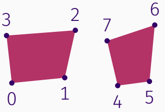
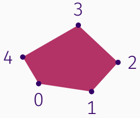
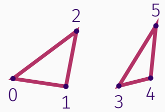
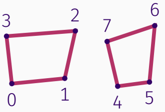
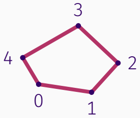
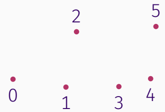
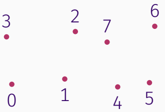
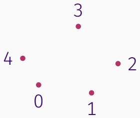
En el caso de las primitivas de polígonos, OpenGL puede visualizarlos de varias formas, según el valor de un parámetro de configuración en el estado de OpenGL, que se llama el **modo de visualización de polígonos**, y que permite seleccionar una de estas opciones:

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

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

Primitivas tipo polígonos (no adyacentes)

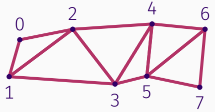
Visualización de una secuencia de n vértices: $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_{n-1}$

Primitivas/ Modo de pol	Triángulos **GL_TRIANGLES**	Cuadriláteros **GL_QUADS**	Polígono **GL_POLYGON**
Modo relleno **GL_FILL**			
Modo líneas **GL_LINE**			
Modo puntos **GL_POINT**			

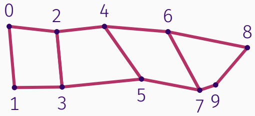
Primitivas tipo polígonos (adyacentes)

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

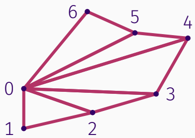
Tira de triángulos
GL_TRIANGLE_STRIP
Polígonos:
$(0,1,2),\ (2,1,3),\ (2,3,4),$
$(4,3,5),\ (4,5,6),\ (6,5,7),\ \ldots$



Tira de cuadriláteros
GL_QUAD_STRIP
Polígonos:
$(0,1,3,2),\ (2,3,5,4),$
$(4,5,7,6),\ (6,7,9,8),\ \ldots$



Abanico de triángulos
GL_TRIANGLE_FAN
Polígonos:
$(0,1,2),\ (0,2,3),\ (0,3,4),$
$(0,4,5),\ (0,5,6),\ \ldots$



Polígonos de más de tres vértices

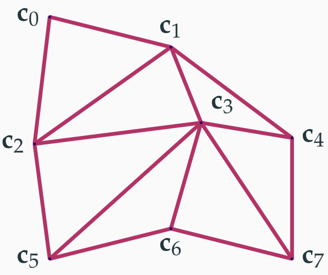
Respecto de las primitivas de tipo polígono de más de 3 vértices y los cuadriláteros:

- Deben cumplir estos requisitos:
  - Deben tener todos sus vértices en el mismo plano
  - Las aristas no deben intersecarse entre ellas
  - Deben de ser convexos

  (si no cumplen alguno de ellos, no se visualizan correctamente).
- Internamente, se convierten en triángulos (las GPUs solo rasterizan triángulos). Se dice que los polígonos son *teselados*.
- En la versión 3.0 de OpenGL (2008), se declararon *obsoletas* este tipo de primitivas, y en posteriores se eliminaron (no existen las constantes **`GL_POLYGON`**, **`GL_QUADS`** ni **`GL_QUAD_STRIP`**).

Problema de vértices replicados

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



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

$$\{ \quad \mathbf{c}_0, \mathbf{c}_2, \mathbf{c}_1, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_1, \mathbf{c}_3, \mathbf{c}_4, \mathbf{c}_2, \mathbf{c}_5, \mathbf{c}_3, \mathbf{c}_3, \mathbf{c}_5, \mathbf{c}_6, \mathbf{c}_3, \mathbf{c}_6, \mathbf{c}_7, \mathbf{c}_3, \mathbf{c}_7, \mathbf{c}_4 \}$$

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

Secuencias indexadas

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

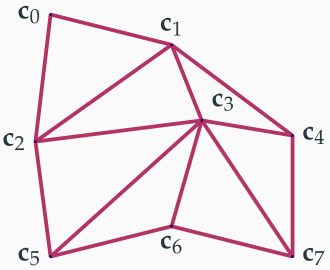
- Se parte de una secuencia V_n de n coordenadas arbitrarias de vértices $V_n = \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\}$, que pueden ser únicos.
- Se usa una secuencia I_m de m **índices** $I_m = \{i_0, i_1, \ldots, i_{m-1}\}$ donde cada valor i_j es un entero entre 0 y $n-1$ (ambos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

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

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

Ejemplo de secuencia indexada

En este ejemplo que hemos visto antes



Haríamos:

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

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

Subsección 3.4.
Atributos de vértices.

Atributos de vértices

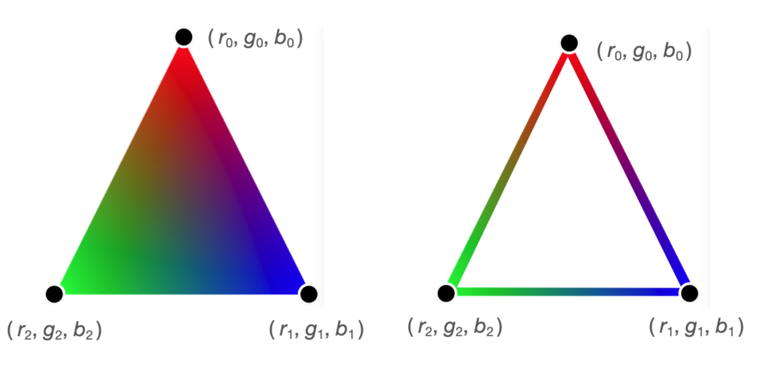
Las coordenadas de su posición se considera un **atributo** de los vértices, es un atributo imprescindible, pero en rasterización se asignan a cada vértice otros:

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

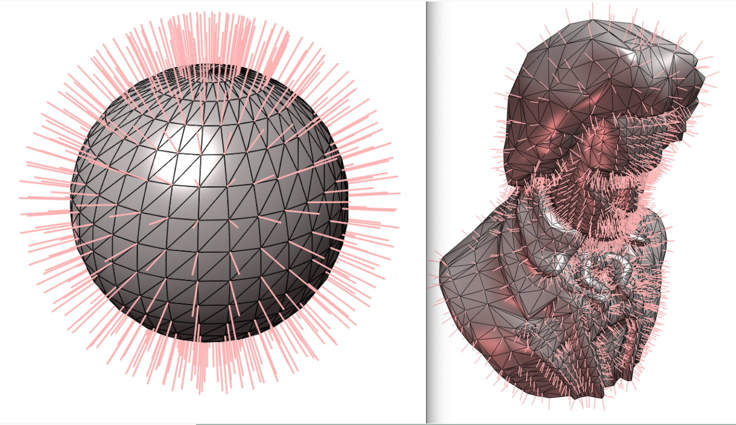
En el cauce progamable moderno de OpenGL se pueden definir tantos atributos como queramos. Nosotros veremos como usar estos tres junto con la posición, usando llamadas de OpenGL 2.1.

Atributos: colores de vértices

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

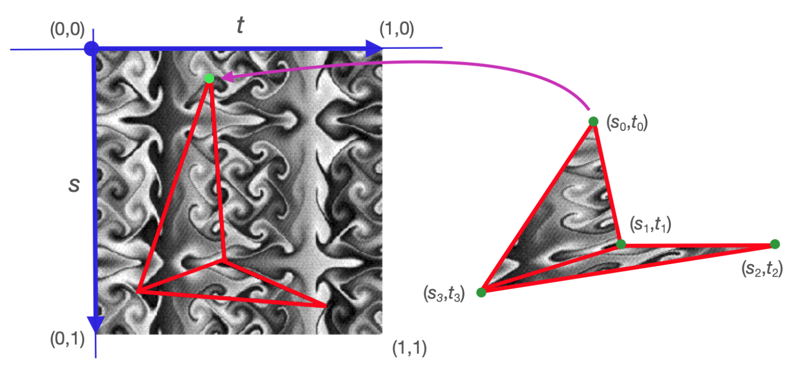


En visualización 3D, a cada vértice se le puede asociar un vector de 3 componentes (x, y, z) (su vector **normal**) que determina la orientación de la superficie en ese vértice y sirve para hacer el sombreado y la iluminación:



Atributos: coordenadas de textura

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



Definición de valores de atributos

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

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

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

Subsección 3.5.
Modos de envío..

Modos de envío

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

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

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

Envío en *Modo Inmediato*

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

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

Envío en *Modo Diferido*

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

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

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

Subsección 3.6.
Almacenamiento de vértices y atributos..

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

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

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

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

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

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

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

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

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

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

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

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

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

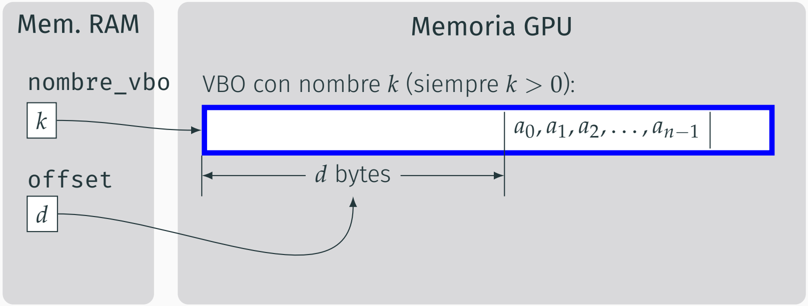
Vertex Buffer Objects para modo diferido

El modo diferido requiere reservar memoria en la GPU, para ello se usan los ***Vertex Buffer Objects*** (**VBOs**). Un VBO es un bloque de bytes contiguos en la memoria de la GPU:

- La aplicación puede crear VBOs (reservar un bloque de memoria) en cualquier momento, indicando el tamaño en bytes de cada VBO.
- La aplicación identifica cada VBO por un valor entero único, mayor estricto que cero, único para cada VBO, y que se llama **nombre de VBO**. Es de tipo **`GLuint`** (equivale a **`unsigned int`**).
- Una dirección de memoria en la GPU viene determinada por un par formado por un nombre de VBO y un **desplazamiento** (***offset***). Este offset es el número de bytes desde el inicio del VBO hasta la dirección de memoria.

Tablas en VBOs: identificador y offset

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



Descriptores de tabla (1/3)

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

1. **Tipo de tabla** (`tipo_tabla`): un valor entero para indicar si es una tabla de índices (`GL_ELEMENT_ARRAY_BUFFER`) o bien una tabla de coordenadas o atributos (`GL_ARRAY_BUFFER`)

2. **Atributo** (`atributo`): en el caso de tablas de coordenadas o atributos, se usa otro valor entero para discriminar de que tabla concreta se trata. Puede valer:
   - `GL_VERTEX_ARRAY` : coordenadas (posiciones)
   - `GL_COLOR_ARRAY` : colores.
   - `GL_NORMAL_ARRAY` : normales.
   - `GL_TEXTURE_COORD_ARRAY`: coordenadas de textura.

Descriptores de tabla (2/3)

3. **Tipo de valores** (`tipo_valores`): valor que codifica el tipo de datos. Usaremos `GL_UNSIGNED_INT` para los índices, y podemos usar los valores `GL_FLOAT` (para `float`) o `GL_DOUBLE` (para `double`), tanto para las coordenadas como para los atributos.

4. **Número de bytes por valor** (`num_bytes_valor`), se puede calcular con `sizeof` exclusivamente en función de `tipo_valores` (> 0).

5. **Número de valores por tupla** (`num_vals_tupla`): número de valores reales por vértice, puede ser 2, 3 o 4 para las tablas de atributos, y debe ser 1 para las tablas de índices (> 0).

6. **Número de tuplas o índices** (`num_tuplas_ind`): Para tablas de atributos es el número de vértices, para las de índices es el número de índices (> 0).

Descriptores de tabla (3/3)

7. **Tamaño en bytes** (**`tamano_en_bytes`**) tamaño de la tabla completa, se calcula como producto de **`num_bytes_valor`**, **`num_vals_tupla`** y **`num_tuplas_ind`**.

8. **Puntero a datos** (**`datos`**): puntero a la dirección de memoria del programa donde está el primer byte de la tabla (no puede ser nulo) (se usa para envío en modo inmediato).
9. **Nombre del VBO** (**`nombre_vbo`**): vale 0 si la tabla solo está en la memoria del programa, y un valor positivo si se ha hecho una copia en la memoria de la GPU (se usa para envío en modo diferido)

La clase DescTabla para descriptores de tablas

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

```
class DescrTabla
{
 public: // constructor: inicializa y comprueba que los datos son correctos
 DescrTabla(const GLenum  p_tipo_tabla,     const GLenum  p_atributo,
            const GLenum  p_tipo_valores,   const GLint   p_num_vals_tupla,
            const GLsizei p_num_tuplas_ind, const GLvoid* p_datos );
   .......
  private:
   GLenum         tipo_tabla      = 0 ;
   GLenum         atributo        = 0 ;
   GLenum         tipo_valores    = 0 ;
   GLsizeiptr     num_bytes_valor = 0 ;
   GLint          num_vals_tupla  = 0 ;
   GLsizei        num_tuplas_ind  = 0 ;
   GLsizeiptr     tamano_en_bytes = 0 ;
   GLuint         nombre_vbo      = 0 ; // inicialmente no hay nada en GPU
   const GLvoid * datos = nullptr ;  // los datos no se modifican desde aquí
} ;
```

Datos sobre secuencias de vértices

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

- localización en memoria y formato de la tabla de coordenadas
- para cada tabla de atributos, y para la tabla de índices
  - un valor lógico que indica si se usará o no dicha tabla.
  - si se usa, la localización en memoria y el formato de la tabla.

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

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

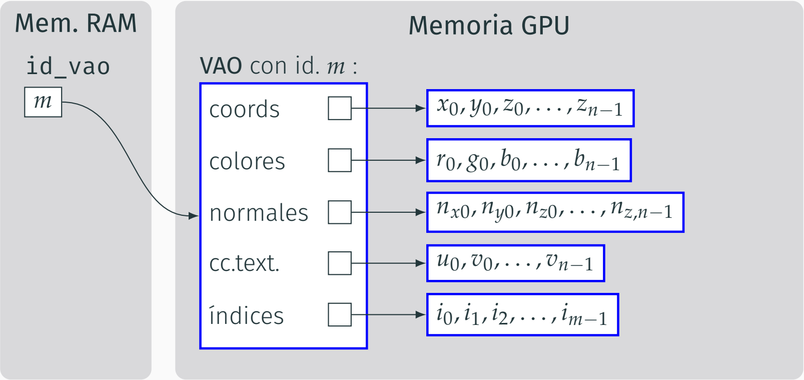
Vertex Array Objects (VAOs) de OpenGL

Un ***Vertex Array Objects*** (VAO) es una estructura de datos que forma parte del estado de OpenGL y que contiene toda la información sobre una secuencia de vértices.

- Para visualización en modo inmediato OpenGL guarda una único VAO en su estado. Por tanto, es necesario actualizar el VAO antes de visualizar una nueva secuencia de vértices distinta.
- Para visualización en modo diferido, la aplicación puede crear tantos VAOs distintos como sea necesario (alojados en la GPU).
- Cada VAO creado por la aplicación tiene asociado un entero único, > 0, que es el *nombre del VAO*.
- El VAO por defecto es el VAO con nombre 0, ya creado y activo al inicio. Es el único que se puede usar en modo inmediato.

Esquema de un VAO para modo diferido

La aplicación únicamente necesita tener el identificador de VAO (**`nombre_vao`**) en una variable:



Cada tabla referenciada en un VAO (con identificador > 0) debe estar en la GPU.

La clase `ArrayVertices` (1/2)

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

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

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

La clase ArrayVertices (2/2)

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

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

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

Representación de secuencias. Estructras de datos en C++11.

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

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

Declaraciones de tablas

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

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

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

Creación e inicialización de objetos ArrayVertices

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

```
// crear un objeto (av) que encapsula toda la información de las tablas, dar coords.
// (el objeto 'av' no modifica ni destruye las tablas, solo las lee)
ArrayVertices * av = new ArrayVertices( GL_FLOAT, 3,
                          coordenadas.size(), coordenadas.data() );

// opcionalmente: añadir a 'av' información sobre atributos o índices
av->fijarColores  ( GL_FLOAT, 3, colores.data() );
av->fijarNormales ( GL_FLOAT,    normales.data() );
av->fijarCoordText( GL_FLOAT, 2, coord_text.data() );
av->fijarIndices  ( GL_UNSIGNED_INT, indices.size(), indices.data() );
```

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

Subsección 3.7.
Envío de vértices y atributos..

Funciones de envío en modo inmediato

Las coordenadas se pueden enviar de tres formas:

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

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

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

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

Envío con glBegin/glEnd

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

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

- Aquí **tipo_prim** es una expresión de tipo **GLenum** (entero sin signo) que codifica el tipo de primitiva.
- Podríamos usar **GL_TRIANGLES** para triángulos (o bien **GL_POINTS**, **GL_LINES**, **GL_LINE_STRIP**, **GL_LINE_LOOP**, etc...)

Envío de atributos con `glBegin/glEnd`

Si queremos enviar atributos, usamos las funciones:

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

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

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

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

Envío con glBegin/glEnd: coords. en arrays

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

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

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

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

Variantes de `glVertex`

Existen otras funciones en la familia de **`glVertex`**

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

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

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

Visualización de arrays de vértices con `begin/end`

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

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

El método visualizarGL_MI_BVE de ArrayVertices

El método de visualización accede a las tablas del array de vértices:

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

Envío con una única llamada

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

- Para una secuencia **no indexada**:

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

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

- Para una secuencia **indexada**:

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

En ambos casos es necesario especificar antes donde se encuentran la tabla de vértices y las de atributos que queramos usar.

Especificación de localización y estructura de tablas

Este método de **DescrTabla** indica a OpenGL la localización (puntero a memoria u offset de VBO) y formato de una tabla de atributos:

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

Activación de una tabla de atributos en M.I.

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

```
void DescrTabla::activar_mi()
{
   assert( tipo_tabla != GL_ARRAY_BUFFER );
   fijarPuntero( datos ); // especifica localización y formato
   glEnableClientState( atributo ); // habilita el uso de la tabla en OpenGL
}
```

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

Visualización en M.I.

Este método activa las tablas necesarias y luego visualiza una secuencia de vértices con una única llamada, en M.I. (el tipo de primitiva es el parámetro `tp`)

```
void ArrayVertices::visualizarGL_MI_DAE( const GLenum tp )
{
  glBindVertexArray( 0 );     // activa VAO por defecto
  deshabilitar_tablas();      // deshab. todas la tablas
  coordenadas->activar_mi(); // activar la tabla de coordenadas
  // activar las tablas de atributos que corresponda (en modo inmediato)
  if ( colores  != nullptr ) colores->activar_mi() ;
  if ( normales != nullptr ) normales->activar_mi() ;
  if ( coords_textura != nullptr ) coords_textura->activar_mi() ;
  // invocar 'glDrawElements' o 'glDrawArrays' en función de si hay o no hay índices
  if ( indices != nullptr )
   glDrawElements( tp, num_indices, indices->tipo_valores, indices->datos);
  else
   glDrawArrays( tp, 0, num_vertices );
  // dejar tablas deshabilitadas para visualizar otros arrays de vértices
  deshabilitar_tablas();
}
```

Deshabilitación de tablas

El método **`deshabilitar_tablas`** usa **`glDisableClientState`** para deshabilitar todas las tablas de atributos.

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

Debe usarse al inicio y al final de **`visualizarGL_MI_DAE`** para desahbilitar las tablas anteriormente habilitadas (al inicio) y para que no se quede ninguna habilitada (al final).

Funciones de envío en modo diferido

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

- almacenar las secuencias de coordenadas, atributos e índices (si procede) en uno o varios VBOs en la GPU,
- almacenar la información de la localización y formato de las tablas en un VAO,
- antes de la primera visualización, debemos
  - crear e inicializar los VBOs necesarios.
  - crear e inicializar el VAO
- en cada visualización solo es necesario activar el VAO y visualizar.

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

Activación de tablas en modo diferido

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

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

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

Visualización de un ArrayVertices en modo diferido

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

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

Subsección 3.8.
Visualización y parámetros.

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

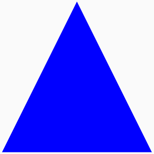
El modo de visualización de polígonos se cambia usando la llamada:

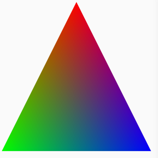
```
glPolygonMode( GL_FRONT_AND_BACK, nuevo_modo )
```

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

Color de vértices y modos de sombreado

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

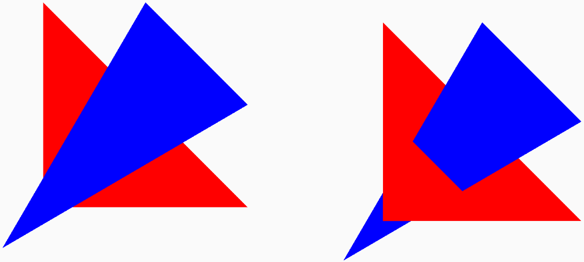
Eliminación de partes ocultas (EPO) con Z-buffer

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*).
- Permite dibujar primitivas en 3D con posibles ocultaciones entre ellas.
- Inicialmente (por defecto) en un pixel, una primitiva A con una Z menor estará por delante de otra B con una Z mayor (A oculta a B).
- Esto puede activarse o desactivarse, con `glEnable` y `glDisable`, usando `GL_DEPTH_TEST` como argumento. Inicialmente, **está desactivado**.

Ejemplo de EPO con Z-buffer.

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



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

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

Ejemplo de función de redibujado

Un ejemplo sencillo para la función de redibujado es esta:

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

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

   // envío de secuencias de vértices (dibujamos varios objetos)
   objeto1.visualizarGL_MD_VAO( tp1 );
   objeto2.visualizarGL_MD_VAO( tp2 );
   .....

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

Detección de errores de OpenGL

Las funciones OpenGL pueden activar 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);
   }
}
```

Otros parámetros de visualización

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

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

Inicialización de OpenGL

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

```
void Inicializa_OpenGL( )
{
   // comprobar si el flag de error de OpenGL ya estaba activiado (si estaba aborta)
   CError();
   // establecer color de fondo: (1,1,1) (blanco)
   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 );
   // 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.

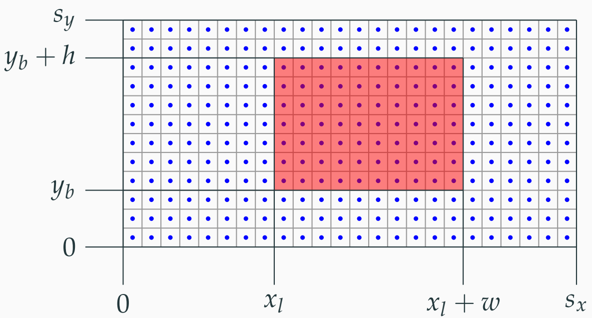
```
glViewport( izqui, abajo, ancho, alto ) ;
```

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.

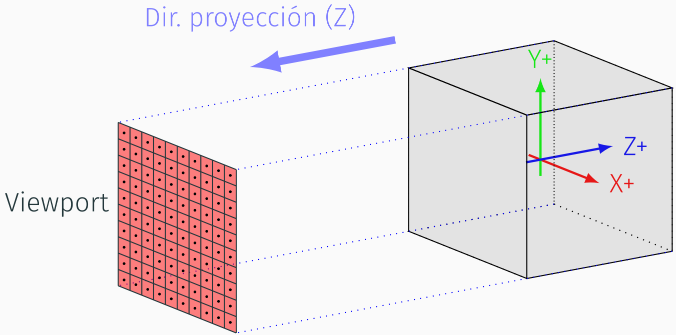
El viewport y la ventana como rejillas de pixels

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.

Subsección 4.1.
El cauce gráfico. Tipos. Shaders..

Transformación y sombreado

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

1. **Transformación:**
   En esta etapa se parte de 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.

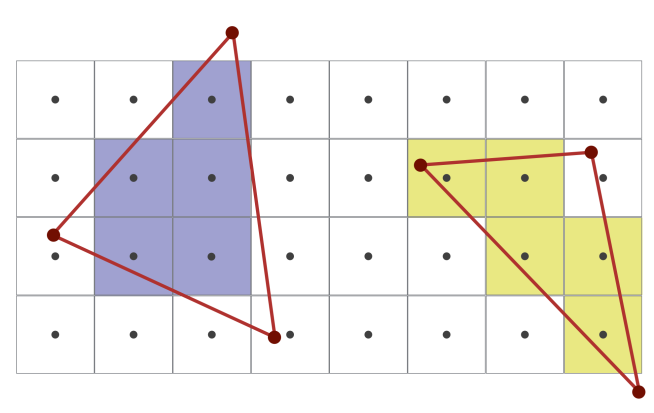
Los shaders. Tipos.

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

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

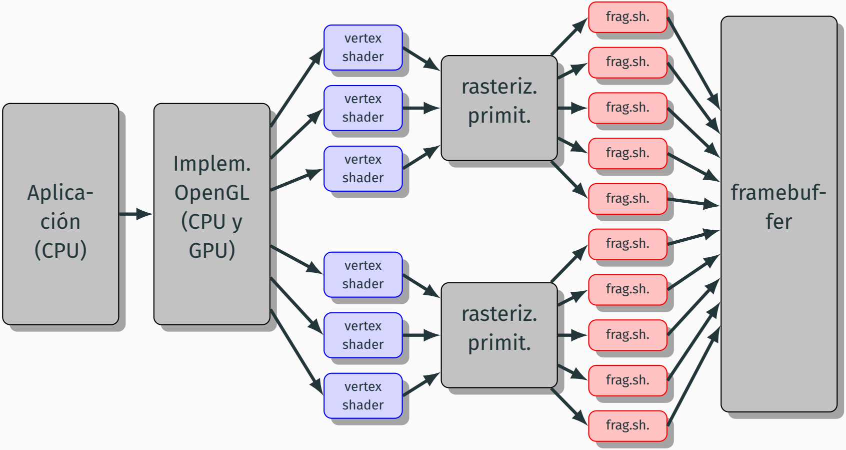
Ejemplo: Visualización de 2 triángulos

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



Cauce gráfico: DFD simplificado

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



Tipos de cauces gráficos:

Hay dos tipos de cauce gráfico:

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

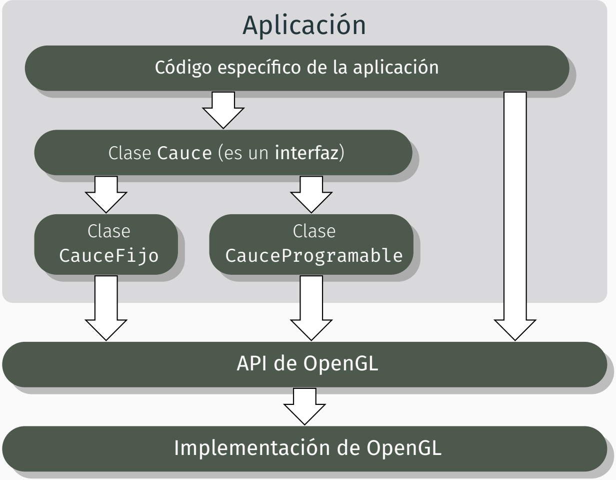
Implementación de cauce fijo/programable

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

- Algunas órdenes de OpenGL para configurar el cauce fijo difieren de las usadas para el cauce programable
- Algunas órdenes de OpenGL son iguales para ambos tipos de cauce.
- Usaremos la clase abstracta (`Cauce`) para realizar un acceso uniforme a ambos tipos de cauce. Esta clase incluye **métodos virtuales puros**.
- De esa clase se derivan dos clases: `CauceFijo` y `CauceProgramable`, cada una de ellas contiene implementaciones distintas de cada método virtual de la clase base `Cauce`.

Clases para acceso al cauce

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



Ejemplo: fijar evaluación de iluminación

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

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

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

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

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

Subsección 4.2.
Estructura de los shader. Un ejemplo sencillo..

Creación y uso de shaders

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

- Los fuentes de los dos shaders deben estar almacenados en memoria en variables de tipo `char *` (vectores de caracteres o cadenas, acabados en 0). Es conveniente 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.

Elementos del fuente de los shaders.

El código fuente de una shader tiene declaraciones de:

- Parámetros **uniform**: valores proporcionados por la aplicación (son constantes para cada secuencia de vértices).
- Variables **varying**: valores calculados el vertex shader en cada vértice, y legibles (interpolados) por el fragment shader en cada pixel. En GLSL moderno se llaman variables **out** (en el vertex shader) o variables **in** (en el fragment shader).
- Atributos de vértices: no son siempre necesarias: podemos usar declaraciones implícitas.
- Función **main**: es la única función obligatoria.
- Funciones auxiliares: llamadas directa o indirectamente desde **main**.

Entradas y salidas según el tipo de shader.

Vertex Shaders (se ejecutan una vez por vértice)

- Entrada: **parámetros uniform** y la posición y atributos de cada vértice enviados por la aplicación en las variables predefinidas: **`gl_Vertex`**, **`gl_Color`**, **`gl_Normal`** y **`gl_MultiTexCoord0`**.
- Salida: **variables varying** (u **out**).

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

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

Vertex shader elemental:

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

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

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_Color, gl_MultiTextCoord0

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

Fragment shader elemental:

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

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

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

Subsección 4.3.
Creación y ejecución de programas..

Identificación y activación de *shader programs*

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

Funciones para compilar y enlazar shaders

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

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

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

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

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

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

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

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

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

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

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

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

```
CauceProgramable::CauceProgramable()
{
   // inicializar los nombres de los archivos fuente:
   frag_fn = "../recursos/shaders/cauce21-frag.glsl" ;
   vert_fn = "../recursos/shaders/cauce21-vert.glsl" ;

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

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

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

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

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

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

```
   ....
   // obtener localizaciones de params. uniforms
   loc_eval_mil    = glGetUniformLocation( id_prog, "eval_mil" );
   loc_sombr_plano = glGetUniformLocation( id_prog, "sombr_plano" );
   loc_eval_text   = glGetUniformLocation( id_prog, "eval_text" );

   // inicializar parámetros uniform a valores por defecto
   glUniform1i( loc_eval_mil,    0 );
   glUniform1i( loc_sombr_plano, 0 );
   glUniform1i( loc_eval_text,   0 );

} // fin del constructor
```

Inicialización del cauce programable.

En la función de inicialización de OpenGL es necesario:

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

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

void Inicializa_OpenGL()
{
   Inicializar_GLEW() ;
   // hacer el resto de inicializaciones (igual que antes)
   .......
   // compilar shaders, crear cauces programable y fijo
   Cauce * cauce_prog = new CauceProgramable(),
         * cauce_fijo = new CauceFijo(),
         * cauce      = ..... ; // usar uno de los dos.
}
```

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

Uso de un programa.

Lo usual es que durante la inicialización de OpenGL

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

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

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

Lo usual es que todo esto se encapsule en clases que permitan portabilidad y sencillez. Nosotros usamos las clases **`CauceFijo`** y **`CauceProgramable`**, ya citadas.

Uso de la clase Cauce y derivadas

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

```
void CauceFijo::activar()        { glUseProgram( 0 );       }
void CauceProgramable::activar() { glUseProgram( id_prog ); }
```

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

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

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

Subsección 4.4.
Funciones auxiliares..

Verificar errores de compilación

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

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

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

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

Verificar errores de enlazado

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

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

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

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

Lectura de un archivo

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

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

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

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

Inicialización de GLEW

En Linux y Windows es necesario leer punteros a las funciones de OpenGL nuevas de la versión 2.0 o poesteriores. Para eso usamos la librería GLEW (en macOS la función no hace nada, es innecesario).

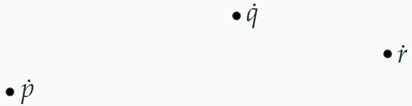
```
void Inicializar_GLEW()
{
   // hacer init de GLEW y comprobar errores
   GLenum codigoError = glewInit();
   if ( codigoError != GLEW_OK )
   {  std::cout << "Imposible inicializar 'GLEW', mensaje: "
                << glewGetErrorString(codigoError) << std::endl ;
      exit(1);
   }
   // comprobar si OpenGL ver 2.0 + está soportado
   if ( ! GLEW_VERSION_2_0 )
   {  cout << "OpenGL 2.0 no soportado." << endl << flush ;
      exit(1);
   }
}
```

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

Subsección 5.1.
Puntos y vectores.

Puntos y vectores

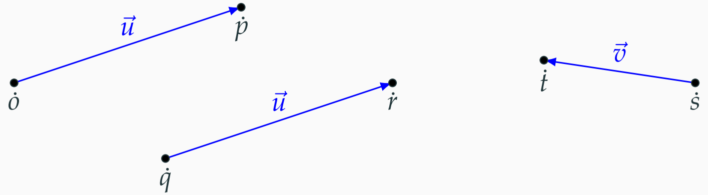
Los modelos 3D y 2D de objetos y figuras que vamos a repsesentar se pueden construir cada uno de ellos en base a una conjunto abstracto (con estructura de **espacio afín**), cuyos elementos son puntos de un determinado espacio donde imaginamos el modelo. Cada uno de estos **puntos** o **localizaciones**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



- 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

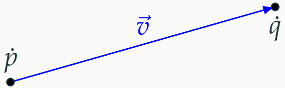
Además del conjunto de puntos, cada modelo tiene asociado un conjunto o espacio de vectores. Cada par de puntos del espacio tiene asociado un **vector** (o **vector libre**), los representamos con flechas $\vec{u}, \vec{v}, \ldots$)



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

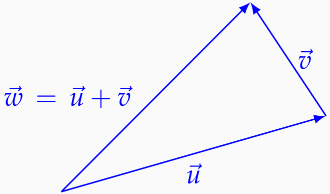
Resta de puntos, suma de vectores

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



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

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

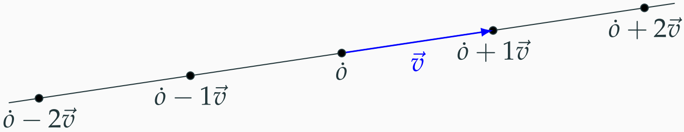


$$\vec{u} + \vec{v} = \vec{w} \iff \vec{v} = \vec{w} - \vec{u}$$

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

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

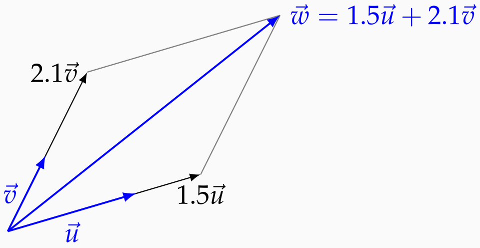
Como consecuencia, todos los puntos de la forma $\dot{o} + t\vec{v}$ (para todos los valores reales posibles de t) están en la recta que pasa por $\dot{o}$ y es paralela a $\vec{v}$



Subsección 5.2.
Marcos de referencia y coordenadas.

Bases de vectores

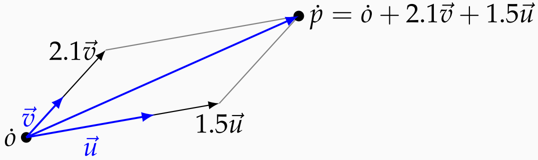
Usando dos vectores cualquiera $\vec{u}$ y $\vec{v}$ del plano (no paralelos ni nulos), podemos escribir cualquier otro vector $\vec{w}$ como una combinación lineal de ellos:



- 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 $\dot{p} = \dot{o} + a\vec{u} + b\vec{v}$:

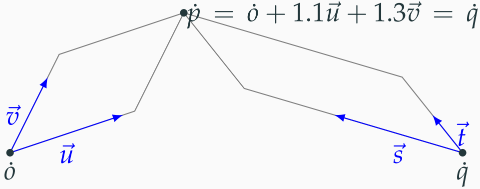


- La terna $\mathcal{R} = [\vec{u},\vec{v},\dot{o}]$ forma un **marco de referencia** (*reference frame*) del plano 2D.
- Un marco sirve para **identificar puntos y vectores usando distancias (valores reales)**.
- Al par (a,b) se le llaman las **coordenadas** del punto $\dot{p}$ en el marco de referencia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

Coordenadas y puntos

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

$$\dot{p} \;=\; \dot{o} + 1.1\vec{u} + 1.3\vec{v} \;=\; \dot{q} + 1.7\vec{s} + 2.0\vec{t}$$



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

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

Unas coordenadas **no tienen signficado** 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 $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots)$ 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:

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

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

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

Coordenadas homogéneas de vectores

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

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

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

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

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

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

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

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

Subsección 5.4.
Operaciones entre vectores: producto escalar y vectorial.

El marco de referencia especial

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

- los vectores $\vec{x}$, $\vec{y}$ y $\vec{z}$ **tienen longitud unidad**: por tanto estos vectores determinarán la longitud de todos los demás, es decir: definen la unidad de longitud en el espacio de coordenadas.
- los vectores $\vec{x}$, $\vec{y}$ y $\vec{z}$ son **perpendiculares entre ellos dos a dos**: por tanto, esos vectores forman ángulos de 90 grados, y determinan los angulos entre cualquiera dos vectores.

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

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

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

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

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

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

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

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

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

Longitud y perpendicularidad de vectores

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

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

Es fácil demostrar que se cumple

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

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

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

Producto vectorial o externo en 3D

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

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

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

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

Marcos cartesianos en 2D

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

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

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

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

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

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

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

En estas condiciones, decimos que $\mathcal{C}$ es un marco de referencia **cartesiano**. (el marco de referencia $\mathcal{E}$ es cartesiano por definición).

Marcos cartesianos en 3D

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

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

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

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

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

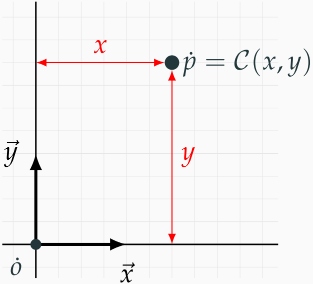
- La *orientación es semejante a la de $\mathcal{E}$*, es decir:

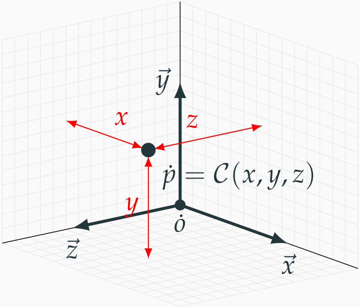
$$\vec{e}_x \times \vec{e}_y \;=\; \vec{e}_z \qquad \vec{e}_y \times \vec{e}_z \;=\; \vec{e}_x \qquad \vec{e}_z \times \vec{e}_x \;=\; \vec{e}_y$$

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).

Marcos ortogonales y ortonormales. Orientación.

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

Calculo del producto escalar y el módulo

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

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

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

  (este valor sería el mismo si usaemos las coordenadas de cualquier otro marco cartesiano distinto de $\mathcal{C}$).
- Como consecuencia, el módulo se puede obtener como:

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

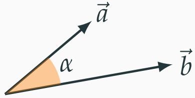
- El módulo de un vector coincide con su **longitud** en el espacio (ya que de los versores de $\mathcal{E}$ dijimos que tenían longitud unidad por definición). El módulo, calculado así, es siempre el mismo en cualquier marco cartesiano.

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:

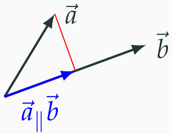
- El producto escalar es proporcional al coseno de α:

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



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

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



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

Cálculo del producto vectorial

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

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

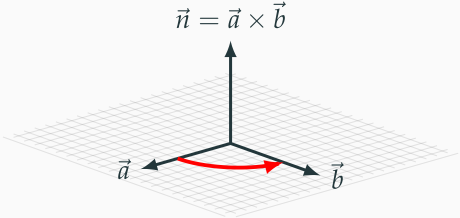
$$\mathcal{C}\begin{pmatrix} a_x \\ a_y \\ a_z \\ 0 \end{pmatrix} \times \mathcal{C}\begin{pmatrix} b_x \\ b_y \\ b_z \\ 0 \end{pmatrix} = \mathcal{C}\begin{pmatrix} a_y b_z - a_z b_x \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \\ 0 \end{pmatrix}$$

- 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)

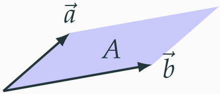
En un marco cartesiano

- 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 parelepípedo 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**

> Una **transformación geométrica** T es una aplicación que asocia a cualquier punto $\dot{p}$ de un espacio afín otro punto $\dot{q}$ del mismo (u otro) espacio afín. Escribimos
>
> $$\dot{q} \quad = \quad T(\dot{p})$$
>
> decimos: $\dot{q}$ es T aplicado a $\dot{p}$, o bien $\dot{q}$ es la imagen de $\dot{p}$ a través de T.

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

Transformación de coordenadas

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

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

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

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

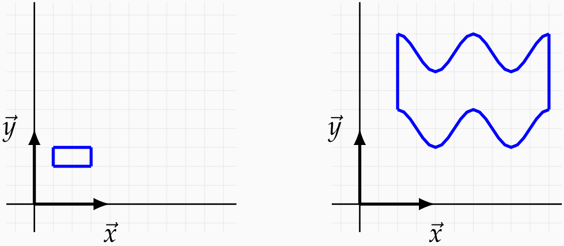
Lógicamente, para una única transformación T, las funciones f_0, f_1 y f_2 **dependen del sistema de referencia** $\mathcal{R}$ en uso.

Ejemplo de transformación en 2D

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

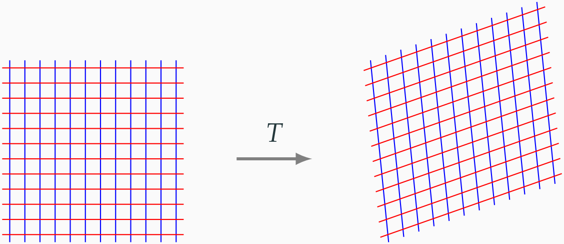
$$f_0(x,y) \;=\; 4x-1 \qquad f_1(x,y) \;=\; 2y + \frac{2+\cos\left((8x-2)\,\pi\right)}{4}$$

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 conserva el paralelismo, por tanto:

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

Esto permite extender T a los vectores del espacio afín:

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

Como consecuencia, podemos caracterizar las transformaciones afines:

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

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

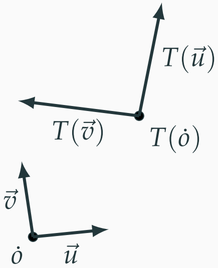
Subsección 5.6.
Matrices de transformación.

Transformación de marcos

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

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

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



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

Transformación de coordenadas

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

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

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

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

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:

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

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 4x4

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

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

$$\begin{tikzcd} \dot{p} \arrow[r, "A"] \arrow[rr, bend left, "C"] & \dot{q} \arrow[r, "B"] & \dot{r} \end{tikzcd} = B(\dot{q}) = B(A(\dot{p})) = (B \circ A)(\dot{p}) = C(\dot{p})$$

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

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

la composición es **asociativa**

Transformación inversa

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

$$
\begin{tikzcd}
T^{-1}(T(\dot{p})) = \dot{p} \arrow[rr, bend left, "T"] & & \dot{q} = T(T^{-1}(\dot{q})) \arrow[ll, bend left, "T^{-1}"]
\end{tikzcd}
$$

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,...,T_n de n transformaciones afines. Si consideremamos la transformación compuesta

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

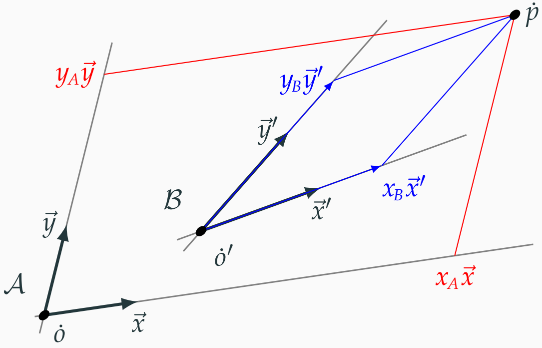
Entonces, la matriz M_C asociada a C será el producto de las matrices M_i asociadas a cada una de las T_i:

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

Nótese que el producto es derecha a izquierda (primero aparecen las mtrices que se aplican después). Esta propieda 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:

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

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

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

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

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

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

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

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

de cualquiera de estas dos igualdades se hace evidente que:

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

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

Descomposición de la inversa

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

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

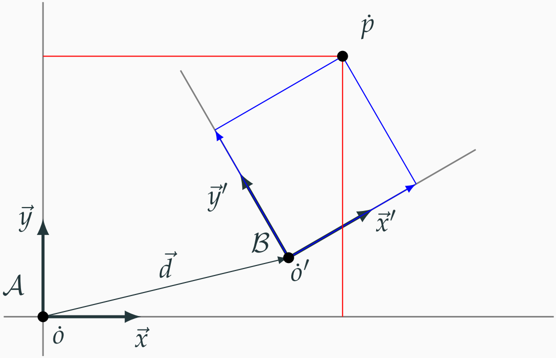
la matriz $D_{\mathcal{A},\mathcal{B}}^{-1}$ es un desplazamiento que lleva $\dot{o}'$ a $\dot{o}$, es decir, un desplazamiento por el vector $\dot{o} - \dot{o}' = \mathcal{A}(-d_x, -d_y, -d_z, 0)^t$.

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

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

Matricialmente, por tanto:

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

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

Subsección 5.7.
Representación y operaciones con tuplas.

Representación en memoria de coordenadas.

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

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

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

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

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

Creación, consulta y modificación de tuplas.

Este código válido ilustra las distintas opciones:

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

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

// accesos de solo lectura, usando su posición o índice en la tupla (0,1,2,...),
// o bien varias constantes predefinidias 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) ;   // apropidado para colores

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

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

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

Operaciones entre tuplas y escalares.

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

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

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 t, t4( 1.0,2.0,3.0,4.0 ) ;  Tupla3f t3(1.0,1.0,3.0);
t = m2*t4 ;  t = m2 * t3 ;
// conversión a puntero a 16 flotantes (float *) (formato OpenGL)
float * pm = m ;   const float * pcm = m ;
// escritura en la salida estándar (varias líneas)
cout << m << endl ;
```

Matrices más usuales

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

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

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

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

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

$\boxed{\text{Fin de la presentación.}}$