Informática Gráfica:
Teoría. Tema 1. Introducción.

Carlos Ureña

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

2017-18

Teoría. Tema 1. Introducción.
Índice.

Sección 1
Introducción

1.1. Concepto y metodologías
1.2. Aplicaciones.

Subsección 1.1
Concepto y metodologías

Concepto

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)

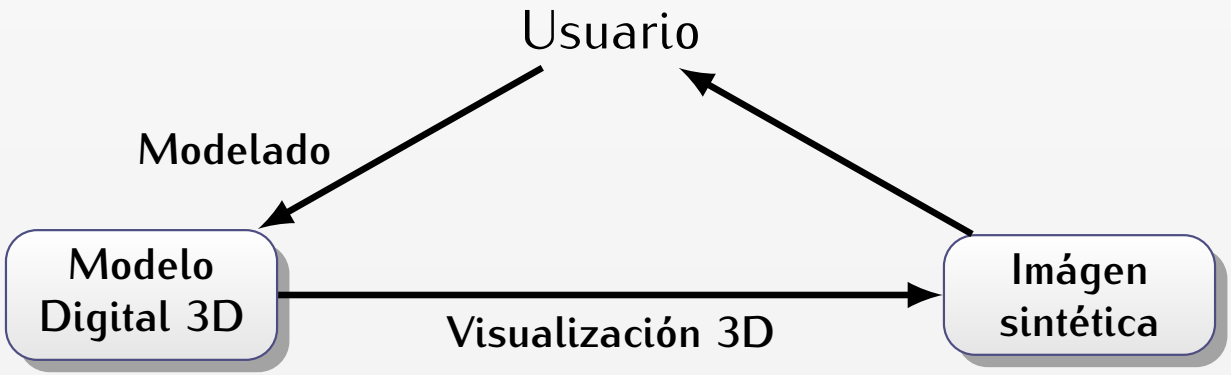
en aplicaciones específicas, se usan otros campos de la Informática en particular o la Ciencia en general (p.ej. para desarrollo de videojuegos se usan también técnicas de Inteligencia Artificial).

Informática Gráfica 3D interactiva

Los elementos esenciales de una aplicación gráfica son:

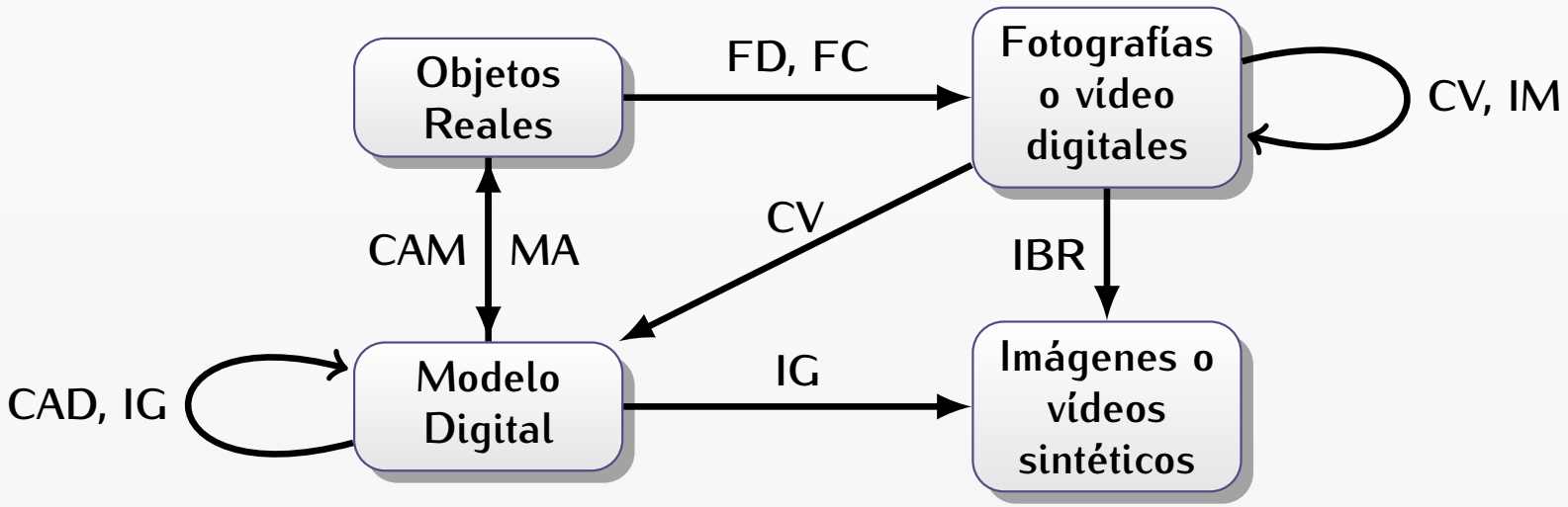
- **Modelos digitales** de objetos reales, ficticios o de datos
- **Imágenes o vídeos digitales** que se usan para visualizar dichos objetos.

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

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

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

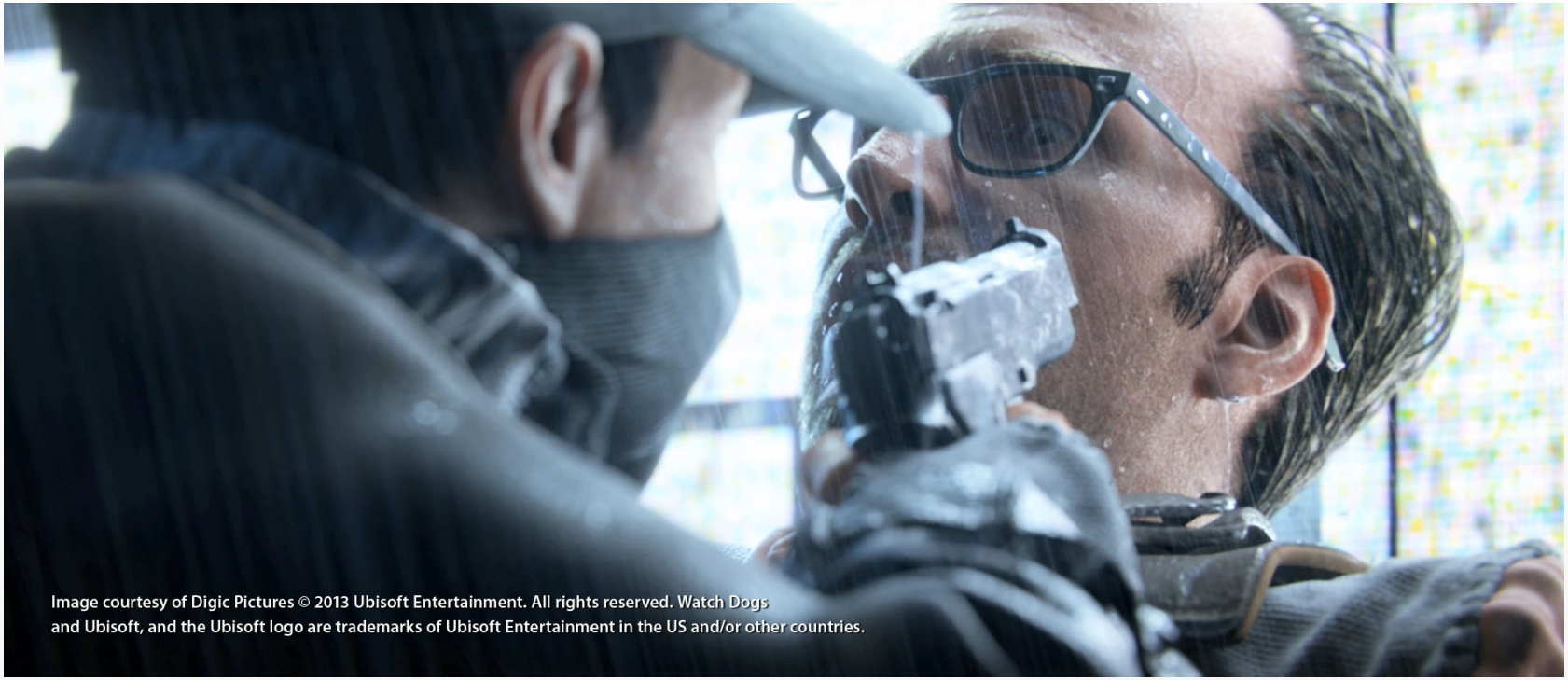
Fotograma del videojuego *Watch Dogs* de Ubisoft.
Vídeo: ☞ http://www.youtube.com/watch?v=kPYgXvgS6Ww

Realidad Aumentada (AR)

Imagen:

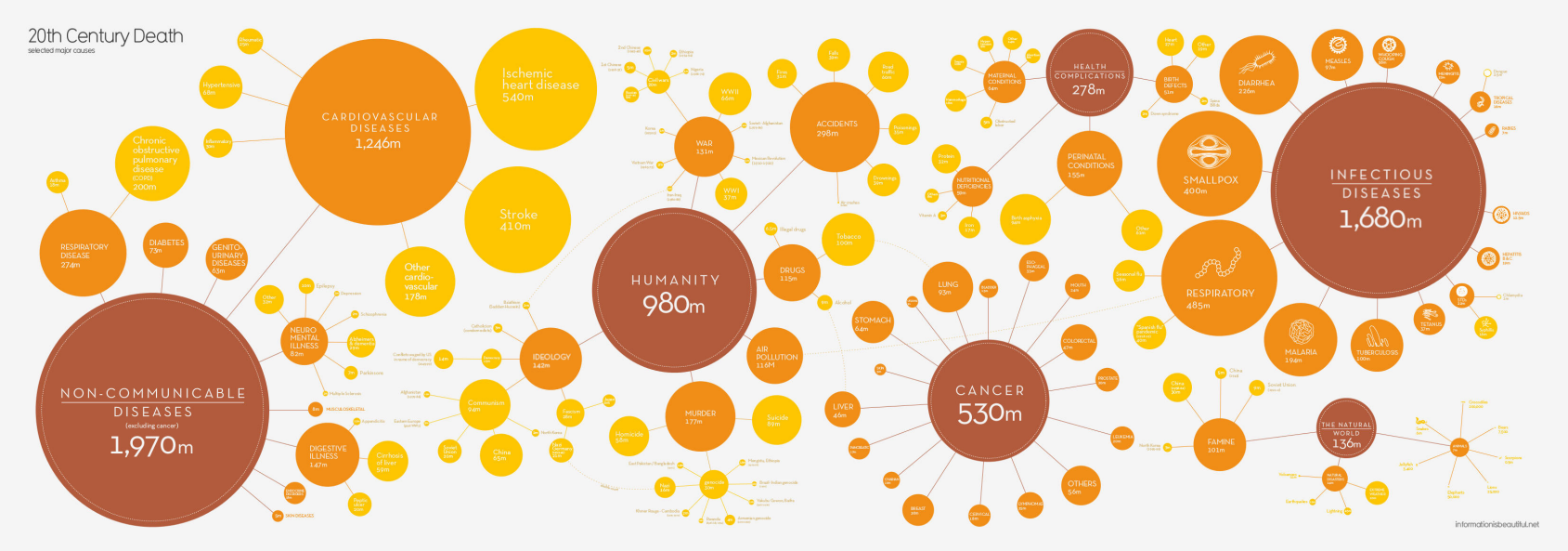
☞ http://technomarketer.typepad.com/technomarketer/2009/04/firstlook-augmented-reality.html

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

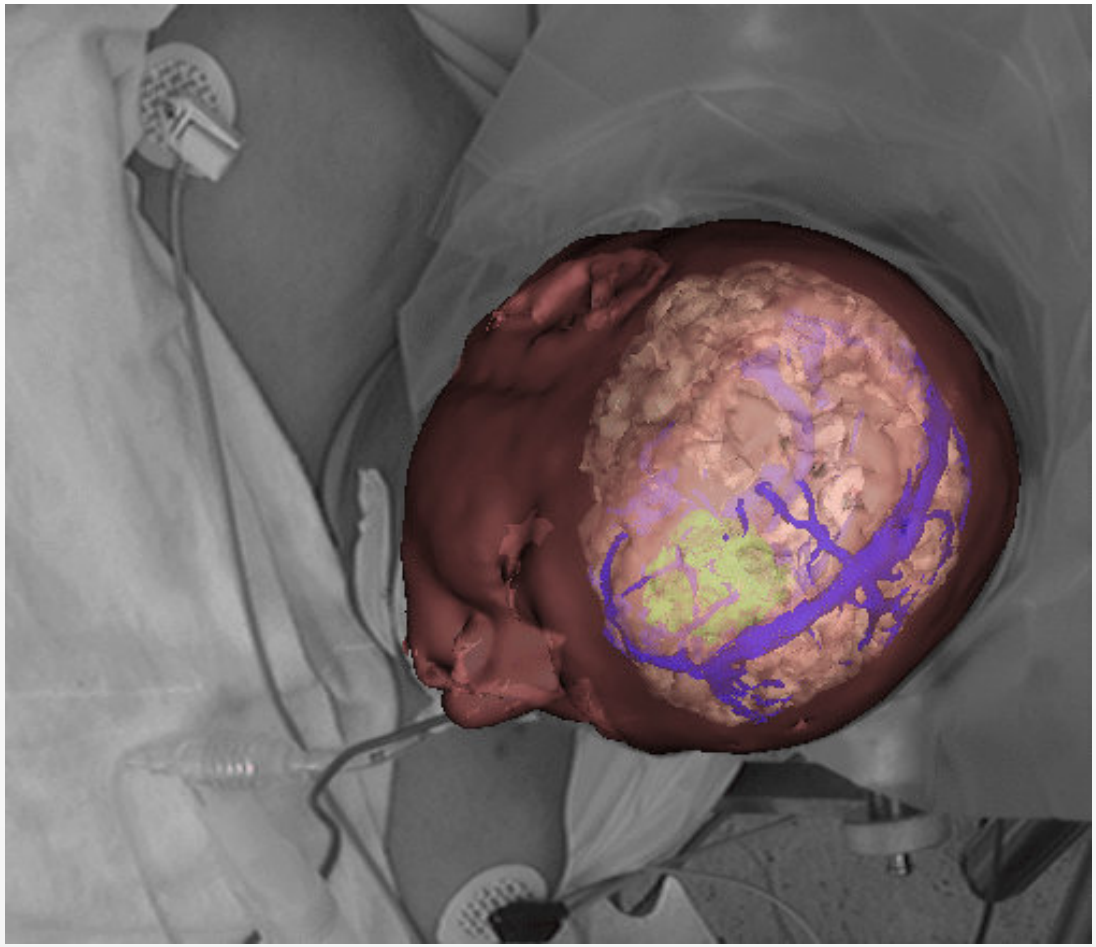
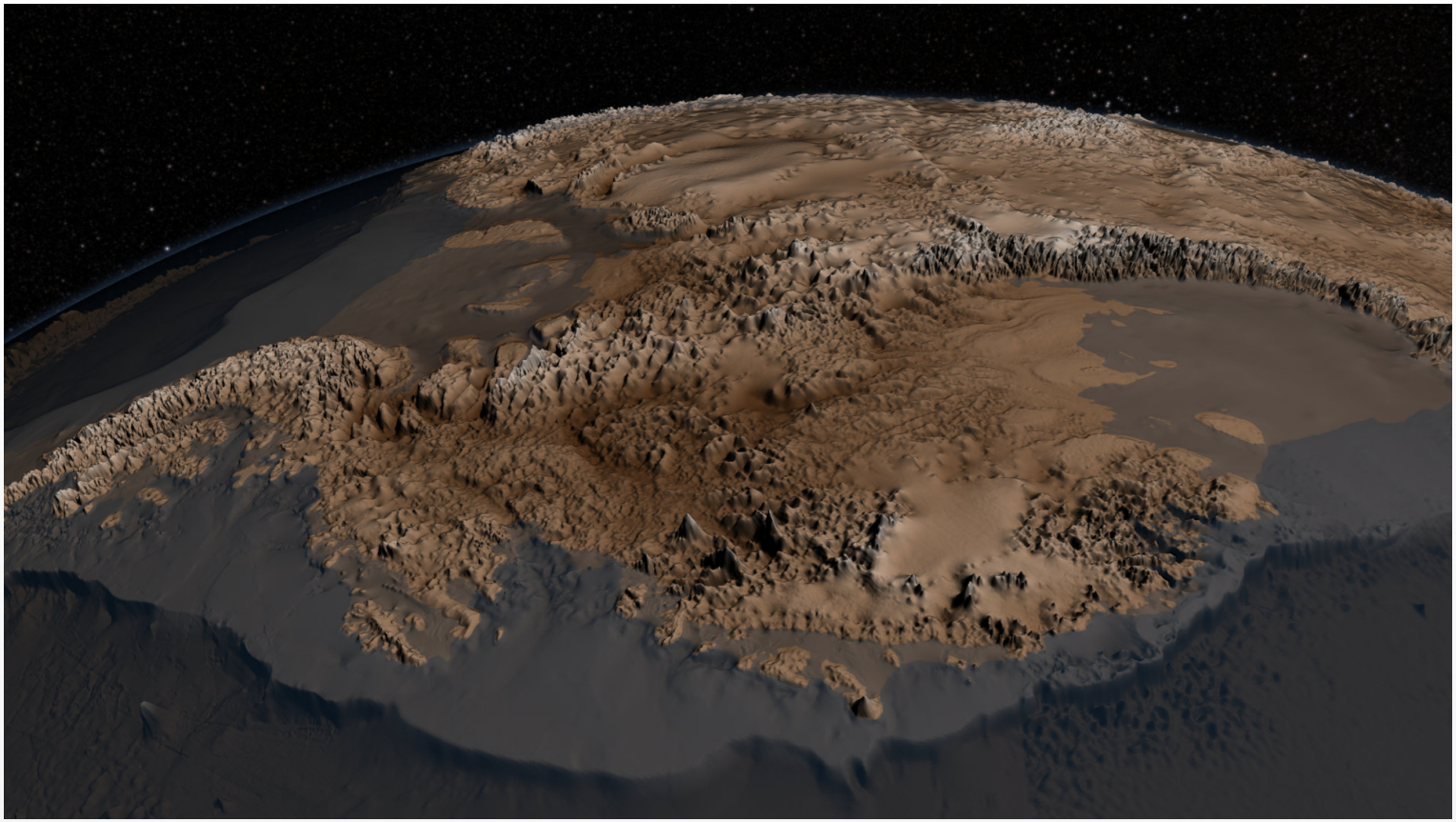


Imagen creada por Christopher Brown, Universidad de Rochester:

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

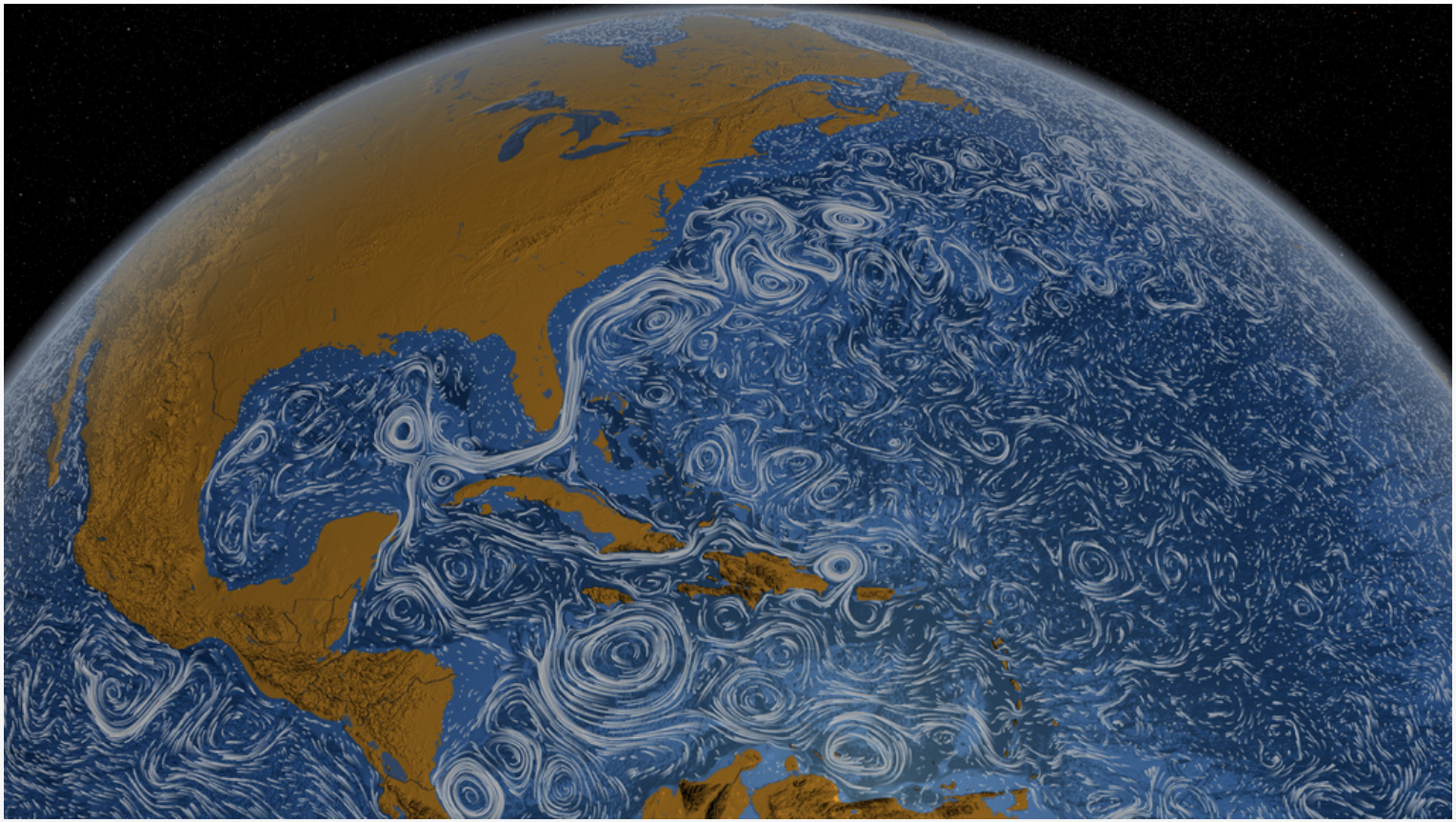
Visualización científica (geología)



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

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

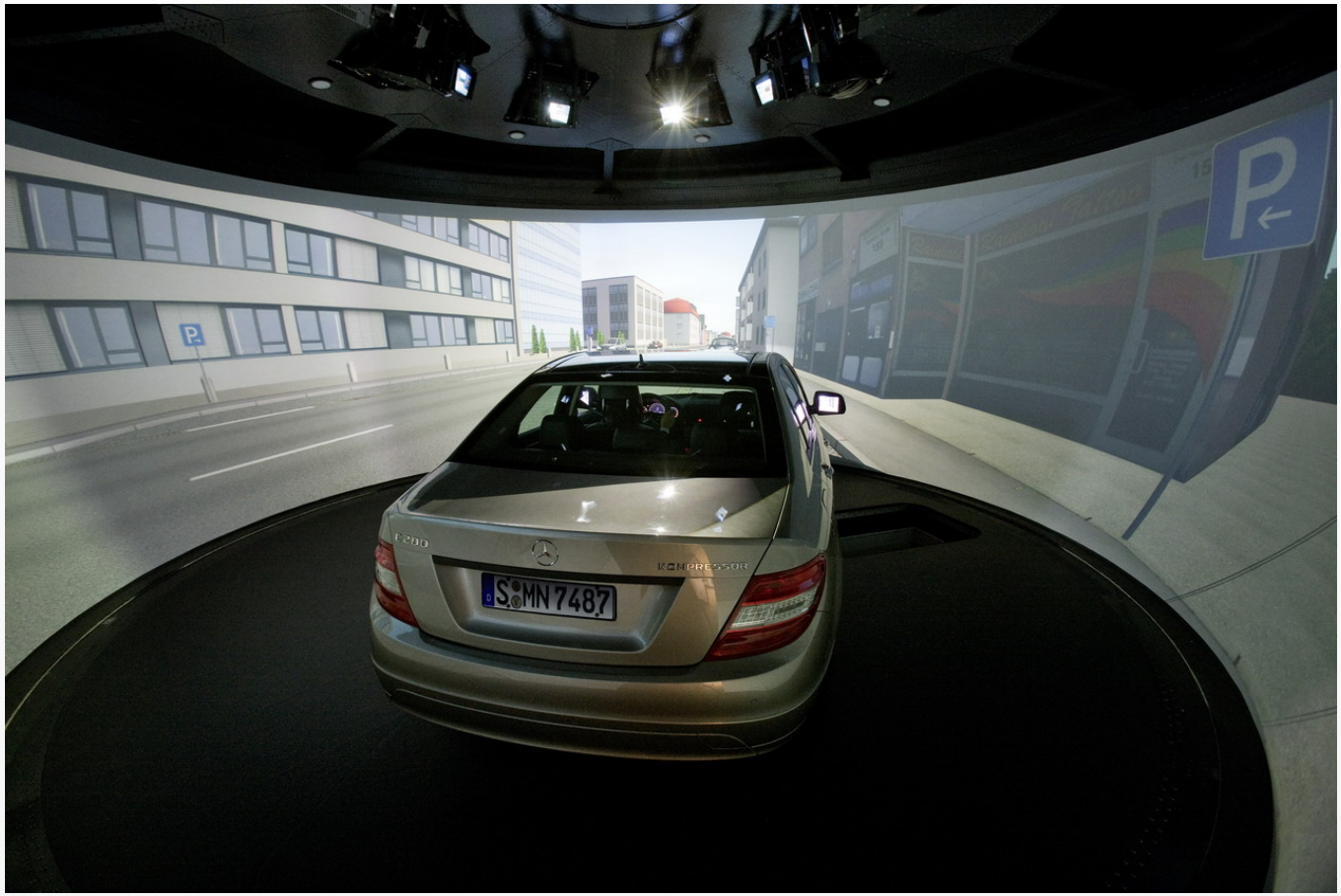
Visualización científica (climatología)



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

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

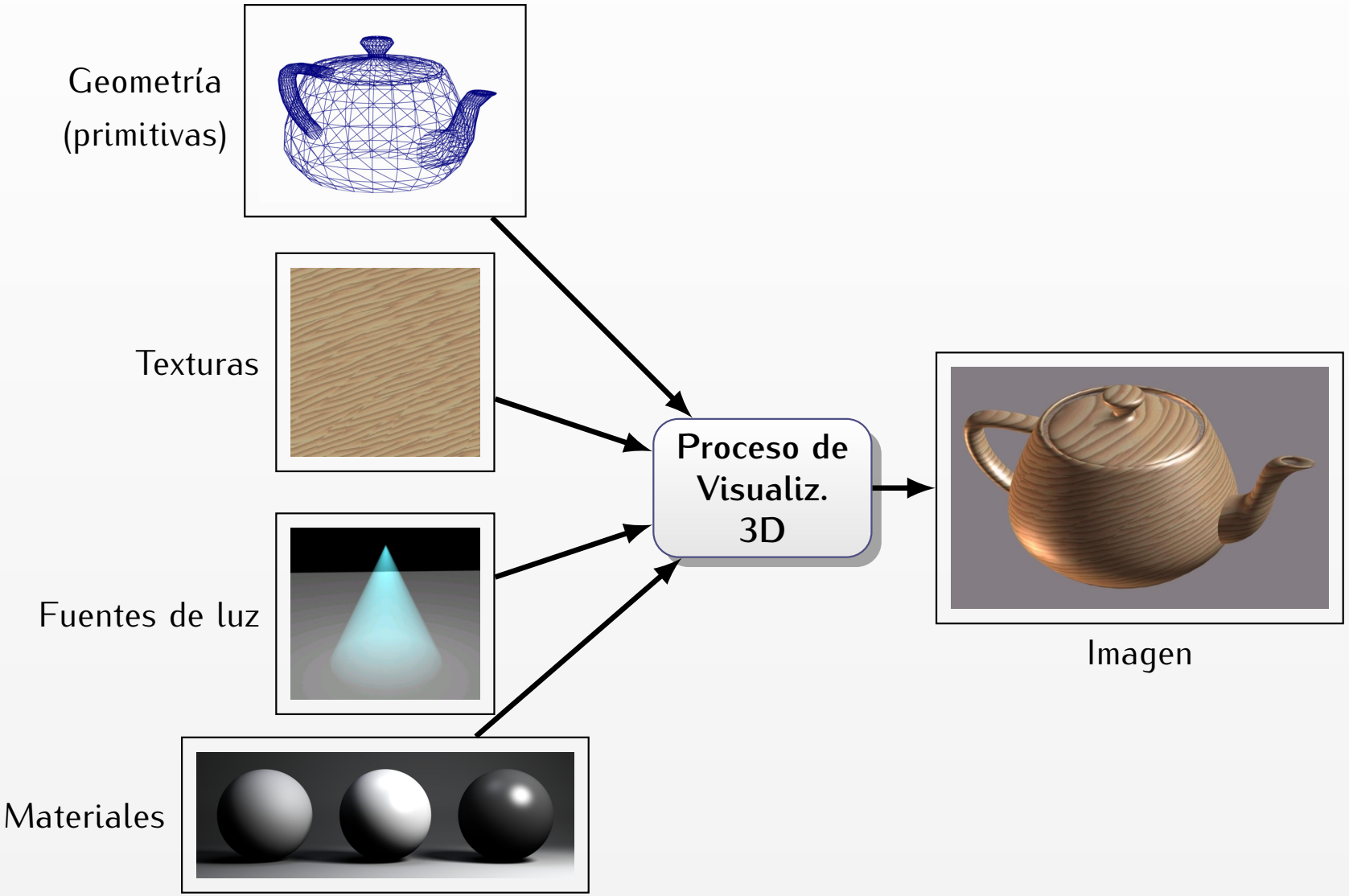
Subsección 2.1
Introducción

Visualización 3D (1)

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

- **Modelo de escena:** estructura de datos en memoria que representa lo que se quiere ver, esta formado por varias partes:
  - **Modelo geométrico**: conjunto de **primitivas** (típicamente polígonos planos), que definen la forma de los objetos a visualizar
  - **Modelo de aspecto**: conjunto de parámetros que definen el aspecto de los objetos: tipo de material, color, texturas, fuentes de luz
- **Parámetros de visualización**: es un conjunto amplio de valores, algunos elementos esenciales son:
  - **Cámara virtual**: posición, orientación y ángulo de visión del observador ficticio que vería la escena como aparece en la imagen
  - ***Viewport***: Resolución de la imagen, y, si procede, posición de la misma en la ventana.

Visualización 3D (2)



Subsección 2.2
Rasterización y ray-tracing.

Visualización basada en rasterización

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

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

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

Otros métodos de visualización

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.

Comparativa

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

- **Rasterización**
  - Las **unidades de procesamiento gráfico** (GPUs) son un hardware diseñado originalmente para ejecutar la rasterización de forma eficiente en tiempo.
  - El método de rasterización es preferible para **aplicaciones interactivas**, y es el que se usa actualmente para **videojuegos**, normalmente asistido por GPUs.
- **Ray-tracing**
  - El método de Ray-tracing y sus variantes suele ser más lento, pero consigue resultados más realistas cuando se pretende reproducir ciertos efectos visuales.
  - Las variantes y extensiones de Ray-tracing son preferibles para síntesis de imágenes *off-line* (no interactivas), y es el que se usa actualmente para **producción de animaciones y efectos especiales** en películas o anuncios.

Subsección 2.3
El cauce gráfico en rasterización.

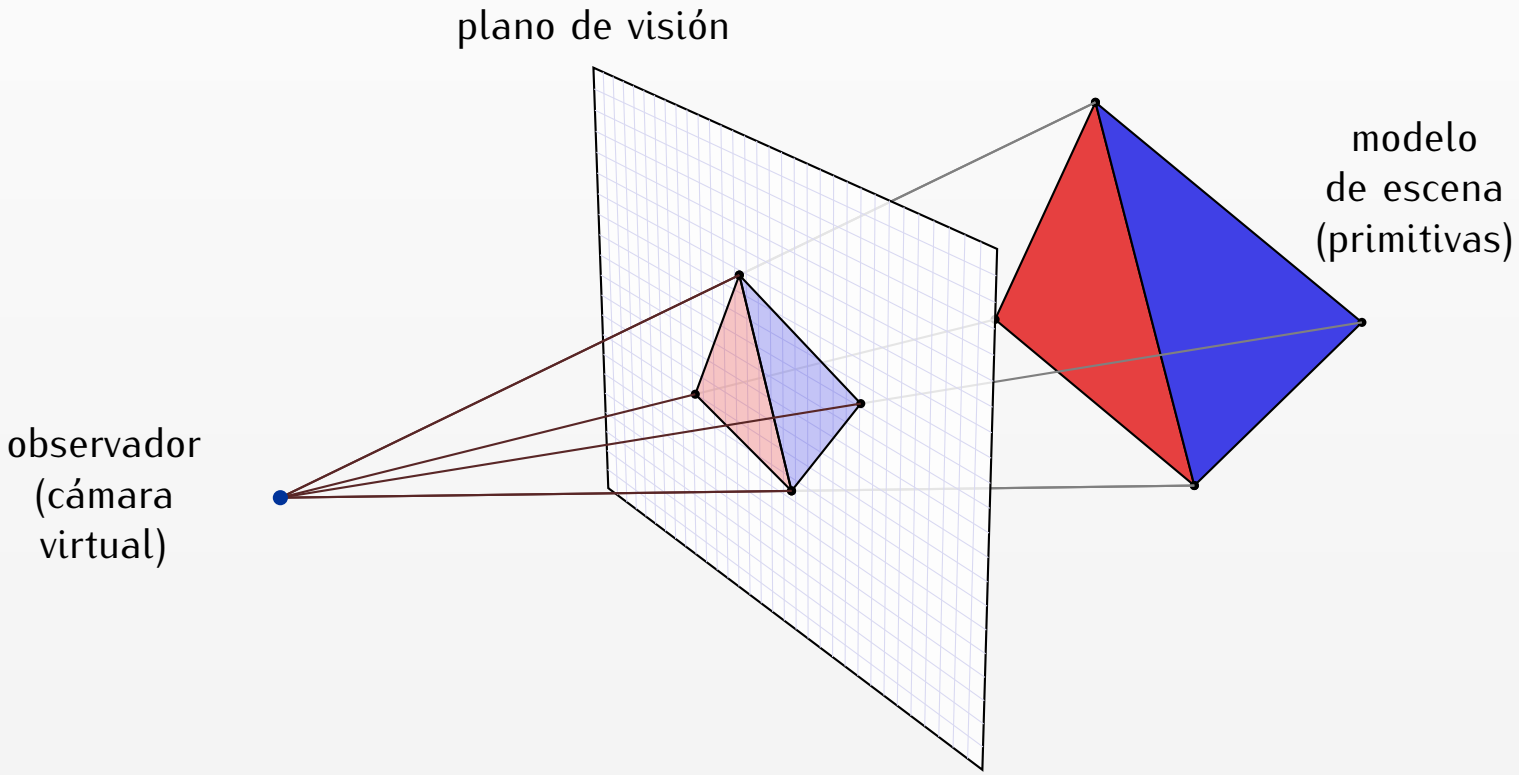
El cauce gráfico en rasterización

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

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

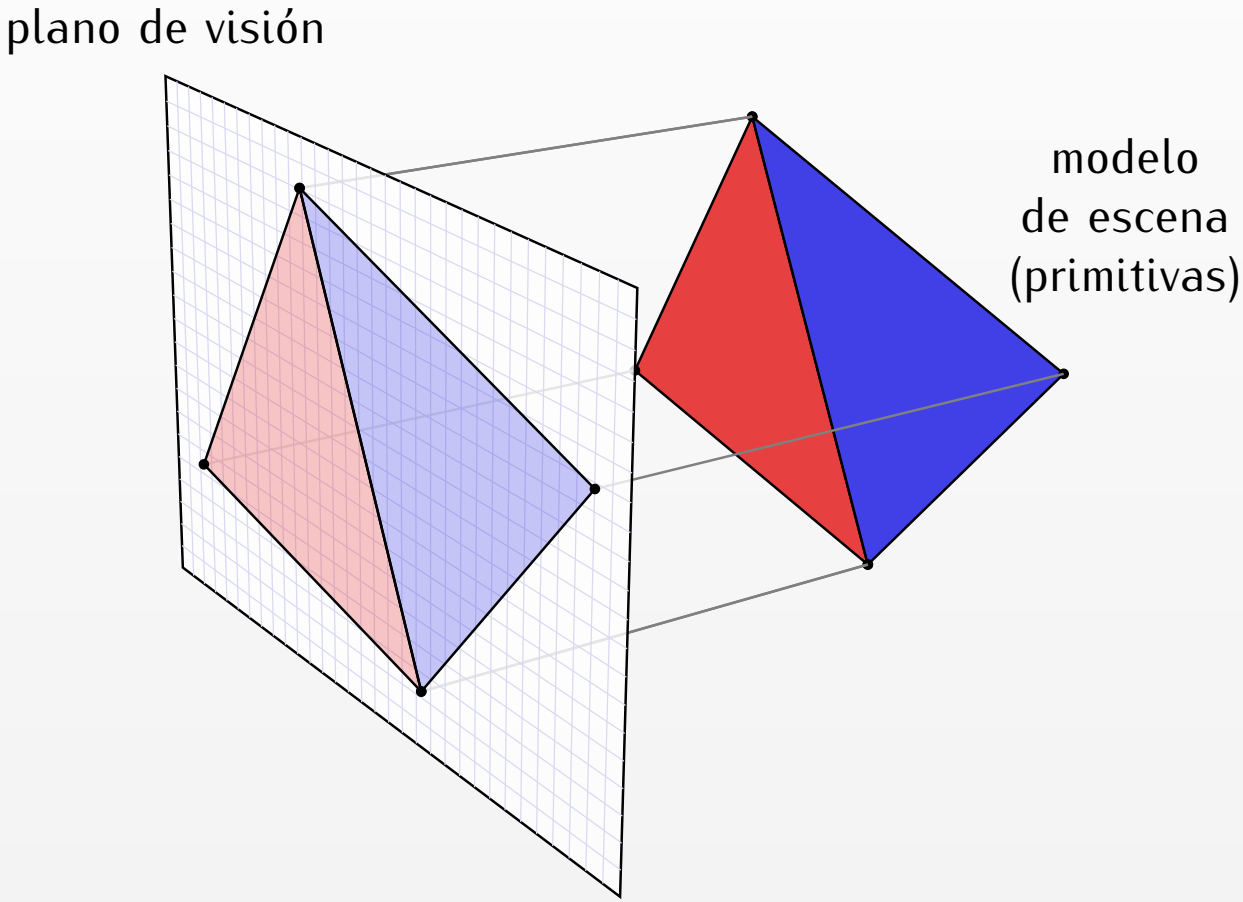
Transformación y proyección

Cada primitiva se 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):



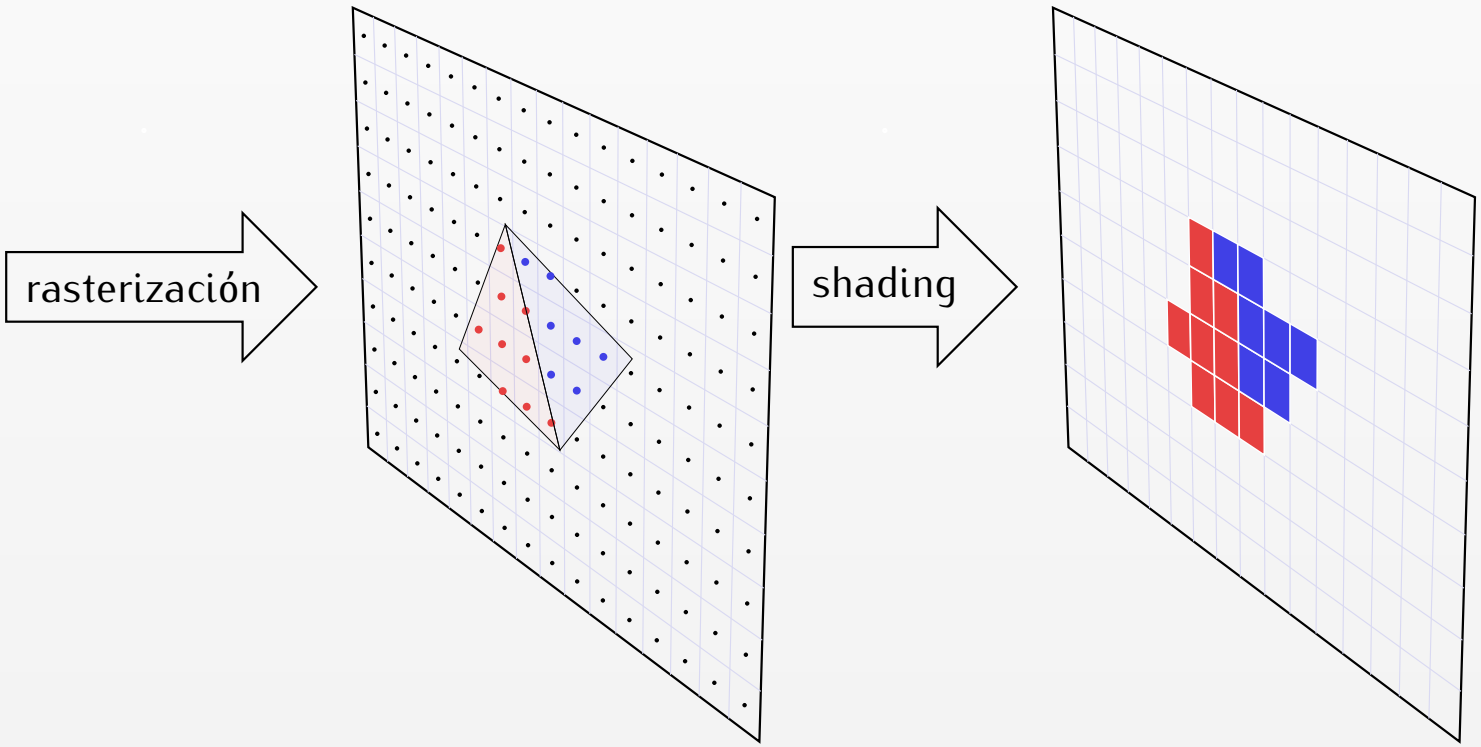
Proyección paralela

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.



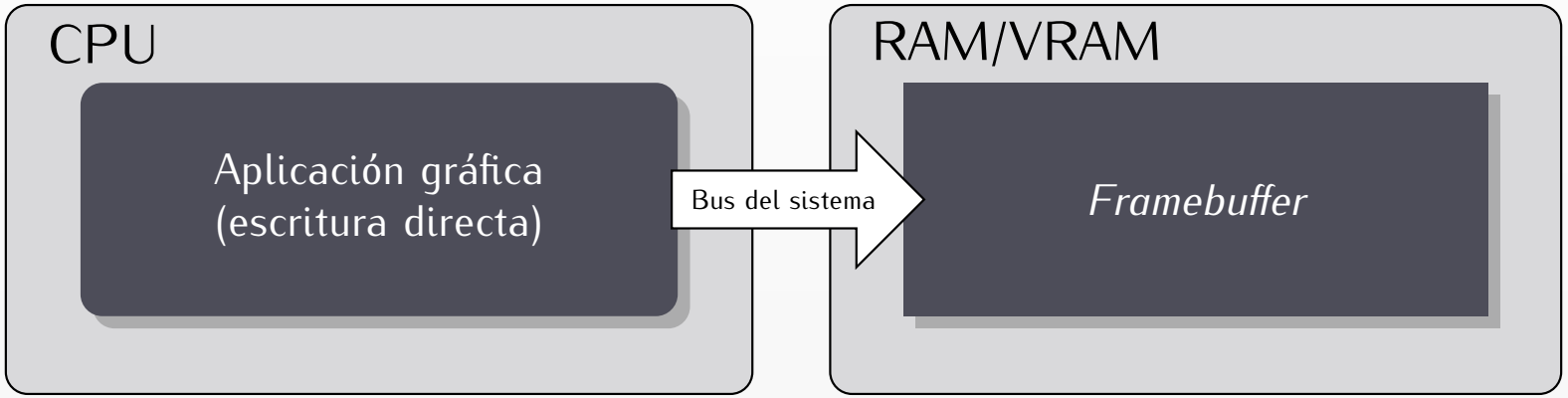
Sección 3
La librería OpenGL. Un programa sencillo.

3.1. Aplicaciones y bibliotecas gráficas con GPUs.
3.2. La librería OpenGL.
3.3. Programación y eventos en OpenGL
3.4. Visualización de primitivas.

Subsección 3.1
Aplicaciones y bibliotecas gráficas con GPUs.

Aplicaciones de escritura directa

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

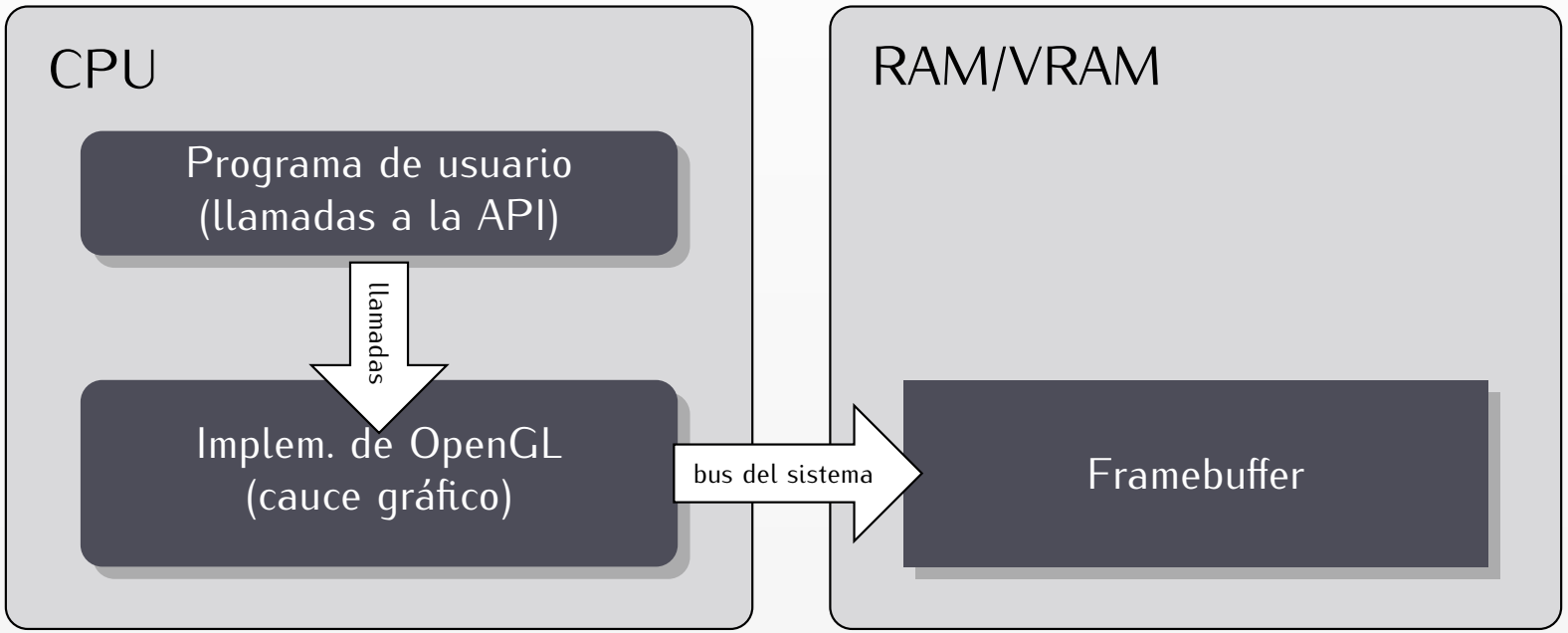


Desventajas

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

Uso de librerías

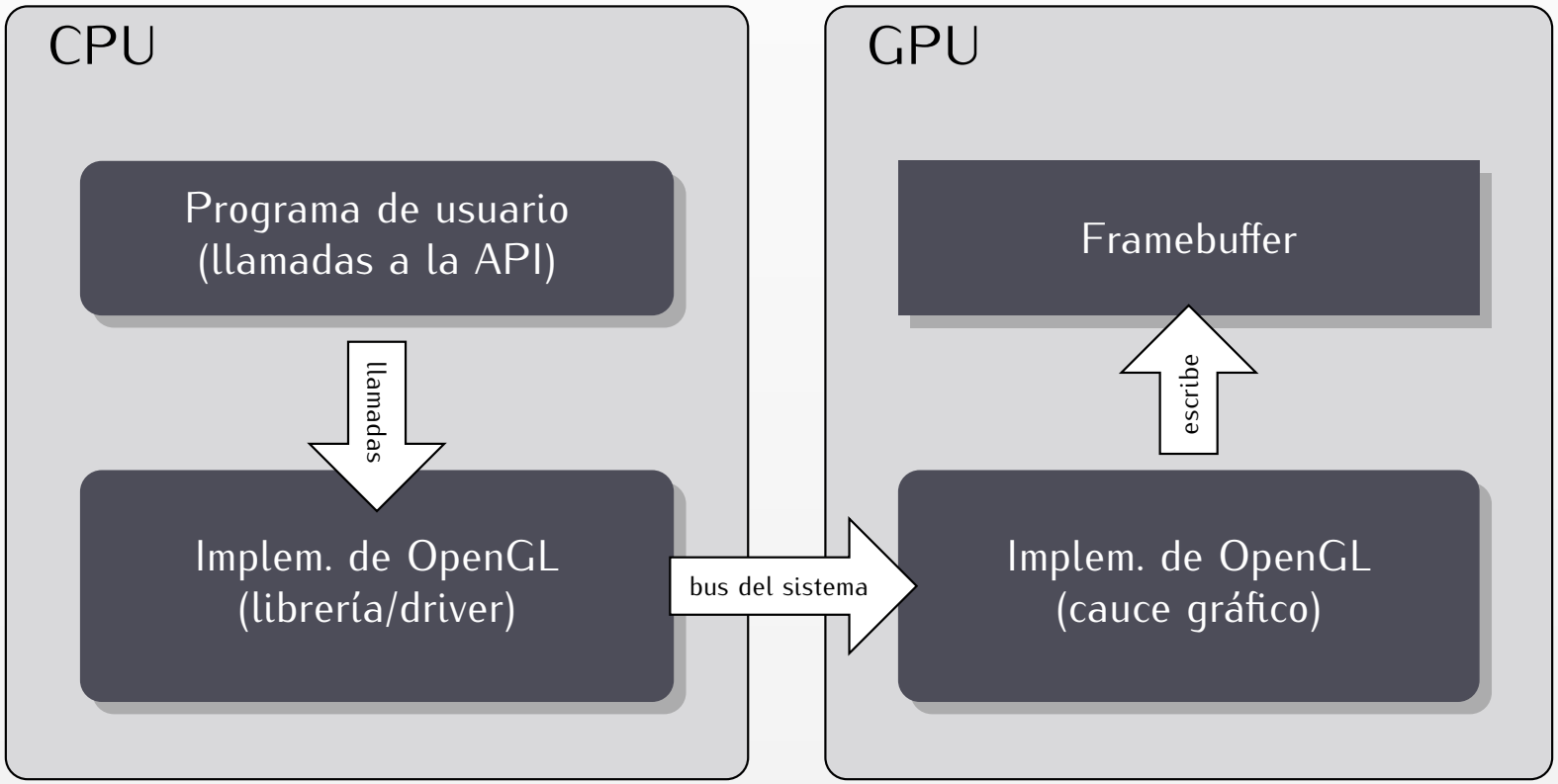
El uso de librerías gráficas (OpenGL) proporciona portabilidad y posibilidad de acceso simultáneo de varias aplicaciones



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

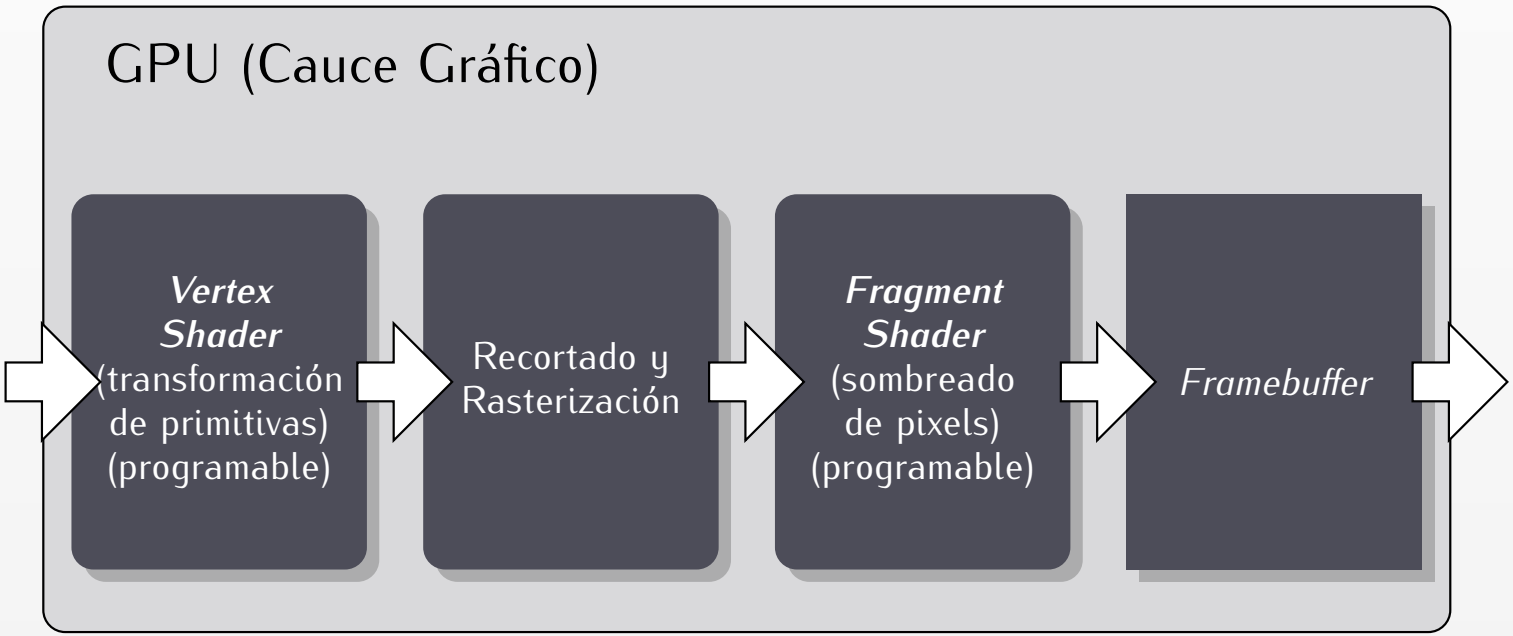
Uso de librerías y GPUs

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



GPUs de cauce programable

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



Subsección 3.2
La librería OpenGL.

La librería OpenGL

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

Características de OpenGL

- Existen implementaciones de la API para las principales plataformas (Windows,MacOS,Linux,Android,iOS...) y lenguajes de programación (C/C++, Java, Python,...)
- OpenGL hace que las aplicaciones sean independientes del hardware.
- Para gestionar ventanas y eventos de entrada se deben usar librerías auxiliares, que pueden o no ser dependientes del entorno hardware/software.
- Utiliza las capacidades de aceleración de las tarjetas gráficas (GPUs).
- Realiza por software las funciones no disponibles en la tarjeta.
- 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.
- 2012: Se publica la versión 3.0 de OpenGL ES.
- 2013: Se publica la versión 4.4 de OpenGL.

Bibliotecas complementarias: GLU y GLUT

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).

GLUT (*OpenGL Utility Toolkit*) es una biblioteca auxiliar que permite:

- Gestión de ventanas.
- Gestión de eventos de entrada.
- Dibujo de primitivas complejas (esferas, conos, toroides, teteras).
- Gestión de menús.

Los nombres de las funciones de GLU comienzan con `glu` y las de GLUT con `glut`.

Subsección 3.3
Programación y eventos en OpenGL

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.
- **Ratón**: pulsación o levantado de botones del raton, o su movimiento.
- **Cambio de tamaño**: cambio de tamaño de alguna ventana de la apl.
- **Redibujado**: necesidad de redibujar la ventana.
- **Reloj**: ocurrencia de un instante dentro una serie periódica (cronometrada) de ellos.
- **Desocupado**: ocurre cuando la aplicación no tiene ningún otro evento pendiente de procesar.

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

- Tras invocar a una de estas funciones, se dice que el correspondiente evento ya ha sido **procesado o gestionado**.
- En el programa se puede establecer que tipos de eventos se deben procesar y que funciones deben hacerlo.
- Para cada tipo de evento, la función que lo gestione debe aceptar unos determinados parámetros.
- Los parámetros permiten indicar a la función alguna información adicional relativa al evento, a modo de ejemplo:
  - tecla que ha sido pulsada o levantada
  - nueva posición del ratón tras moverse
  - botón del ratón que ha sido pulsado o levantado
  - nuevo tamaño de la ventana

Estructura de un programa

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

- Variables, estructuras de datos y definiciones globales.
- Código de las funciones gestoras de eventos.
- Código de inicialización:
  - Creación y configuración de la ventana (o ventanas) donde se visualizan las primitivas,
  - Establecimiento de las funciones del programa que actuarán como gestoras de eventos (como mínimo, la de redibujado)
  - Configuración inicial de OpenGL, si es necesario.
- Llamada a la función que realiza el **bucle de gestión de eventos**.

Bucle de gestión de eventos

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

- GLUT mantiene una **cola de eventos**: es una lista (FIFO) con información de cada evento que ya ha ocurrido pero que no ha sido gestionado aún por la aplicación.
- En cada iteración del bucle se dan estos dos pasos:
  1. Si la cola de eventos está vacía, se inserta un evento de desocupado.
  2. Se extrae el siguiente evento de la cola: si hay designada una función gestora para ese tipo de evento, se ejecuta dicha función.
- El bucle termina típicamente cuando alguna función gestora cierra la aplicación.

Las aplicaciones ejecutan el bucle con una llamada a la función **`glutMainLoop`**.

Estructura del programa.

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

```
void FGE_Redibujado( )
    { ....}
void FGE_CambioTamano( int nuevoAncho, int nuevoAlto )
    { .... }
// otros callbacks:
// ......
void Inicializa_GLUT( int argc, char * argv[] )
    { .... }
void Inicializa_OpenGL( )
    { ..... }
int main( int argc, char *argv[] )
{
    Inicializa_GLUT(argc,argv) ;  // crea un 'rendering context'
    Inicializa_OpenGL() ;         // usa el 'rendering context'
    glutMainLoop() ;
}
```

Código de inicialización de GLUT

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

```
void Inicializa_GLUT( int argc, char * argv[] )
{
   // inicializa glut:
   glutInit( &argc, argv );
   // establece posicion inicial de la ventana:
   glutInitWindowPosition( 100, 100 );
   // establece tamaño inicial de la ventana:
   glutInitWindowSize( 512, 512 );
   // establece atributos o tipo de ventana:
   // (pixels en formato RGBA, double buffering activado, Z-buffer activado)
   glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
   // crea y visualiza una ventana:
   glutCreateWindow("IG. Ejemplo 1.");
   // establece función gestora del evento de redibujado
   glutDisplayFunc( FGE_Redibujado );
   // establece función gestora del evento de cambio de tamaño de la ventana
   glutReshapeFunc( FGE_CambioTamano );
}
```

Subsección 3.4
Visualización de primitivas.

Envio de primitivas en modo inmediato

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

- Las primitivas se describen usando una **secuencia de coordenadas de vértices**.
- 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.

Tipos de primitivas

Una misma secuencia de coordenadas de vértices (n en total) puede interpretarse de varias formas:

- un conjunto de **puntos** aislados (n arbitrario)
- un conjunto de **segmentos** de recta entre pares de puntos (n par)
- una única **polilínea abierta** (n arbitrario)
- una única **polilínea cerrada** (n arbitrario)
- un único **polígono relleno** de color (n arbitrario)
- una serie de **triángulos** rellenos (n multiplo de 3)
- una serie de **cuadriláteros** rellenos (n multiplo de 4)

estas primitivas simples permiten construir otras más complejas, como iremos viendo a lo largo del temario.

Formas de envio de la secuencia

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

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

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

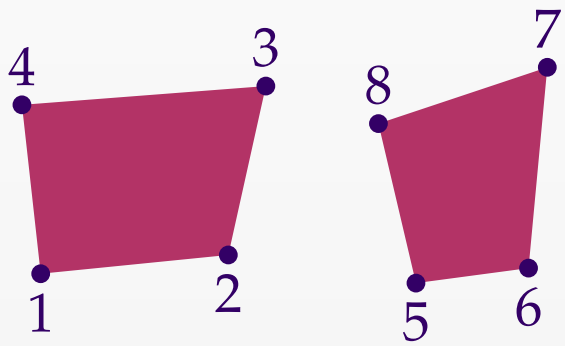
La primera forma de visualización es usando las funciones **`glBegin`**, seguida de **`glVertex`** (una por cada vértice) y **`glEnd`**:

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

Visualización de cuadriláteros

La constante de OpenGL **GL_QUADS** permite enviar uno o más cuadriláteros, indicando una secuencia de vértices (para cada uno se indican tres expresiones reales):

```
glBegin( GL_QUADS ) ;
    // primer cuadrilátero
    glVertex3f( x1 , y1 , z1 ) ;
    glVertex3f( x2 , y2 , z2 ) ;
    glVertex3f( x3 , y3 , z3 ) ;
    glVertex3f( x4 , y4 , z4 ) ;
    // segundo cuadrilátero
    glVertex3f( x5 , y5 , z5 ) ;
    glVertex3f( x6 , y6 , z6 ) ;
    glVertex3f( x7 , y7 , z7 ) ;
    glVertex3f( x8 , y8 , z8 ) ;
    // ..... otros cuadr. .....
glEnd() ;
```

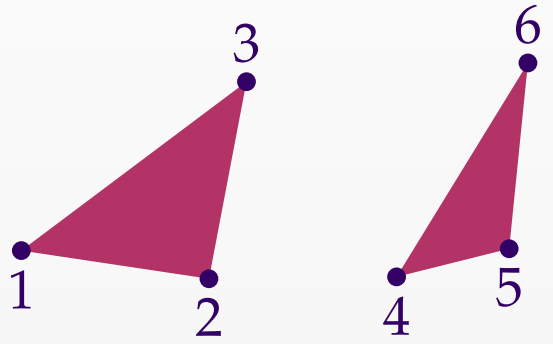


- Con **GL_QUADS**, el número de llamadas a **glVertex** entre **glBegin** y **glEnd** debe ser un múltiplo de 4 (nunca cero).

Visualización de triángulos

Se puede usar la constante de OpenGL **GL_TRIANGLES** para enviar uno o varios triángulos:

```
glBegin( GL_TRIANGLES ) ;
    // primer triángulo
    glVertex3f( x1 , y1 , z1 ) ;
    glVertex3f( x2 , y2 , z2 ) ;
    glVertex3f( x3 , y3 , z3 ) ;
    // segundo triángulo
    glVertex3f( x4 , y4 , z4 ) ;
    glVertex3f( x5 , y5 , z5 ) ;
    glVertex3f( x6 , y6 , z6 ) ;
    // ... otros triángulos ...
glEnd() ;
```

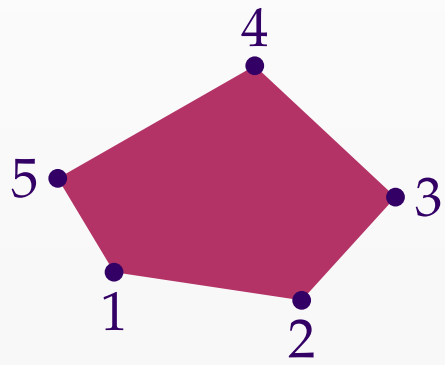


- Ahora, con **GL_TRIANGLES**, el número de llamadas a **glVertex** entre **glBegin** y **glEnd** debe ser múltiplo de 3.

Polígonos planos genéricos

Una secuencia de vértices (de tamaño arbitrario) puede describir un polígono con un número arbitrario de aristas, con **GL_POLYGON**

```
glBegin( GL_POLYGON ) ;
   // un polígono de n vértices
   glVertex3f( x1 , y1 , z1 );
   glVertex3f( x2 , y2 , z2 );
    .......
   glVertex3f( xn-1 , yn-1 , zn-1 );
   glVertex3f( xn , yn , zn );
glEnd() ;
```

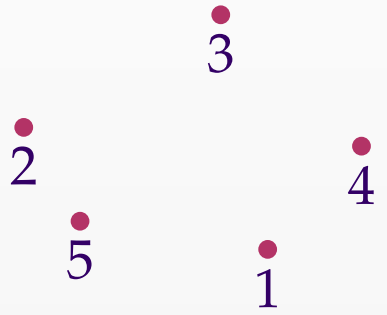


- Es necesario que las posiciones de todos los vértices de un polígono **estén en el mismo plano en 3D**, de otra forma se pueden obtener resultado incorrectos (esto es cierto para los cuadriláteros y los polígonos arbitrarios).

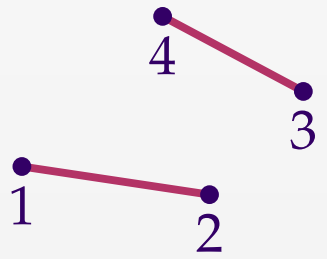
Puntos aislados

Usando **GL_POINTS** y **GL_LINES**, no se visualizan polígonos rellenos, sino únicamente puntos aislados o segmentos de líneas (un número arbitrario de ellos):

```
glBegin( GL_POINTS ) ;
   // puntos aislados
   glVertex3f( x1 , y1 , z1 );
   glVertex3f( x2 , y2 , z2 );
    .......
   glVertex3f( xn , yn , zn );
glEnd() ;
```



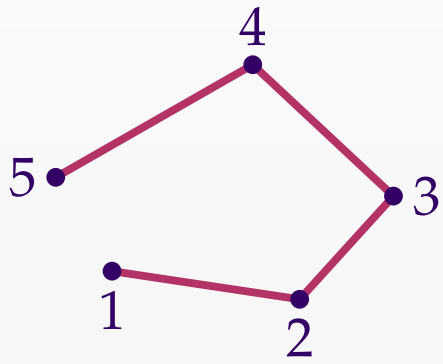

```
glBegin( GL_LINES ) ;
   // primer segmento:
   glVertex3f( x1 , y1 , z1 );
   glVertex3f( x2 , y2 , z2 );
   // segundo segmento:
   glVertex3f( x3 , y3 , z3 );
   glVertex3f( x4 , y4 , z4 );
glEnd() ;
```



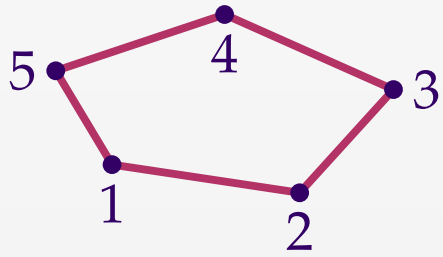
Segmentos de recta o polilíneas

Con **GL_LINE_STRIP** se pueden dibujar una seríe de $n-1$ segmentos de recta unidos (con n vértices arbitrarios). Con **GL_LINE_LOOP** el resultado es similar, pero en este caso el último vértice se une al primero con un segmento adicional (n segmentos).

```
glBegin( GL_LINE_STRIP ) ;
  glVertex3f( x1 , y1 , z1 );
   .......
  glVertex3f( xn , yn , zn );
glEnd() ;
```




```
glBegin( GL_LINE_LOOP ) ;
  glVertex3f( x1 , y1 , z1 );
   .......
  glVertex3f( xn , yn , zn );
glEnd() ;
```



Otras formas de enviar cada vértice

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

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

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

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

Array de coordenadas de vértices.

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

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

	x_1	y_1	z_1	x_2	y_2	z_2	x_3	$\cdots$	z_{n-1}	x_n	y_n	z_n

	x_1	y_1	x_2	y_2	x_3	$\cdots$	z_{n-1}	y_n	z_n

- La dirección de memoria y la primera coordenada (x_1) es un valor (un puntero) conocido de tipo **float** $*$ (o **double** $*$)
- Para especificar dicha dirección y la estructura del array se debe usar **glVertexPointer**.
- Las coordenadas se pueden almacenar en un vector de tipo **std::vector**<**GLfloat**>.

Uso de `glDrawArrays`

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

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

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

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

Arrays de vértices en 2D

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

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

el envio se haría de forma muy similar:

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

Envio de arrays de vértices

En lugar de usar **GL_POINTS** podríamos usar también:

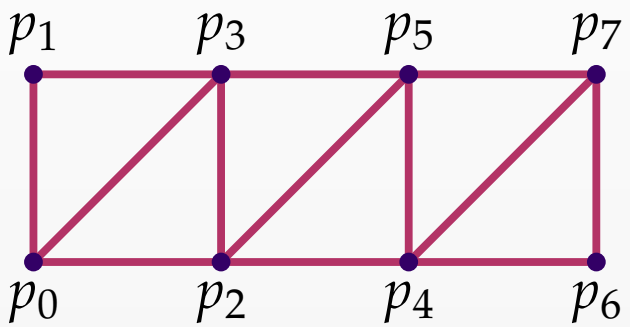
- **GL_LINE_STRIP** : una polilínea abierta (n arbitrario)
- **GL_LINE_LOOP** : una polilínea cerrada (n arbitrario)
- **GL_LINES** : $n/2$ segmentos no conectados (n par)
- **GL_TRIANGLES** : $n/3$ triángulos (n múltiplo de 3)
- **GL_QUADS** : $n/4$ cuadriláteros (n múltiplo de 4, no disponible a partir de OpenGL 3.3)
- **GL_POLYGON** : un polígono relleno (n arbitrario) (no disponible a partir de OpenGL 3.3)

En lugar de usar un array de simple precisión, podríamos enviar uno de doble:

- El array sería de tipo **GLdouble** en lugar de **GLfloat**
- En **glVertexPointer** usaríamos la constante **GL_DOUBLE** en lugar de **GL_FLOAT**

Primitivas con vértices replicados

Muchas veces necesitamos usar un vértice para varias primitivas:



Si usamos **GL_TRIANGLES**, la secuencia es:

$$p_0, p_3, p_1, \quad p_0, p_2, p_3$$
$$p_2, p_5, p_3, \quad p_2, p_4, p_5$$
$$p_4, p_7, p_5, \quad p_4, p_6, p_7$$

Supone **emplear más memoria y/o tiempo para visualizar del necesario**. En este ejemplo necesitamos una secuencia de 18 coordenadas de vértices, de las cuales solo hay 8 distintas.

Uso de `glDrawElements`

Para evitar tener que repetir coordenadas, se puede usar **`glDrawElements`** en lugar de **`glDrawArrays`**:

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

Arrays de índices y vértices

Para el ejemplo anterior, se requieren estas tablas:

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	$\cdots$	x_7	y_7	z_7

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

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

Uso de `glDrawElements`

Los arrays se pueden declarar como sigue:

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

GLfloat vertices[num_verts*3] =        // array con las coords. de verts.
   {  x0, y0, z0, x1, y1, z1, ···
      xn−1, yn−1, zn−1
   } ;
unsigned indices[num_indices] =        // array con los índices
   {  i0, i1, i2, i3, ···,
      im−2, im−1
   } ;
```

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

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

Modo de visualización de polígonos

Cuando se visualizan primitivas de tipo **GL_TRIANGLES** o **GL_POLYGON** (u otros modos que producen polígonos, como **GL_TRIANGLE_FAN**), OpenGL permite visualizar dichos polígonos de varias formas, usando la llamada:

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

donde *modo* es un valor **GLenum** que puede valer alguna de estas tres constantes:

- **GL_POINT** se visualizan únicamente los vértices como puntos.
- **GL_LINE** se visualizan únicamente las aristas como segmentos.
- **GL_FILL** se visualizan el triángulo relleno del color actual.

el nuevo modo se aplica hasta que se cambie (forma parte del estado de OpenGL). El valor inicial es **GL_FILL**.

Asignación de colores a vértices.

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

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

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

Envío de colores con begin/end

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

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

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

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

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

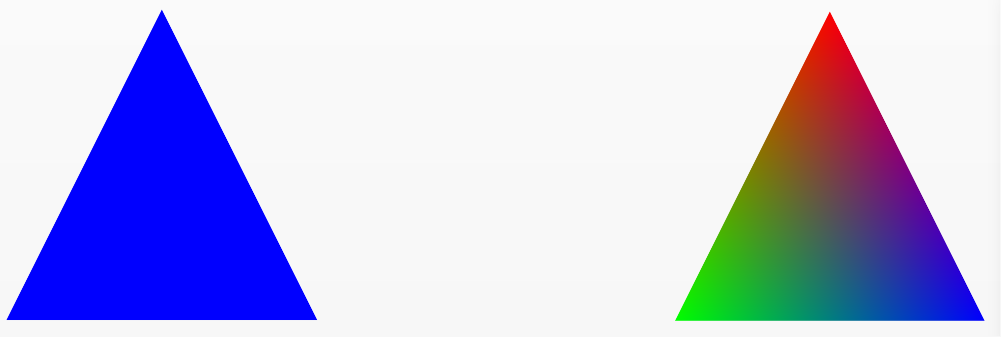
Envío de colores en un array de colores

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

```
// declaramos los vectores (usando vectores STL)
const std::vector<float>
   vertices = { 0.0,0.9,0.0,  -0.9,-0.9,0.0,  0.9,-0.9,0.0 } ,
   colores  = { 1.0,0.0,0.0,   0.0, 1.0,0.0,  0.0, 0.0,1.0 } ;

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

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

// dibujar
glDrawArrays( GL_TRIANGLES, 0, vertices.size()/3 ) ;

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

Modo de sombreado

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



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

Eliminación de partes ocultas (EPO) con Z-buffer

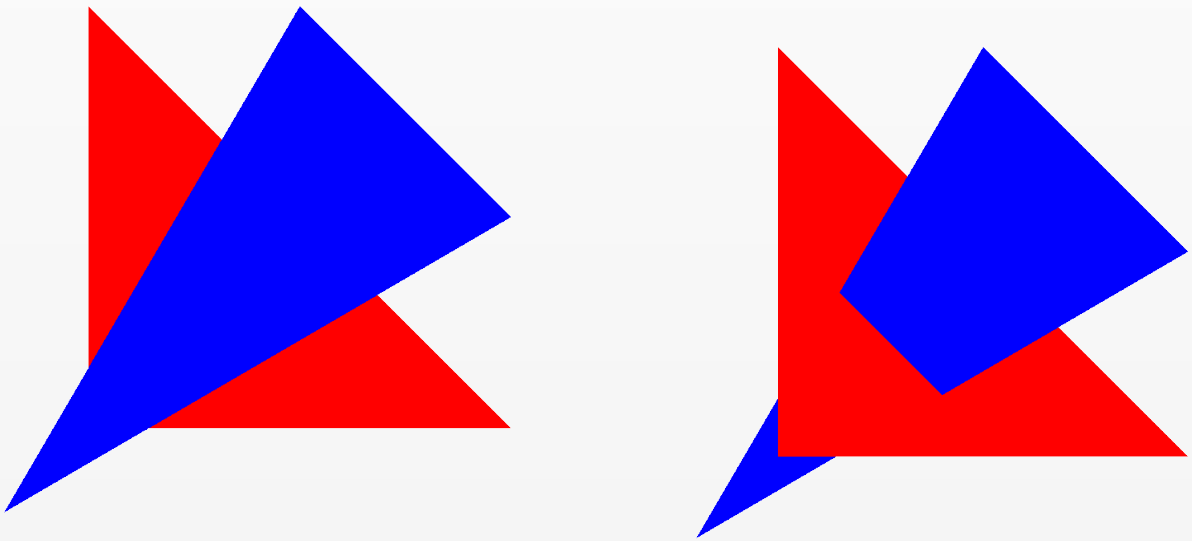
OpenGL usa las coordenadas Z de los vértices para calcular (por interpolación) la profundidad en Z en cada pixel de cada primitiva visualizada (Z es la dirección perpendicular a la pantalla).

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

- Esto permite dibujar primitivas 3D con cálculo correcto de las posibles ocultaciones entre ellas.
- Inicialmente (por defecto) en un pixel, una primitiva A con una Z menor estará por delante de otra B con una Z mayor (A oculta a B).
- Esto puede activarse o desactivarse, con **`glEnable`** y **`glDisable`**, usando **`GL_DEPTH_TEST`** como argumento. Inicialmente, está desactivado.
- Cuando se desactiva, al dibujar una primitiva en un pixel siempre se sobreescribe lo que hubiese escrito antes en el pixel, sin considerar la Z.

Ejemplo de EPO con Z-buffer.

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



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

Primitivas de alto nivel.

Las bibliotecas construidas sobre OpenGL permiten primitivas de más alto nivel. A modo de ejemplo:

- Glut incluye funciones para dibujar objetos (cubos, esferas, toros, conos, teteras). Estos objetos se descomponen internamente en triángulos.
- GLU incluye funciones para dibujar curvas y superficies (cuádricas y NURBS), y para descomponer (teselar) polígonos concavos.

Aquí hay dos ejemplos de dos funciones de `glut`

```
// dibuja un toro solido con radios 0.5 y 3.0
glutSolidTorus( 0.5, 3.0, 24, 32 );
// dibuja un cono solido con radio 1 y altura 1
glutSolidCone( 1.0, 1.0, 10, 20 );
```

La función de redibujado

La visualización de primitivas debe hacerse exclusivamente en la función gestora del evento de redibujado (*display function*) (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 **`glutSwapBuffers`** (solo si se usó **`GLUT_DOUBLE`** al inicializar `glut`). Esto **espera a que se rasterizen las primitivas** en el *framebuffer* y después **se visualiza en la ventana la imagen** ya creada en dicho *framebuffer*.

Ejemplo de función de redibujado

Un ejemplo sencillo para la función de redibujado es esta:

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

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

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

Detección de errores de OpenGL

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

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

y se define así:

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

Atributos de las primitivas

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

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

Inicialización de OpenGL

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

```
void Inicializa_OpenGL( )
{
    // comprobar si el flag de error de OpenGL ya estaba 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 );
    // establecer modo de visualización de prim.
    // (las tres posibilidades son: GL_POINT, GL_LINE, GL_FILL)
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    // habilitar eliminación de partes ocultas usando el Z-buffer
    glEnable( GL_DEPTH_TEST );
    // comprobar si ha habido algún error en esta función
    CError();
}
```

Definición del *viewport*

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

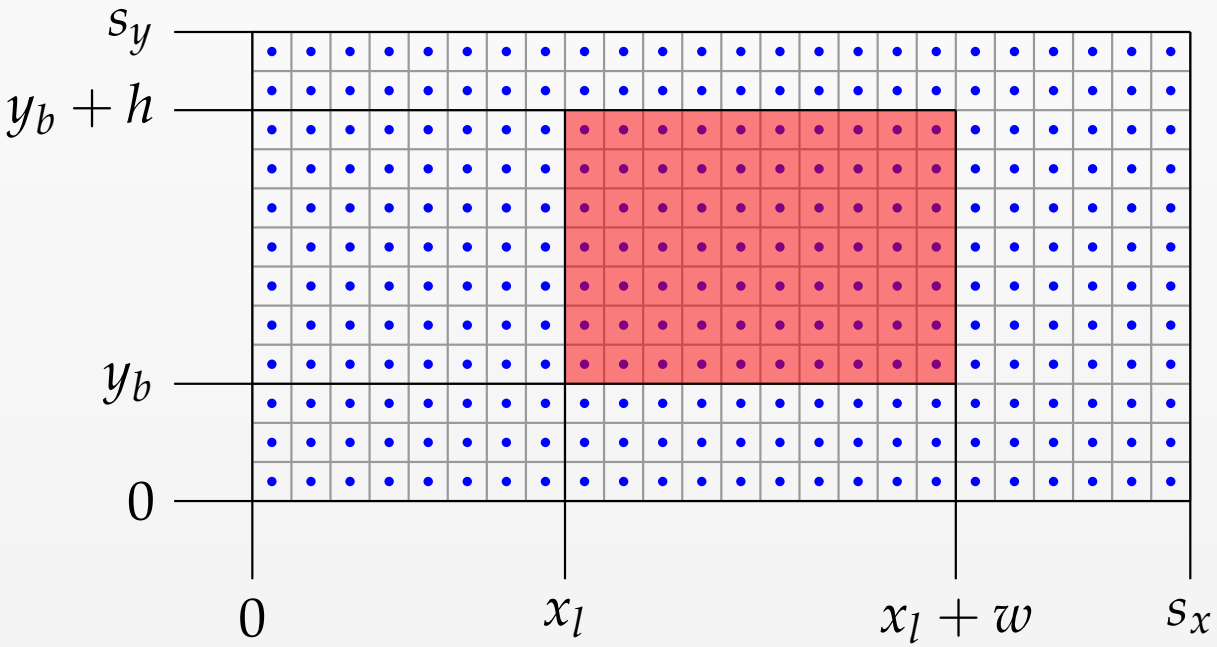
```
glViewport( izqui, abajo, ancho, alto ) ;
```

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

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

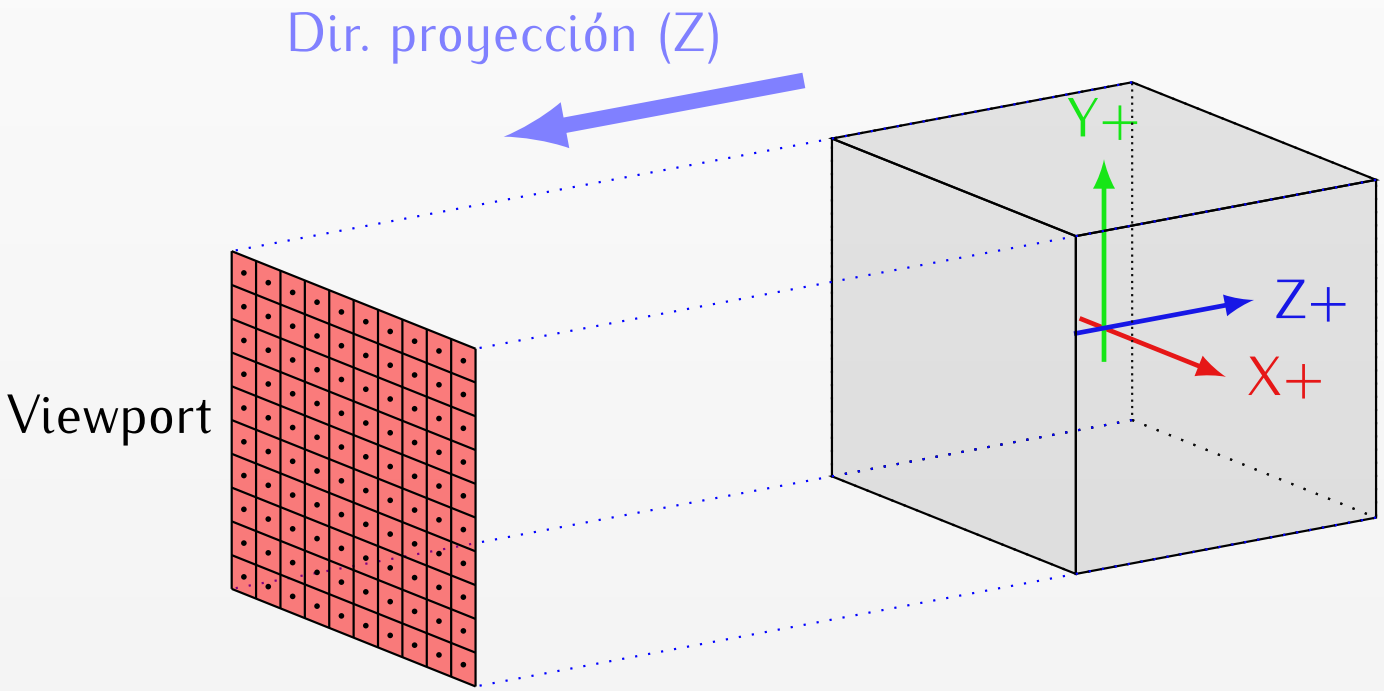
El viewport y la ventana como rejillas de pixels

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



Región visible y proyección sobre el viewport

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



La función gestora del cambio de tamaño

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

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

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

Documentación on-line sobre OpenGL

- 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 4.4): ☞ www.opengl.org/registry/#apispecs
  - Versiones anteriores: ☞ www.opengl.org/registry/#oldspecs
- Librería GLUT (especificaciones e implementación)
  - API v.3:☞ www.opengl.org/resources/libraries/glut/spec3/spec3.html
  - *Freeglut* (implementación de la API ver.3): ☞ freeglut.sourceforge.net
- Página de referencia de GLSL:
  - Todas las ver.:☞ www.opengl.org/sdk/docs/manglsl/

Sección 4
Programación básica del cauce gráfico

Subsección 4.1
Cauce programable

Transformación y sombreado

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

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

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

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

Vertex y Fragment Shaders

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

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

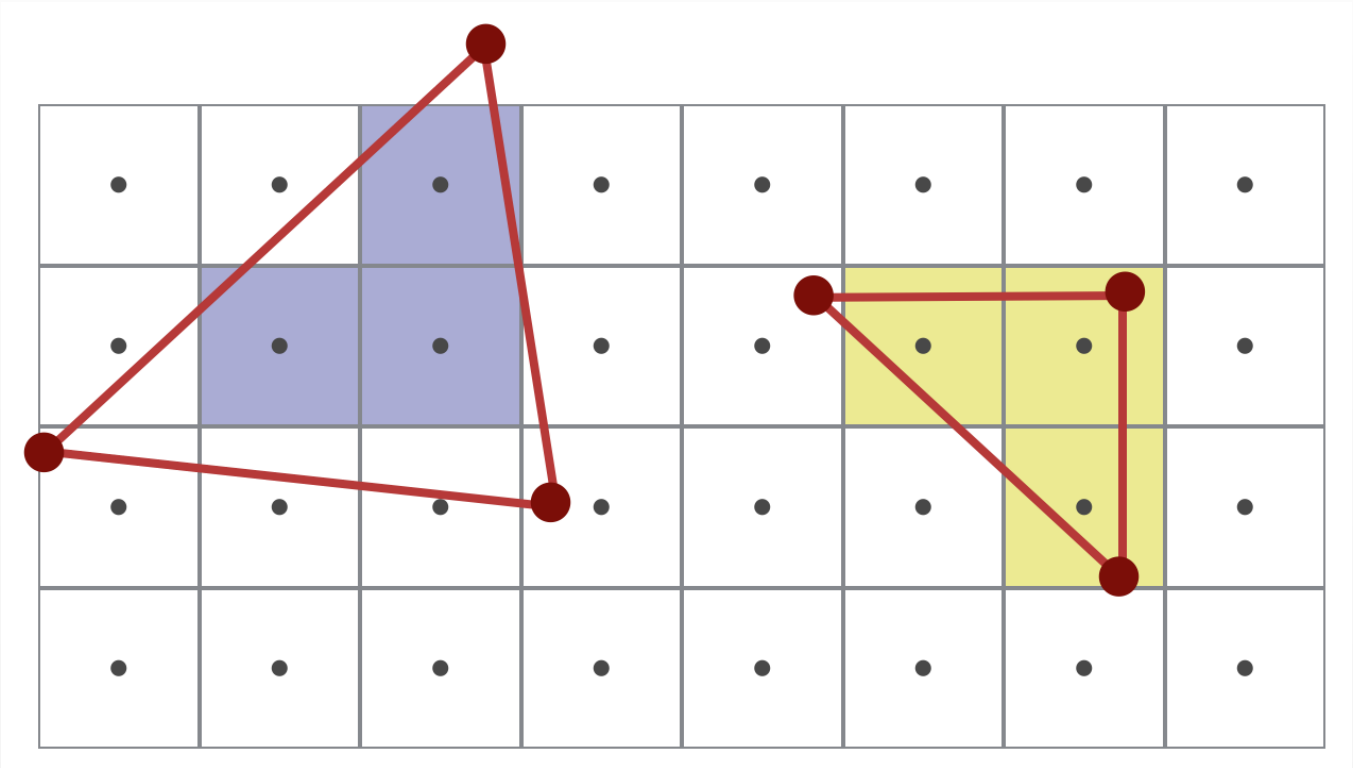
Programación de shaders

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

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

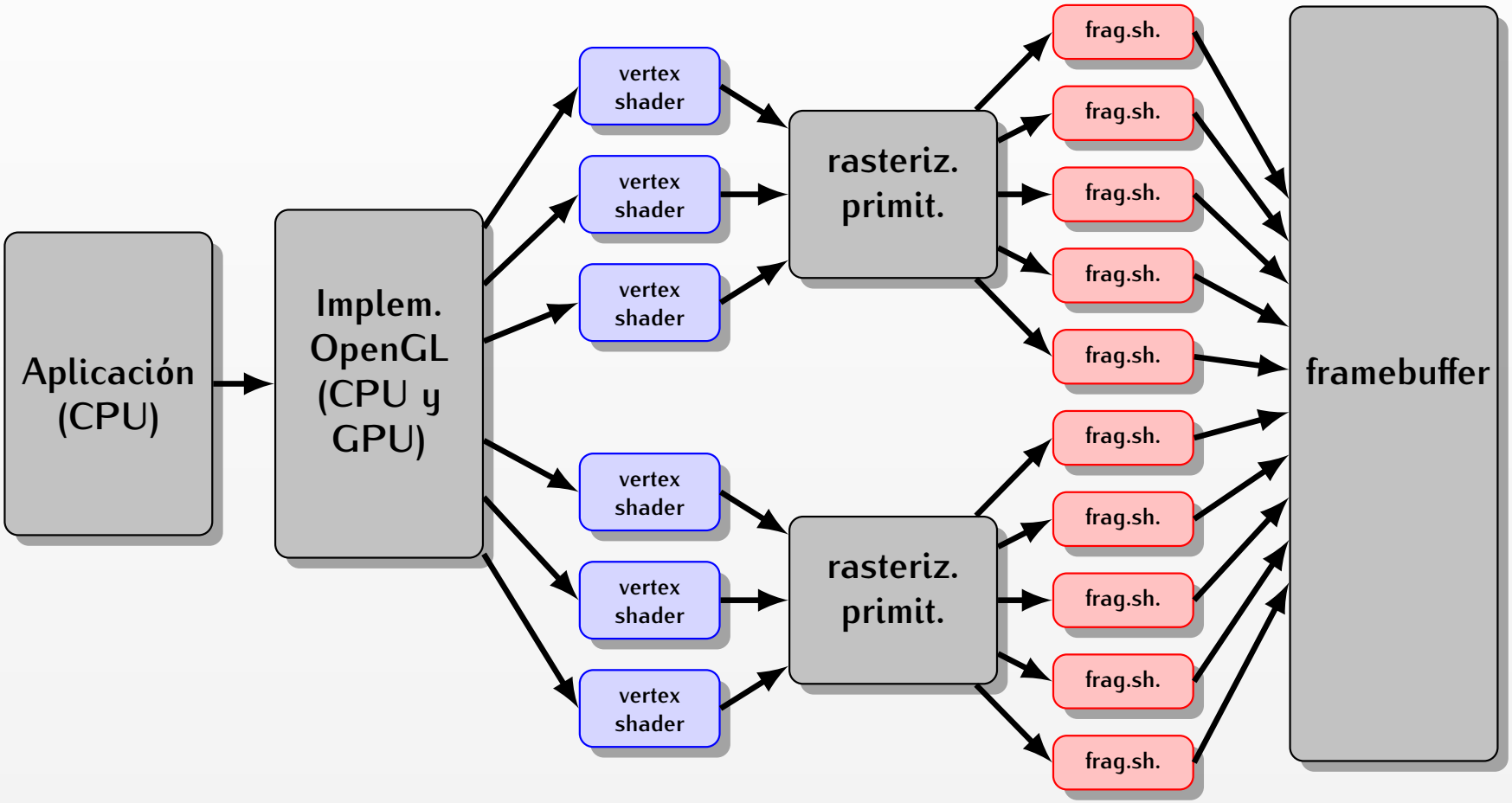
Rasterización de 2 triángulos

En este ejemplo, tenemos 6 vértices que definen 2 triángulos y que cubren 6 pixels (3 pixels cada triángulo):



Cauce gráfico: DFD simplificado

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



Subsección 4.2
Ejemplo de shaders básicos.

Creación y uso de shaders

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

- Los dos shaders deben estar almacenados en memoria en variables de tipo **`char *`** (vectores de caracteres o cadenas, acabados en 0). Es conveniente almacenarlos en archivos en el sistema de archivos.
- Los dos shaders deben compilarse usando llamadas a OpenGL (puede haber errores al compilar).
- Una vez compilados correctamente, los dos shaders se enlazan, creándose un programa.
- Una aplicación puede generar uno o varios programas. En OpenGL 3.0 y anteriores, siempre está disponible, además, el programa del cauce fijo.
- En cada momento hay un programa activo, que se usa para visualizar, y que se puede cambiar en cualquier momento.

Vertex shader elemental:

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

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

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

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

Fragment shader elemental:

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

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

    gl_FragColor = gl_Color;
}
```

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

Subsección 4.3
Creación y ejecución de programas.

Identifición y funcionalidad para shaders y programas

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

- Cada shaders o programa se identifica en la aplicación con un valor entero (**`GLuint`**), que llamamos su **identificador**.
- Existen funciones para:
  - Crear un shader (**`glCreateShader`**).
  - Asociar su código fuente a un shader (**`glShaderSource`**).
  - Compilar un shader (**`glCompileShader`**).
  - Crear un programa (**`glCreateProgram`**).
  - Asociar sus dos shader a un programa (**`glAttachShader`**).
  - Enlazar un programa (**`glLinkProgram`**).
  - Ver log de errores al compilar o enlazar.
  - Activar un programa (**`glUseProgram`**).

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

Compilar shaders

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

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

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

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

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

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

Crear y enlazar un programa

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

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

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

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

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

Inicialización y creación de shaders (1/2)

En la función de inicialización de OpenGL es necesario:

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

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

Inicialización y creación de shaders (2/2)

```
.....

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

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

// compilar shaders, crear programa
idProg = CrearPrograma("fragment-shader.glsl","vertex-shader.glsl");
```

Uso del programa

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

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

```
GLuint idProg ;
.....
void FGE_Redibujado()
{
   glUseProgram( 0 ) ; // activar programa de funcionalidad fija
   // envio de primitivas con los shaders estándard de OpenGL
   .......

   glUseProgram( idProg ); // activar nuestro programa
   // envio de primitivas con el shader creado por nosotros:
   ......
}
```

Subsección 4.4
Funciones auxiliares.

Verificar errores de compilación

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

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

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

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

Verificar errores de enlazado

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

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

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

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

Lectura de un archivo

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

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

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

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

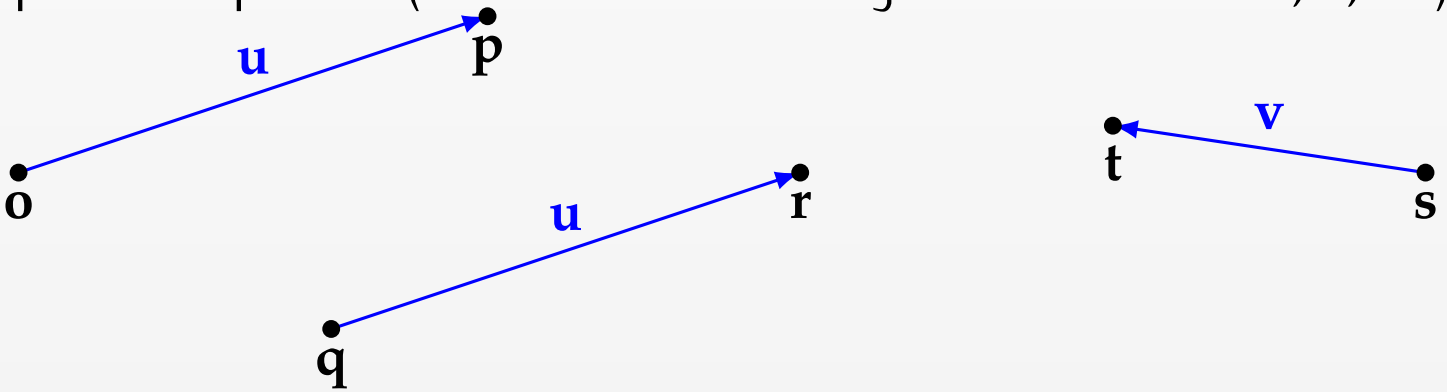
Sección 5
Apéndice: puntos, vectores y marcos de referencia

Subsección 5.1
Puntos y vectores

Puntos y vectores

El espacio 2D o 3D se puede considerar como un conjunto infinito de **puntos** o **localizaciones** (que notaremos con negrita minúscula: $\mathbf{p}, \mathbf{q}, \ldots$)

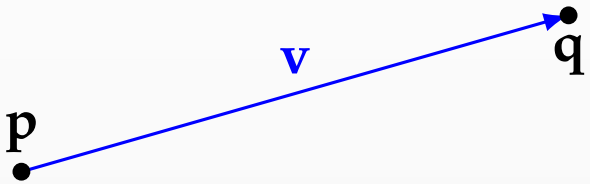
Un **vector** (o **vector libre**) es una entidad que representa las diferencias entre pares de puntos (usamos también negrita minúscula: $\mathbf{u}, \mathbf{v}, \ldots$)



(los vectores se puede dibujar con el origen en cualquier punto, p.ej. en esta figura aparece el vector $\mathbf{u}$ dibujado en dos sitios distintos).

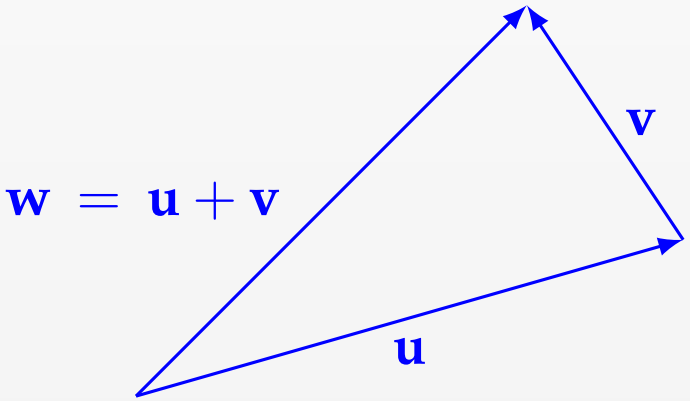
Resta de puntos, suma de vectores

La diferencia de dos puntos produce un vector (o, lo que es lo mismo, un punto más un vector produce otro punto)



$$\mathbf{q} - \mathbf{p} = \mathbf{v} \iff \mathbf{p} = \mathbf{q} + \mathbf{v}$$

Dos vectores $\mathbf{u}$ y $\mathbf{v}$ se pueden sumar entre si, produciendo otro vector $\mathbf{w} = \mathbf{u} + \mathbf{v}$, según la regla del paralelogramo:



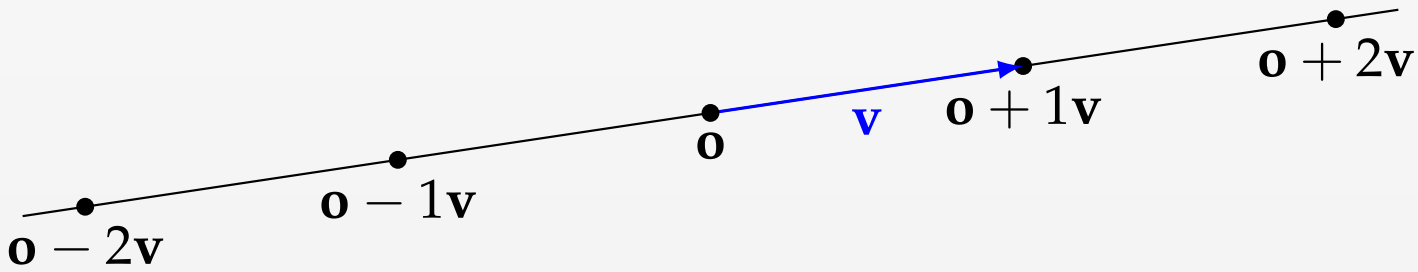
$$\mathbf{u} + \mathbf{v} = \mathbf{w} \iff \mathbf{v} = \mathbf{w} - \mathbf{u}$$

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

Producto de vectores y valores escalares

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

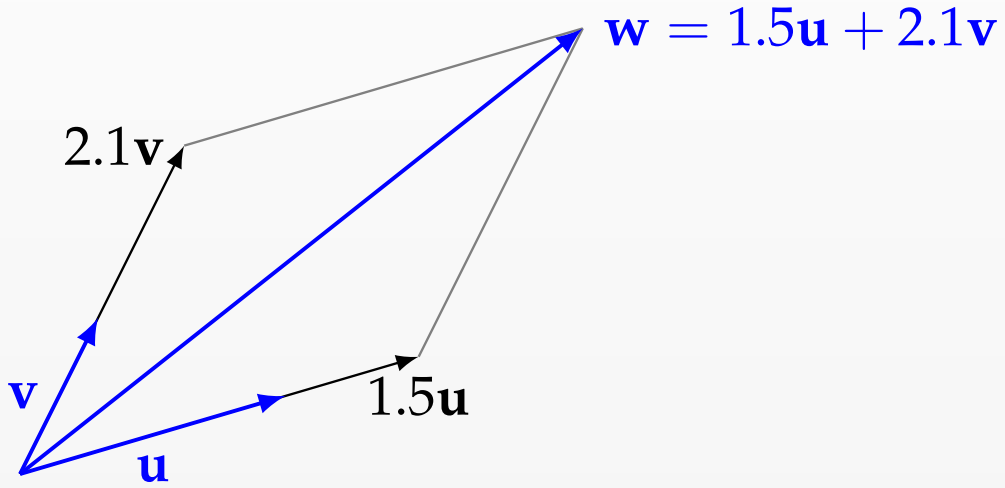
Como consecuencia, todos los puntos de la forma $\mathbf{o} + t\mathbf{v}$ (para todos los valores reales posibles de t) están en la recta que pasa por $\mathbf{o}$ y es paralela a $\mathbf{v}$



Subsección 5.2
Marcos de referencia y coordenadas

Bases de vectores

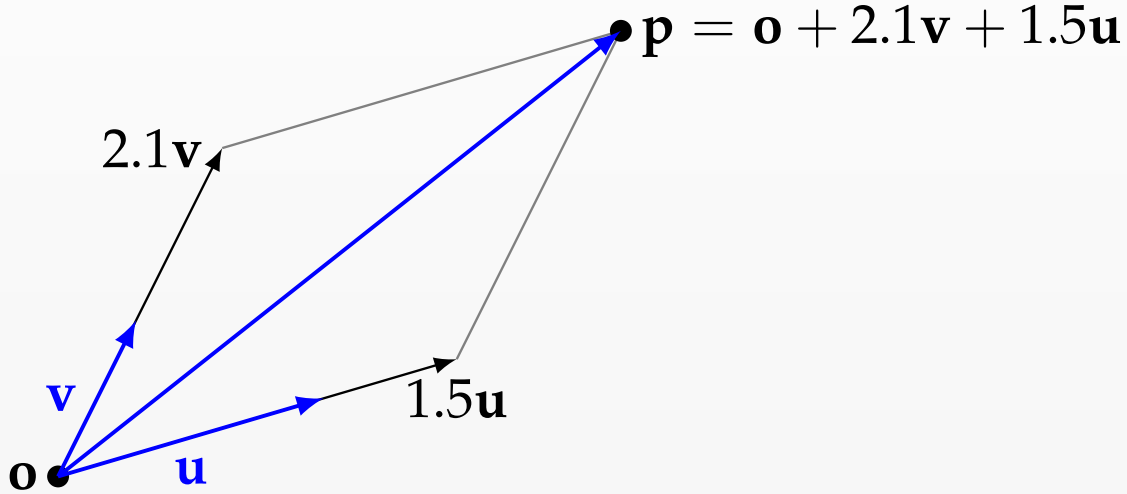
Usando dos vectores cualquiera $\mathbf{u}$ y $\mathbf{v}$ del plano (no paralelos ni nulos), podemos escribir cualquier otro vector $\mathbf{w}$ como una combinación lineal de ellos:



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

Marcos de referencia y coordenadas

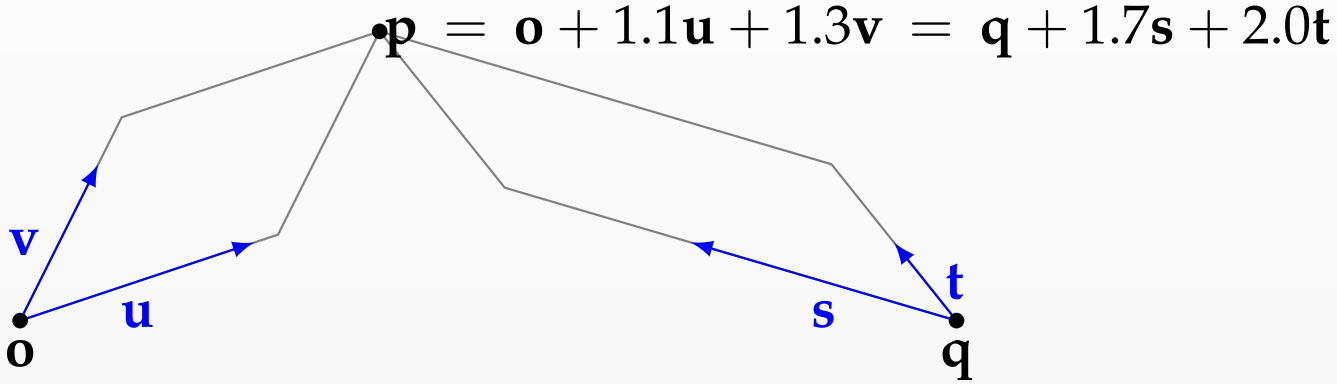
Si fijamos un punto $\mathbf{o}$ (origen) y una base $\{\mathbf{u}, \mathbf{v}\}$, cualquier punto $\mathbf{p}$ del plano se puede escribir como $\mathbf{p} = \mathbf{o} + a\mathbf{u} + b\mathbf{v}$:



- La terna $\mathbf{R} = [\mathbf{u}, \mathbf{v}, \mathbf{o}]$ forma un **marco de referencia** (*reference frame*) del plano 2D. (sirve para **identificar puntos y vectores usando distancias**).
- Al par (a, b) se le llaman las **coordenadas** del punto $\mathbf{p}$ en el marco de referencia $\mathbf{R}$.
- El conjunto de los puntos forma un **espacio afín** (*affine space*).

Coordenadas y puntos

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



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

$$\begin{aligned} \mathbf{p} &\equiv (1.1, 1.3) && \text{en el marco de referencia } \mathbf{R} = [\mathbf{u}, \mathbf{v}, \mathbf{o}] \\ \mathbf{p} &\equiv (1.7, 2.0) && \text{en el marco de referencia } \mathbf{S} = [\mathbf{s}, \mathbf{t}, \mathbf{q}] \end{aligned}$$

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

Operaciones usando coordenadas

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

- Supongamos dos vectores $\mathbf{u} \equiv (u_x, u_y, u_z)$ y $\mathbf{v} \equiv (v_x, v_y, v_z)$, y dos puntos $\mathbf{p} \equiv (p_x, p_y, p_z)$ y $\mathbf{q} \equiv (q_x, q_y, q_z)$ (todas las coordenadas en $\mathbf{R}$), y un valor real a.
- En estas condiciones las operaciones entre puntos y vectores se puede definir en términos de operaciones con sus coordenadas:

$$
\begin{aligned}
\mathbf{u} + \mathbf{v} &\equiv (u_x, u_y, u_z) + (v_x, v_y, v_z) = (u_x + v_x\,,\, u_y + v_y\,,\, u_z + v_z) \\
\mathbf{p} - \mathbf{q} &\equiv (p_x, p_y, p_z) - (q_x, q_y, q_z) = (p_x - q_x\,,\, p_y - q_y\,,\, p_z - q_z) \\
\mathbf{p} + \mathbf{v} &\equiv (p_x, p_y, p_z) + (v_x, v_y, v_z) = (p_x + v_x\,,\, p_y + v_y\,,\, p_z + v_z) \\
a\mathbf{u} &\equiv a(u_x, u_y, u_z) = (au_x\,,\, au_y\,,\, au_z)
\end{aligned}
$$

Subsección 5.3
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** $\mathbf{E} = [\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}]$, en ese marco **por definición**:

- los vectores $\mathbf{x}$, $\mathbf{y}$ y $\mathbf{z}$ **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 $\mathbf{x}$, $\mathbf{y}$ y $\mathbf{z}$ son **perpendiculares entre ellos dos a dos**: por tanto, esos vectores forman ángulos de 90 grados, y determinan los angulos entre cualquiera dos vectores.

Nótese que estas propiedades se refieren a la interpretación en el espacio de $\mathbf{E}$, no son por tanto, propiedades formales sino intuitivas. Más adelante se formalizan.

Producto escalar y módulo

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

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

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

$$\mathbf{x} \cdot \mathbf{x} = \mathbf{y} \cdot \mathbf{y} = \mathbf{z} \cdot \mathbf{z} = 1 \qquad \text{y} \qquad \mathbf{x} \cdot \mathbf{y} = \mathbf{y} \cdot \mathbf{z} = \mathbf{z} \cdot \mathbf{x} = 0$$

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

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

Marcos cartesianos

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

- sus vectores tienen longitud unidad:

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

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

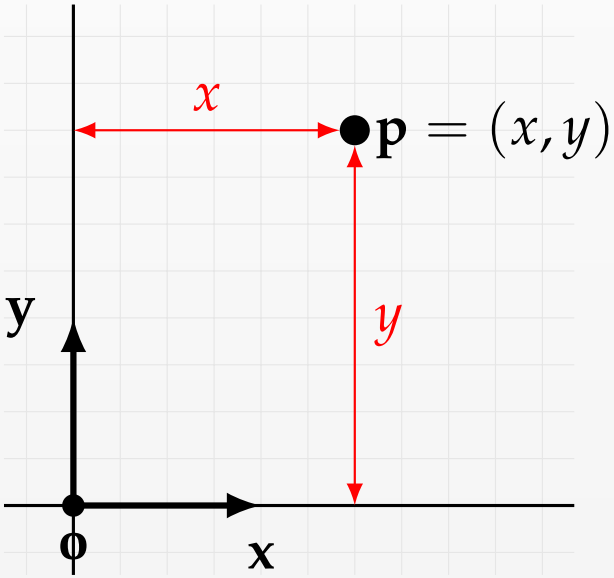
$$\mathbf{e}_x \cdot \mathbf{e}_y \;=\; \mathbf{e}_y \cdot \mathbf{e}_z \;=\; \mathbf{e}_z \cdot \mathbf{e}_x \;=\; 0$$

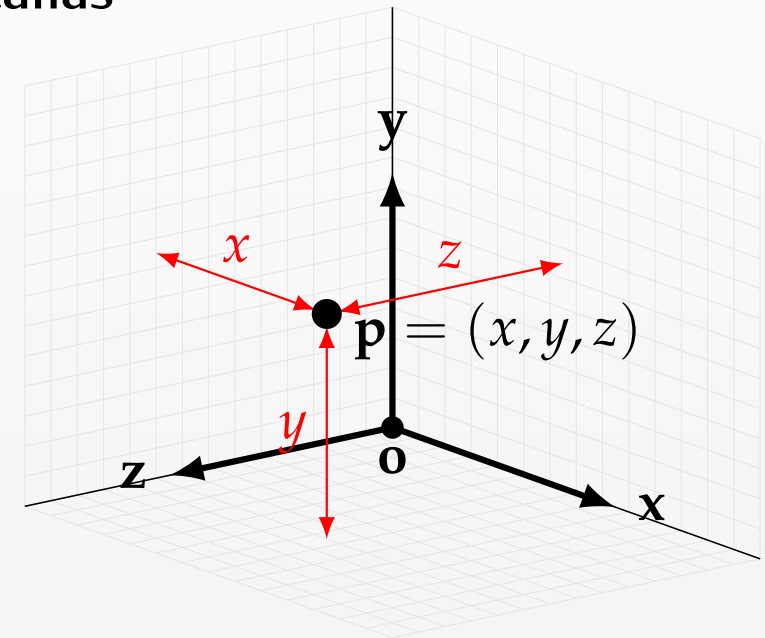
En estas condiciones, decimos que $\mathbf{R}$ es un marco de referencia **cartesiano**.

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

Marcos y coordenadas cartesianas

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





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

Calculo del producto escalar y el módulo

Se puede calcular fácilmente el producto escalar y el módulo de vectores usando sus coordenadas relativas a un marco cartesiano $\mathbf{R}$. Sean dos vectores $\mathbf{a} \equiv (a_x, a_y, a_z)$ y $\mathbf{b} \equiv (b_x, b_y, b_z)$ (coordenadas en $\mathbf{R}$):

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

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

  (este valor **es el mismo** para cualquier marco cartesiano $\mathbf{R}$).
- Como consecuencia, el módulo se puede obtener como:

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

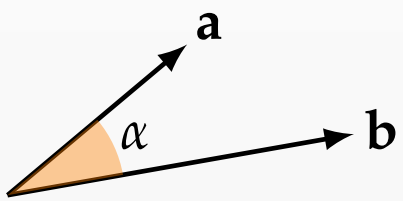
- El módulo de un vector coincide con su **longitud** en el espacio (ya que de los versores de $\mathbf{E}$ dijimos que tenían longitud unidad por definición).

Interpretación geométrica del producto escalar

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

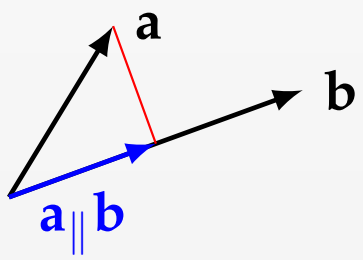
- El producto escalar es proporcional al coseno de α:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos \alpha$$



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

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



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

Producto vectorial

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

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

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

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

Cálculo del producto vectorial

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

- Se puede demostrar a partir de los axiomas, que si $\mathbf{a} \equiv (a_x, a_y, a_z)$ y $\mathbf{b} \equiv (b_x, b_y, b_z)$, entonces:

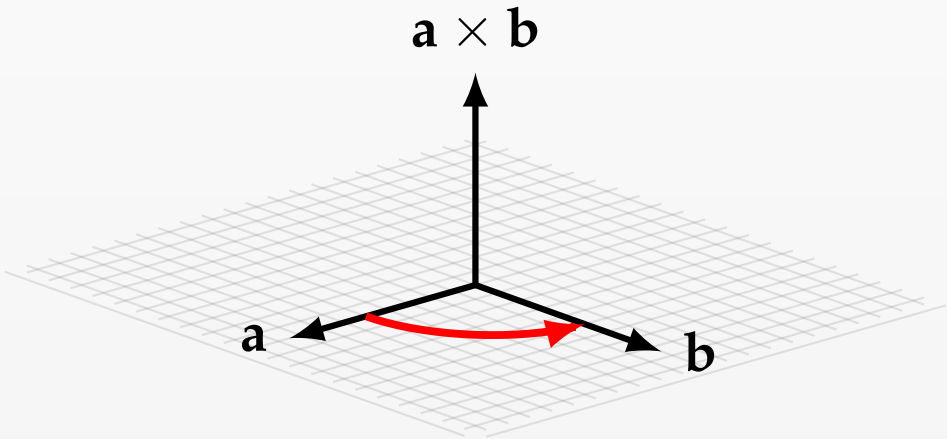
$$\mathbf{a} \times \mathbf{b} \equiv (a_y b_z - a_z b_x \, , \, a_z b_x - a_x b_z \, , \, a_x b_y - a_y b_x)$$

- Esta propiedad se cumple siempre que $\mathbf{R}$ 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 $\mathbf{a} \times \mathbf{b}$ es perpendicular al plano que forman $\mathbf{a}$ y $\mathbf{b}$ (y por lo tanto, perpendicular tanto a $\mathbf{a}$ como a $\mathbf{b}$)



En los marcos de referencia a derechas, la dirección de $\mathbf{n} = \mathbf{a} \times \mathbf{b}$ es la dirección en la que avanza un tornillo paralelo a $\mathbf{n}$ cuando se gira desde $\mathbf{a}$ hacia $\mathbf{b}$

Interpretación geométrica del producto vectorial (2)

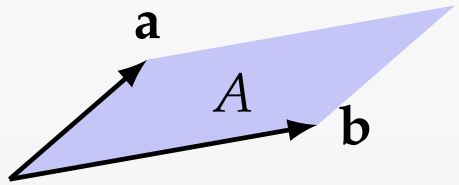
En un marco cartesiano

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

$$\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \, \|\mathbf{b}\| \, \sin \alpha$$

- Esa longitud es igual al área A del parelepípedo formado por $\mathbf{a}$ y $\mathbf{b}$

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



- Por lo tanto, si $\|\mathbf{a}\| = 1$ y $\|\mathbf{b}\| = 1$, entonces:

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

Subsección 5.4
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 vectores 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:

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

nótese que hemos usado el símbolo $\vec{c}$ para denotar una tupla de coordenadas. Usaremos este tipo de símbolos ($\vec{a},\vec{b},\vec{c},\ldots$) para las tuplas de coordenadas homogéneas.

Esta notación pretende dejar clara la diferencia entre los puntos y sus coordenadas, algo especialmente relevante para entender las transformaciones.

Puntos, vectores y coordenadas homogéneas

En un marco de referencia cualquiera $\mathbf{R} = [\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}]$, una tupla de coordenadas homogéneas $\vec{c} = (c_x, c_y, c_z, c_w)^t$ representa un punto o un vector $\mathbf{s}$ del espacio, lógicamente definido como:

$$\mathbf{s} = c_x\mathbf{x} + c_y\mathbf{y} + c_z\mathbf{z} + c_w\mathbf{o}$$

(aquí hemos definido $0\mathbf{o} = \mathbf{0}$ (el vector nulo) y $1\mathbf{o} = \mathbf{o}$ (el mismo punto)). Con esto, la igualdad anterior se puede expresar de forma matricial:

$$\mathbf{s} = c_x\mathbf{x} + c_y\mathbf{y} + c_z\mathbf{z} + c_w\mathbf{o} = [\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}] \begin{pmatrix} c_x \\ c_y \\ c_z \\ c_w \end{pmatrix} = \mathbf{R}\,\vec{c}$$

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

$$\mathbf{s} = \mathbf{R}\,\vec{c}$$

Subsección 5.5
Representacion y operaciones con tuplas

Representación en memoria de coordenadas.

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

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

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

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

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

Creación, consulta y modificación de tuplas.

Este código válido ilustra las distintas opciones:

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

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

// accesos de solo lectura, usando su posición o índice en la tupla (0,1,2,...),
// o bien varias constantes 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)
```

Fin de la presentación.