GUÍA PARA EL TRABAJO AUTÓNOMO

TEORÍA

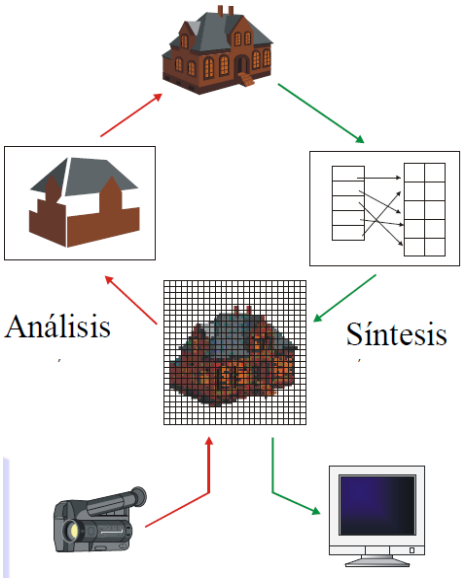
TEMA 1. INTRODUCCIÓN A LA INFORMÁTICA GRÁFICA

Objetivos:

- Conocer los fundamentos de la Informática Gráfica.
- Comprender el funcionamiento del cauce gráfico de visualización.
- *Tomar conciencia de los avances producidos en los últimos 50 años para valorar el estado actual*
- *Entender el proceso de síntesis de imágenes*
- *Conocer un cierto abanico de soluciones gráficas aplicables en distintas disciplinas*

INTRODUCCIÓN

Es muy posible que usted viva cada hora varias situaciones que no se habrían podido suceder sin la participación de un ingeniero informático, y menos sin la concurrencia de la Informática Gráfica. Hagamos un breve repaso:



Pensemos por ejemplo en un gesto tan habitual como hablar por el móvil. ¿Cómo se ha diseñado el aparato? Muy posiblemente utilizando una herramienta CAD, con un interfaz gráfico de usuario en el cual el especialista en diseño industrial ha dibujado líneas, superficies y ha aplicado colores y tomado medidas hasta llegar al diseño del objeto que ahora mismo alberga su tarjeta SIM. Pero es bastante probable que su móvil no sea un simple terminal telefónico, sino que estará funcionando sobre un sistema Android, iOS o Windows. ¿Ha visto la resolución de su pantalla? ¿Y cómo se realizan los efectos gráficos? ¿Y lo fácil que es hacer clic sin ratón? Seguro que tiene algún juego instalado... gráfico, por supuesto.

Es posible que el próximo fin de semana haya quedado para ir al cine. Independientemente de la película escogida, es completamente seguro que en su producción se han utilizado gráficos por ordenador. No es necesario que sean películas de animación (2D ó 3D), sino que en películas "normales" también se recrean escenarios y objetos virtuales que pasan desapercibidos para el espectador no avezado. Normalmente, con un buen presupuesto, los efectos 3D se notan menos que el retoque fotográfico de los carteles anunciadores.

Quizá alguien de su entorno se haya visto sometido en algún momento a una Resonancia Magnética Nuclear, o a una Tomografía Axial Computerizada (TAC), y los resultados posiblemente los haya examinado el galeno en la pantalla de un ordenador. Gracias a estos avances, y la posibilidad de visualizar en 2D y 3D los datos numéricos obtenidos, la medicina actual es capaz de detectar enfermedades y anomalías de una forma más precisa que hace treinta años. También gracias a la Informática Gráfica, los cirujanos del mañana se entrenan con pacientes virtuales y no con seres humanos reales, lo que tranquiliza bastante ¿no?

Todos hemos visto proyectos urbanísticos con recreaciones virtuales de edificios que aún se encuentran sólo en plano. Además, no sólo debe reflejarse el edificio y sus volúmenes, sino también integrarse con realismo en el entorno, para que creamos que vemos el edificio real. Ello se consigue con modelado geométrico 3D de los edificios y síntesis realista de la imagen.

Dicho de otra manera, y sin ánimo de ser prepotentes: **el mundo tal y como lo conocemos hoy no sería posible sin los elementos desarrollados por la Informática Gráfica.**

EJERCICIOS

1.En la página anterior se han citado cuatro ámbitos en los que la informática gráfica forma parte fundamental: el diseño industrial, el cine, la medicina y la arquitectura. Describa usted otras cuatro disciplinas profesionales o científicas que requieran de la participación inexcusable de los gráficos por ordenador para su correcto desempeño.

2. Utilice el espacio disponible en el recuadro para describir en lenguaje natural con el máximo detalle la figura mostrada (no más de 50 palabras)

3.Cuando ve una película de animación en el cine, ¿en qué se fija, en el guion o en los efectos más o menos realistas? ¿Recuerda el movimiento del pelo de "Sulley" en "Monstruos SA"? ¿Y el resbalar de la capa de "Encantador" sobre su caballo al llegar al castillo en "Shrek"? Si no lo recuerda, es buen momento para volver a verlo.

4.Comparta con los amigos una sesión de cine: Gravity. Documéntese y coménteles todos los elementos virtuales utilizados en la producción de la película.

Hasta ahora hemos estado hablando de *informática gráfica*, dando por supuesto que todos hablamos de lo mismo. Pero echemos un vistazo a dos definiciones:

- *"Gráficos por ordenador es la creación, manipulación, análisis e interacción de las representaciones pictóricas de objetos usando ordenadores (Carlson 93)*
- *"Informática gráfica es la síntesis pictórica de objetos reales o imaginarios basada en sus modelos de ordenador" (Foley 90)*

EJERCICIOS

5.Analice ambas definiciones (Carlson93 y Foley90) y comente las diferencias entre ambas. ¿Cuál le parece más acertada?

6.¿Qué ventajas tiene el ordenador frente al humano u otros medios técnicos (p.ej. Cámaras) para la generación de imágenes? Lea http://es.gizmodo.com/no-son-fotos-el-75-del-catalogo-de-ikea-esta-generado-1628534358 (consultado 15/09/2021)

7.Consulte en la bibliografía y comente la diferencia entre análisis y síntesis de imágenes. Sirva de ayuda la imagen que acompaña el principio del tema.

Historia de la Informática Gráfica

No siempre se ha trabajado con los ordenadores en modo gráfico. El ratón, el monitor en color con 16 millones de colores y super resolución no han existido siempre. Para valorar lo que disponemos ahora, podemos echar un vistazo a las videotecas históricas:

- La primera tableta digitalizadora (1963)
- Presentación pública del ratón (1968)

Repasar la historia de la informática gráfica puede resultar una experiencia sin duda sorprendente para muchos de ustedes, que nacieron con el ratón en la mano y la pantalla plana a color en el escritorio.

Marcamos algunos hitos que sin duda ilustrarán los inicios de esta disciplina, y que podrá usted ampliar indagando en la bibliografía o en la web:

- 1947. La US Air Force desarrolla el Whirlwind, un ordenador para control de aviones.
- 1963. Sutherland desarrolla su tesis doctoral sobre la tableta digitalizadora
- 1964. General Motors presenta su sistema CAD de diseño de coches
- 1965. Algoritmo de Bresenham para el dibujado de líneas rectas. (PDF)
- 1967. Interpolación de superficies (FORD)
- 1968. Engelbart presenta el primer ratón
- 1969. Line Drawing System 1: El primer hardware de CAD
- 1970. Primera representación 3D: Univ. Cornell. Program of Computer Graphics

Una completa referencia se puede encontrar en este libro de la National Academies Press.

EJERCICIOS

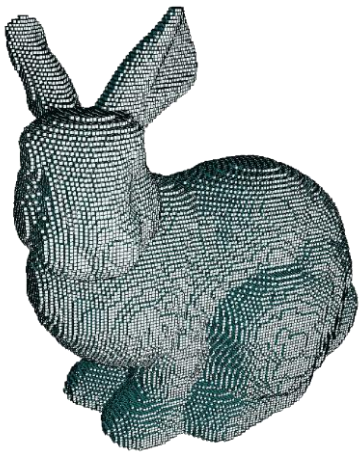
8. Averigüe el origen de las curvas de Bèzier y D'Casteljau. ¿Es casualidad que ambos trabajaran en la industria automovilística?

9. Documente al menos otros diez hitos de la Informática Gráfica a partir de 1970, al estilo de los indicados anteriormente.

Áreas de la Informática Gráfica

Lo que se describe con tan sólo dos palabras en realidad puede ser refinado. Cuando hablamos de Informática Gráfica hablamos de disciplinas tan diversas como, por ejemplo:

- Modelado (*modelling*)
- Síntesis de imágenes (*rendering*)
- Animación (*animation*)
- Realidad Virtual (*virtual reality*)
- Interacción (*user interaction*)
- Visualización (*visualization*)
- Digitalizacíon 3D (*3D scanning*)



EJERCICIOS

10. *Enumere las diferencias que hay entre modelar un edificio y digitalizarlo en 3D*
11. *En su opinión ¿es correcto, como hace Shirley, considerar la Realidad Virtual una disciplina independiente?*
12. *¿Por qué cree he extraído de la lista original del libro [Shirley09] las disciplinas "procesamiento de imágenes" y "fotografía computacional"?*

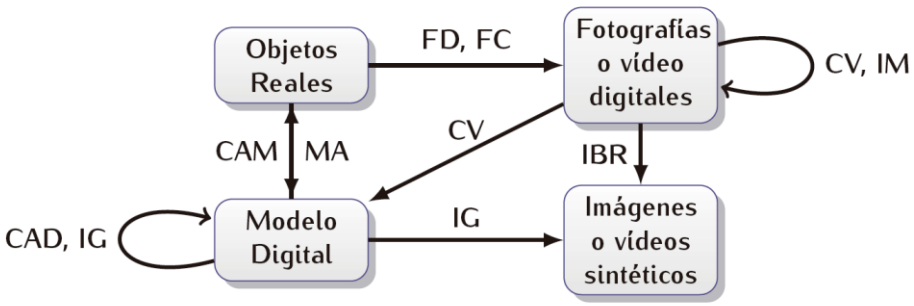
La definición de estas áreas se puede encontrar en la referencia [Shirley09], por lo que obviamos en esta guía dicho trabajo que habrá de realizar usted para comprender a qué nos referimos.

La Informática Gráfica puede considerarse [Ureña14], también, un campo multidisciplinar que hace uso de otras disciplinas, destacando entre todas:

- Programación Orientada a Objetos
- Ingeniería del Software
- Geometría Computacional
- Matemática Aplicada (métodos numéricos)
- Física (óptica, dinámica, cinemática)
- Psicología y Medicina (percepción visual)

También se hace uso de otros campos de la informática, como la Inteligencia Artificial en la creación de videojuegos o la Estadística para la visualización realista.

A su vez, la Informática Gráfica se encuentra dentro de un área de Computación Visual (*visual computing*) en la que se relaciona con otras disciplinas científicas, como la Visión por Computador (CV), el Diseño Asistido por Computador (CAD) o la Fotografía Computacional (FC).



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

Como hemos visto al inicio del tema, la Informática Gráfica es una de las disciplinas que están presentes de una forma más perceptible en el día a día:

- Videojuegos
- Producción de animaciones y efectos especiales para cine y televisión
- Diseño industrial
- Modelado y Visualización en Ingeniería y Arquitectura
- Simuladores y juegos serios para entrenamiento y aprendizaje
- Visualización de datos
- Visualización científica y médica
- Arte Digital
- Patrimonio Cultural

Es por ello por lo que esperamos que esta asignatura sea de su interés y le saque el máximo provecho a la misma, pues con una alta probabilidad, bien como usuario o como desarrollador, tendrá que enfrentarse a problemas donde los gráficos por ordenador serán imprescindibles.

EJERCICIOS

13. ¿A qué aplicación de la IG de las anteriores pertenece cada una de las siguientes ilustraciones?

Ilustración 1 Fotograma del videojuego Watch Dogs de Ubisoft.
Vídeo: http://www.youtube.com/watch?v=kPYgXvgS6Ww

Ilustración 4. Realidad aumentada.
Imagen: http://technomarketer.typepad.com/technomarketer/2009/04/firstlook-augmented-reality.html

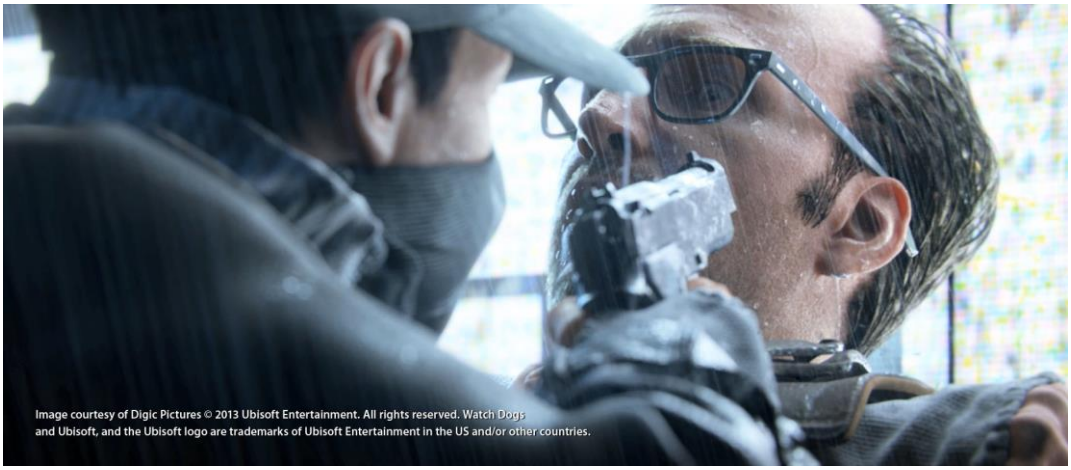


Ilustración 3 Fotograma del tráiler cinemático del videojuego Wacth Dogs. Imagen creada por Digic Pictures para Ubisoft, Img: _ http://www.fxguide.com/featured/the-state-of-rendering-part-2/#arnold.

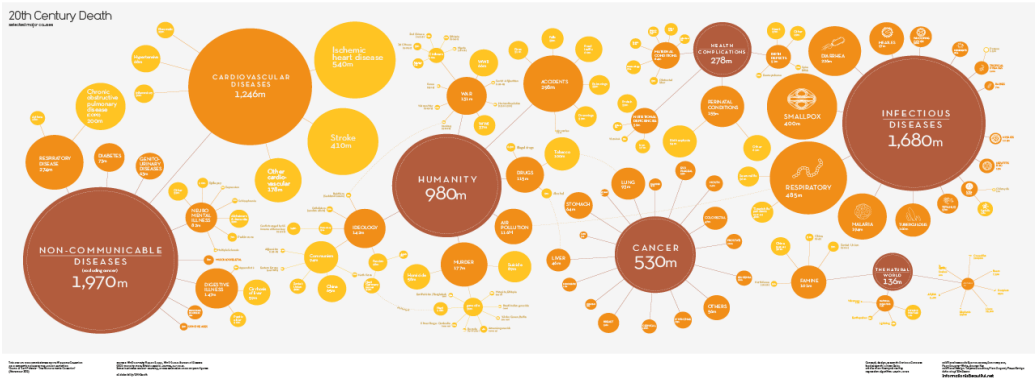


Ilustración 2. Frecuencia de causas de muerte en el siglo XX:
http://www.informationisbeautiful.net/visualizations/20th-century-death/

Ilustración 5: MIT Technology Review.
http://www.technologyreview.com/view/428134/the-future-of-medical-visualisation/

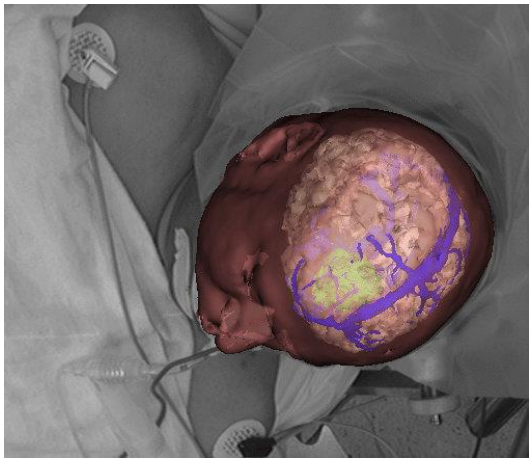


Ilustración 6: Cirugía guiada por realidad aumentada.
http://www.cs.rochester.edu/u/brown/projects.html

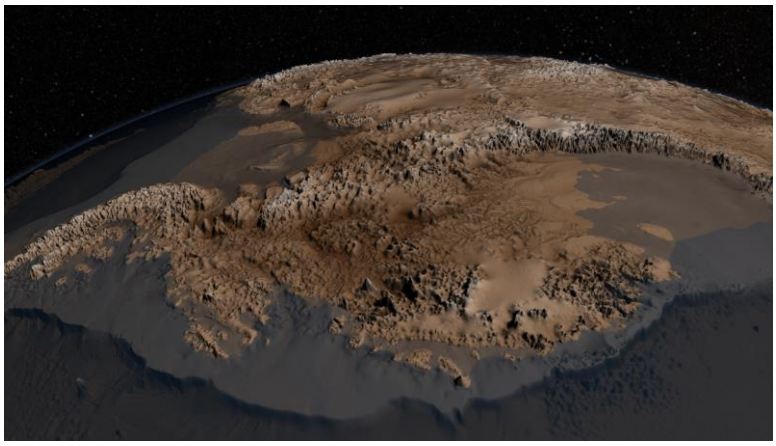


Ilustración 7: Visualización de la topografía del suelo de la Antártica (NASA):

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

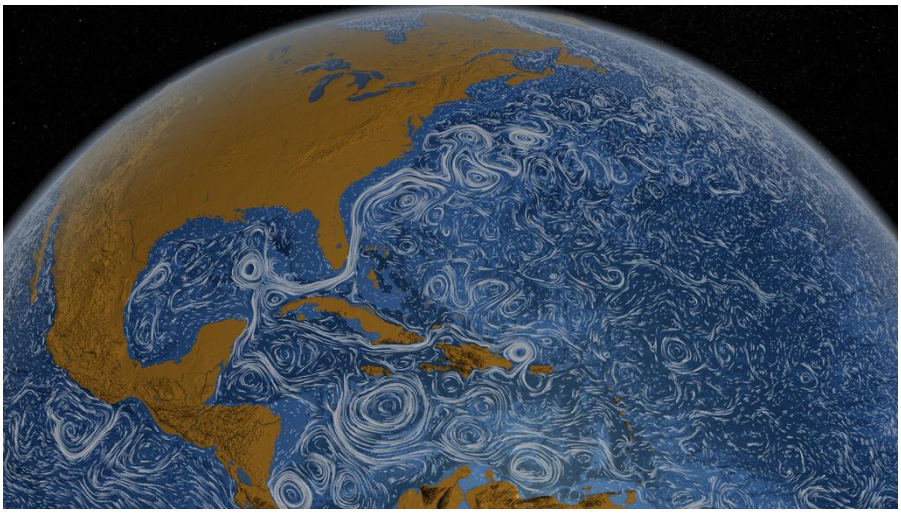


Ilustración 8: Visualización de las corrientes oceánicas.

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

Ilustración 9: Simulador de conducción Mercedes-Benz
(http://mercedesbenzblogphotodb.wordpress.com/2010/10/06/)

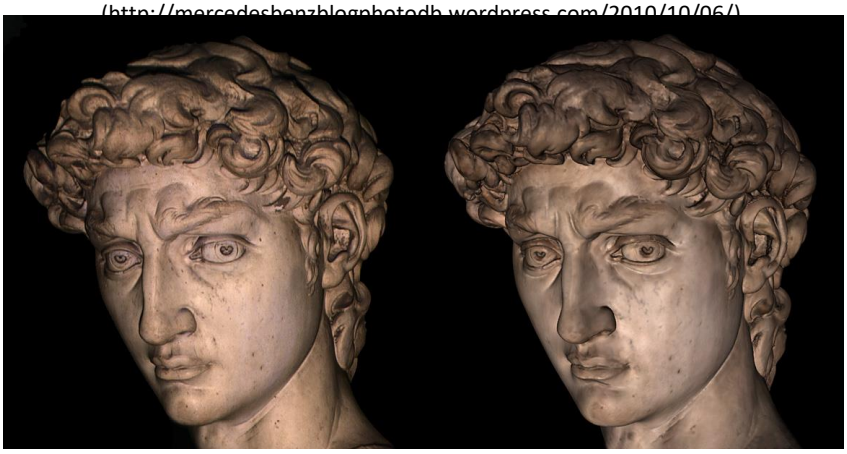


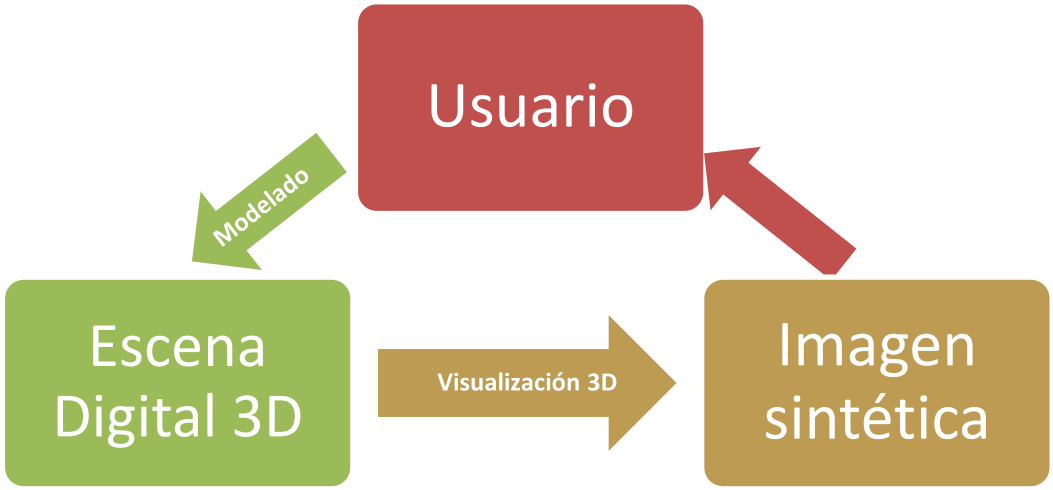
Ilustración 10: Fotografía (izda) y visualización 3D del modelo digital (dcha.) del David de Miguel Ángel ©The Digital Michelangelo, Stanford University

EJERCICIOS

14. *Observe la escena que tiene ahora mismo en la sala donde se encuentra. Intente describir el proceso mediante el cual se forma la imagen en su retina.*
15. *¿Conoce el mecanismo de funcionamiento de una cámara de fotos simple, denominado pinhole? Si no es así, búsquelo e intente establecer la analogía con el ejercicio anterior.*

El Proceso de Visualización

En las aplicaciones interactivas 3D hay tres elementos básicos: el usuario, el modelo digital 3D y la imagen sintética generada:



Para generar una imagen sintética, podemos seguir la analogía de la toma de una fotografía con la cámara, representada en la Ilustración 13Ilustración 13: Elementos que intervienen en la generación de la imagen 2D. Necesitamos:

- Objetos o personas en un entorno concreto, con una forma y colores determinados.
- Una cámara, para capturar la fotografía.
- Luz (pues sin luz se vería todo negro).

Pues en el caso de la Informática Gráfica, para la generación de imágenes por ordenador, es la misma necesidad la que nos encontramos. Necesitamos:

- Modelos digitales 3D de lo que queremos representar:
  - Geometría (primitivas geométricas, normalmente triángulos)
  - Propiedades sobre la apariencia (textura, material, color)
- Iluminación (posición de luces, modelo de sombreado, etc)
- Una o varias cámaras, indicando para cada una
  - Lente que usar (ángulo de visión)
  - Posición y orientación
- Información sobre la resolución de la imagen de salida

Estos elementos forman los dos componentes principales: la escena y los parámetros de visualización. Volviendo a la analogía con el mundo real, la escena es lo que está en el mundo, independientemente de que hagamos la foto o no. Los parámetros de visualización son los que hacen que la foto salga de una manera u otra (posición de la cámara, lente, formato de salida, filtros, etc.)

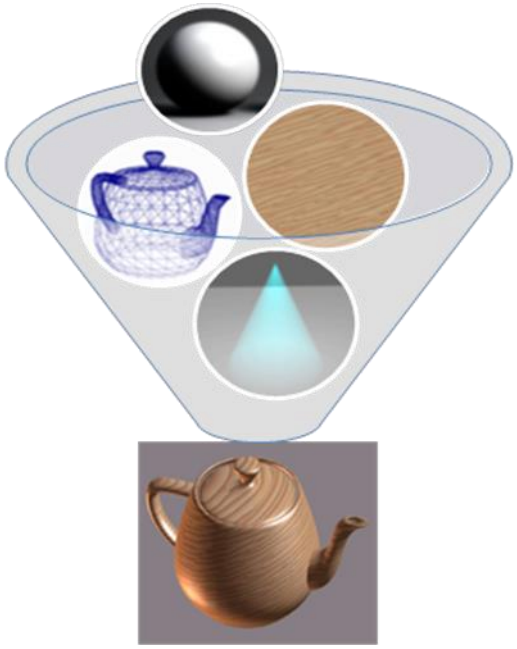


Ilustración 11 Diversos elementos necesarios para componer una imagen por ordenador.

En la Ilustración 12 se representa de forma muy simplificada el proceso para generar una imagen 2D a partir de un modelo 3D.

1. **Transformaciones del modelo**
   - Situarlo en la escena
   - Cambiarlo de tamaño
   - Crear modelos compuestos de otros más simples
2. **Transformación de vista**
   - Poner al observador en la posición deseada
3. **Transformación de perspectiva**
   - Pasar de un mundo 3D a una imagen 2D
4. **Rasterización**
   - Calcular para cada píxel su color, teniendo en cuenta la primitiva que se muestra, su color, material, texturas, luces, etc.
5. **Transformación de dispositivo**
   - Adaptar la imagen 2D a la zona de dibujado

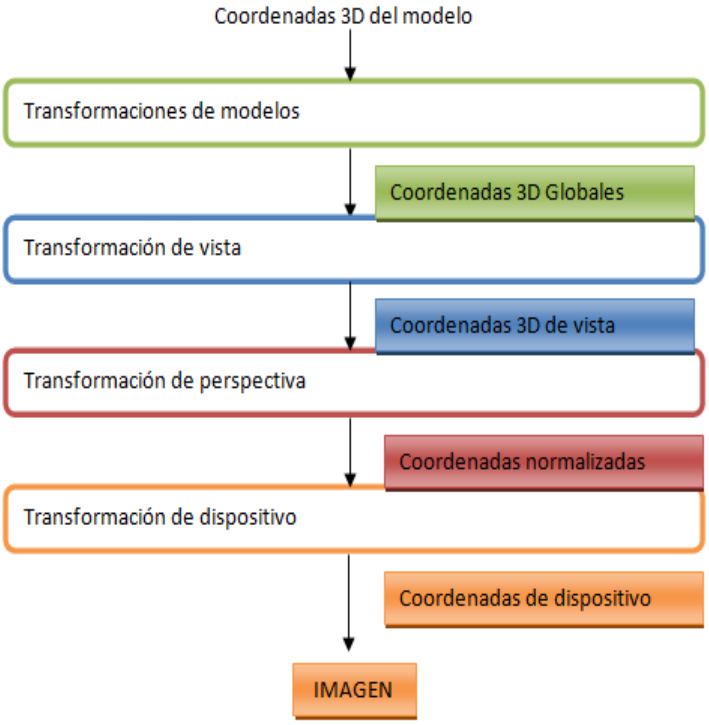


Ilustración 12 Pipeline Gráfico Fijo.

Todos estos pasos requieren docenas de operaciones que iremos desgranando a lo largo de la asignatura, y que en este tema introductorio sólo requieren de usted la comprensión de los que supone cada uno de los pasos a alto nivel.

Ilustración 13: Elementos que intervienen en la generación de la imagen 2D

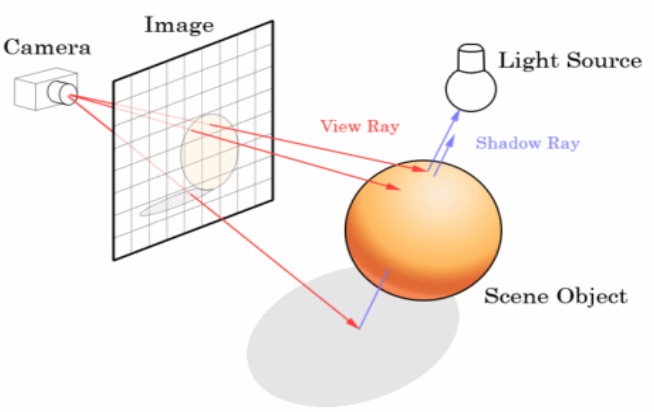


Ilustración 14: Trazado de rayos.

Rasterización y ray-tracing

Le quedará claro a estas alturas que el paso de un punto en el espacio cartesiano 3D a un pixel de color RGB en el *framebuffer* no es para nada trivial. De hecho, no hemos indicado nada de cómo se calcula el color de cada pixel de la imagen 2D.

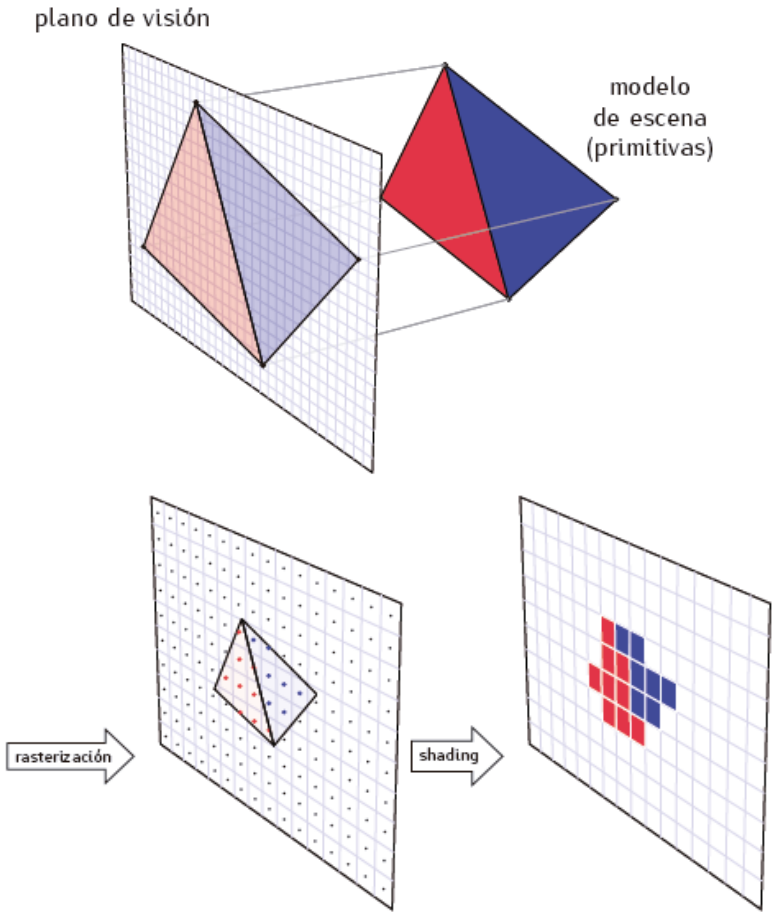


Ilustración 15: Desde las primitivas a los pixeles por rasterización.

En esta asignatura nos vamos a centrar en los algoritmos relacionados con la generación de imágenes por rasterización, pero no es la única forma, como veremos. De una forma esquemática, se puede relatar como:

```
Inicializar el color de todos los pixeles
Para cada primitiva P de la escena a visualizar
        Encontrar el conjunto S de pixeles cubiertos por P
Para cada pixel s de S:
        Calcular el color de P en s
        Actualizar el color de s
```

Por primitiva entendemos los elementos más pequeños que pueden ser visualizados. Normalmente serán triángulos, pero también pueden ser polígonos, puntos, segmentos de recta, círculos, etc. La complejidad en tiempo es del orden del número de primitivas por el número de píxeles, por lo que escenas muy grandes lleva mucho más tiempo en ser rasterizadas.

Otra opción es el algoritmo de ray-tracing (trazado de rayos), que suele ser más lento pero consigue resultados más realistas, por lo que se utiliza para síntesis de imágenes realistas *off-line*, como la producción de animaciones y efectos especiales en películas o alumnos. Sintéticamente, el algoritmo consiste en invertir los bucles de la rasterización:

```
Inicializar el color de todos los pixeles
Para cada pixel s de I:
        Calcular T el conjunto de primitivas que cubren s
        Para cada primitiva P de T
              Calcular el color de P en s
              Actualizar el color de s
```

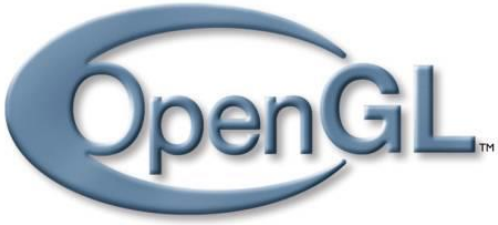
En el trazado de rayos lo que se hace es seguir el camino que sigue un rayo de luz desde la imagen hasta la fuente de luz, pasando por la cámara y rebotando entre todos los objetos de la escena (Ilustración 14).

Con la Ilustración 14 también mostramos todos los elementos que forman parte de la creación de la escena, y cómo esta se proyecta en un plano de imagen. En clase iremos explicando este proceso poco a poco.

La diferencia de estas dos técnicas se puede ver los resultados en la Ilustración 16. En la asignatura nos quedaremos en el resultado de la imagen superior, la de rasterización.

Ilustración 16: Rasterización vs Trazado de Rayos (Intel)

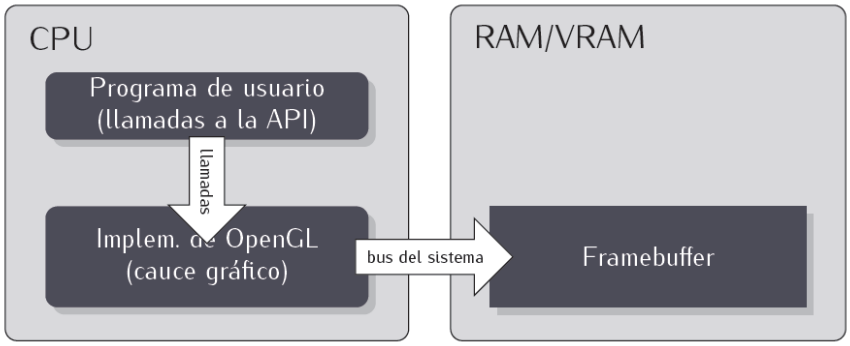
INTRODUCCION A OPENGL



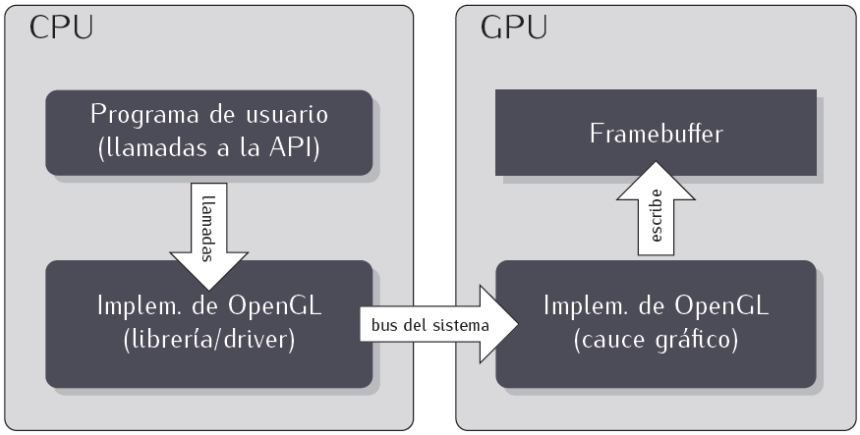
OpenGL es una API independiente de plataforma que se utiliza para la visualización 2D/3D basada en rasterización. Eso quiere decir que no está limitada a un lenguaje concreto, sino que define una lista de nombres de funciones, sus parámetros y el comportamiento esperado de cada una.

En la década de los 80, los programas gráficos se escribían para cada hardware específico, escribiendo directamente sobre la memoria de video, y a finales de esta década, Silicon Graphics era la líder en estaciones gráficas, ofreciendo IrisGL, que era su API propietaria.

En 1992, surgió OpenGL como evolución libre de IrisGL, permitiendo abstraer el uso de primitivas gráficas, sin necesidad de escribir directamente en la memoria de video. En el desarrollo del estándar intervinieron Microsoft, IBM, Intel, etc.



En 2004, con la aparición de OpenGL 2.0, se añadió la especificación de OpenGL SL (Shading Language), un lenguaje similar a C que permitía la programación del pipeline gráfico en la tarjeta gráfica. En 2017 apareció la versión 4.6.



Para más información sobre su evolución, ver http://www.opengl.org/wiki/History_of_OpenGL

OpenGL no es la única API de gráficos 3D, pero sí la más extendida entre las libres y multiplataforma. Su principal competidor, por denominarlo así, es la API de Microsoft© (Direct3D©), pero como es de esperar, ésta sólo funciona en sistemas Windows© y es privativa.

OpenGL tiene diversos sub-lenguajes o especificaciones, como pueden ser OpenGL ES (OpenGL for Embedded Systems) o WebGL.

El objetivo final del uso de OpenGL es generar imágenes mediante la rasterización de escenas 3D, independientemente del lenguaje de programación utilizado, del hardware existente y del sistema operativo que se esté utilizando.

Una cosa que a los que comienzan en el mundo de OpenGL les llama la atención es que OpenGL no maneja nada relacionado ni con la gestión de ventanas ni con los eventos sobre éstas. OpenGL se centra única y exclusivamente en procesar unos datos, referentes a la escena 3D y generar un buffer de memoria, *framebuffer*, con la imagen 2D rasterizada. La API no se preocupa si esa imagen va por pantalla, a un archivo o en un holograma. Por tanto, será necesario utilizar una librería de interfaz de usuario distinta a OpenGL para gestionar las ventanas y los eventos, y aquí las posibilidades son casi infinitas: GLU, Qt, WxWidgets, Java, FLTK, Tcl-TK, etc.

Otra característica de OpenGL es su arquitectura cliente/servidor. ¿Cómo puede ser cliente/ servidor sin salir del PC o del móvil? Pensemos en la CPU como cliente, y la GPU como servidor. OpenGL funciona como una máquina de estados, que envía a la GPU instrucciones y cambios de estado que son aplicados mientras no se diga lo contrario.

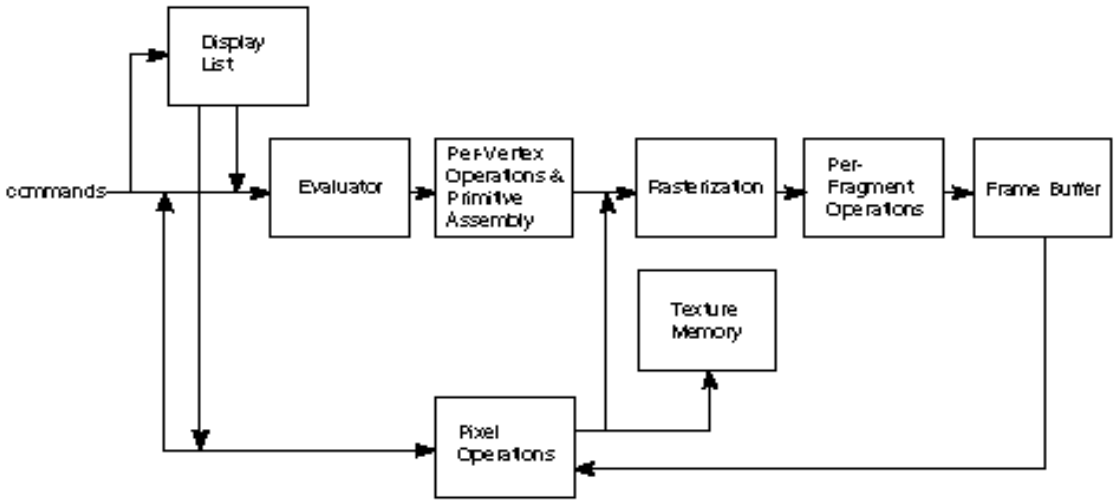


Ilustración 17: Diagrama de bloques de OpenGL

Una cuestión interesante es cómo resuelve OpenGL el polimorfismo en C. Ya sabemos que no existe el polimorfismo en C, sin embargo, hay funciones de OpenGL que pueden recibir parámetros de distinto tipo. Por ello, siempre se sigue el siguiente esquema:

```
gl funcion #parámetros tipo
```

Por ejemplo, `glVertex{2,3,4}{b,s,i,f,d,ub,us,ui}{v}` da lugar a

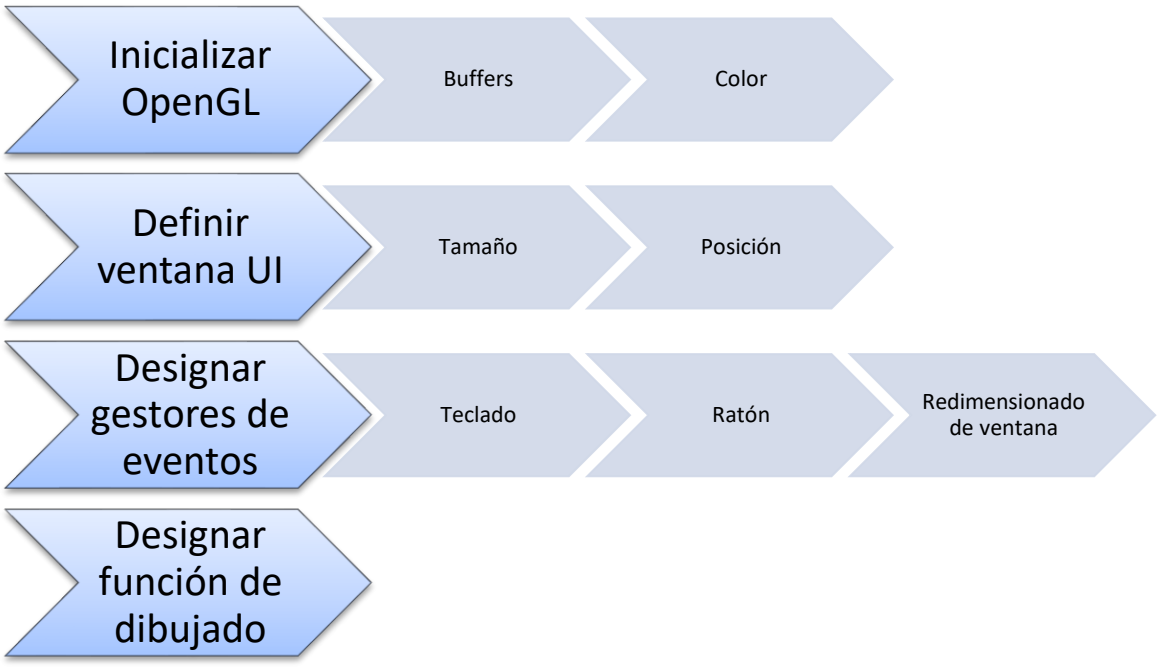
- `glVertex2i` (dos parámetros enteros)
- `glVertex3f` (tres parámetros reales)
- `glVertex3fv` (un vector de tres reales)
- `glVertex4d` (cuatro parámetros dobles)

```
glVertex2i(1,2) ; // Dos coordenadas enteras
glVertex3i(1,2,4) ; // Tres coordenadas enteras
glVertex3f(1.,2.2,43.56) ; // Tres coordenadas reales
glVertex4f(1.0,0.5,3.4,5) ; // Cuatro coordenadas reales
GLfloat vector[4]={1.0,0.5,3.4,5}; // Vector de 4 reales
glVertex4fv(vector) ;
```

En todos los ejemplos glVertex estamos definiendo un punto, un vértice, pero con distintos datos.

Esquema básico de una aplicación gráfica. GLUT

Como hemos dicho anteriormente, toda aplicación que use OpenGL, independientemente del sistema operativo, el lenguaje de programación y la librería de gestión de usuario utilizada, necesita de unos pasos básicos o una funcionalidad que debe existir, definiéndose de una forma u otra en función de los requisitos del lenguaje.




```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdio.h>
void dibuja() {
..
}
int main(&argc, argv)
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
      glutInitWindowPosition(50,50);
      glutInitWindowSize(800,600);
      glutCreateWindow("Ejemplo");
      glutDisplayFunc(dibuja());
      glutReshapeFunc(dibuja());
      glutMainLoop();
      return(true);
}
```

Código 1: Esquema básico de un programa en GLUT

La forma en que se establece qué función se encarga de cada cosa es específica de cada librería de interfaz de usuario utilizada. Un ejemplo básico, utilizando GLUT/freeGLUT es el que se ve en el Código 1:

- `glutInitDisplayMode` inicializa OpenGL
- `glutInitWindowPosition`, glutInitWindowSize y glutCreateWindow inicializan la ventana.
- `glutDisplayFunc` establece la función gestora del evento de redibujado

- `glutReshapeFunc` establece la función gestora del evento de redimensionado de ventana.
- `glutMainLoop` echa a andar la función de dibujado anteriormente indicada en un bucle infinito.

En FreeGLUT se usan lo que se denominan *callbacks*, que requieren de punteros a las funciones C que se encargan de la gestión, por ejemplo:

```
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
void glutDisplayFunc(void (*func)(void));
```

EJERCICIOS

16. *¿Cómo se realizan las mismas operaciones descritas en el* Código 1 *pero usando Qt en lugar de FreeGLUT?*
17. *Dado que FreeGLUT es C, y el paradigma imperante hoy en día es la orientación a objetos. ¿Cómo se puede hacer para usar objetos y a la vez los callback de FreeGLUT?*

En https://www.opengl.org/documentation/specs/glut/spec3/spec3.html puede encontrarse toda la especificación de GLUT, paralizada desde 1998. FreeGLUT tampoco es que haya evolucionado mucho más en cuanto a funcionalidad, pero su última versión liberada es de 2013 (http://freeglut.sourceforge.net/)

En las sesiones de prácticas presentaremos FreeGLUT con más detalle, y veremos cada uno de los callbacks que vamos a usar en la asignatura, pero lo interesante es el bucle de gestión de eventos, el *mainLoop* que tiene GLUT:

- GLUT mantiene una cola de eventos con información de cada evento que ha ocurrido pero no ha sido gestionado por la aplicación
- En cada iteración del bucle:
  - Si la cola está vacía,
    - se inserta el evento *idle* (desocupado)
  - Si no,
    - se extrae el siguiente evento y, si hay función designada para su gestión, se ejecuta esta función
  - El bucle termina cuando alguna función gestora cierra la aplicación.

Construcción de primitivas en OpenGL

Si usted ha cursado Química, sabrá que toda sustancia química, toda molécula, se compone de átomos, y éstos a su vez de elementos más pequeños aún. Da igual que tengamos ácido sulfúrico que agua, que ambos están formados de protones, neutrones, electrones, y demás partículas subatómicas.

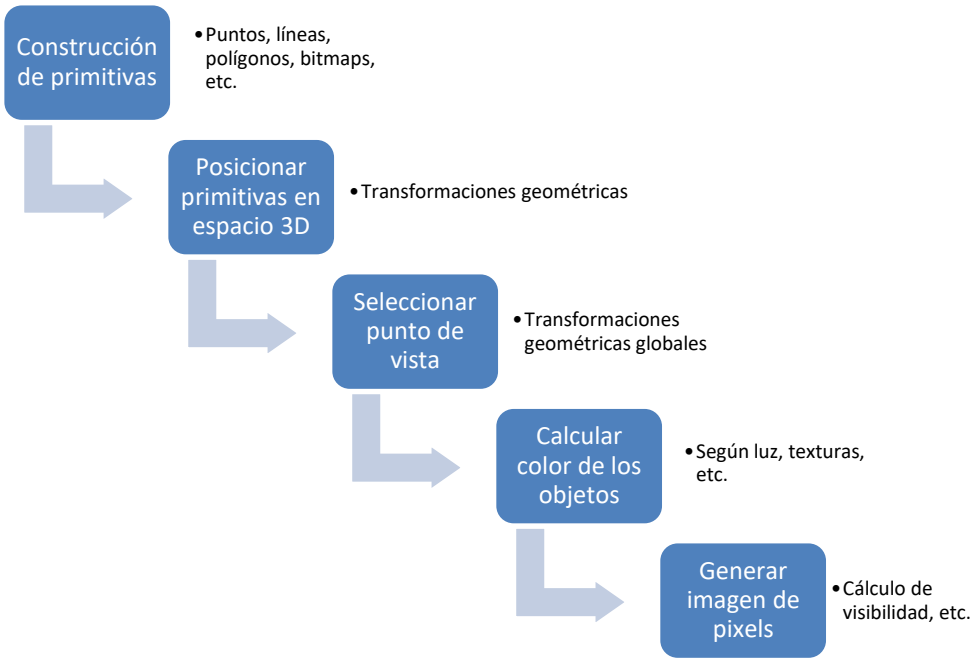
Pues bien, en los gráficos por ordenador ocurre igual. Da igual la escena que tengamos, que por muy compleja que sea, en el fondo no son más que puntos, y la forma en que éstos se unen y organizan dan lugar a una escena u otra. En OpenGL, estas "partículas subatómicas" se denominan primitivas: puntos, líneas, triángulos o polígonos. En todos los casos, cualquier primitiva no es más que un conjunto de vértices organizados de una forma muy concreta. Un punto es una primitiva con un único vértice. Una línea está formada por dos vértices, y un triángulo por tres.

Las aplicaciones normalmente descomponen las superficies complejas en un gran número de triángulos, y éstos son enviados a OpenGL donde son rasterizados. ¿Por qué triángulos? Pues porque son relativamente fáciles de dibujar, ya que son siempre convexos. Si el modelo 3D tiene partes o

qt/glut crea el canvas, la ventana, pintar, redimensionar

polígonos cóncavos, se descomponen en triángulos. Además, el hardware gráfico está optimizado para el dibujado de triángulos

Antes de ver en detalle las primitivas, recordemos qué ocurre en la función de dibujado. Se puede ver el proceso de generación de una imagen por rasterización anteriormente comentado como los siguientes pasos:



Por tanto, vamos a ir presentando algunas funciones de OpenGL en el contexto de las operaciones indicadas en esta figura.

Primitivas en OpenGL < 2.0

En las primeras versiones de OpenGL, las primitivas se describían usando una secuencia de vértices, que como hemos visto anteriormente se pueden definir con las funciones `glVertex`, en el caso de OpenGL < 2.0 y, como veremos, usando arrays o buffers en OpenGL>=2.0. Un vértice no es más que un punto en un sistema de coordenadas, normalmente en el cartesiano.

El mecanismo, independientemente de la primitiva que queramos dibujar, es el siguiente:

```
glBegin(PRIMITIVA);
  glVertex3f(x,y,z);
  …
  glVertex3f(x,y,z);
glEnd();
```

En función de la PRIMITIVA, los vértices incluidos en ella se interpretan o agrupan de una forma u otra.

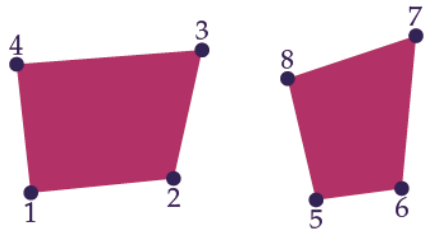
A partir de OpenGL 2.0, el concepto de primitiva se mantiene, pero veremos que en lugar de dar los vértices uno a uno, se almacenan en un array y se mandan en bloque a los *shaders*.

Pasamos a ver los tipos de primitivas, usando para su explicación la forma en que se definían en OpenGL 1.0 (**OJO: no se puede usar en la asignatura esta forma de definir las primitivas**).

Cuadriláteros

La constante GL_QUADS ordena a OpenGL que agrupe los vértices de cuatro en cuatro, y dibuje cuadriláteros siguiendo el sentido **antihorario**.

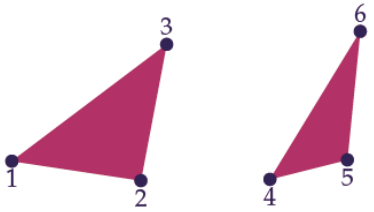
```
glBegin( GL_QUADS ) ;
/ / primer cuadrilátero
    glVertex3f( x1 , y1 , z1 ) ;
    glVertex3f( x2 , y2 , z2 ) ;
    glVertex3f( x3 , y3 , z3 ) ;
    glVertex3f( x4 , y4 , z4 ) ;
    / / segundo cuadrilátero
    glVertex3f( x5 , y5 , z5 ) ;
    glVertex3f( x6 , y6 , z6 ) ;
    glVertex3f( x7 , y7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



Triángulos

La constante GL_TRIANGLES ordena a OpenGL que agrupe los vértices de tres en tres, y dibuje triángulos siguiendo el sentido **antihorario**.

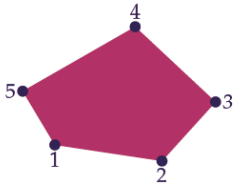
```
glBegin( GL_TRIANGLES ) ;
        / / primer triángulo
        glVertex3f( x1 , y1 , z1 ) ;
        glVertex3f( x2 , y2 , z2 ) ;
        glVertex3f( x3 , y3 , z3 ) ;
        / / segundo triángulo
        glVertex3f( x4 , y4 , z4 ) ;
        glVertex3f( x5 , y5 , z5 ) ;
        glVertex3f( x6 , y6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



Polígonos

La constante GL_POLYGON ordena a OpenGL que use todos los vértices para generar un polígono, siguiendo el sentido **antihorario**. Si los vértices no son coplanares, el resultado es incierto.

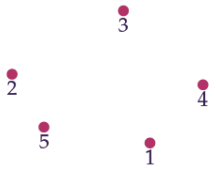
```
glBegin( GL_POLYGON ) ;
        glVertex3f( x1 , y1 , z1 ) ;
        glVertex3f( x2 , y2 , z2 ) ;
        glVertex3f( x3 , y3 , z3 ) ;
        …
        glVertex3f( xn-1 , yn-1 , zn-1 ) ;
        glVertex3f( xn , yn , zn ) ;
glEnd()
```



Puntos aislados

La constante GL_POINTS ordena a OpenGL que use todos los vértices para generar un punto para cada vértice. En este caso, al no ser una secuencia, el orden da igual.

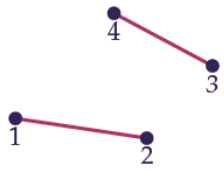
```
glBegin( GL_POINTS ) ;
        glVertex3f( x1 , y1 , z1 ) ;
        glVertex3f( x2 , y2 , z2 ) ;
        …
        glVertex3f( xn , yn , zn ) ;
glEnd()
```



Segmentos

La constante GL_LINES ordena a OpenGL que use los vértices de dos en dos para generar un segmento por cada par.

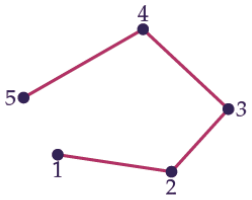
```
glBegin( GL_LINES ) ;
        / / primer segmento:
        glVertex3f( x1 , y1 , z1 );
        glVertex3f( x2 , y2 , z2 );
        / / segundo segmento:
        glVertex3f( x3 , y3 , z3 );
        glVertex3f( x4 , y4 , z4 );
glEnd()
```



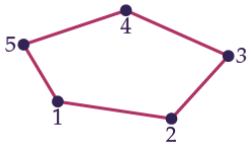
Polilíneas

Dada una secuencia de vértices, podemos generar una polilínea abierta o cerrada, en función de si usamos GL_LINE_STRIP o GL_LINE_LOOP. En este último caso, hay que distinguir entre el GL_POLYGON y el GL_LINE_LOOP. Aunque pueda parecer lo mismo, **no lo es**. La polilínea cerrada no es más que los "alambres" que unen los puntos.

```
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() ;
```



Primitivas en OpenGL >= 2.0

A partir de OpenGL, el estándar propone el uso de buffers para, en lugar de hacer miles de llamadas a `glVertex`, realizar una única llamada a `glDrawArrays`() o bien de `glDrawElements`().

Con `glDrawArrays` la idea que subyace es tener un array con las coordenadas de los vértices, y en el momento de dibujar, pasar todos en bloque a la tarjeta gráfica. Si hay más propiedades, como la normal, el color o la coordenada de textura, se almacenan en otros buffer, y se activan para ser tenidos en cuenta.

`void glDrawArrays(GLenum mode, GLint first, GLsizei count);`

PARÁMETROS

`mode`. Especifica cómo interpreter la secuencia de datos (GL_POINTS, GL_LINE_STRIP,GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY o GL_PATCHES)

`first` Especifica el índice inicial de los array habilitados

`count` Especifica el número de elementos del array que se van a usar

`glDrawArrays` permite dibujar diferentes tipos de primitivas geométricas con una única llamada. En lugar de llamar a una función OpenGL para cada propiedad (un `glVertex` para pasar cada vértice individual, `glNormal` para cada normal, `glTextureCoord` para cada coordenada de textura, `glColor` para cada color de vértice, etc.), se puede preparar un array para cada una de estas propiedades y visualizar todo con una única llamada a `glDrawArrays`.

Cuando se llama a `glDrawArrays`, se usan **count** elementos secuencialmente de cada uno de los arrays habilitados, comenzando por el elemento **first**.

Ahora bien, ¿cómo se especifica cual es el array de datos donde están los vértices? Vamos a ver cómo se definen los arrays.

`void glVertexPointer (GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);`

`size` Número de coordenadas por vértice: 2, 3 o 4.

`type` Tipo de cada coordenada del array (GL_SHORT, GL_INT, GL_FLOAT, o GL_DOUBLE)

`stride` Offset en bytes entre dos vertices consecutivos. Si es 0, se entiende que los vértices están consecutivos.

`pointer` Puntero a la primera coordenada del primer vértice del array.

Con estas funciones, se puede realizar el dibujado de un triángulo indicando a OpenGL en qué vector están los vértices, y que los tome como triángulos (de 3 en 3, en sentido antihorario):

Sin embargo, OpenGL nos permite utilizar otra función, llamada `glDrawElements` para no tener que repetir coordenadas de vértices en objetos con múltiples triángulos, mediante el uso de un buffer de vértices y otro de índices que nos definirá las caras. Lo veremos en el próximo tema cuando veamos las distintas formas de representar los objetos 3D.

En el guion de la práctica 1, además de este modo **inmediato**, explicaremos el modo **diferido** del uso de buffers, que será necesario implementar también en prácticas.

```
float vertices[] = {   -1.0f, 1.0f, 0.0f,
                        1.0f, 1.0f, 0.0f,
                        0.0f, 0.0f, 0.0f,
                       };
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

La función de redibujado

La llamada a las instrucciones de definición de las primitivas puede hacerse única y exclusivamente desde la función de redibujado (o cualquiera de las invocadas desde ésta). Además, no se puede llamar a dicha función sin haber inicializado la máquina de estados de OpenGL.

Un ejemplo sencillo, en C y con GLUT, de función de redibujado es:

```
void draw () {
// limpiar la ventana
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
// envio de primitivas
glEnableClientState(GL_VERTEX_ARRAY);
// designación del vector de vértices
glVertexPointer(3, GL_FLOAT, 0, vertices);
//  dibujado de triángulos con el vector de vértices activos
glDrawArrays(GL_TRIANGLES, 0, 3);.....
// Intercambio de buffers
glutSwapBuffers() ;
}
```

En otras librerías de GUI, el esquema sería similar:

- limpiar la ventana (con `glClear`)
- enviar las primitivas
- intercambiar los buffers

¿Qué es eso de intercambiar los buffers? Con OpenGL se puede utilizar, y de hecho se utiliza, un sistema de doble buffer para el dibujado, de forma que la rasterización se realiza en una imagen que no se ve, está oculta, hasta que no ejecutamos el intercambio de buffers. De esta forma no se ve cómo avanza el barrido del algoritmo raster (que va de arriba debajo de izquierda a derecha), y especialmente en las escenas animadas esto es muy importante. Se puede usar un único buffer, pero no es recomendable.

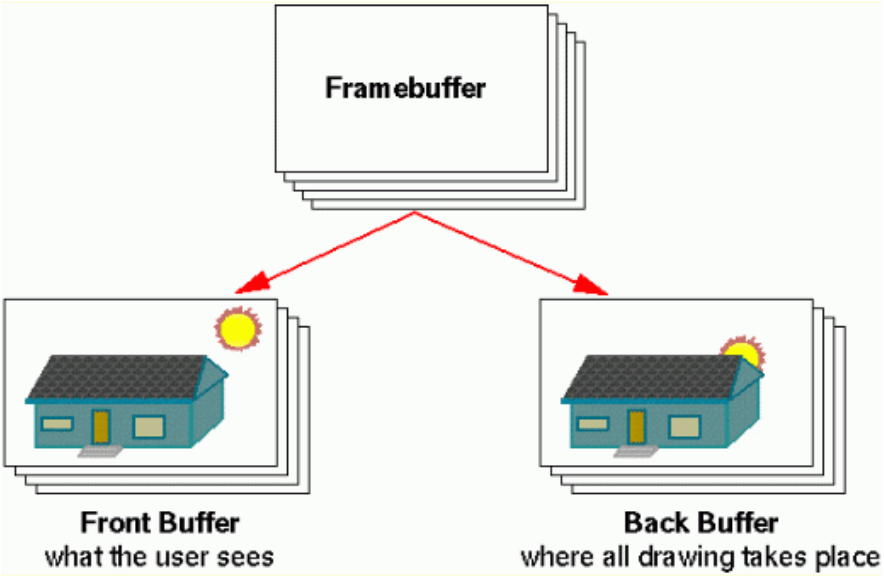


Ilustración 18 Mecanismo de Doble Buffer

Atributos de las primitivas en OpenGL

OpenGL almacena, dentro de las múltiples variables de su máquina de estados, varios atributos que se usan para la visualización de las primitivas anteriormente descritas. Los datos asociados a la geometría y la topología se almacenan en buffers. Por ejemplo:

- **color.** Con la instrucción glColorPointer le indicamos a OpenGL dónde se encuentra el buffer que almacena los valores de color, en RGB o RGBA, a utilizar por cada uno de los vértices.

```
void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);
```

- **ancho** (en pixeles) **de las líneas.**

```
glLineWidth( )
```

- **ancho** (en pixeles) **de los puntos.**

```
glPointSize()
```

- **modo de dibujado raster de polígonos**. Se indica que los polígonos (primitivas GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_POLYGON, GL_QUADS, GL_QUADSTRIP) se han de dibujar de un modo concreto por la cara indicada

```
glPolygonMode(cara, modo)
```

```
glPolygonMode(GL_FRONT, GL_POINT) // Cara delantera solo puntos
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE) //Ambas caras en alambres
glPolygonMode(GL_BACK, GL_FILL) // Cara trasera rellena
```

Es importante destacar que no es lo mismo dibujar líneas que dibujar **en modo líneas**. Para lo primero, se usa la primitiva GL_LINES en glDrawElements. Para dibujar en modo líneas hay que usar `glPolygonMode`.

Mientras no se indique un modo contrario, la máquina de estados que es OpenGL seguirá con las mismas condiciones para todas las primitivas poligonales.

EJERCICIOS

18. *Escriba la función de dibujado para pintar un pentágono regular con los vértices en verde, las líneas rojas y relleno azul.*
19. *¿Qué cara es la GL_FRONT de un polígono? ¿Es aleatorio?*

También es un atributo, aunque de la escena, el color con el que se limpia o borra el framebuffer:

```
glClearColor(R,G,B,A);
```

Normalmente, el valor de color de borrado del framebuffer se define en la función de inicialización de OpenGL y no se vuelve a cambiar, pero ello no quiere decir que no se pueda.

EJERCICIOS

20. ***Documente en sus apuntes*** *cómo se codifican los colores en formato RGB entero y real. ¿Qué formato ofrece una mayor riqueza de colores?*
21. *¿Es correcto un color RGBA(423,123,121,0)?* **no**
22. *¿Es correcto un color RGBA(2.3,0.123,0.121,1.0)?* **si**

El viewport

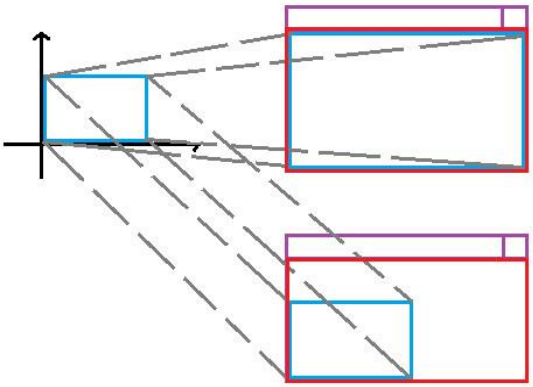


Ilustración 19: Una imagen puede ser representada en dos viewport de distinto tamaño en

La función `glViewport` permite establecer que parte de la ventana será usada para visualizar (Ilustración 19). Dicha parte (llamada viewport) es un bloque rectangular de pixels.

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

Es importante que las dimensiones del viewport sean **proporcionales** a las de la proyección 2D de la escena, pues de lo contrario, nos encontraremos con que la imagen sale distorsionada, como se muestra en la Ilustración 20.

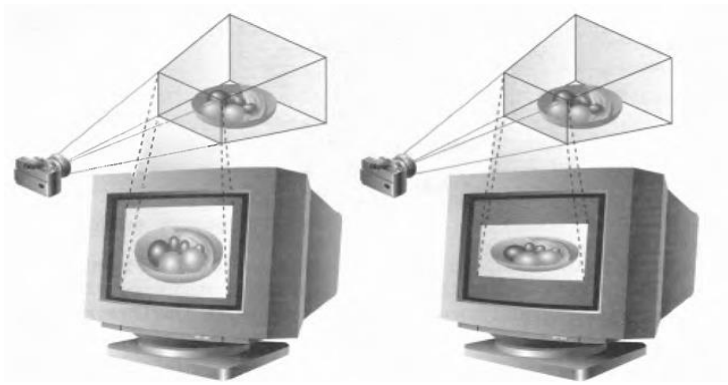


Ilustración 20: Izda. Viewport proporcional a la imagen proyectada (imagen sin distorsión)

Dcha. Viewport que genera una imagen distorsionada por tener proporciones distintas.

Por defecto, OpenGL utiliza una cámara situada en el eje Z, mirando hacia el sentido negativo de la Z, como se muestra en la Ilustración 21. El volumen de visualización, es decir, lo que se ve por defecto, es un cubo de lado 2 con el centro en el origen de coordenadas.

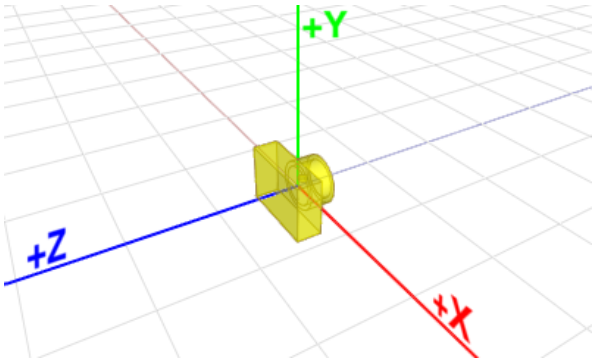


Ilustración 21 La cámara de OpenGL

Cuando se realiza un cambio de tamaño de la ventana, **siempre es necesario redimensionar el viewport** para adaptarlo al nuevo tamaño de la ventana, si queremos que siga conservando el 100% del área, por ejemplo. Además, habrá que cambiar el volumen de visualización, pues si tenemos una proyección cuadrada y la ventana es rectangular, se producirá una distorsión en la imagen.

Es lo que se llama una transformación de coordenadas de vista a coordenadas de dispositivo.

Por tanto, cuando se produzca el evento de redimensionado, podemos llamar a la función siguiente con las nuevas dimensiones de la ventana:

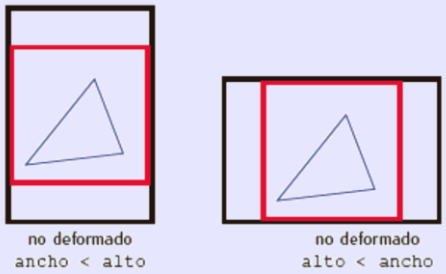
```
void FGE_CambioTamano( int nuevoAncho, int nuevoAlto )  {
        glViewport(0,0,nuevoAncho,nuevoAlto);
}
```

EJERCICIOS

23. *Escribe una nueva versión de FGE_CambioTamano para que el viewport sea siempre cuadrado, ocupando el cuadrado más grande posible dentro de la ventana, y centrado con respecto a esta.*

24. *La función glClear ¿limpia sólo el viewport o todo el framebuffer?*

25. *Escribe una función de dibujado que muestre el viewport cuadrado del ejercicio 25 en blanco y el resto de la ventana en gris.*



BIBLIOGRAFÍA

[Möller18] **Real-time rendering** (Fourth edition.). Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., & Hillaire, S. (2018). CRC Press, Taylor & Francis Group.

[Foley97] **Computer Graphics: Principles and Practice in C**; Foley, Van Dam, Feiner y Hughes. de. Addison-Wesley , 1997.

[Shirley09] **Fundamentals of Computer Graphics;** P. Shirley. AK Peters, 2009

[Angel08] **Interactive Computer Graphics: A Top Down Approach** (5ª Ed); E. Angel. Addison Wesley, 2008

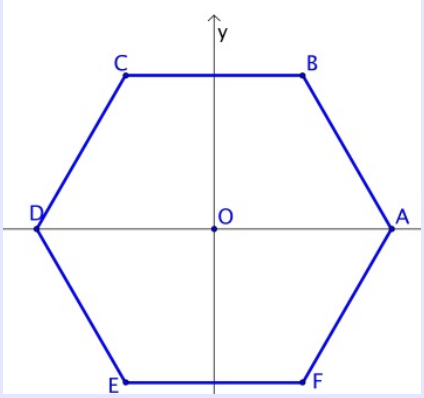
[Hearn10] **Computer Graphics with Open GL** (4ª Ed) ; D. Hearn, P. Baker, W. Carithers; Prentice Hall, 2010

Recursos online (**un gran apoyo** para la asignatura):

- https://www.ugr.es/~demiras/cgex/#home
- https://open.gl/
- http://www.xmission.com/~nate/tutors.html
- http://graphics.stanford.edu/courses/
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-837-computer-graphics-fall-2003/
- http://www.student.cs.uwaterloo.ca/~cs488/
- http://www.opengl-tutorial.org/
- https://content.byui.edu/file/2315e65e-a34a-48d3-814d-4175a2b74ed5/1/intro/165-opengl-visualStudio2017.html

EJERCICIOS PREPARATORIOS PARA EL SEMINARIO DE GEOMETRÍA

I. *Vea este video: http://www.youtube.com/watch?v=vSDDgL6Cq6g*
II. *Escriba el código en C para generar los seis vértices de un hexágono regular centrado en el origen de coordenadas. El vértice A estará en la posición (2,0,0). Los valores de los vértices deberán ser generados con automáticamente, utilizando trigonometría.*



III. *Escribir el código en C que calcule el ángulo entre los segmentos AB y CB*
IV. *Escribir el código en C que calcule la longitud del segmento DC*
V. *Si consideramos el origen de coordenadas el punto C, ¿qué valores coordenados tendrán los otros cinco puntos?*
VI. *Calcule el producto vectorial de ABxAF y AFxAB. Dibújelo el resultado en 3D.*