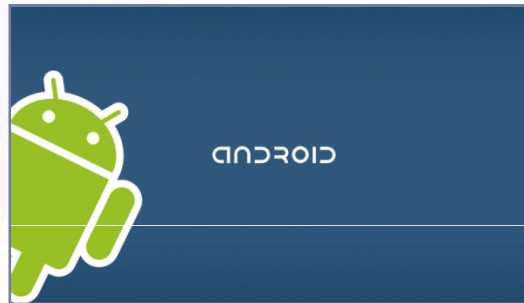




OpenGL ES con Android



✓ ¿Qué es?

- ✓ Variante simplificada de la API OpenGL para dispositivos integrados
- ✓ La promueve y define el grupo Khronos, consorcio de empresas dedicadas a hardware y software gráfico
- ✓ OpenGL es una API multilenguaje y multiplataforma para realizar gráficos 2D y 3D.
- ✓ Permite desarrollar escenas a partir de primitivas gráficas (puntos, líneas, triángulos...)



Utilidad:

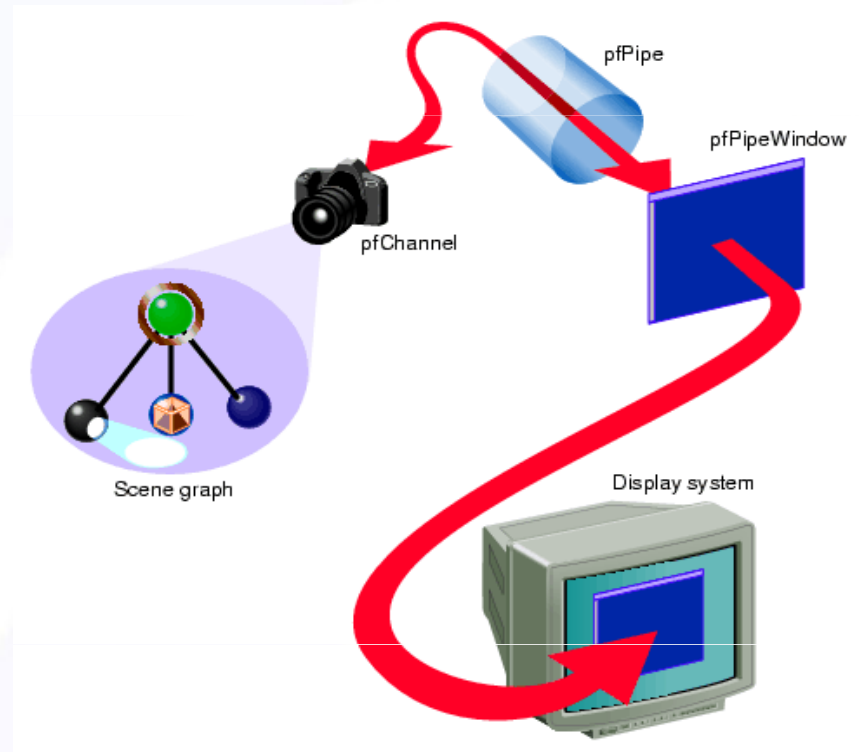
- *Uso de OpenGL ES:*
 - API para gráficos 3D en Symbian OS y Android
 - OpenGL ES 2.0 para Nokia N900 con SO Maemo (basado Linux)
 - SDK de Iphone
 - PlayStation 3

symbian OS

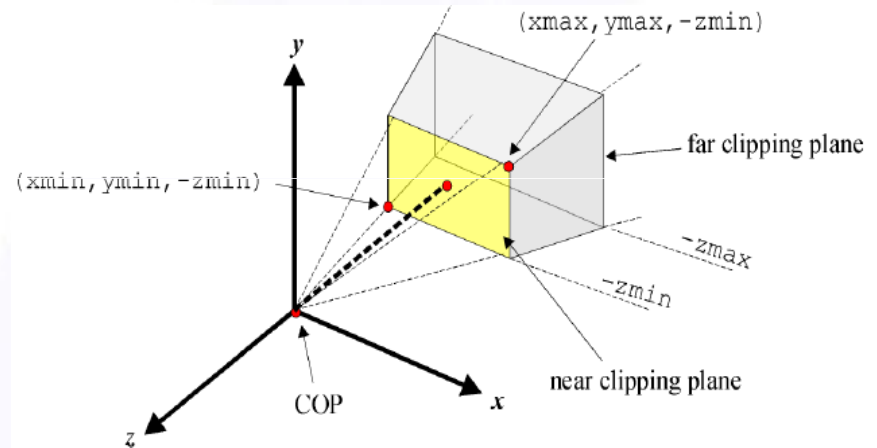
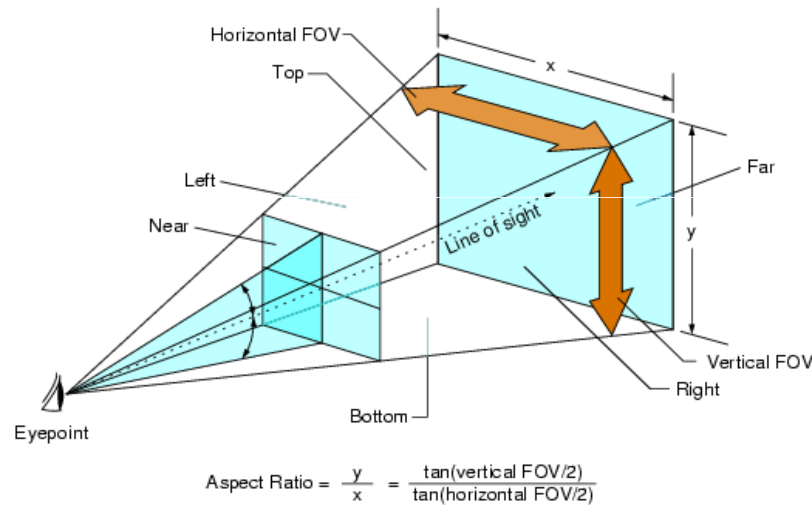


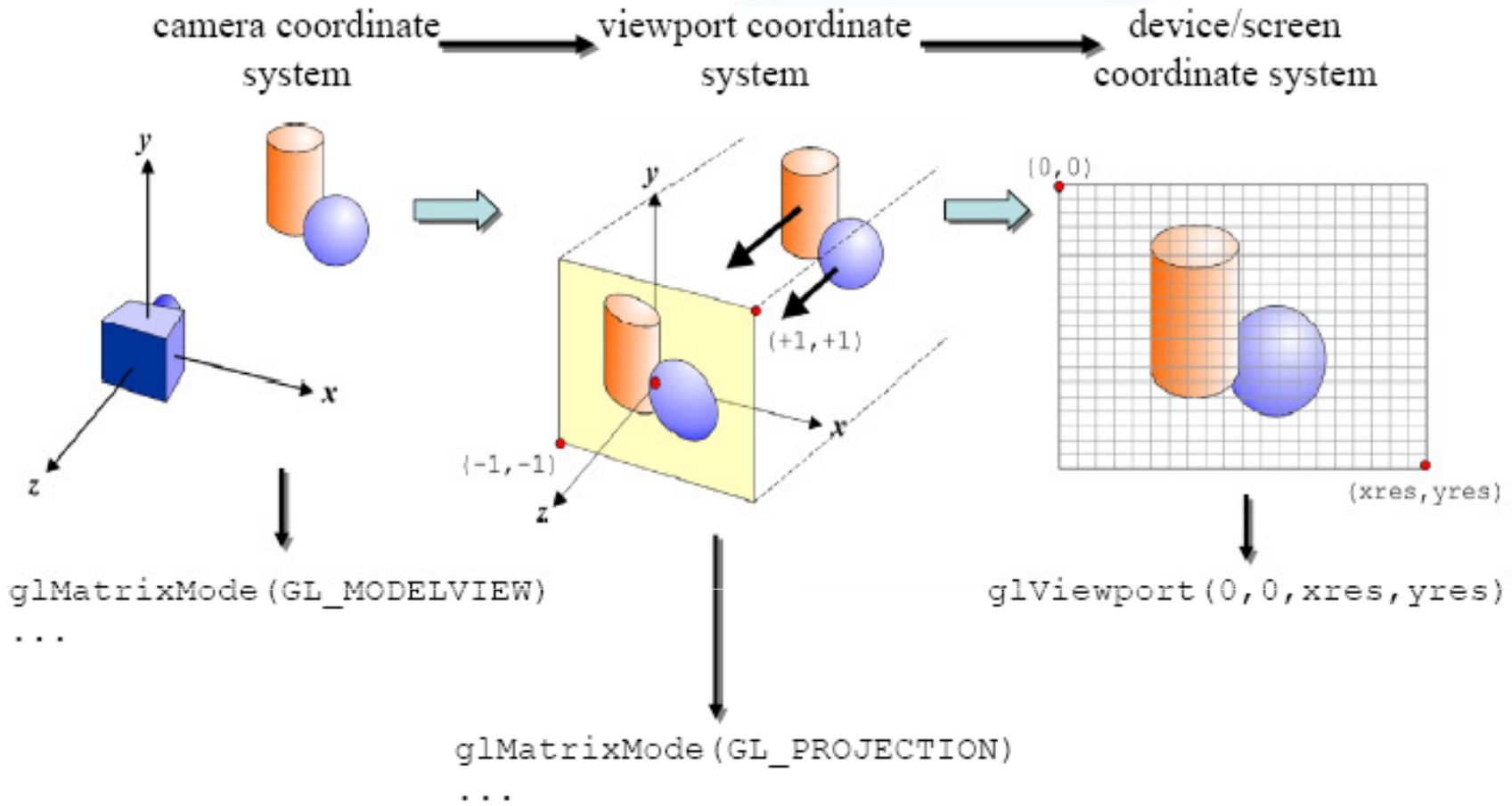
➤ Modelo OpenGL

- Antes de visualizar objetos por pantallas se debe realizar un modelado.
- Los objetos del mundo real se pueden trasladar al mundo virtual utilizando vértices, caras, aristas...
- El modelado no es trivial: cómo se modela una esfera?
- OpenGL utiliza su propio modelo 3D y su pipeline para visualizar los objetos por pantalla.
- Los vértices, caras y demás tienen una posición conocida en el espacio 3-dimensional, ahora falta saber cómo se visualizan los objetos.



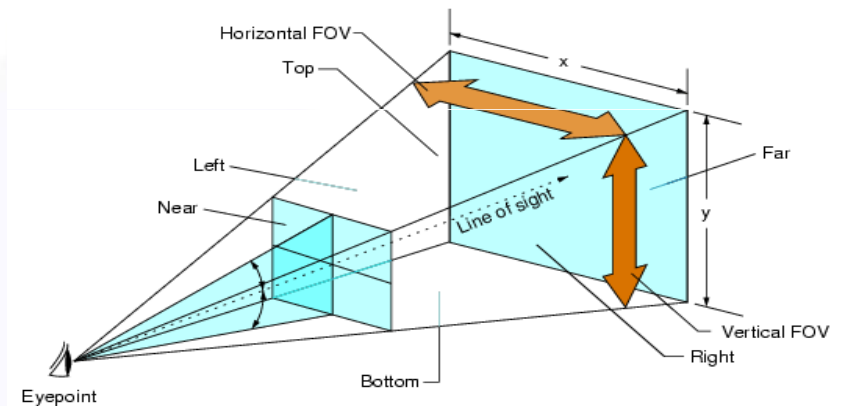
- *Visualización de un objeto: la cámara*
 - Tenemos almacenado un objeto en 3D en nuestro modelo: ¿qué se necesita saber para visualizarlo?
 - ¿Desde donde lo observamos.?
 - ¿Con qué orientación lo observamos?
 - ¿A qué distancia estamos del objeto?
 - ¿Cual es nuestro ángulo de apertura?
 - En OpenGL estas preguntas se traducen utilizando la pirámide de visión (frustrum)





➤ Visualización de un objeto

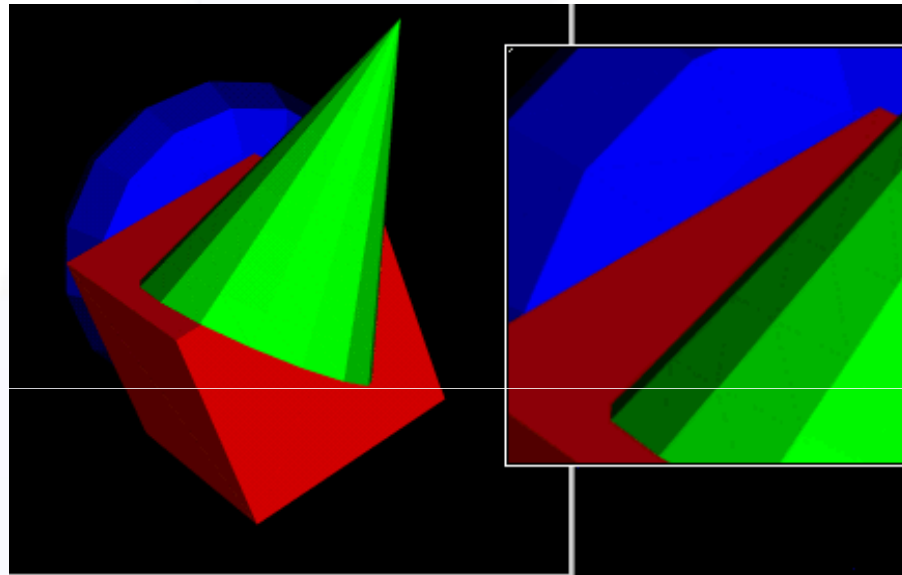
- Ajustando los parámetros de la pirámide de visión se pueden simular muchos de los aspectos de una cámara real.
- Parámetros relevantes:
 - Aspect ratio: relación ancho / alto
 - Ángulo de altura (FOV): ángulo de apertura superior de la cámara
 - Znear y Zfar: planos respecto a la cámara que recortan la escena.
 - Posición de la cámara: ¿en que posición del SC global está?
 - Up (view-up vector): vector orientación de la cámara



$$\text{Aspect Ratio} = \frac{y}{x} = \frac{\tan(\text{vertical FOV}/2)}{\tan(\text{horizontal FOV}/2)}$$

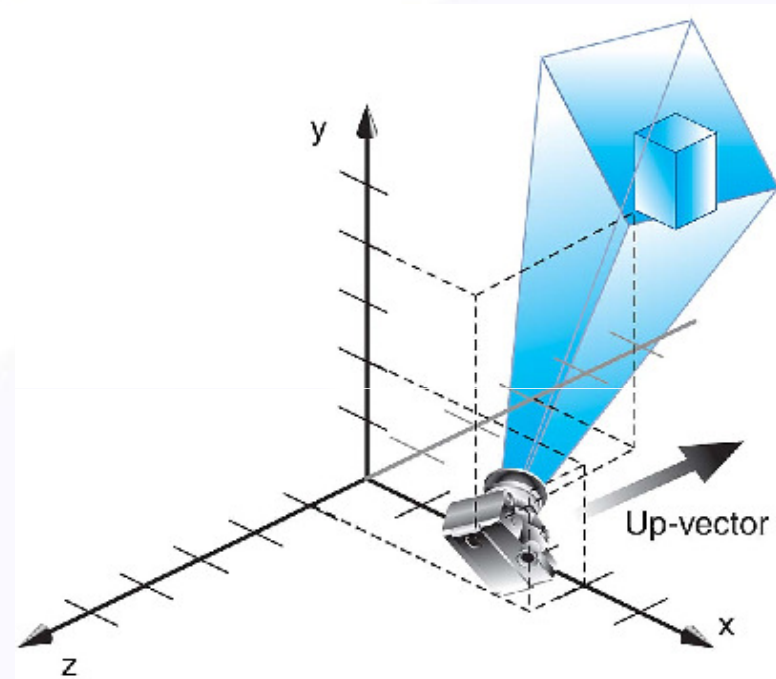
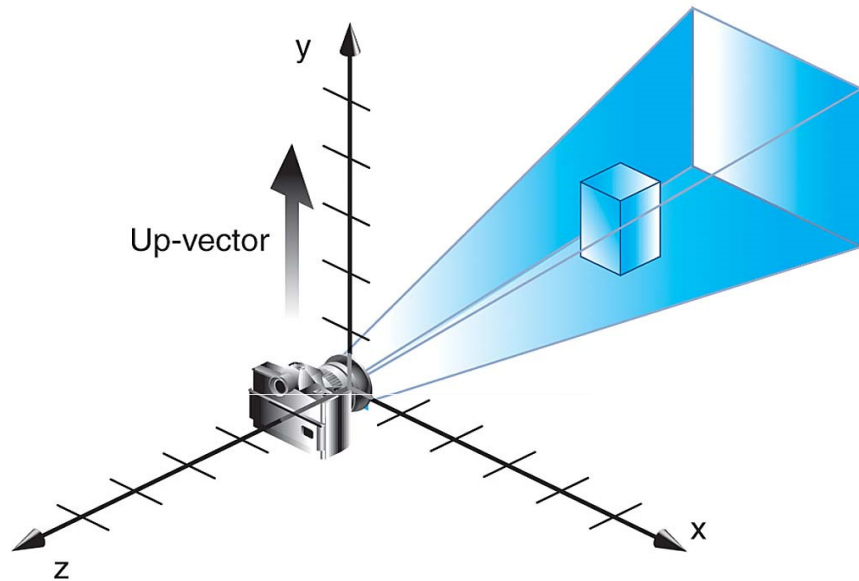
➤ *Ángulo de visión*

- Cambiar el ángulo de visión afecta a cómo vemos la escena.
- Un ángulo de visión mayor implica ver los objetos más pequeños
- En OpenGL daremos el fovy (ángulo superior) y el aspect ratio. Con el AR OpenGL sabe cómo calcular el fovx.



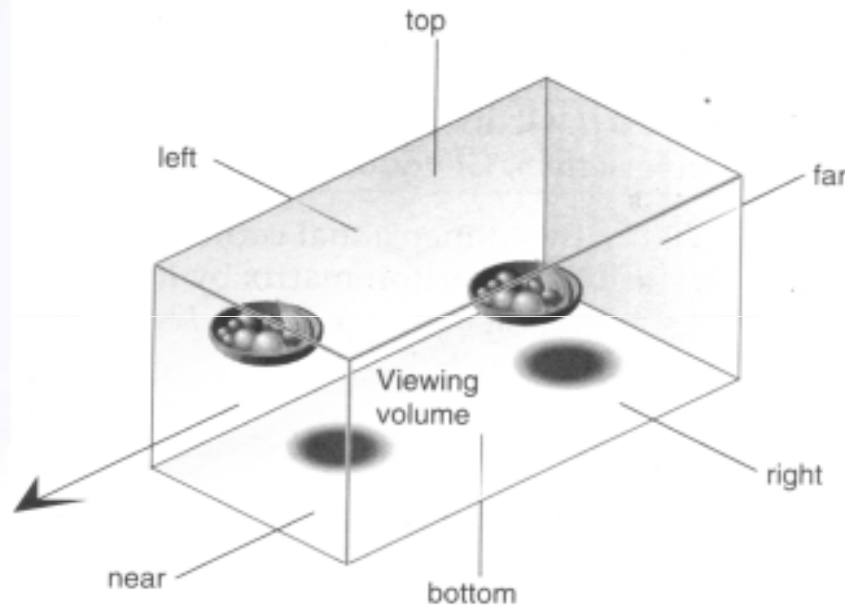
➤ *View-up vector*

- Define la orientación de la cámara
- Se da en tres coordenadas: (x,y,z)
- Es el vector orientación: $(0,1,0)$ indica que la cámara está orientada como en la imagen 1
- Un $(1,0,0)$ indicaría que la cámara está girada 90 grados



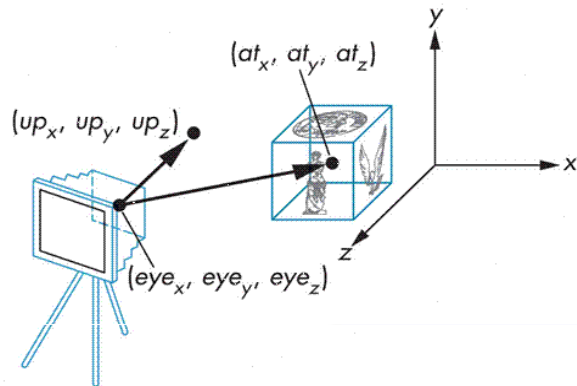
➤ Planos de corte

- En OpenGL la pirámide de visión no es perfecta: se le pueden aplicar dos planos de corte.
- La escena captada será la que esté DENTRO de esos dos planos de corte
- Se representan como Znear y Zfar, con dos número que representan la distancia a la cámara.
- Lo que quede fuera de ese Volumen de Visión no aparecerá.

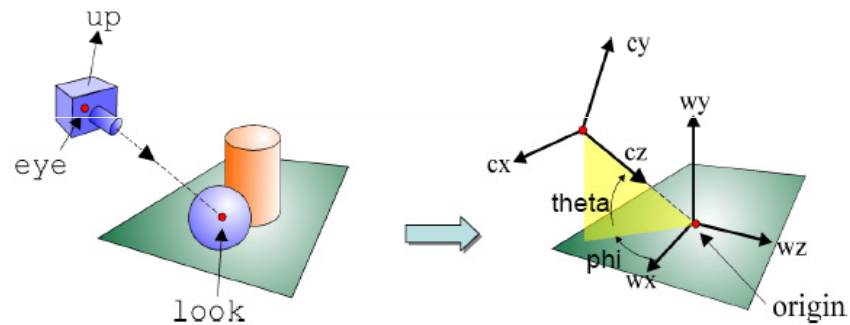


➤ Definición de la cámara

- Para crear nuestra cámara perspectiva, utilizaremos:
`gluPerspective(fvy, aspectratio, znear, zfar);`
- Para decirle a OpenGL donde está nuestra cámara y a donde mirar, se utilizará:
`gluLookAt(camx, camy, camz, //posición de la cámara
lookx, looky, lookz, //a donde apunta la cámara
upx, upy, upz); // orientación de la cámara (view-up vector)`
- No confundir el SC absoluto con el SC de la cámara

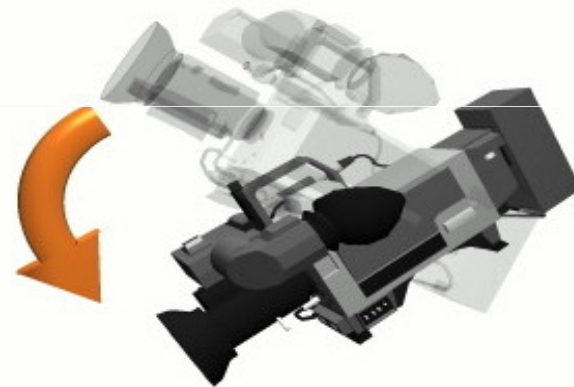


```
gluLookAt(eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz);
```



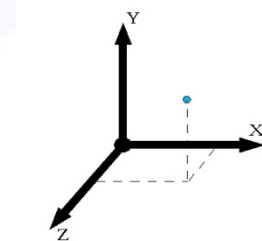
➤ *Movimiento de la cámara*

- Para cambiar la ubicación y orientación de la cámara se aplican transformaciones geométricas.
- Virtualmente, desde el punto de vista de la imagen, es lo mismo acercar la cámara a un objeto que acercar el objeto a la cámara.
- Implicaciones: las transformaciones se realizarán a nivel de modelo, es decir, la cámara se puede interpretar como algo fijo y lo que se mueve son los objetos.
- Al mover los objetos, todo lo que entre dentro del frustum de visión de la cámara aparecerá en la pantalla.
- La función `gluLookAt(..)` posiciona la cámara respecto al mundo.



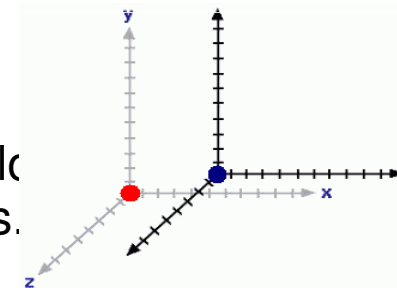
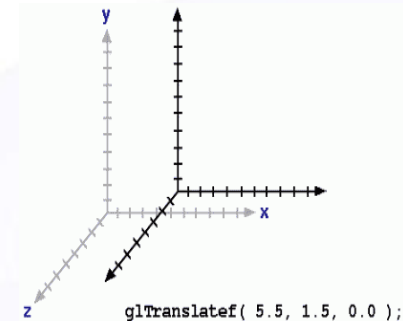
➤ Sistema de coordenadas

- Al iniciar OpenGL tiene el siguiente sistema de coordenadas:
- Esto quiere decir que a la hora de dibujar, OpenGL utilizará ese sistema de referencias (la posición 0,0,0 hace referencia al centro de ese sistema).



➤ Translación del sistema de coordenadas

- Podemos desplazar el sistema de referencias donde OpenGL tiene que dibujar: `glTranslate(x,y,z)`
- No es lo mismo dibujar en la posición 0,0,0 del sistema ANTES y DESPUÉS de moverlo. Ejemplo:
 - ✓ Dibujamos un círculo rojo en el SC Original (0,0,0)
 - ✓ Movemos el SC Original a (5,2,0) y dibujamos un círculo azul en (0,0,0): los puntos están en posiciones distintas.



➤ Rotación del sistema de coordenadas

- Similar a la idea anterior, podemos rotar el sistema de coordenadas respecto a uno de sus ejes:

`glRotatef(grados, x, y, z);` // 0 o 1 rota o no en el eje.

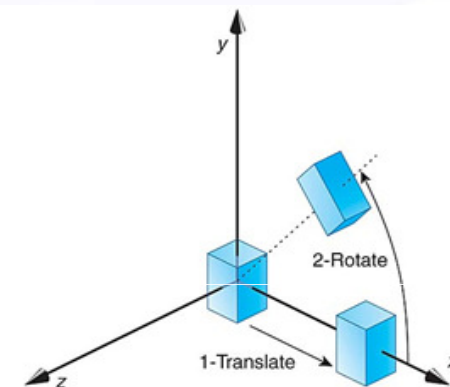
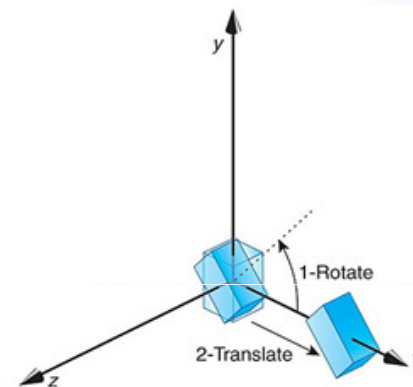
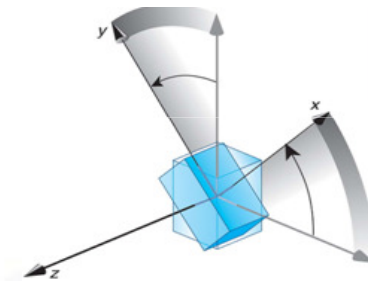
- Ojo!: no es lo mismo hacer:

1. Trasladar el SC
2. Dibujar un objeto
3. Rotar el SC

que

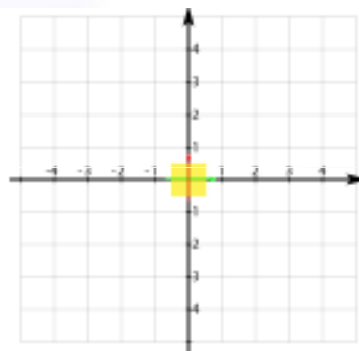
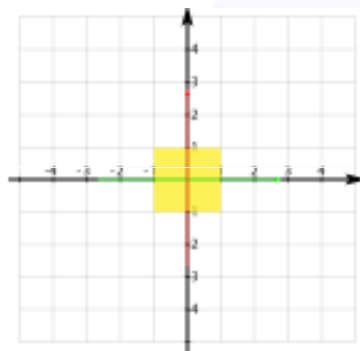
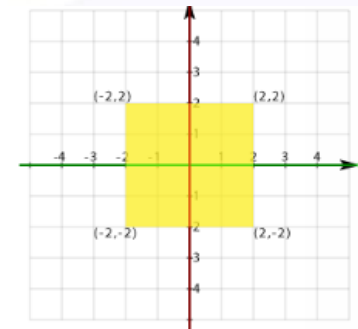
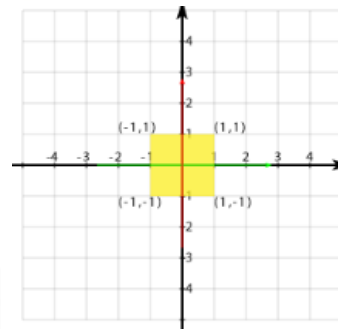
1. Rotar el SC
2. Dibujar un objeto
3. Trasladar el SC

El resultado es totalmente distinto!



➤ Escalado del sistema de coordenadas

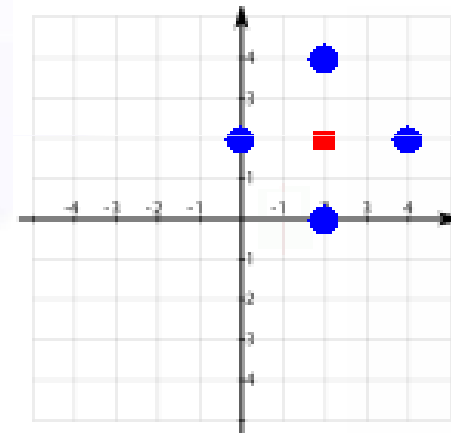
- Esta operación es la única que puede modificar la forma del objeto
- `glScalef(2f, 2f, 2f)`; multiplica por 2 las coordenadas del eje X, Y y Z: es decir, lo que antes en el eje X medía 1, ahora mide 2, lo mismo para el resto
- Se puede hacer más pequeño si multiplicamos por un número menor que 1: `glScale(0.5, 0.5, 0.5)`:



➤ *Salvar el estado del SC*

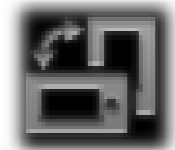
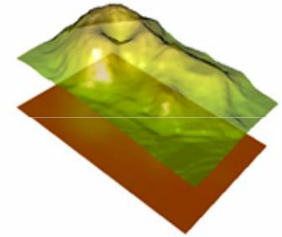
- A veces interesa no perder el estado del SC en un momento determinado.
- `glLoadIdentity()`: inicializa el SC
- `glPushMatrix()` apila el estado del SC
- `glPopMatrix` recupera el estado del SC
- Ejemplo de utilidad: dibujar un cuadrado en la posición (2,2) y cuatro esferas a una distancia de 2 unidades respecto al cuadrado, en las posiciones (4,2), (2,4), (0,2), (2,0):
- ```
glLoadIdentity(); //Inicializa el SC a su origen
translatef(2.0, 2.0); //Nuevo SC en 2,2
dibujarCuadrado(); //dibujamos cuadrado en 2,2
glPushMatrix(); //guardamos el SC
translatef(2.0, 0); //Movemos SC a 4,2
dibujarCirculo(); //círculo derecho
glPopMatrix(); //recuperamos la matrix
glPushMatrix(); //guardamos el SC de nuevo
translatef(0, 2.0) //Movemos SC a 2,4
dibujarCirculo(); //Circulo superior
```

...



### ➤ *Características:*

- Proporciona mecanismo para enlazar OpenGL con las Views y la Activity
- Facilita la inicialización de la parte gráfica
- Facilita herramientas de debug para controlar las llamadas a la API de OpenGL y localización de errores
- La interfaz GLSurface.Renderer se deberá implementar para la renderización de cada frame:
  - ✓ *onSurfaceCreated(...)*: inicialización de la superficie donde se dibujará. Aquí se deben incluir cosas que no cambien a menudo (limpieza de pantalla, z-buffer...)
  - ✓ *onDrawFrame(...)*: método que realiza el dibujo.
  - ✓ *onSurfaceChanged(...)*: se invoca al cambiar el dispositivo de landscape a portraite. Incluir aquí el nuevo *aspect ratio*.



### ➤ ¿Qué es?

- Interface que sirve de capa de abstracción entre JAVA y OpenGL
- Se utilizarán la mayoría de sus métodos estáticos para utilizar funciones OpenGL
- Constantes que utilizan los métodos también definidas aquí.
- Se pasará una instancia de este objeto a los métodos de la clase GLSurfaceView:
  - ✓ *onSurfaceCreated(GL10 gl, EGLConfig config)*
  - ✓ *onDrawFrame(GL10 gl)*
  - ✓ *onSurfaceChanged(GL10 gl...)*



- *Ubicación del ejemplo:*
  - Carpeta de workspace: /OpenGL/OpenGL-inicioRenderer
- *Ubicación del ejemplo:*
  - Carpeta de workspace: /OpenGL/OpenGL-inicio
- *Comportamiento:*
  - En este ejemplo se puede ver cómo se inicializa la GLSurfaceView, y se prepara para poder pintar en la pantalla. En el Renderer (que implementa GLSurfaceView.Renderer) se han sobrescrito los tres métodos principales para inicializar y redibujar información.
- *Resultado:*
  - En el primer ejemplo se muestra cómo cambiar de color la pantalla realizando eventos de touch.
  - En este ejemplo únicamente se visualiza una pantalla oscura, que indica que todo está listo para empezar a dibujar

## Ejemplo de transformaciones

### ➤ *Ubicación del ejemplo:*

- Carpeta de workspace: /OpenGL/OpenGL-transformaciones

### ➤ *Comportamiento:*

- Importante: interpretar las transformaciones en orden inverso!
- Primer cuadrado: se dibuja y se gira N grados.
- Segundo cuadrado: se dibuja, se escala a la mitad, se mueve 2 unidades en el eje X, y se rota N grados respecto al eje Z.
- Tercer cuadrado: en este caso NO se hace pop de la matriz, por lo que el SC sigue siendo el del segundo cuadrado. Se dibuja en el mismo origen que el segundo, se gira  $N \cdot 10$  grados sobre Z, se escala a la mitad (una cuarta parte del primero).

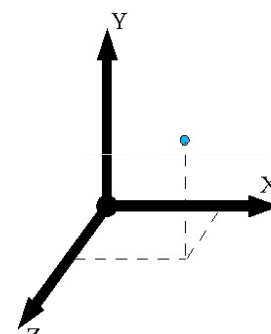
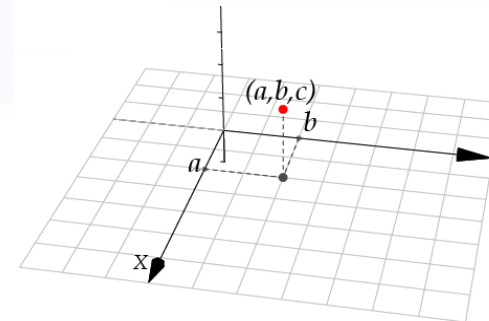
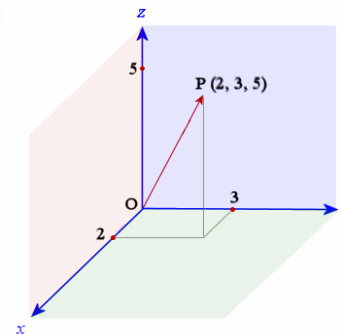
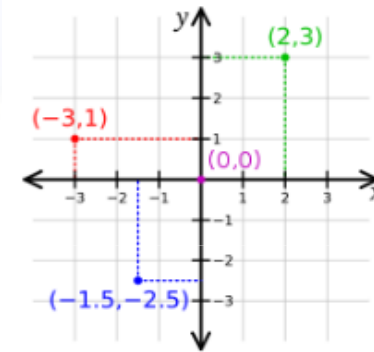
### ➤ *Resultado:*

- En este ejemplo se visualiza una pantalla con 3 cuadrados rotando respecto a distintos ejes de coordenadas.

## ➤ *Vértice (Vertex):*

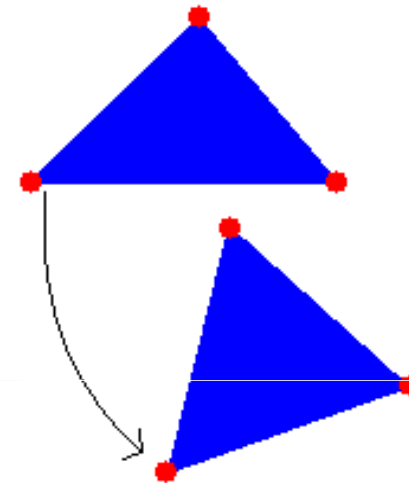
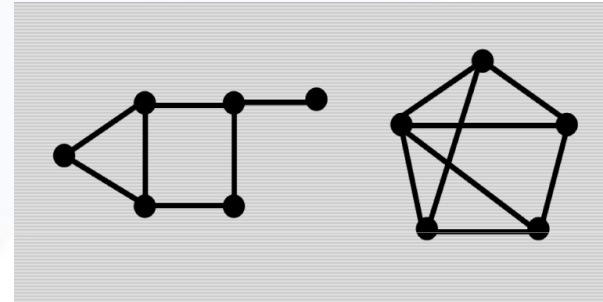
- Unidad básica para dibujado. Representa una posición en el espacio y es donde convergen dos aristas o más.
- Se define usando dos coordenadas (X,Y) en el espacio 2-dimensional y tres coordenadas (X, Y, Z) en el espacio 3-dimensional
- En OpenGL los creamos en arrays, en una matriz desde -1 a +1:

```
private float vertices[] = {
 -1.0f, 1.0f, 0.0f, // 0, Top Left
 -1.0f, -1.0f, 0.0f, // 1, Bottom Left
 1.0f, -1.0f, 0.0f, // 2, Bottom Right
 1.0f, 1.0f, 0.0f, // 3, Top Right
};
```



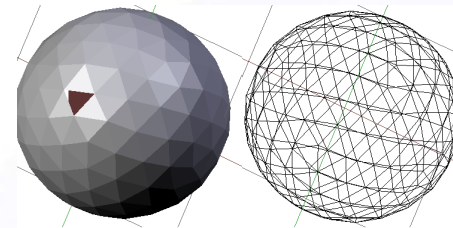
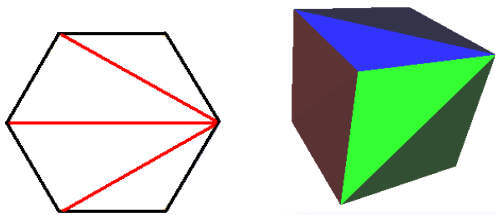
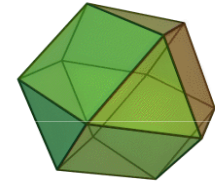
### ➤ *Arista (Edge):*

- Es una línea entre dos vértices. Representan las esquinas de polígonos. Una arista une dos caras adyacentes.
- En OpenGL no se utiliza este concepto: modificar una arista implica modificar uno de sus dos vértices, ya que esto hace cambiar la forma de la arista.



### ➤ *Cara (Face):*

- Conceptualmente, una cara es una lista de vértices. El área que queda entre todos esos vértices es la cara.
- Para OpenGL, una cara (face) se representará por un triángulo.
- ¿Por qué? Para simplificar: cualquier polígono se puede aproximar por triángulos.

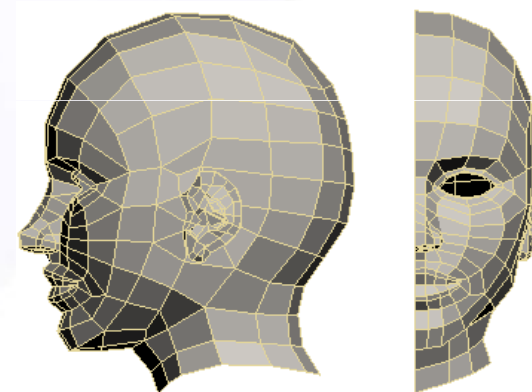
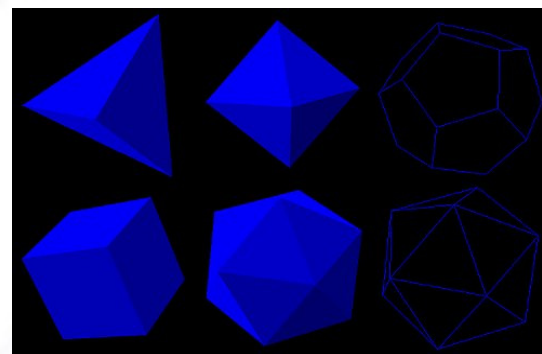
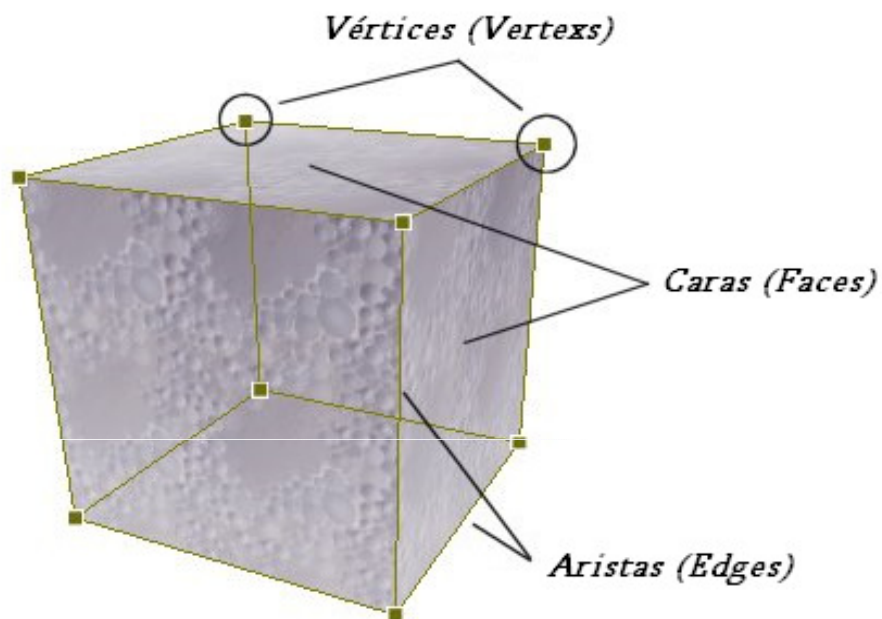


- Importante: el orden de la lista de vértices es importante: OpenGL la utiliza para determinar que cara es interior y que cara exterior (utilizado para ocultar caras, luminosidad, sombras, etc)

### ➤ *Cara (Face):*

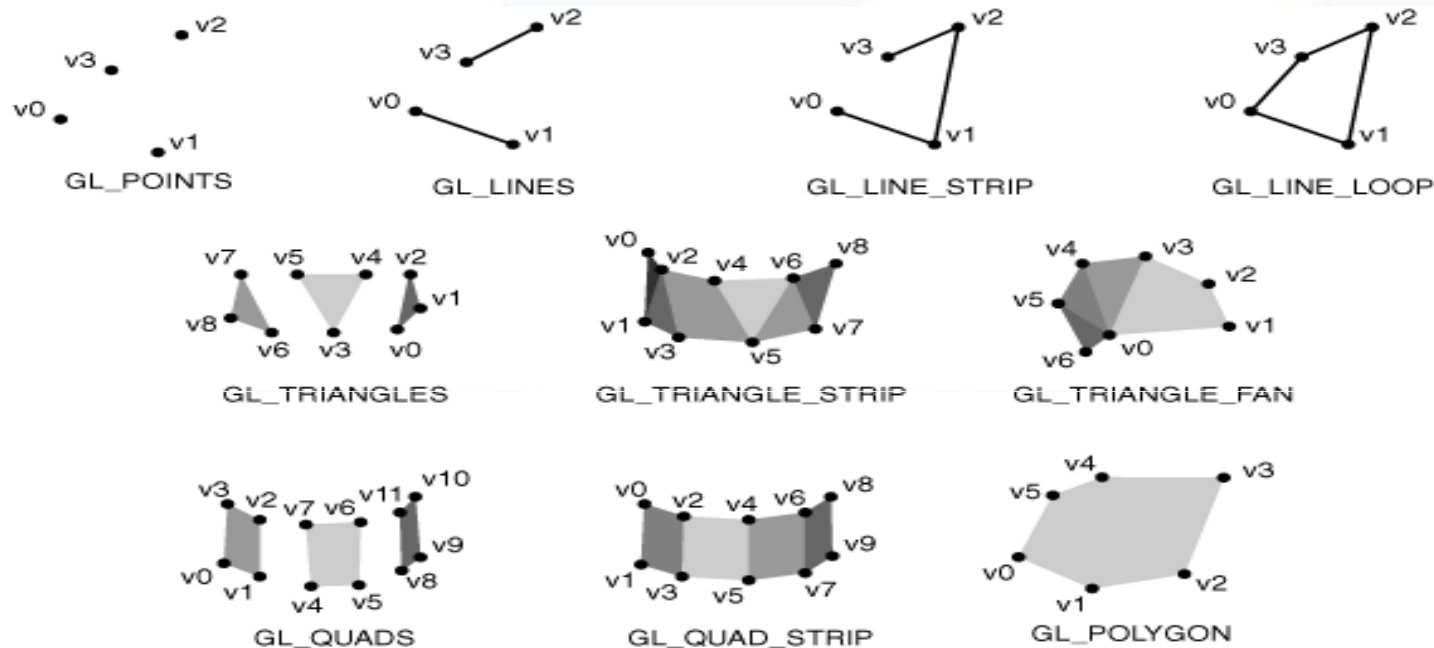
- Los métodos más utilizados para modificar el comportamiento de las caras son:
  - ✓ `GL10.glFrontFace(GL10.GL_CCW)`: indica si una cara es exterior si los vértices se han dado en el sentido contrario a las agujas del reloj. Con `GL_CW` en el sentido opuesto.
  - ✓ `GL10.glEnable(GL10.GL_CULL_FACE)`: Elimina las caras no visibles de un objeto.
  - ✓ `GL10.glCullFace(GL10.GL_BACK)`: Se indica que las caras que quieren ser eliminadas son las traseras respecto al punto de visión.

- *Polígono (Polygon):*
  - Agrupación de caras, vértices y aristas que se agrupan para crear formas deseadas, desde simples a más complejas



➤ *Primitiva*

- OpenGL ofrece unidades básicas de dibujo
- Estas primitivas las referencia por una constante de la clase GL10
- OpenGL sólo necesita saber los vértices para dibujarlas

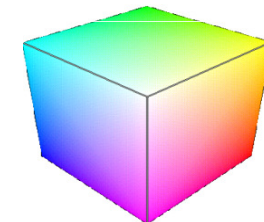
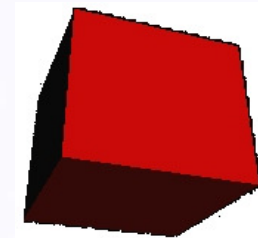
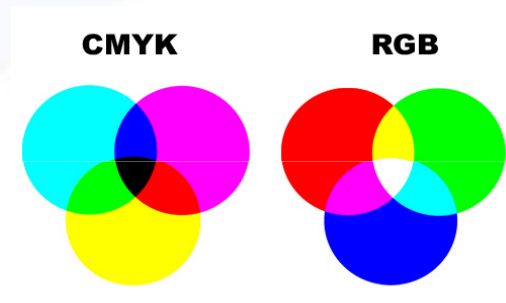


## Ejemplo de dibujo de primitiva

- *Ubicación del ejemplo:*
  - Carpeta de workspace: /OpenGL/OpenGL-poligonos
- *Comportamiento:*
  - En la Activity principal, se pasa un objeto OpenGLRenderer a la GLSurfaceView.
  - Inicialmente se llama al método onSurfaceCreated que se usará para inicializar el aspecto visual.
  - Cada redibujado llamará automáticamente el método onDrawFrame, que redibujará el objeto (square.draw() )
  - En caso de que rotemos el dispositivo, se invocará onSurfaceChanged() para calcular el nuevo aspect ratio.
- *Resultado:*
  - En este ejemplo se visualiza una pantalla oscura con un cuadrado de color en el centro.

## ➤ Colores

- OpenGL utiliza RGBA (Red, Green, Blue y Alpha)
- Se utilizan valores del 0 al 255 decimal, o 0..FF en hexadecimal, o 0..1 en %.
- RGB=(0,0,0) es negro (ausencia de color)
- RGB=(1,1,1) es blanco (todos los colores)
- Flat coloring (coloración plana): se indica a OpenGL con el método `glColor4f(r,g,b,a)`; y renderiza en ese color.
- Se pueden obtener colores degradados en las caras, dándoles a los vértices distintos colores.



## Ejemplo de coloreado

- *Ubicación del ejemplo:*
  - Carpeta de workspace: /OpenGL/OpenGL-colores
- *Comportamiento:*
  - El método onDrawFrame() crea dos cuadrados, uno encima del otro, uno utilizando coloreado Flat(plano) y otro Smooth (degradado)
  - La diferencia principal reside en que el coloreado Flat se realiza en el método draw(...) directamente antes de dibujar el objeto, y en Smooth se debe pasar al método glColorPointer un array de floats con los colores. Este relacionará cada vértice con cada color en el coloreado.
- *Resultado:*
  - En este ejemplo se visualiza una pantalla con 2 cuadrados de distintos colores.