

# *Principios básicos de OpenGL*

*Prof. Wílmer Pereira*

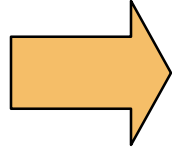
*<http://www ldc usb ve/~wpereira>*

# *Funcionalidades Básicas de OpenGL*

Es una librería de proposito general para el desarrollo animaciones e imágenes en 2D y 3D, inicialmente a partir de GL definido por Silicon Graphic

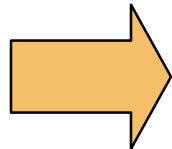
Permite construir gráficos vectoriales con:

Primitivas gráficas:



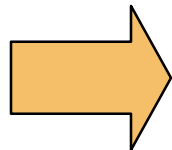
Puntos, rectas curvas, círculos, elipses, ...  
esferas, elipsoides, toroides, ...

Atributo:



Color, estilo de líneas,  
relleno de superficies, ...

Transformaciones:



Traslaciones, rotaciones y escalamiento  
Además reflexiones e inclinaciones

# *Extensiones de OpenGL en distintas plataformas*

AGL	→	MAC
WGL	→	Window
PGL	→	IBM
GLX	→	Xwindow

Además de OpenGL están:

DirectX:	Sólo para Window aunque muy popular en el desarrollo de videojuegos.
SDL:	Librerías LGPL multiplataforma, popular en la gestión de música y sonido
Allegro:	Específico para videojuegos montado sobre OpenGL en C.
Vulkan:	Sucesor de OpenGL multiplataforma que aprovecha los núcleos e incorpora programación de GPU
Java2D, Java3D:	Funciona sobre OpenGL o DirectX

# Tipos de Coordenadas

Lo común es usar coordenadas cartesianas en 4 fases: Modelado, Universales, Visualización y Vinculadas al Dispositivo

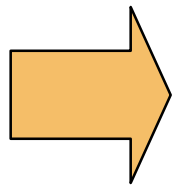
- Modelado: Son locales para definir una escena por partes ... por ejemplo un objeto que se repite como la rueda de una bicicleta
- Universales: Coordenadas para componer toda la escena. Pueden absolutas o normalizadas
- Visualización: Transformar hacia posición y orientación de la cámara
- Dispositivo: Coordenadas que consideran la resolución de la pantalla y tamaño de la ventana. Este último paso se conoce como *rasterización*.

# Notación OpenGL

Se usa el prefijo `GL` para las primitivas básicas y los prefijo `glut` o `glu` para el gestor de ventanas (independiente del sistema operativo)

- Funciones: `glBegin`, `glCopyPixels`, ...
- Constantes: `GL_2D`, `GL_RGB`, ...
- Tipos: `GLbyte`, `GLint`, ...

Para ejecutar programas en OpenGL con GLUT deben tener:



Biblioteca de vínculo dinámico (`.dll`)

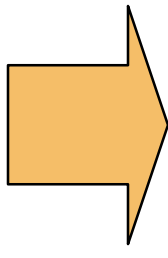
La librería de GLUT para (`.lib`)

Una carpeta GL con varias funciones para incluir (`.h`)

# Ventana y Visualización

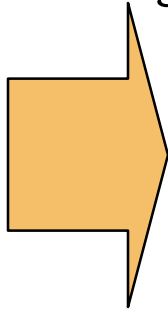
Inicialmente se fija una ventana y sobre ella uno o varios espacios de visualización en píxeles. Las coordenadas pueden estar normalizadas para adaptarse a cualquier tamaño de pantalla

- Ventanas con posición inicial y dimensiones sobre un cuarto cuadrante:



```
glutCreateWindow(Encabezado)
glutInitWindowPosition(Xpixel , Ypixel)
glutInitWindowSize(Xlongitud , Ylongitud)
gluOrtho2D(Xmin , Ymin , Xmax , Ymax) ...
```

- Estrategia de pintado:

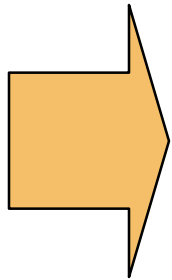


```
glutDisplayFunc(NombreFuncion)
glutMainLoop() Ciclo infinito en espera de eventos
glutInitDisplayMode(Modos)
    GLUT_SINGLE, GLUT_RGB...
Las opciones por defecto ...
```

# Colores

Se define sobre la paleta RGB (cada componente entre 0 y 1). Además es posible un cuarto parámetro que indica la transparencia

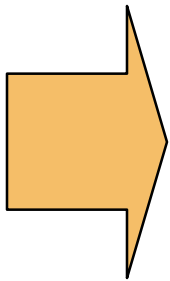
- Define el color actual sobre todo lo que se pinte:



`glColor*(Rojo , Verde , Azul , Transparencia)`

Donde \* representan el tipo del valor que contienen los colores (byte con b, float con f, ...).

- Para fijar el color de fondo de la ventana se usa:



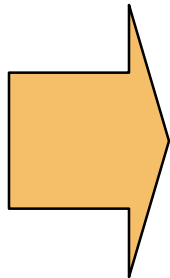
`glClearColor (Rojo , Verde , Azul , Transparencia)`

Donde **Transparencia** con valor 1 es opaco y 0 completamente traslucido.

# Matrices

Definen el tipo de proyección en una matriz inicial. Sobre ella se aplicarán las distintas transformaciones para cada objeto

- Indica inicialmente :

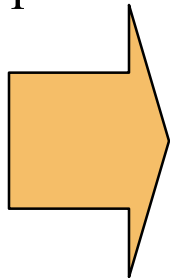


`glLoadIdentity()`

`glMatrixMode(TipoProyeccion)`

Ejemplos: `GL_MODELVIEW`  
`GL_PROJECTION`

- Por otro lado los *buffers* contienen datos para ser plasmados eficientemente (color, profundidad, ...):



`glClear(TiposBuffers)`

Ejemplos: `GL_COLOR_BUFFER_BIT`  
`GL_DEPTH_BUFFER_BIT`



## *Ejemplo ...*

```
#include "GL/glut.h"
void init(void) {
    glClearColor(1.0,1.0,1.0,0.0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0.0,200.0,0.0,150.0);
}
void lineSegment(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,0.4,0.2);
    glBegin(GL_LINES);
        glVertex3f(180,15,-1);
        glVertex3f(10,145,-1);
    glEnd();
    glFlush();
}
void main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(50,100);
    glutInitWindowSize(400,300);
    glutCreateWindow("Ejemplo");
    init();
    glutDisplayFunc(lineSegment);
    glutMainLoop();
}
```

# Primitivas de dibujo

Las imágenes 2D se componen inicialmente con líneas y polígonos, (para el teselado) y curvas (*spline* o Bézier, círculos y elipses)

- En primer lugar las coordenadas de cada punto pueden ser en coordenadas absolutas o relativas. Cada punto se puede pintar individualmente con `setPixel(X , Y)` y se obtiene su *status* con `getPixel(X , Y , color)`. El sistema de coordenadas lo define `gluOrtho2D(Xmin , Xmax , Ymin , Ymax)` ...
- La función `glVertex*` define los vértices de un polígono. El `*` define la cantidad de coordenadas (2D o 3D) y el tipo. Además se puede pasar sólo un parámetro con un vector, una estructura (C) o un objeto (C++).

# *Figuras con líneas*

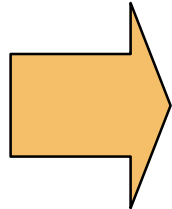
Pintar polígonos se hace en un bloque de vértices que se conectan dependiendo del tipo de trazado

```
glBegin(Figura);
```

vértices ...

```
glEnd();
```

Cuando se trata de líneas, Figura puede ser:



GL\_LINES

GL\_LINE\_STRIP

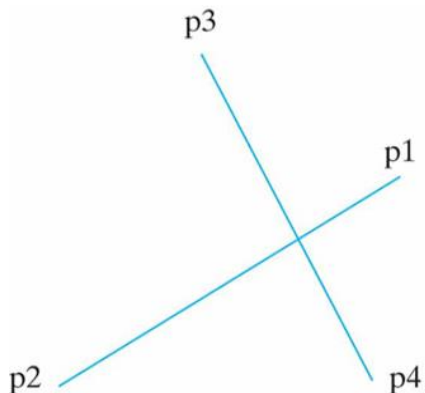
GL\_LINE\_LOOP

# Ejemplo con líneas

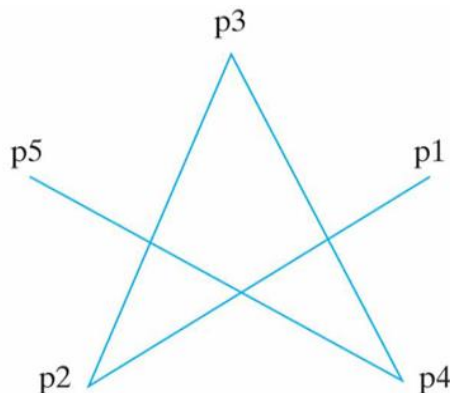
```
glBegin(X);  
    glVertex2i(p1);  
    glVertex2i(p2);  
    glVertex2i(p3);  
    glVertex2i(p4);  
    glVertex2i(p5);  
glEnd();
```

Si X es:

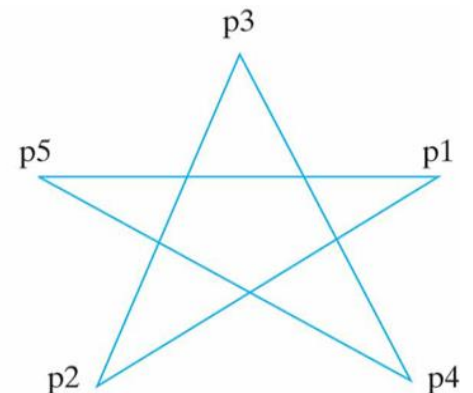
- (a) GL\_LINES                      Una línea discontinua
- (b) GL\_LINE\_STRIP                Una polilínea
- (c) GL\_LINE\_LOOP                Una polilínea cerrada



(a)



(b)



(c)

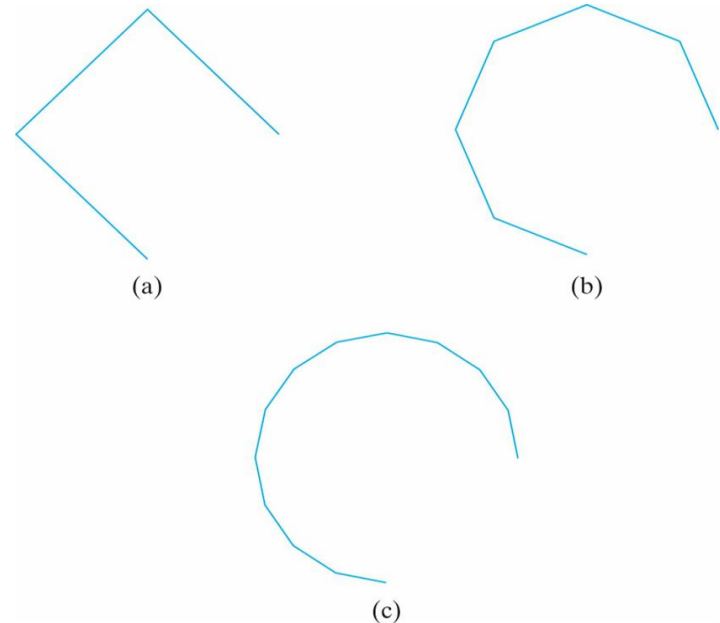
# Curvas con múltiples líneas

Se puede utilizar una aproximación con puntos discretos con Bézier, *splines* o simplemente una polilínea

Tanto las curvas de Bézier como los diferentes tipos de *spline*, usan interpolación con polinomios de grado bajo. Esto para aproximar un conjunto de puntos con una curva suave

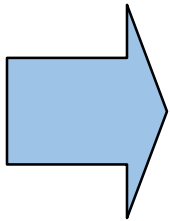
Aproximación con:

- (a) Tres líneas
- (b) Seis líneas
- (c) Doce líneas



# Llenado de áreas en 2D

- El llenado de superficies con texturas no es tan simple sobre áreas de cualquier forma. Los polígonos deben satisfacer ciertas condiciones mínimas:



- Debe ser una polilínea cerrada sin arcos que se crucen lo que define un polígono
- El polígono debe ser convexo para evitar superficies sin rellenar

- Un polígono es convexo cuando los ángulos internos son estrictamente menores a  $180^\circ$ . Sino es cóncavo.

... 0 ...

- Un polígono es convexo si la prolongación de cualquiera de sus arcos deja el resto de los arcos de un lado. Si la prolongación de alguno de sus arcos corta otro arco entonces es cóncavo.

... 0 ...

- Un polígono es convexo si el segmento que une dos puntos del interior, está completamente contenido en el polígono. Sino es cóncavo.

# Identificación de polígonos convexos

Se logra asociando los arcos a vectores (trasladándolos al origen) y calculando el producto cruz o producto vectorial. Si los resultados son todos del mismo signo el polígono asociado a los vectores es convexo

Si  $V_1=(x_1, y_1)$  y  $V_2=(x_2, y_2)$  entonces

$$V_1 \times V_2 = x_1 * y_2 - y_1 * x_2$$

El resultado del producto vectorial es un vector unitario, perpendicular al plano que forman los dos vectores. Si el ángulo es  $< 180$  es positivo. Sino es negativo

... Se cumple la regla de la mano derecha por lo que no es conmutativo ...

# Polígono cóncavo

$$\mathbf{E}_1 = (1, 0, 0)$$

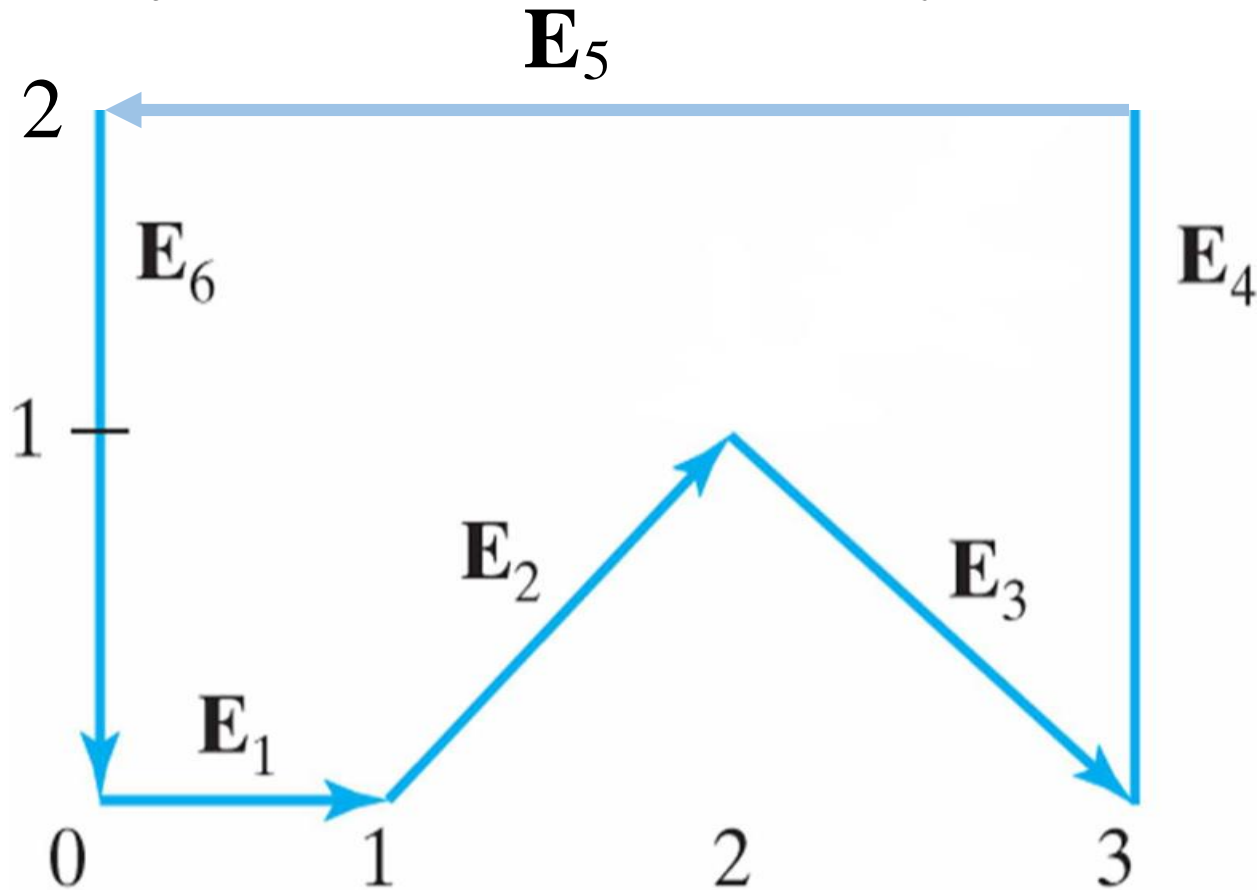
$$\mathbf{E}_3 = (1, -1, 0)$$

$$\mathbf{E}_5 = (-3, 0, 0)$$

$$\mathbf{E}_2 = (1, 1, 0)$$

$$\mathbf{E}_4 = (0, 2, 0)$$

$$\mathbf{E}_6 = (0, -2, 0)$$



$$\mathbf{E}_1 \times \mathbf{E}_2 = (0, 0, 1)$$

$$\mathbf{E}_4 \times \mathbf{E}_5 = (0, 0, 6)$$

$$\mathbf{E}_2 \times \mathbf{E}_3 = (0, 0, -2)$$

$$\mathbf{E}_5 \times \mathbf{E}_6 = (0, 0, 6)$$

$$\mathbf{E}_3 \times \mathbf{E}_4 = (0, 0, 2)$$

$$\mathbf{E}_6 \times \mathbf{E}_1 = (0, 0, 2)$$

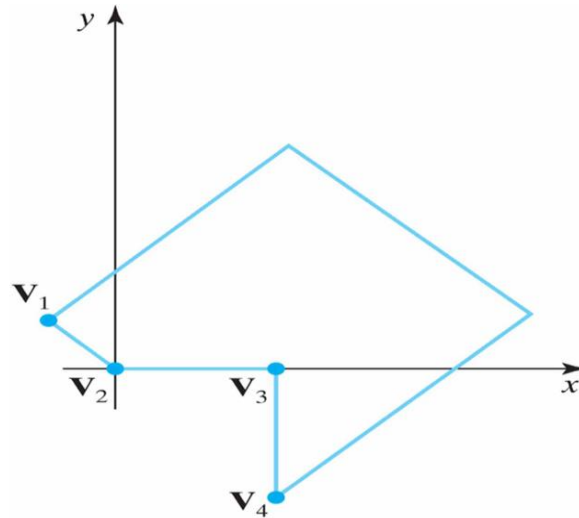


# *Condiciones adicionales a los polígonos*

- La mayoría de las librerías (incluyendo OpenGL) exigen que los polígonos sean convexos (de hecho algunas librerías sólo aceptan triángulos). La razón es que los polígonos cóncavos pueden presentar dificultades para ser rellenados
- Si un conjunto de vértices son colineales, el producto vectorial será  $(0,0,0)$  lo que describe un polígono degenerado. Esto debe evitarse ...
- Lo ideal sería que las superficies se aproximen con polígonos del mismo tipo (triángulos, cuadrados, hexágonos, octógonos, ...), es decir, poligonizar o teselar un área así como se aproxima una curva con una polilínea.

# *Construir polígonos convexos a partir de cóncavos*

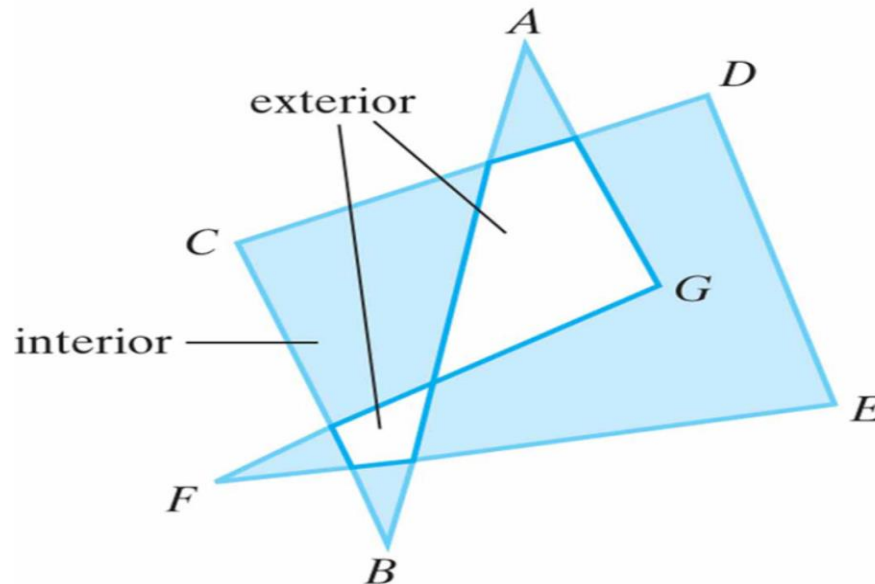
- Rotar el polígono para hacer coincidir uno de los vectores (con vector perpendicular diferente) hacia uno de los ejes coordenados



- Poligonizar o teselar directamente el polígono cóncavo no es buena idea porque puede haber problemas ... siempre debe ser “convexizado” antes de hacer el teselado

# Prueba para puntos interiores o exteriores de un polígono

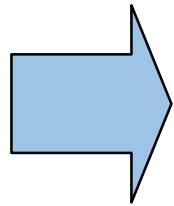
Partir de un punto lejano al polígono convexo y contar los lados. Si el número de segmentos cruzados es impar, es un punto interior y se debe pintar ... sino es un punto exterior y no se pinta



Hay otras estrategias que pueden ser convenientes cuando se cruzan polígonos, calculando el número de rotación (*winding-number*).  
... Puede dar resultados diferentes ...

# *Otras consideraciones a las superficies poligonales*

- Es convenientes construir estructuras de datos o tablas de polígonos para:



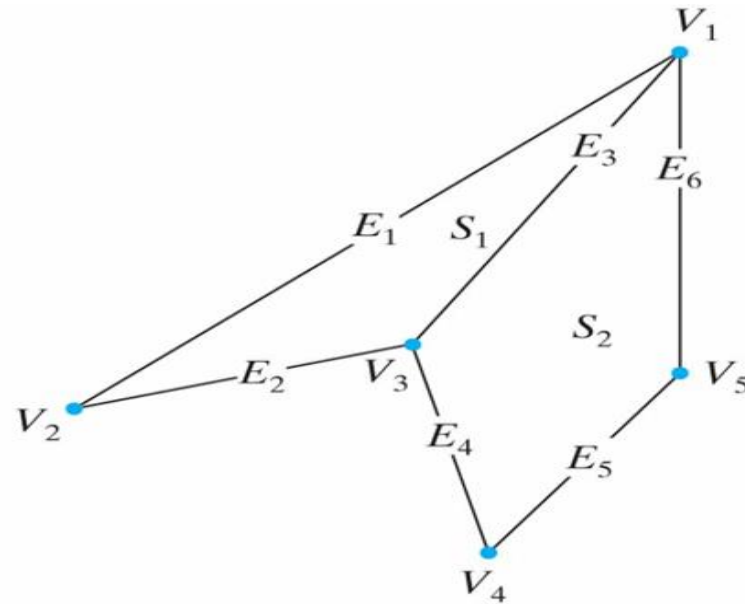
Atributos como color, transparencia, reflexión de luz, ...

Arcos que componen el polígono

Tabla de vértices ...

- También puede ser adecuado definir la ecuación del plano de esa área (sobre todo si se está modelando el objeto en 3D).
- Además caracterizar lo que sería la cara frontal o visible y la cara posterior u oculta del área que define el polígono. Esto para la aplicación del color, textura o efectos visuales y de iluminación.

# Estructuras de datos a las superficies poligonales



VERTEX TABLE	
$V_1$ :	$x_1, y_1, z_1$
$V_2$ :	$x_2, y_2, z_2$
$V_3$ :	$x_3, y_3, z_3$
$V_4$ :	$x_4, y_4, z_4$
$V_5$ :	$x_5, y_5, z_5$

EDGE TABLE	
$E_1$ :	$V_1, V_2$
$E_2$ :	$V_2, V_3$
$E_3$ :	$V_3, V_1$
$E_4$ :	$V_3, V_4$
$E_5$ :	$V_4, V_5$
$E_6$ :	$V_5, V_1$

SURFACE-FACET TABLE	
$S_1$ :	$E_1, E_2, E_3$
$S_2$ :	$E_3, E_4, E_5, E_6$

# Funciones para relleno de polígonos convexos

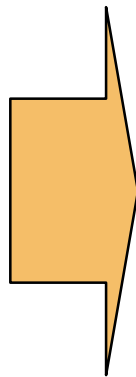
Al igual que con las líneas, se describe entre `glBegin` y `glEnd` y los vértices que conforman un polígono convexo. OpenGL ofrece para triángulos y cuadrados.

```
glBegin(Figura);
```

```
vértices ...
```

```
glEnd();
```

Cuando se trata de polígonos convexos, esta vez `Figura` puede ser:



```
GL_POLYGON  
GL_TRIANGLES  
GL_TRIANGLE_STRIP  
GL_TRIANGLE_FAN  
GL_QUADS  
GL_QUAD_STRIP
```

# Ejemplo con polígonos convexos y triángulos

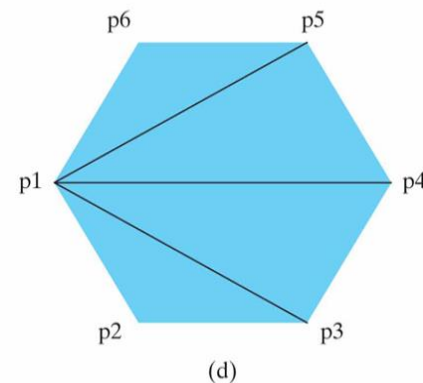
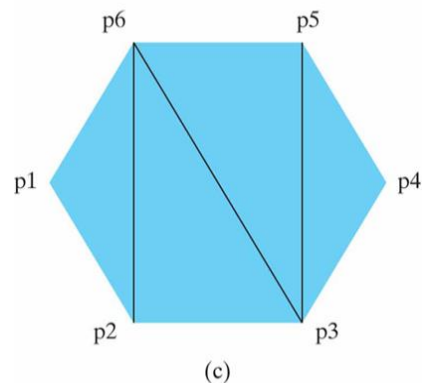
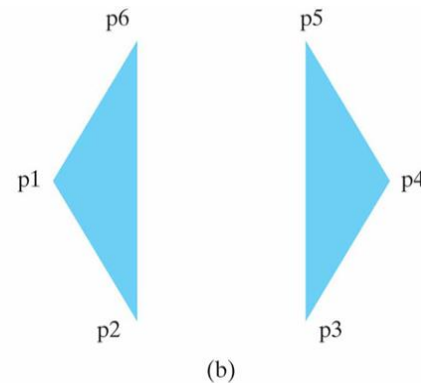
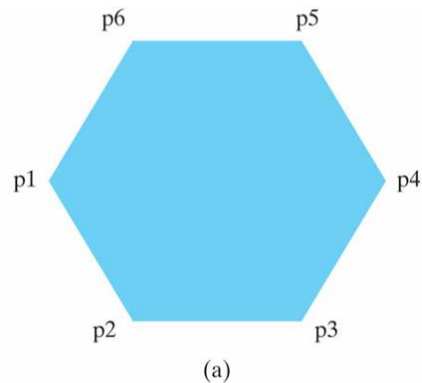
`glBegin(X);`

```
glVertex2i(p1);  
glVertex2i(p2);  
glVertex2i(p3);  
glVertex2i(p4);  
glVertex2i(p5);  
glVertex2i(p6);
```

`glEnd();`

Donde X es:

- (a) `GL_POLYGON`
- (b) `GL_TRIANGLE`
- (c) `GL_TRIANGLE_STRIP`
- (d) `GL_TRIANGLE_FAN`



# Ejemplo con cuadrados

```
glBegin(X);
```

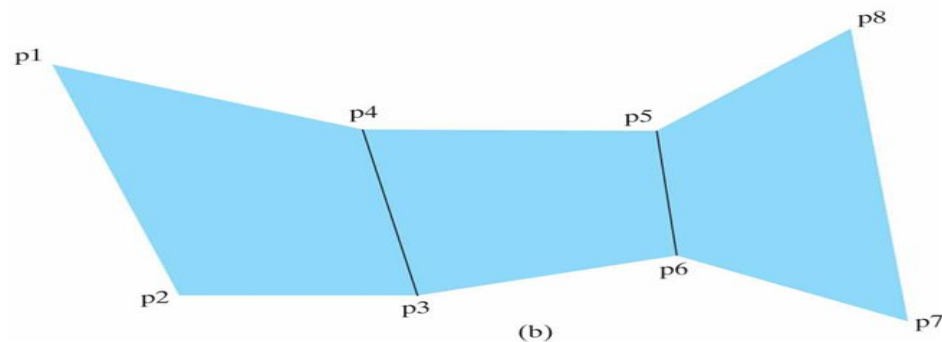
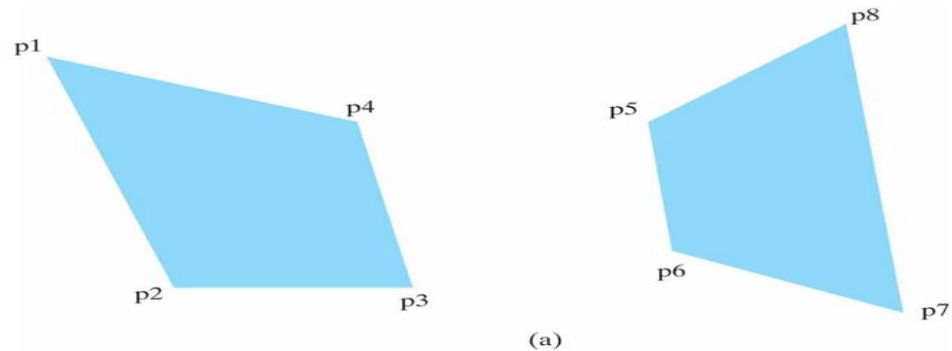
```
glVertex2i(p1);  
glVertex2i(p2);  
glVertex2i(p3);  
glVertex2i(p4);  
glVertex2i(p5);  
glVertex2i(p6);  
glVertex2i(p7);  
glVertex2i(p8);
```

Donde X es:

(a) GL\_QUADS

(b) GL\_QUAD\_STRIP

```
glEnd();
```





# bitmap vs pixelmap

Cuando se desea generar patrones sobre una superficie o polígono, se pueden usar imágenes predefinidas como fotografías (*pixelmap*) o construir el patrón a partir de colores básicos usando una máscara (*bitmap*)

- El *pixelmap* incluye información de colores por cada pixel. En cambio *bitmap* tiene valores 0/1 indicando cuando pintar o no, cada pixel, al color actual.
- Ambos se manejan con *buffer* al rasterizar (pasar a coordenadas del dispositivo)

`glBitmap(ancho , alto ,  $x_0$  ,  $y_0$  ,  $x_{prox}$  ,  $y_{prox}$  , buffer)`

Donde:

ancho , alto:	Dimensiones del buffer
$x_0$ , $y_0$ :	Coordenadas de inicio en la ventana
$x_{prox}$ , $y_{prox}$ :	Próximas coordenadas de rasterización
buffer:	Arreglo con 0 y 1.

`glRasterPos(Coordenadas)`

Fija posición inicial de rasterización

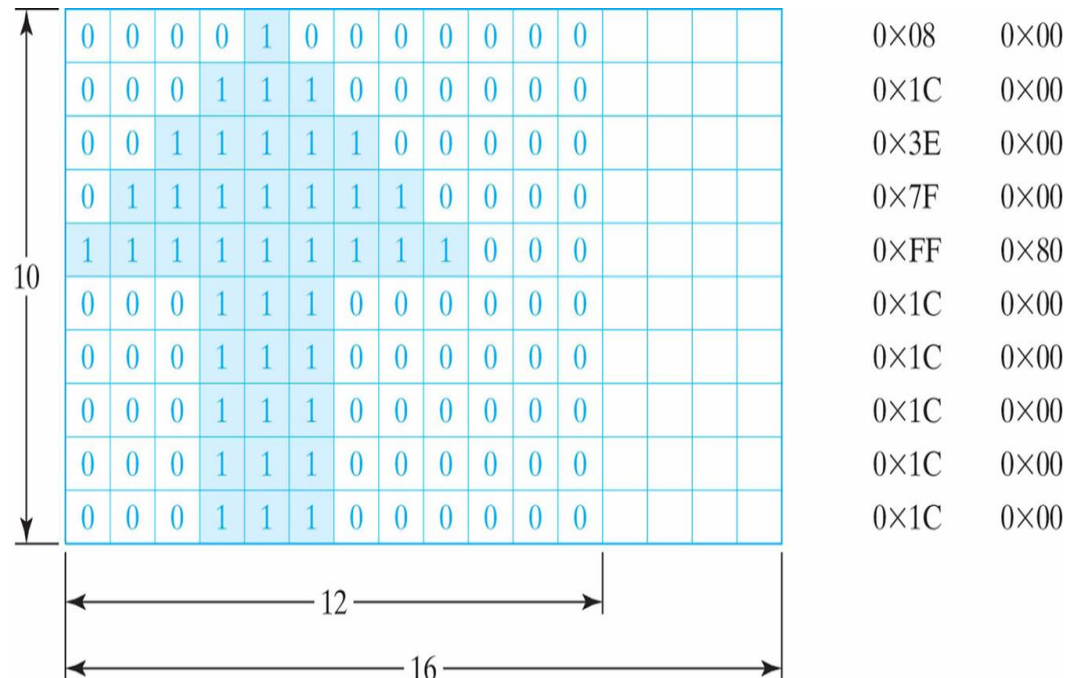
# Ejemplo bitmap

```
GLubyte bitShape [20] = { 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00,  
                           0x1c, 0x00, 0xff, 0x80, 0x7f, 0x00, 0x3e, 0x00,  
                           0x1c, 0x00, 0x08, 0x00};
```

```
glPixelStorei (GL_UNPACK_ALIGNMENT, 1); // Coloca modo de  
                                         // almacenamiento de los  
                                         // pixeles.
```

```
glRasterPos2i (30, 40);
```

```
glBitmap (9, 10, 0.0, 0.0, 20.0, 15.0, bitShape);
```



# Funciones para pixelmap

En principio se utiliza un *buffer* de colores con el patrón pero pueden haber varios *buffers* para efectos estereoscópicos

`glDrawPixels(ancho , alto , FormatoDatos , Tipo , buffer)`

Donde:

ancho , alto:	Dimensiones del buffer
FormatoDatos:	Presentación de colores
Tipo:	Tipo de los colores
buffer:	Arreglo

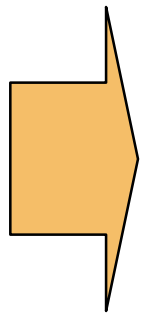
Ejemplo:

`glDrawPixels(128,128,GL_RGB,GL_UNSIGNED_BYTE,colorBuffer)`

# Buffer para visualización

Los *buffers* de profundidad (`GL_DEPTH_COMPONENT`) y bordes (`GL_STENCIL_INDEX`) también tienen incidencia en los colores

- Además también están los *buffers* para estereoscopía que permiten dar sensación de profundidad con al menos dos imágenes tomadas desde diferentes ángulos
- Se proponen máximo 4 *buffers* (derecho, izquierdo, frontal y posterior)



`GL_FRONT_LEFT`  
`GL_FRONT_RIGHT`  
`GL_BACK_LEFT`  
`GL_BACK_RIGHT`

- Además están las operaciones para leer, copiar y transferir los buffers

# Tratamiento de errores OpenGL

Además de las facilidades que ofrece C++ (try/catch) y las de C (perror), OpenGL ofrece captura de errores para las llamadas a la librería

- El código del último error se registra en una variable de tipo `Glenum`, y se salta la llamada que generó el incidente.
- Con la función `glGetError` se recupera el código y con `gluErrorString` se obtiene un mensaje describiendo la falla. Si retorna `GL_NO_ERROR` la ejecución no tuvo problemas.

```
#include <stdio.h>
Glenum errorCheck() {
    Glenum codigo;
    cons Glubyte *string;
    codigo=glGetError();
    if (codigo!=GL_NO_ERROR) {
        string=gluErrorString(codigo);
        fprintf(stderr,"OpenGL error: %s\n",string);
    }
    return codigo;
}
```

# *Código de errores OpenGL*

Symbolic Constant	Meaning
GL_INVALID_ENUM	A GLenum argument was out of range
GL_INVALID_VALUE	A numeric argument was out of range
GL_INVALID_OPERATION	An operation was illegal in the current OpenGL state
GL_STACK_OVERFLOW	The command would cause a stack overflow
GL_STACK_UNDERFLOW	The command would cause a stack underflow
GL_OUT_OF_MEMORY	There was not enough memory to execute a command