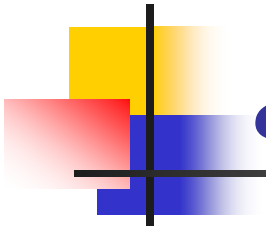
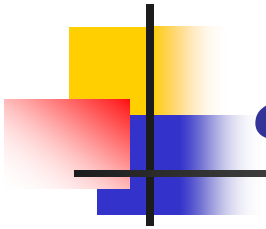


¿Qué es Open GL?

- Es un SW (en forma de librería C) que **permite la comunicación entre el programador y el HW** de la máquina para el diseño de gráficos.
- Es **portable**. Consta de unos 150 comandos muy básicos.
- **No ofrece comandos para la manipulación de ventanas**, ni para leer datos introducidos por un usuario.
- No dispone de comandos de alto nivel para describir escenas 3D. En Open GL debemos construir el gráfico a partir de sus **primitivas geométricas**:
  - **Puntos, Líneas y Polígonos**

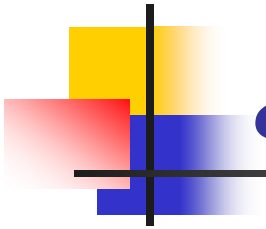


¿Qué es Open GL?

- Open GL **está diseñado para trabajar en red**. En un ordenador (servidor) podemos ejecutar nuestro programa Open GL y en otro (cliente) podemos mostrar el gráfico que hayamos diseñado.

- Open GL es una **máquina de estados**. Tenemos una colección de variables de estado a las que vamos cambiando su valor.

- Open GL se distribuye siempre con la librería GLU (**Open GL Utility Library**), que está construida a partir de Open GL y suministra comandos de alto nivel para el dibujo de gráficos 3D.



¿Qué es Open GL?

- Para la manipulación de ventanas y E/S puede utilizarse la librería GLUT (**Open GL Utility ToolKit**), que tiene la ventaja de ser portable.

- Nosotros utilizaremos el entorno de desarrollo **C++ Builder** que permite el desarrollo de interfaces gráficas de usuario de una manera simple.

C++ Builder y Open GL (estructura de un programa)

UFP.h (declaración de la clase)

```
#ifndef UEsqueletoH
#define UEsqueletoH
//---------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>          // # indica que es una directiva para el
#include <Forms.hpp>             //    el precompilador

 #include <gl\gl.h>
 #include <gl\glu.h>
//---------------------------------------
```

C++ Builder y Open GL (estructura de un programa)

```
Class TGLForm2D : public TForm
{
__published:
  void __fastcall FormResize(TObject *Sender);
  void __fastcall FormPaint(TObject *Sender);
  void __fastcall FormDestroy(TObject *Sender);
  void __fastcall FormCreate(TObject *Sender);
```

C++ Builder y Open GL (estructura de un programa)

```
private:
    HDC hdc;
    HGLRC hrc;
    //definen el tamaño del volumen de vista
    GLfloat xLeft,xRight,yTop,yBot;
    //guarda el radio del puerto de vista
    GLfloat RatioViewPort;
```

- HDC (Handle Device Context): Para fijar el "***área cliente***" de la ventana en la que deseamos pintar.

- HGLRC (Handle Open GL Rendering Context): Para fijar **el contexto** en el que Open GL dibujará sus gráficos.

C++ Builder y Open GL (estructura de un programa)

```
void __fastcall TGLForm2D::FormCreate(TObject *Sender){
    hdc = GetDC(Handle);
    SetPixelFormatDescriptor();
    hrc = wglCreateContext(hdc);
    if(hrc == NULL)
      ShowMessage(":-)~ hrc == NULL");
    if(wglMakeCurrent(hdc, hrc) == false)
      ShowMessage("Could not MakeCurrent");
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

C++ Builder y Open GL (estructura de un programa)

```
    //inicialización del volumen de vista
    xRight=200.0; xLeft=-xRight;
    yTop= xRight; yBot=-yTop;
    //Radio del volumen de vista == 1

    //inicialización del puerto de vista
    //ClientWidth=400;
    //ClientHeight=400;
    RatioViewPort=1.0;

    // inicialización de variables de la escena
}
```

C++ Builder y Open GL (estructura de un programa)

```
void __fastcall TGLForm2D::SetPixelFormatDescriptor() {
  PIXELFORMATDESCRIPTOR pfd = {
     sizeof(PIXELFORMATDESCRIPTOR),
     1,
     PFD_DRAW_TO_WINDOW | PFD_SUPPORT_Open_GL | PFD_DOUBLEBUFFER,
     PFD_TYPE_RGBA,
     32, 0,0,0,0,0,0,0,0,0,0,0,0,32,0,0,
     PFD_MAIN_PLANE,0, 0,0,0
  };
  int PixelFormat = ChoosePixelFormat(hdc, &pfd);
  SetPixelFormat(hdc, PixelFormat, &pfd);
}
```

C++ Builder y Open GL (estructura de un programa)

```
void __fastcall TGLForm2D::FormResize(TObject *Sender)
   //se actualiza puerto de vista y su radio
   if ((ClientWidth<=1)||(ClientHeight<=1)){
     ClientWidth=400;
     ClientHeight=400;
     RatioViewPort=1.0;
   }
   else RatioViewPort=
                  (float)ClientWidth/(float)ClientHeight;

   glViewport(0,0,ClientWidth,ClientHeight);
```

C++ Builder y Open GL (estructura de un programa)

```
// se actualiza el área visible de la escena
// para que su radio coincida con ratioViewPort
GLfloat RatioVolVista=xRight/yTop;

if (RatioVolVista>=RatioViewPort){
 //Aumentamos yTop-yBot
    yTop= xRight/RatioViewPort;
    yBot=-yTop;
}
else{ //Aumentamos xRight-xLeft
    xRight=RatioViewPort*yTop;
    xLeft=-xRight;
}
```

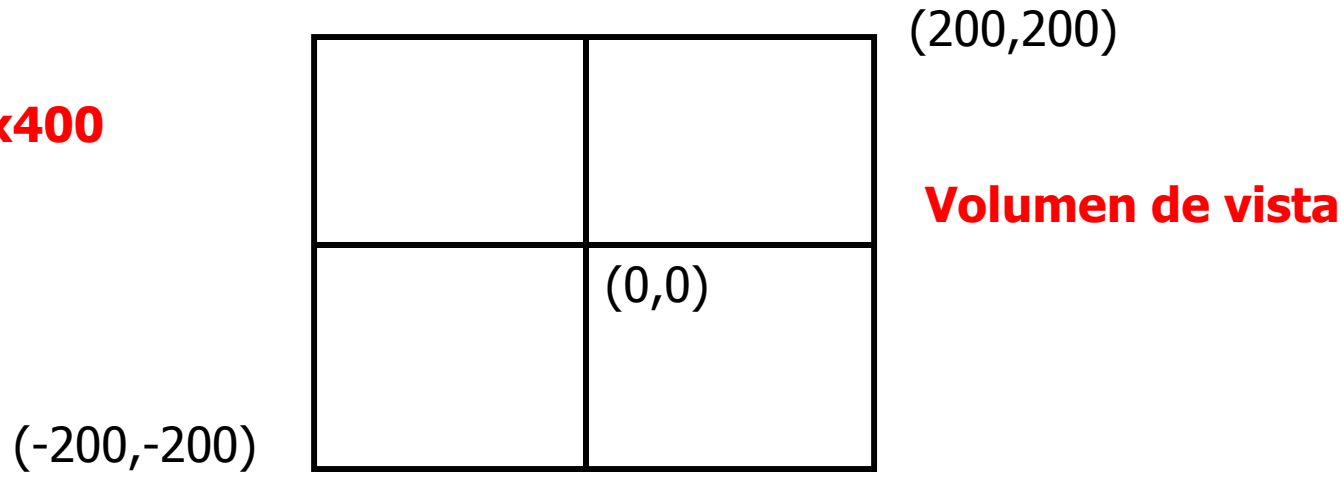
C++ Builder y Open GL (estructura de un programa)

```
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(xLeft,xRight,yBot,yTop);


  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  GLScene();

}
```

C++ Builder y Open GL (estructura de un programa)

- El método **FormResize** establece el volumen de vista (área visible de la escena) y el puerto de vista (zona de la ventana de la pantalla donde aparecerán los objetos).
- El sistema de coordenadas lo establece el comando **gluOrtho2D(..)**
- El puerto de vista lo establece el comando **glViewport(..)**
- La situación inicial es:

Ventana: 400x400
(puerto de vista)



Volumen de vista

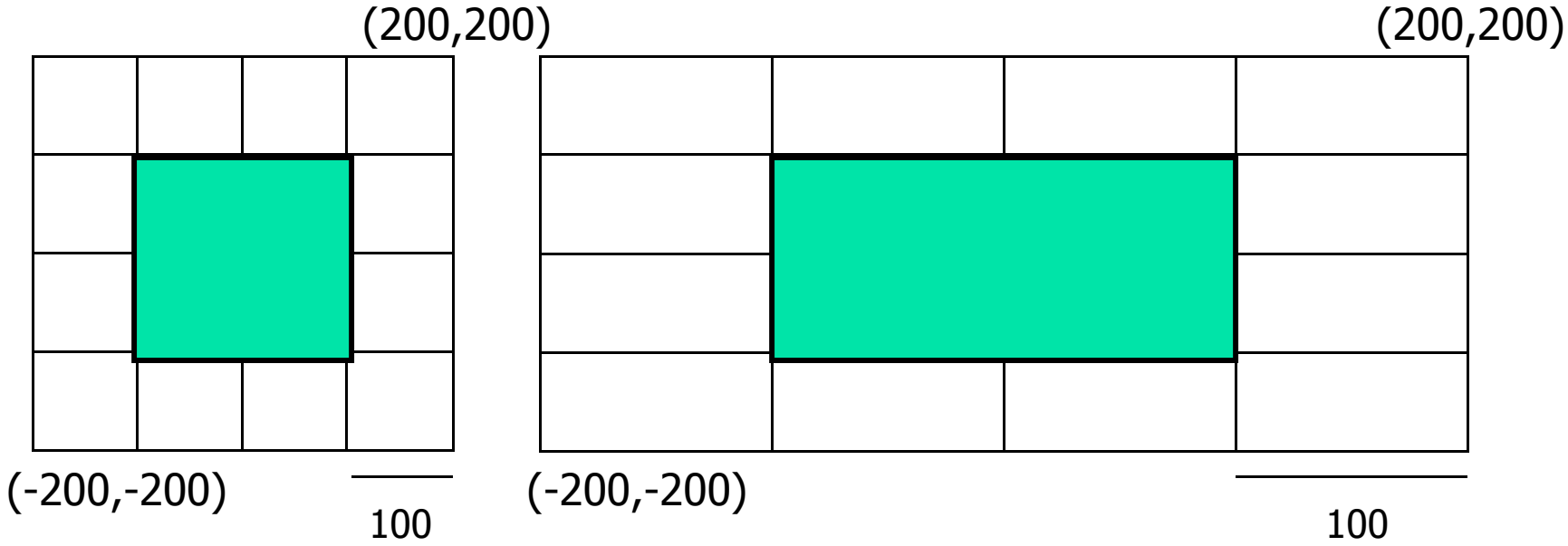
C++ Builder y Open GL (estructura de un programa)

- El comando **glViewport(0, 0, ClientWidth, ClientHeight)** establece el **puerto de vista**: región de la ventana, con forma de rectángulo, en la que vamos a dibujar. (0,0) indica la posición de su esquina inferior izquierda. ClientWidth y ClientHeight indican su anchura y su altura. **(0,0)** y **(ClientWidth, ClientHeight)** vienen dadas en coordenadas de la ventana (integer).

- El comando **gluOrtho2D(l,r,b,t)** hace que (l,b) se sitúe en la esquina inferior izquierda del puerto de vista, y (r,t) en la esquina superior derecha.

C++ Builder y Open GL (estructura de un programa)

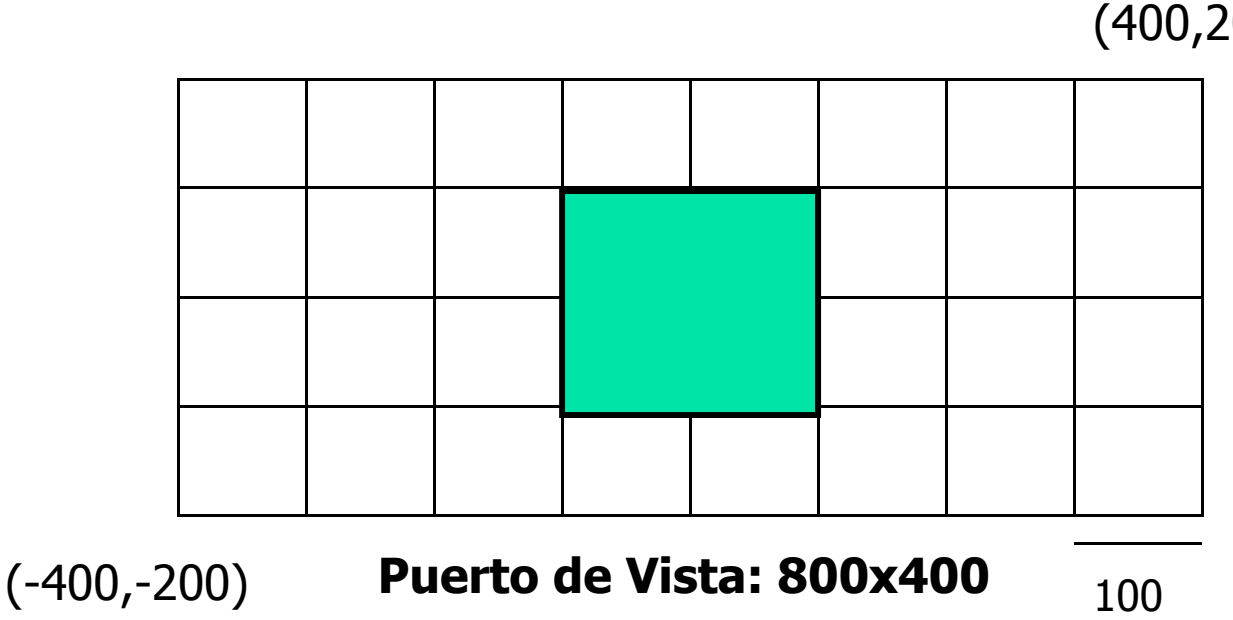
- Si modificamos el tamaño de la ventana sin ajustar el área visible de la escena, posiblemente deformemos el gráfico. El cuadrado es el mismo en dos puertos de vista de distinto tamaño.



Puerto de vista: 400x400 **Puerto de vista: 800x400**

C++ Builder y Open GL (estructura de un programa)

- Para evitar que se deforme la figura, aumentamos el eje de las x's de forma proporcional (**RatioVolVista=RatioViewPort**).



C++ Builder y Open GL (estructura de un programa)

- Open GL trabaja internamente con **matrices**. Más concretamente tiene la matriz de **proyección**, la de **modelado y vista**, y la de **puerto de vista**.

- La **matriz de proyección** almacena la forma en la que debe proyectarse el gráfico en el plano de visión.
- La de **modelado y vista** almacena las transformaciones de *rotación, traslación*, etc.
- La de **puerto de vista** almacena cómo presentar la proyección realizada en el puerto de vista.

C++ Builder y Open GL (estructura de un programa)

- Antes de definir el volumen de vista, es necesario cargar la matriz de proyección, y hacerla la identidad.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(xLeft, xRight, yBot, yTop);
```

- De igual forma, tras establecer el volumen de vista y antes de dibujar nada, es necesario cargar como matriz actual la de modelado y vista, y hacerla la identidad.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
GLScene();
```

C++ Builder y Open GL (estructura de un programa)

```
void __fastcall TGLForm2D::GLScene()
{
   glClear(GL_COLOR_BUFFER_BIT);
  // instrucciones para dibujar la escena
  glFlush();
  SwapBuffers(hdc);
}
```

- Debe contener las instrucciones Open GL para el dibujo del gráfico. Es la parte a rellenar dentro del esqueleto.

C++ Builder y Open GL (estructura de un programa)

```
void __fastcall TGLForm2D::FormPaint(TObject *Sender)
{
  GLScene();
}
```

- Este método se ejecuta cada vez que se "repinta" el formulario, por lo tanto debe llamar a GLScene (contiene las instrucciones de dibujo).

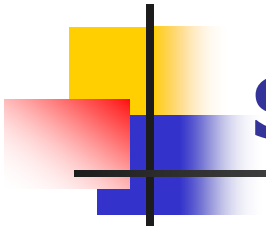
C++ Builder y Open GL (estructura de un programa)

```
void __fastcall TGLForm2D::FormDestroy(TObject *Sender)
{
    ReleaseDC(Handle,hdc);
    wglMakeCurrent(NULL, NULL);
    wglDeleteContext(hrc);
    // liberar lo que hayamos creado
}
```

- Liberamos los recursos escogidos.

C++ Builder y Open GL (estructura de un programa)

- Es conveniente añadir el esqueleto como formulario de C++
  - Sobre el formulario principal seleccionar Add to Repository del menú emergente.
  - En el cuadro de diálogo que se abre, rellenar los campos seleccionando Page como Forms.
- Luego, al crear un proyecto nuevo:
  - Proyect/Remove from Project: Unit1.cpp
  - File/new/Forms: elegir el esqueleto salvado.



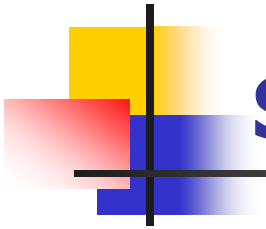
Sintaxis de los comandos Open GL

- Todos los comandos Open GL comienzan con **gl**, y cada una de las palabras que componen el comando comienzan por letra mayúscula.

```
glMatrixMode(....)
glEnable(...)
```

- Las constantes (y variables de estado) en Open GL se escriben en mayúsculas, y todas ellas empiezan por GL. Cada una de las palabras que componen la constante está separada de la anterior por "_".

```
GL_MODELVIEW
GL_COLOR_BUFFER_BIT
```



Sintaxis de los comandos Open GL

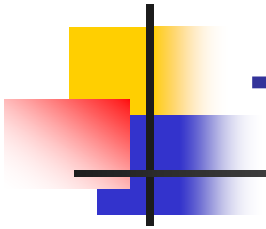
- Existen comandos en Open GL que admiten distinto número de argumentos y distintos tipos. Estos comandos terminan con el sufijo **nt**, donde **n** indica el número de argumentos y **t** el tipo de los mismos.

```
glVertex2i(10,21)
```

indica que estamos usando vértices 2D del tipo entero. Por el contrario

```
glVertex3f(10.0,21.1,11.7)
```

indica que estamos usando vértices 3D del tipo GLfloat.

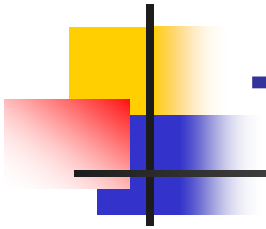


Tipos básicos de Open GL

- Open GL trabaja internamente con tipos básicos específicos que son compatibles con los de C/C++.
- Es fuertemente recomendable utilizar los tipos de Open GL para garantizar la "**independencia del sistema**" en el que estamos ejecutando el programa.
- Por ejemplo, un método:

```
void drawDot(int x, int y) {
    glBegin(GL_POINTS);
        glVertex2i(x,y);
    glEnd();
}
```

tendría que transformar el **int** en **GLint**



Tipos básicos de Open GL

- El programa anterior podría dar problemas en sistemas que empleen un número distinto de bits para almacenar un **int** y un **GLint**.

- La forma idónea de definirse el método anterior sería:

```
void drawDot(GLint x, GLint y) {
   glBegin(GL_POINTS);
      glVertex2i(x,y);
   glEnd();
}
```

Tipos básicos de Open GL

Sufijo	Tipo	Open GL
b	**entero de 8 bits**	**GLbyte**
s	**entero de 16 bits**	**GLshort**
i	**entero de 32 bits**	**GLint, GLsizei**
f	**coma flotante 32 bits**	**GLfloat, GLclampf**
d	**coma flotante 64 bits**	**GLdouble, GLclampd**
ub	**entero sin signo de 8 bits**	**GLubyte, GLboolean**
us	**entero sin signo de 16 bits**	**GLushort**
ui	**entero sin signo de 32 bits**	**GLuint, GLenum, GLbitfield**

Primitivas gráficas: Puntos

- Para dibujar un punto tenemos los comandos:
  - `glVertex2{sifd}[v](x,y).` Especifica un vértice 2D en las coordenadas (x,y).
  - `glVertex3{sifd}[v](x,y,z).` Especifica un vértice 3D en las coordenadas (x,y,z).

- Para dibujar una secuencia de puntos:

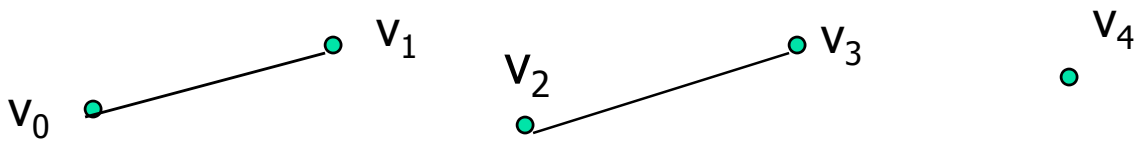
```
glBegin(GL_POINTS);
  glVertex...
  glVertex..
  ......
  glVertex..
glEnd();
```

Primitivas gráficas: Líneas

- Para dibujar líneas debemos usar:

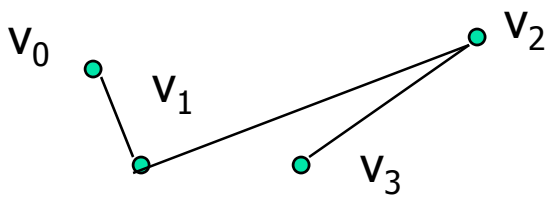
```
glBegin(GL_LINES);
  glVertex...      // v0
  glVertex..       // v1
  ......
  glVertex..       // vn
glEnd();
```

Pinta las líneas $\mathbf{v_0v_1}$, $\mathbf{v_2v_3}$, …, $\mathbf{v_{n-1}v_n}$. Si el número de vértices es impar, el último vértice introducido se ignora.

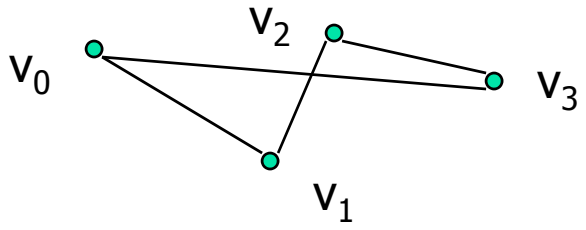


Primitivas gráficas: Líneas

- Si utilizamos la constante **GL_LINE_STRIP**, las líneas dibujadas se conectan, i.e, se pintan las líneas $\mathbf{v_0v_1}$, $\mathbf{v_1v_2}$, $\mathbf{v_3v_4}$, etc. Si el número de vértices es 1, entonces no se pinta nada.



- Con la constante **GL_LINE_LOOP** la poli línea dibujada se cierra. Es decir, se dibujan las líneas $\mathbf{v_0v_1}$, $\mathbf{v_1v_2}$,..., $\mathbf{v_{n-1}v_n}$, $\mathbf{v_nv_0}$.



Primitivas gráficas: Triángulos

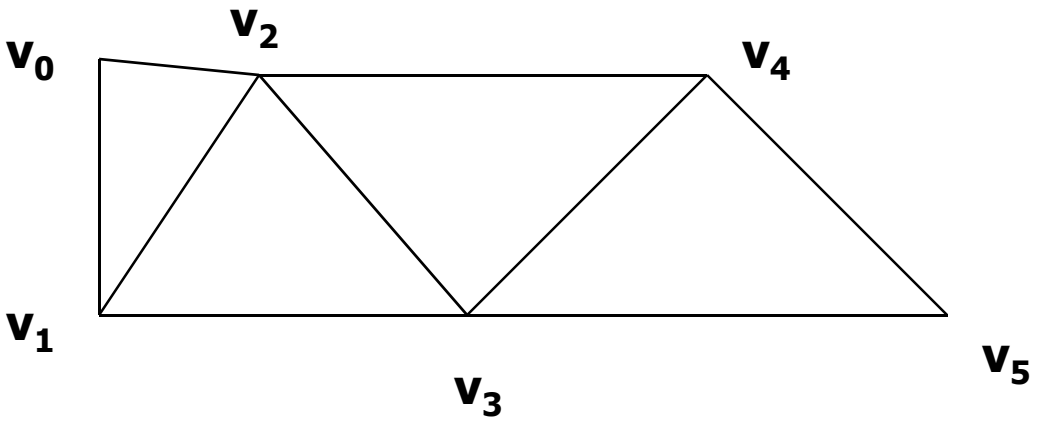
- Para dibujar triángulos debemos usar:

```
glBegin(GL_TRIANGLES);
  glVertex...      // v0
  glVertex..       // v1
  ......
  glVertex..       // vn
glEnd();
```

Pinta las triángulos $\mathbf{v_0v_1v_2}$, $\mathbf{v_3v_4v_5}$, ... Si el número de vértices no es múltiplo de 3, el último (o los dos últimos) vértices se ignoran.

Primitivas gráficas: Triángulos

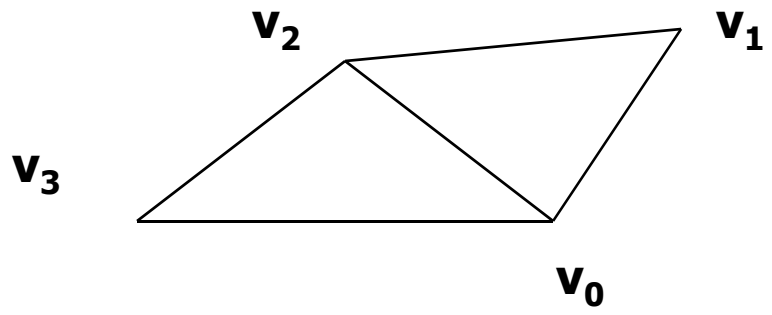
- `GL_TRIANGLE_STRIP`: Dibuja los triángulos $\mathbf{v_0v_1v_2}$, $\mathbf{v_2v_1v_3}$, $\mathbf{v_2v_3v_4}$, ...



El número de vértices ha de ser al menos 3

Primitivas gráficas: Triángulos

- **GL_TRIANGLE_FAN**: Pinta los triángulos $\mathbf{v_0v_1v_2}$, $\mathbf{v_0v_2v_3}$, $\mathbf{v_0v_3v_4}$, ... El número de vértices ha de ser mayor o igual que tres. Todos los triángulos comparten un vértice común $\mathbf{v_0}$.



Todos los triángulos dibujados salen rellenos del color que esté activado

Primitivas gráficas: Cuadriláteros

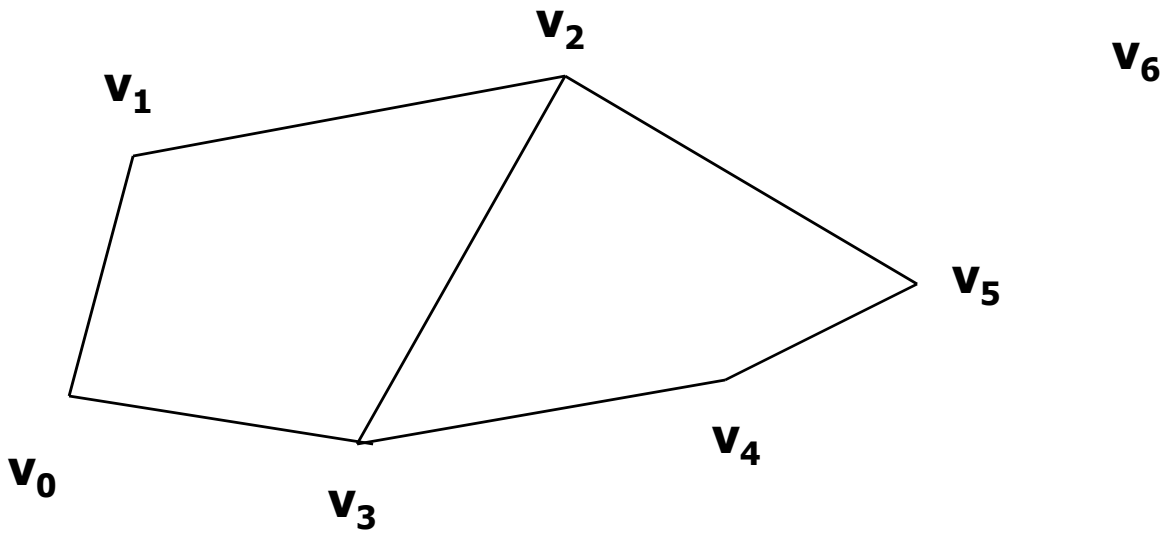
- Para dibujar cuadriláteros debemos usar:

```
glBegin(GL_QUADS);
  glVertex...      // v0
  glVertex..       // v1
  ......
  glVertex..       // vn
glEnd();
```

Pinta los cuadriláteros $\mathbf{v_0v_1v_2v_3}$, $\mathbf{v_4v_5v_6v_7}$, ... Si el número de vértices no es múltiplo de 4, podemos descartar hasta los tres últimos vértices.

Primitivas gráficas: Cuadriláteros

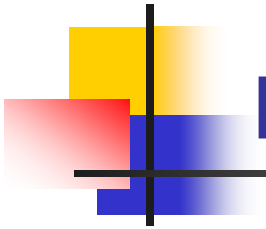
- **GL_QUAD_STRIP**: igual que el anterior, pero conecta un cuadrilátero con otro. Si el número de vértices es impar, el vértice final se ignora.



Primitivas gráficas: Polígonos en general

- Debemos utilizar la constante **GL_POLYGON**. Sólo pinta un polígono, que tiene tantos lados como vértices estén especificados entre glBegin(..) y glEnd().

- Open GL **sólo garantiza un funcionamiento adecuado para el caso de polígonos convexos**. La constante **GL_POLYGON** sólo debe utilizarse para la creación de este tipo de polígonos. En otro caso el comportamiento de Open GL frente a operaciones sobre ese polígono es incontrolable.



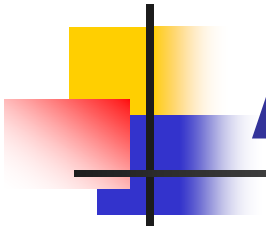
Primitivas gráficas: Consideraciones

- La omisión de **glEnd();** no genera error de compilación, pero impide que se dibuje el gráfico. En general hay que tener cuidado con los errores de sintaxis, porque muchos de ellos no son detectados por el compilador.

- Entre **glBegin(..)** y **glEnd()** puede aparecer información sobre el color, textura, iluminación, etc, de la figura. También pueden aparecer comandos C++.

- La enumeración de los vértices de un polígono conviene realizarla en el sentido contrario de las agujas del reloj.

- Todos los polígonos aparecen rellenos del color que esté activo.



Algunas variables de estado

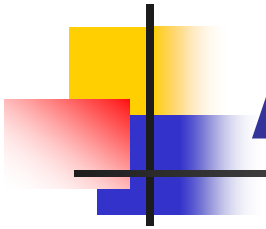
- El tamaño de un punto es una variable de estado que por defecto vale 1.0. Podemos modificar la variable de estado mediante el comando:

```
glPointSize(GLfloat size)
```

`size` debe ser estrictamente mayor que 0.

- El comando **glPointSize** sólo afecta a aquellos vértices definidos con `GL_POINTS` y debe definirse antes del **glBegin** correspondiente. En otro caso no tiene efecto.

- Mientras no se cambie el tamaño del punto con una nueva llamada a **glPointSize**, todos los puntos saldrán de ese tamaño.



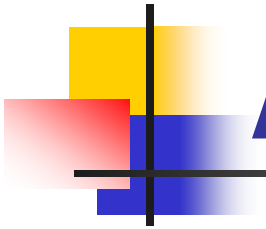
Algunas variables de estado

- El color a utilizar en un gráfico también es una variable de estado. Por defecto el color es blanco ((1.0,1.0,1.0) en el modelo RGB).
- Para modificar la variable de estado, tenemos el comando:

```
glColor3f(GLfloat r, GLfloat g, GLfloat b)
```

donde **r**,**g** y **b** son valores entre 0 y 1, que especifican la cantidad de rojo, verde y azul que deseamos. El color resultante es una mezcla de los tres colores.

(0,0,0): negro	(0,1,0): verde	(1,1,0): amarillo	(0,1,1): cyan
(1,0,0): rojo	(0,0,1): azul	(1,0,1): magenta	(1,1,1): blanco



Algunas variables de estado

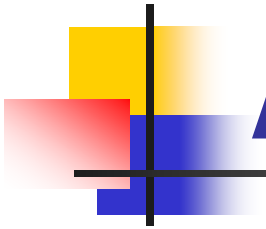
- El color de fondo de la ventana en la que deseamos dibujar podemos modificarlo utilizando el comando:

```
glClearColor(GLclampf r, GLclampf g,
             GLclampf b, GLclampf alfa)
```

Los tres primeros argumentos son similares a los del comando **`glColor3f`**. El último hace referencia al grado de transparencia. Por el momento siempre usaremos 1.0.

- Por defecto, el color de la ventana es **negro**, que es equivalente a ejecutar el comando:

```
glClearColor(0.0,0.0,0.0,1.0)
```

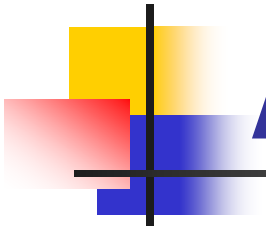


Algunas variables de estado

- Una vez establecido el color de fondo de nuestra ventana, para aplicarlo a la ventana entera debemos utilizar el comando:

```
glClear(GL_COLOR_BUFFER_BIT)
```

- Este comando vacía el buffer donde está almacenado el gráfico. Cada vez que se ejecuta, la ventana pasa a tener enteramente el color del fondo.



Algunas variables de estado

- Podemos modificar el grosor de una línea utilizando el comando:

```
glLineWidth(Glfloat w)
```

- La anchura por defecto de una línea es 1.0.

- Debe aparecer siempre antes del **`glBegin(..)`** correspondiente. El tamaño de línea permanece inalterable hasta que se vuelve a cambiar utilizando este comando.

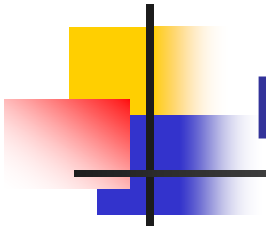
Rectángulos

- En Open GL podemos dibujar un **rectángulo relleno del color activo** y con sus lados paralelos a los ejes de coordenadas utilizando el comando:

```
glRect{sifd}(type x1, type y1, type x2, type y2)
```

- El comando anterior es equivalente a:

```
glBegin(GL_POLYGON);
    glVertex2?(x1,y1); glVertex?(x2,y1);
    glVertex2?(x2,y2); glVertex?(x1,y2);
glEnd();
```



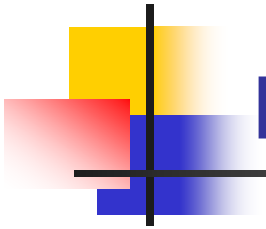
Formas de líneas y polígonos

- Podemos definirnos el tipo de línea que queremos utilizar (discontinua, punteada, etc..) sin más que definirnos el patrón adecuado.
- Primero debemos habilitar el uso de este tipo de líneas (activar la variable de estado) mediante el comando:

```
glEnable(GL_LINE_STIPPLE)
```

- Posteriormente, debemos utilizar el comando:

```
glLineStipple(GLint factor, GLushort pattern)
```



Formas de líneas y polígonos

- `pattern` es una secuencia de 16 bits, donde un **0** indica que no debemos dibujar el píxel, mientras que un **1** indica que el píxel debe ser dibujado.
- La secuencia de bits debe leerse de derecha a izquierda.
- Por ejemplo:

`0011111100000111`

representa: Pintar tres píxeles consecutivos. No pintar los siguientes 5. Pintar los siguientes seis píxeles consecutivos. No pintar los siguientes 2 píxeles.

Formas de líneas y polígonos

- **`factor`** sirve para acortar o agrandar el patrón. Cada secuencia de 0's y 1's consecutivos se multiplica por el factor. En nuestro ejemplo anterior, si tomamos **`factor=2`**, entonces:
  - Pintar 6 píxeles. No pintar 10. Pintar 12 píxeles. No pintar 4.
- Para expresar el patrón se utiliza código hexadecimal. Así, el patrón `0011111100000111` lo expresaríamos como:

`0x3F07`

- **`3 = 0011`** **`F = 1111`** **`0 = 0000`** **`7 = 0111`**

Formas de líneas y polígonos

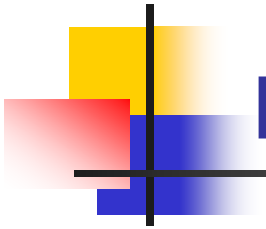
Acrobat Reader - [Hill_02]

File Edit Document View Window Help

pattern	factor	resulting stipple
0xFF00	1	
0xFF00	2	
0x5555	1	
0x3333	2	
0x7733	1	

FIGURE 2.51 Sample stipple patterns.

98% 51 of 55 10.01 x 8.01 in



Formas de líneas y polígonos

- Para deshabilitar el trazado de estas líneas debemos ejecutar:

```
glDisable(GL_LINE_STIPPLE)
```

- También es posible elegir un patrón determinado para rellenar un polígono. Primero debemos habilitar el modo con el comando:

```
glEnable(GL_POLYGON_STIPPLE)
```

- Después debemos usar el comando:

```
glPolygonStipple(const GLubyte *mask)
```

Formas de líneas y polígonos

- **`mask`** es un puntero a un bitmap de 32x32 bits, que se interpreta como una máscara de 0's y 1's. Un 0 indica que no se debe pintar el píxel, mientras que un 1 indica que si se debe pintar el píxel.

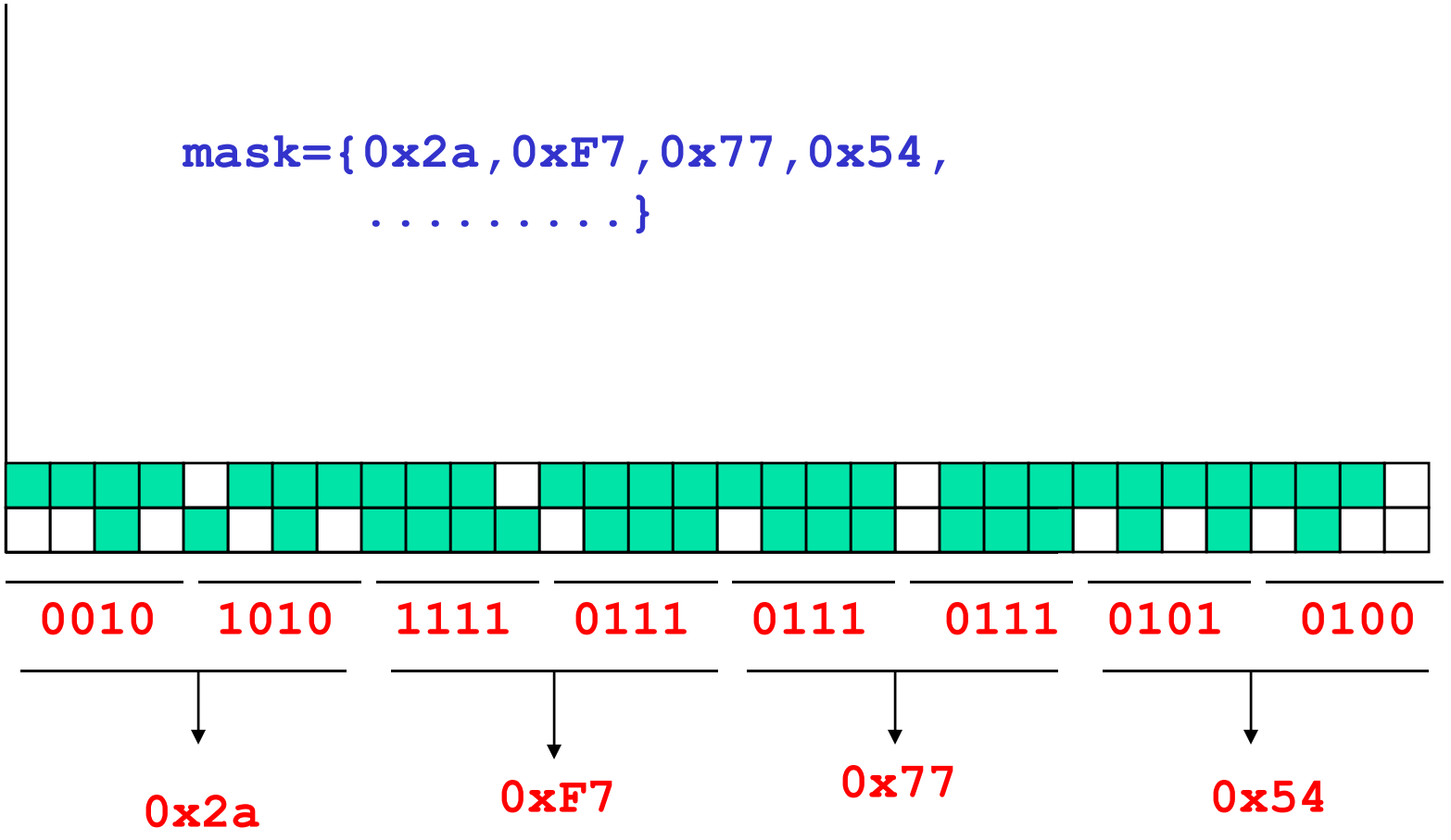
- El dibujo representado por el bitmap debe interpretarse de abajo a arriba y de izquierda a derecha.

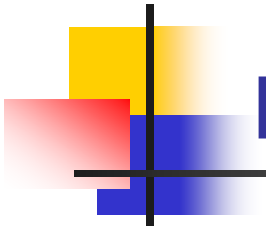
- Cada elemento de la matriz es de 8 bits, y se expresa en **hexadecimal** (**0x??**). Por tanto la matriz tendrá 128 elementos (cada uno un **byte**).

- Para deshabilitar este modo debemos ejecutar:

```
glDisable(GL_POLYGON_STIPPLE)
```

Formas de líneas y polígonos





Formas de líneas y polígonos

```
glPolygonMode(GLenum face,GLenum mode);
```

- Especifica el modo en el cuál aparecerá dibujado un polígono.
- **face** puede ser: `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`
- **mode** especifica como queremos que salga dibujado el polígono: `GL_POINT, GL_LINE, GL_FILL`.