

Introducción a OpenGL

Ana Gil Luezas
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

- ❑ OpenGL (Open Graphics Library) es una API portable que permite la comunicación entre el programador de aplicaciones y el hardware gráfico de la máquina (GPU: Graphics Processing Unit).
- ❑ No es exactamente una API (Application Programming Interface), es una especificación gestionada actualmente por Khronos Group, que implementan los fabricantes de GPUs.
- ❑ OpenGL es una máquina de estados. Tenemos una colección de variables de estado a las que vamos cambiando su valor (el estado se conoce como OpenGL context), y se renderiza sobre el estado actual.

- ❑ No dispone de comandos de alto nivel para cargar imágenes o describir escenas 3D. Tampoco dispone de comandos para gestionar ventanas ni para interactuar con el usuario.
- ❑ Para la gestión de ventanas utilizamos la librería [GLUT \(OpenGL Utility ToolKit\)](#): básica y portable.
- ❑ Para las operaciones matemáticas utilizamos la librería [GLM \(OpenGL Mathematics\)](#): especializada para la programación gráfica.
- ❑ Utilizamos el entorno de desarrollo [VS 2019 C++14](#) con [FreeGLUT](#) y [GLM](#) (plantilla: [proyectosIG_x64_VS2019_FG.zip](#)).
- ❑ Otras utilidades: [GL Image](#), [GL Load](#), ...

- ❑ Todos los comandos OpenGL comienzan con **gl**, y cada una de las palabras que componen el comando comienzan por letra mayúscula (*CamelCase*).

`glClearColor(....)`

`glEnable(...)`

- ❑ Las constantes se escriben en mayúsculas, y comienzan por **GL**. Cada una de las palabras que componen el identificador está separada de la anterior por `_` (SNAKE_CASE).

`GL_DEPTH_TEST`

`GL_COLOR_BUFFER_BIT`

Sintaxis de los comandos OpenGL

- ❑ Existen comandos en OpenGL que admiten distinto número y tipos de argumentos. Estos comandos terminan con el sufijo que indica el tipo de los mismos.

`glColor4ub(GLubyte red, ...)` // 4 (RGBA) unsigned byte

`glColor3d(GLdouble red, ...)` // 3 (RGB) double

`glColor4fv(GLfloat *)` // 4 (RGBA) float*

4fv: indica que el parámetro es un puntero a un array de 4 float

`glLoadMatrixf(const GLfloat * m)`

`glLoadMatrixd(const GLdouble * m)`

Matrixf/d: indica que los parámetros son punteros a un array de 4x4 float/double

- ❑ OpenGL trabaja internamente con tipos básicos específicos que son compatibles con los de C/C++.

Sufijo	Tipo OpenGL
b	GLbyte (entero de 8 bits)
ub	GLubyte (entero sin signo de 8 bits)
s	GLshort (entero con signo de 16 bits)
us	GLushort (entero sin signo de 16 bits)
i	GLint, GLsizei (entero de 32 bits)
ui	GLuint, GLenum (entero sin signo de 32 bits)
f	GLfloat, GLclampf (punto flotante de 32 bits)
d	GLdouble, GLclampd (punto flotante de 64 bits)

GLboolean (GL_TRUE / GL_FALSE)

- ❑ **GLM** ofrece tipos, clases y funciones compatibles con OpenGL, GLSL y C++.

Define el espacio de nombres **glm** y tipos para vectores y matrices

`glm::vec2, glm::vec3, glm::vec4`

`glm::dvec2, glm::ivec3, glm::uvec4, glm::bvec3`

`glm::mat4, glm::dmat4, glm::mat3, glm::dmat3`

- ❑ Para las coordenadas de los vértices de las primitivas gráficas usaremos vectores de `glm::dvec3` (componentes: `v.x`, `v.y`, `v.z`)
- ❑ Para las componentes de los colores RGBA usaremos `glm::dvec4` (componentes `c.r`, `c.g`, `c.b`, `c[0]`, `c[1]`, `c[2]`)
- ❑ Para las matrices `glm::dmat4` `m`: `m[i]` columna *i*-ésima (`dvec4`)
- ❑ Operaciones: `*`, `+`,

- ❑ El **color de fondo** de la ventana en la que vamos a dibujar podemos modificarlo utilizando el comando:

```
glClearColor(GLfloat r, GLfloat g, GLfloat b, GLfloat alpha)
```

Valores de los argumentos en [0,1].

Por ejemplo color de fondo negro:

```
glClearColor(0.0, 0.0, 0.0, 0.0); // valores por defecto
```

- ❑ Función **display()** de la ventana con **doble buffer: Front y Back**:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
// El BackColorBuffer queda del color establecido con glClearColor()
```

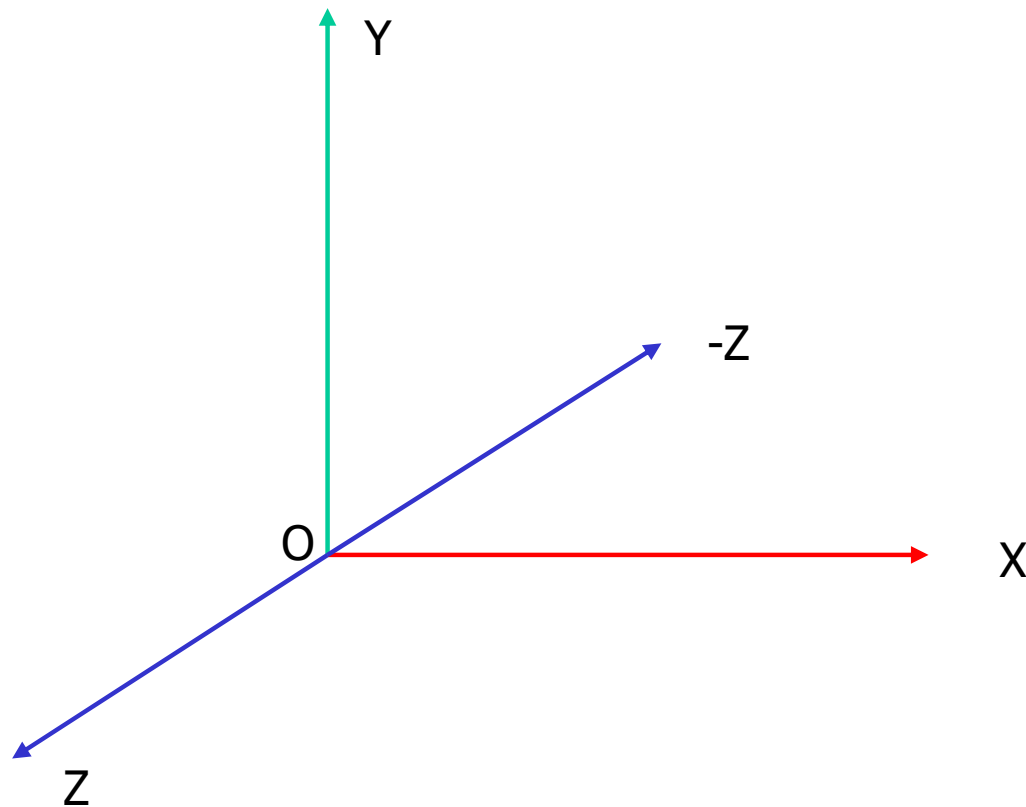
```
// El DepthBuffer queda a 1 (máxima distancia)
```

```
scene.render(); // se dibujan los objetos en el BackColorBuffer
```

```
glutSwapBuffers(); // se intercambian los buffers (Back/Front)
```


Sistema cartesiano en OpenGL

Sistema cartesiano: origen (O)
y tres ejes ortogonales: X, Y, Z



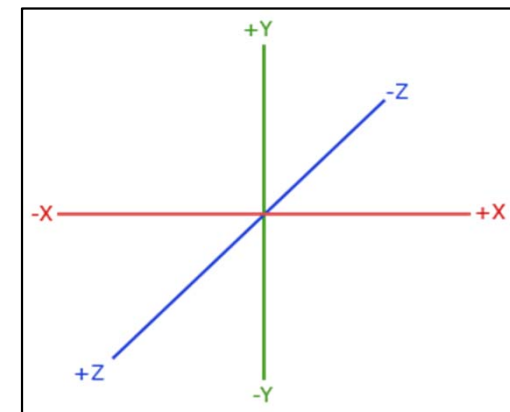
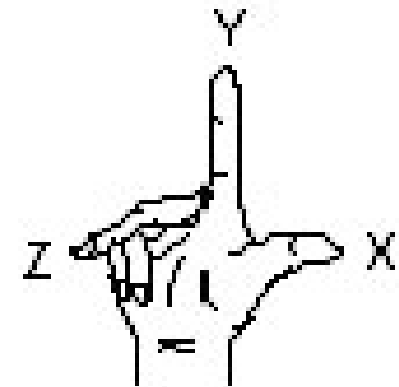
Right-handed system:

Right = X positivo

Up = Y positivo

Backwards = Z positivo

Forwards = Z negativo




- ❑ Por ejemplo, para definir las coordenadas de 4 vértices:

```
GLuint numVertices = 4;  
std::vector<glm::dvec3> vertices .reserve(numVertices);  
vertices.emplace_back(10.0, 0.0, 0.0);  
vertices.emplace_back(0.0, 10.0, 0);  
vertices.emplace_back(0.0, 0.0, 10.0);  
vertices.emplace_back(0.0, 0.0, 0.0);
```

- ❑ Para dibujar puntos (mesh::render) utilizamos la primitiva **GL_POINTS**:

```
glEnableClientState(GL_VERTEX_ARRAY);  
glVertexPointer(3, GL_DOUBLE, 0, vertices.data()); // dvec3  
// nº y tipo de las componentes, paso entre valores, puntero al 1º elemento  
glDrawArrays(GL_POINTS, 0, numVertices);  
glDisableClientState(GL_VERTEX_ARRAY);
```



- ❑ ¿Color y grosor?: Los que estén establecidos en el momento de `glDrawArrays(...)`. OpenGL es una máquina de estados.
- ❑ Para dibujar todos los puntos con un grosor y color determinado: `glPointSize(GLfloat), glColor*(...)`

```
glPointSize(3);  
glColor3d(0.5, 1, 0.25); // -> alpha =1 = opaco  
mesh->render();  
  
glPointSize(1);           // valores por defecto  
glColor4d(1, 1, 1, 1);    // valores por defecto
```

- Si utilizamos la constante `GL_LINES`:

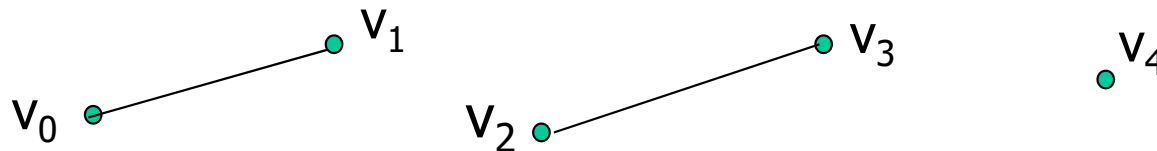
Para `vertices = { \mathbf{v}_0 , \mathbf{v}_1 , \mathbf{v}_2 , \mathbf{v}_3 , ..., \mathbf{v}_{n-1} , \mathbf{v}_n }`

`glDrawArrays(GL_LINES, 0, numVertices);`



Dibuja las líneas $\mathbf{v}_0\mathbf{v}_1$, $\mathbf{v}_2\mathbf{v}_3$, ..., $\mathbf{v}_{n-1}\mathbf{v}_n$.

Si el número de vértices es impar, el último vértice se ignora.

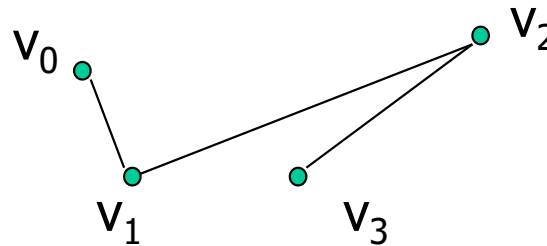


- Atributos de línea: `glLineWidth(GLfloat)`, `glColor*(...)`

Para $\text{vertices} = \{ \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \dots, \mathbf{v}_{n-1}, \mathbf{v}_n \}$

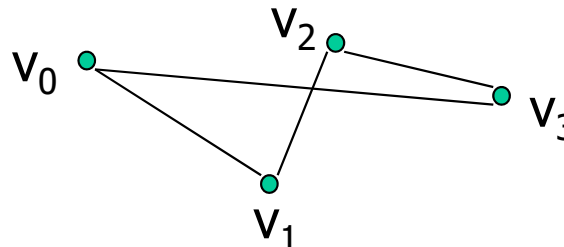
- ❑ Si utilizamos la constante `GL_LINE_STRIP`, las líneas se conectan, i.e., se dibujan las líneas $\mathbf{v}_0\mathbf{v}_1, \mathbf{v}_1\mathbf{v}_2, \mathbf{v}_2\mathbf{v}_3, \dots, \mathbf{v}_{n-1}\mathbf{v}_n$.

Si el número de vértices es 1, no hace nada.



- ❑ Con la constante `GL_LINE_LOOP` la poli-línea se cierra.

Es decir, se dibujan las líneas $\mathbf{v}_0\mathbf{v}_1, \mathbf{v}_1\mathbf{v}_2, \dots, \mathbf{v}_{n-1}\mathbf{v}_n, \mathbf{v}_n\mathbf{v}_0$.



- ❑ Si queremos que cada vértice tenga su propio color, tenemos que asociarlo en la malla.

La dimensión del vector de colores tiene que ser la misma que la de los vértices, y hay que activarlo de forma análoga al vector de vértices.

```
glEnableClientState(GL_COLOR_ARRAY);  
glColorPointer(4, GL_DOUBLE, 0, colores.data()); // dvec4  
// nº y tipo de las componentes, paso entre valores, puntero al 1º elemento  
glEnableClientState(GL_VERTEX_ARRAY);  
glVertexPointer(3, GL_DOUBLE, 0, vertices.data()); // dvec3  
glDrawArrays(GL_POINTS, 0, numVertices); ←  
glDisableClientState(GL_COLOR_ARRAY);  
glDisableClientState(GL_VERTEX_ARRAY);
```

- ❑ La malla **EjesRGB**.

```
generaEjesRGB(GLdouble l);
```

```
int numVertices;  
std::vector<glm::dvec3>  
    vertices;  
std::vector<glm::dvec4>  
    colores;
```

Los vectores tienen que tener
el mismo número de datos

```
{  
    numVertices = 6;  
    vertices.reserve(numVertices);  
    vertices.emplace_back(0, 0, 0);  
    vertices.emplace_back(l, 0, 0);  
    vertices.emplace_back(0, 0, 0);  
    vertices.emplace_back(0, l, 0);  
    vertices.emplace_back(0, 0, 0);  
    vertices.emplace_back(0, 0, l);  
    colors.reserve(numVertices);  
    colores.emplace_back(1, 0, 0);  
    colores.emplace_back(1, 0, 0);  
    colores.emplace_back(0, 1, 0);  
    colores.emplace_back(0, 1, 0);  
    colores.emplace_back(0, 0, 1);  
    colores.emplace_back(0, 0, 1);  
}
```

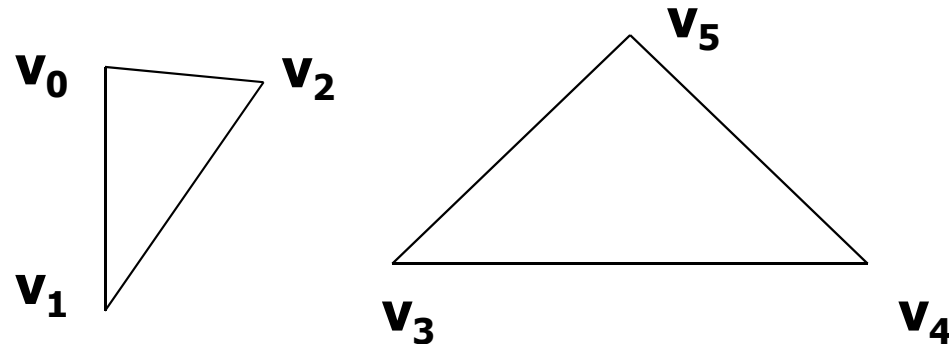
Primitivas gráficas: Triángulos

❑ GL_TRIANGLES: vertices= { $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5$ }

`glDrawArrays(GL_TRIANGLES, 0, numVertices);` ←

Dibuja triángulos
independientes:

$\mathbf{v}_0 \mathbf{v}_1 \mathbf{v}_2, \mathbf{v}_3 \mathbf{v}_4 \mathbf{v}_5$



❑ Los vértices de un triángulo $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ deben estar ordenados en **sentido antihorario** (Counter-Clock Wise). Determina la **cara exterior**.

`glPolygonMode(GLenum face, GLenum mode);`

Especifica el modo en el cuál se rasterizará el polígono.

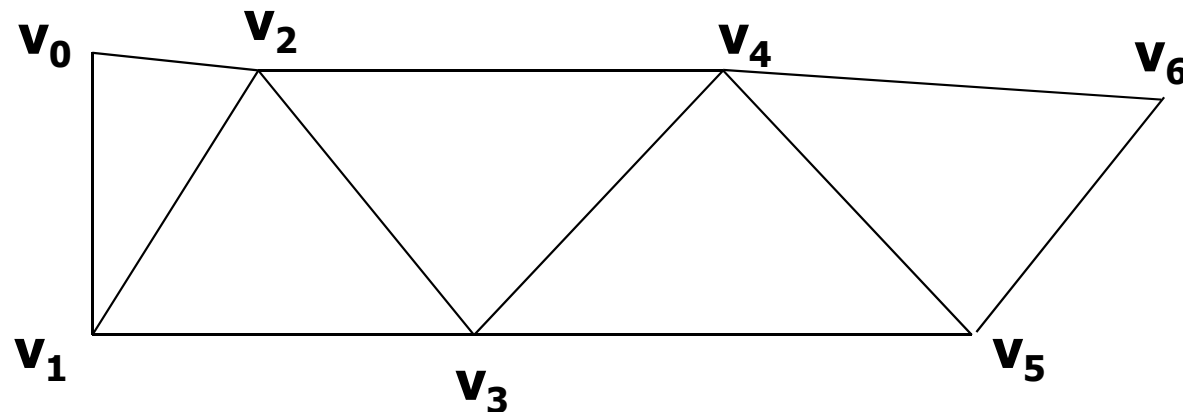
face puede ser: GL_FRONT_AND_BACK, GL_FRONT o GL_BACK

mode puede ser: GL_FILL, GL_LINE o GL_POINT

Primitivas gráficas: Triángulos

❑ `GL_TRIANGLE_STRIP`: `vertices = { $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6$ }`

Dibuja los triángulos: $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$, $\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$, $\mathbf{v}_2\mathbf{v}_3\mathbf{v}_4$, $\mathbf{v}_3\mathbf{v}_4\mathbf{v}_5$, $\mathbf{v}_4\mathbf{v}_5\mathbf{v}_6$
uniformizando el sentido CCW con el del primer triángulo. Por tanto,
dibuja los triángulos: $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$, $\mathbf{v}_2\mathbf{v}_1\mathbf{v}_3$, $\mathbf{v}_2\mathbf{v}_3\mathbf{v}_4$, $\mathbf{v}_4\mathbf{v}_3\mathbf{v}_5$, $\mathbf{v}_4\mathbf{v}_5\mathbf{v}_6$

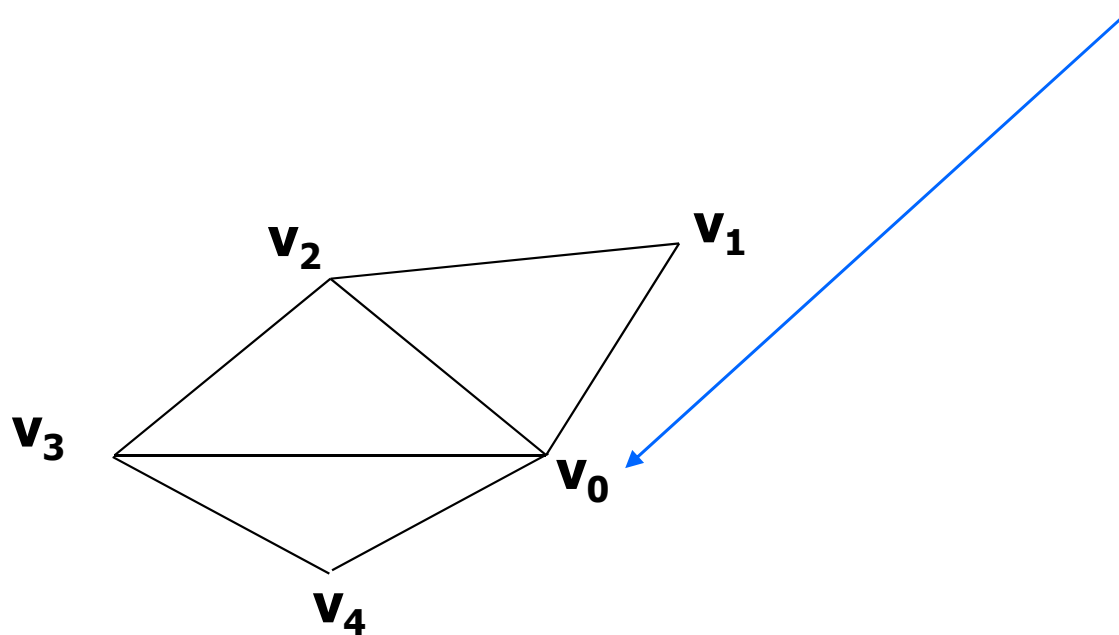


El número de vértices tiene que ser al menos 3

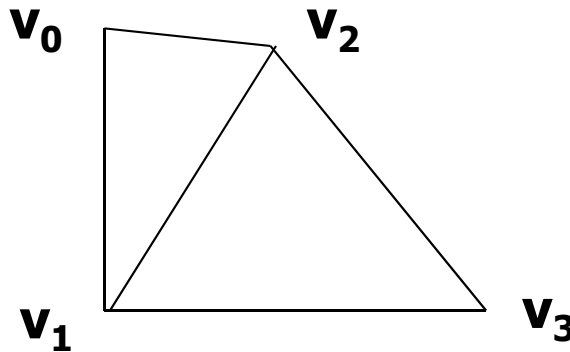
❑ `GL_TRIANGLE_FAN`: `vertices = { \mathbf{v}_0 , \mathbf{v}_1 , \mathbf{v}_2 , \mathbf{v}_3 , \mathbf{v}_4 }`

Dibuja los triángulos $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$, $\mathbf{v}_0\mathbf{v}_2\mathbf{v}_3$, $\mathbf{v}_0\mathbf{v}_3\mathbf{v}_4$

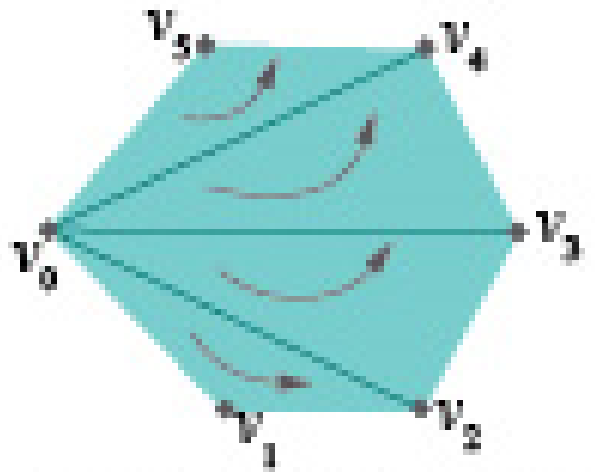
Todos los triángulos comparten un vértice común: \mathbf{v}_0



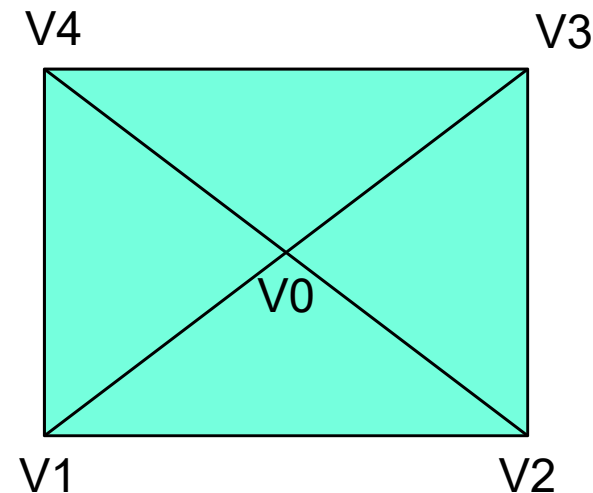
- ❑ Para cuadriláteros utilizamos `GL_TRIANGLE_STRIP`. Para los cuatro vértices del cuadrilátero $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$, dados en el orden $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$, dibuja el cuadrilátero con 2 triángulos: $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ y $\mathbf{v}_2\mathbf{v}_1\mathbf{v}_3$



- ❑ Para polígonos utilizamos `GL_TRIANGLE_FAN` con los vértices del polígono $\mathbf{v_0v_1v_2v_3 \dots v_n}$ en orden contrario a las agujas del reloj



`GL_TRIANGLE_FAN`



`V0, V1, V2, V3, V4, V1`

Matriz del marco cartesiano

$$\begin{matrix} X & Y & Z & O \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{matrix}$$

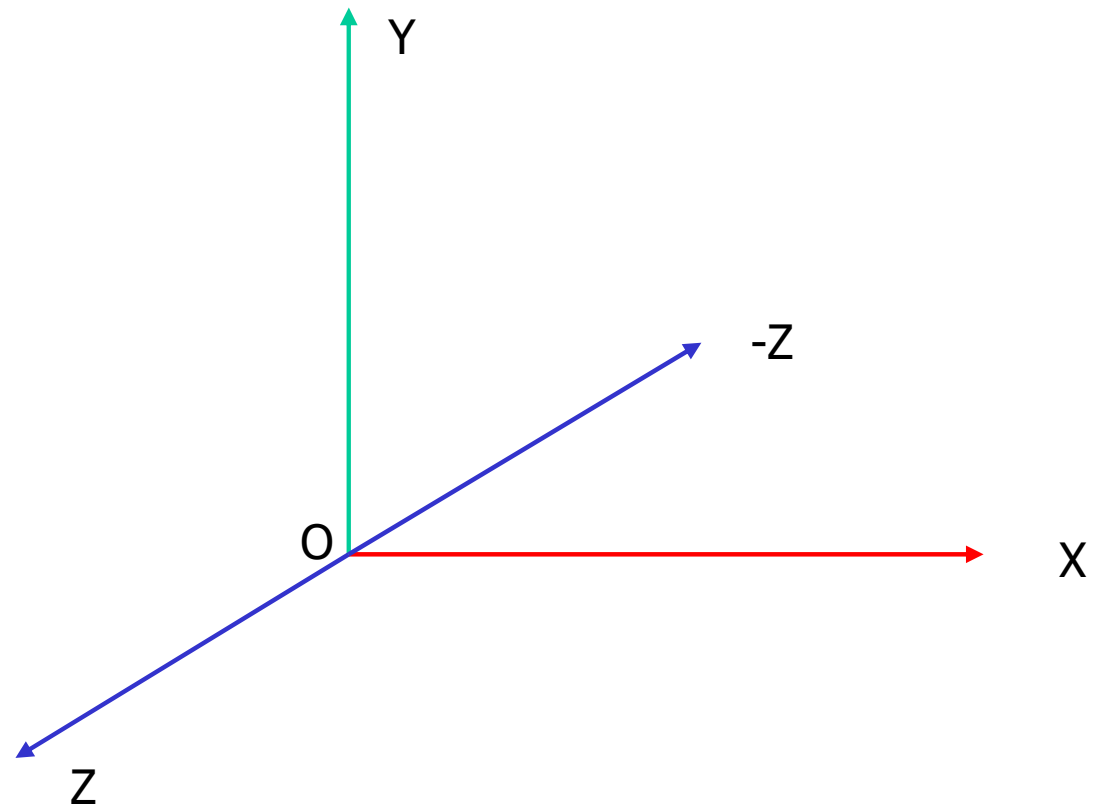
Matriz
identidad

Matrices 4x4 que se aplican
a puntos y vectores en
coordenadas homogéneas:

(x, y, z, w)

$w = 1 \rightarrow$ punto (vértice)

$w = 0 \rightarrow$ vector



En openGL las matrices son 4x4
column-major

- Las **rotaciones**, **traslaciones** y **escalas** se expresan con matrices de la forma

$$F = \left(\begin{array}{c|c} M & T \\ \hline 0 & 1 \end{array} \right) = \begin{pmatrix} M_{11} & M_{12} & M_{13} & T_x \\ M_{21} & M_{22} & M_{23} & T_y \\ M_{31} & M_{32} & M_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Composición.** Las transformaciones se pueden componer multiplicando las matrices.
El producto de matrices es asociativo pero no conmutativo

$$(M1 * M2) * V = M1 * (M2 * V) \qquad M1 * M2 \neq M2 * M1$$

- Las escalas uniformes cambian el tamaño del objeto, las traslaciones cambian la posición del objeto y las rotaciones la orientación, sin deformar el objeto (transformaciones rígidas).
La escala no uniforme deforma el objeto.

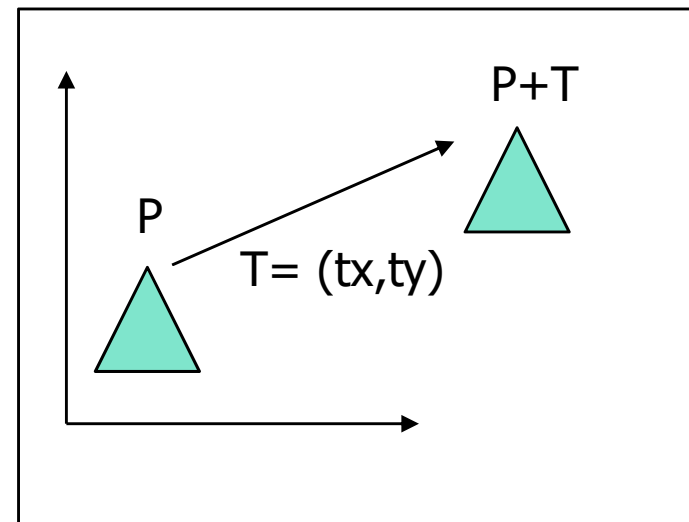
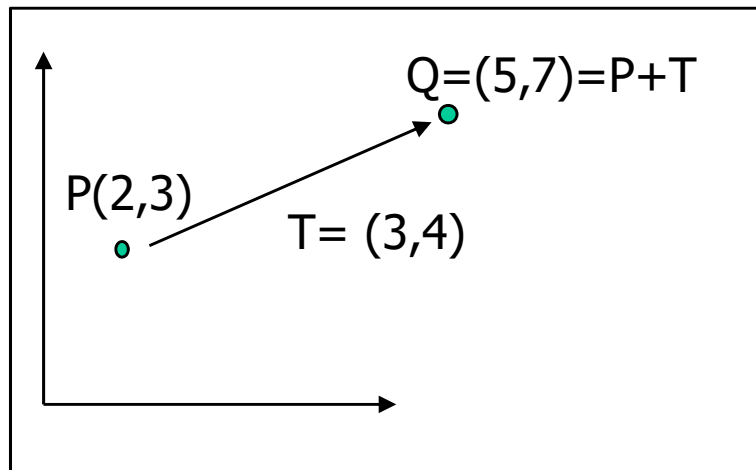
❑ Traslación con vector $T=(tx, ty, tz)$:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} P_x + tx \\ P_y + ty \\ P_z + tz \\ 1 \end{pmatrix}$$

↑
↑

Coordenadas de un punto P

 Coordenadas del punto P una vez trasladado



❏ `glm::dmat4 m = glm::translate(dmat4, dvec3);`

`mT = translate (dmat4(1), dvec3(tx, ty, tz));`

`dmat4(1)`: matriz
identidad (`mI`)

$$mT = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`mT`: matriz de
traslación

Composición:

`m = translate(mat, dvec3(tx, ty, tz));` \rightarrow `m = mat * mT`
 $m * V = (mat * mT) * V = mat * (mT * V)$

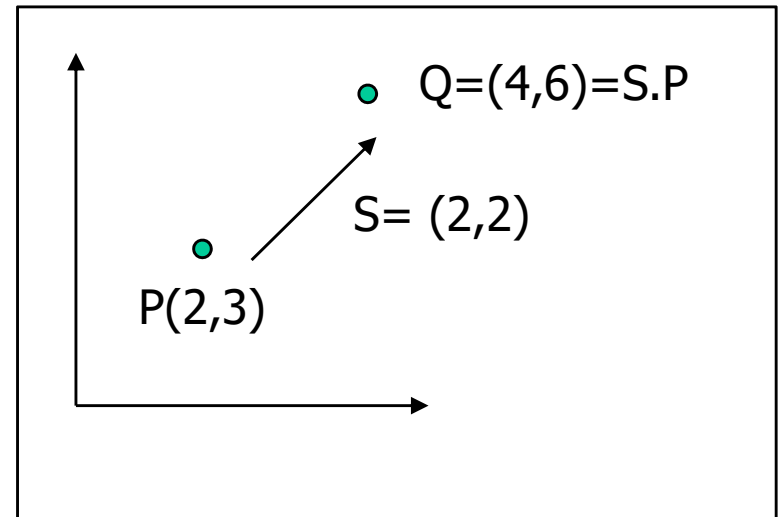
❑ Escala con factor $S=(s_x, s_y, s_z)$:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x P_x \\ s_y P_y \\ s_z P_z \\ 1 \end{pmatrix}$$

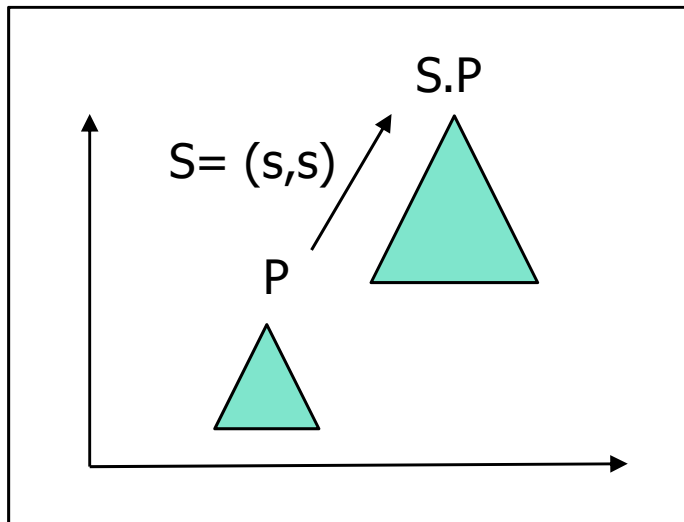
Coordenadas de un punto P

Coordenadas del punto P una vez escalado

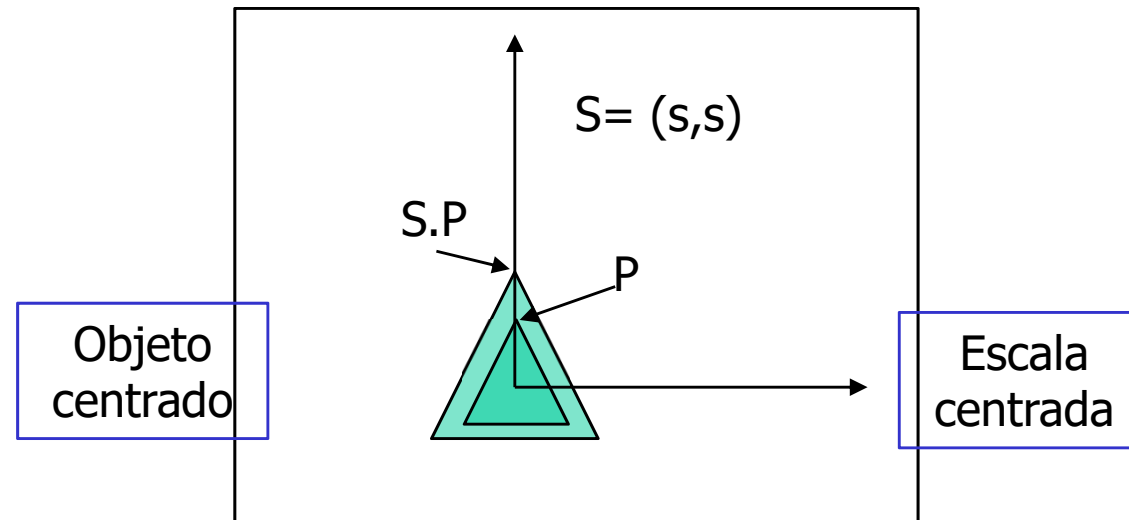
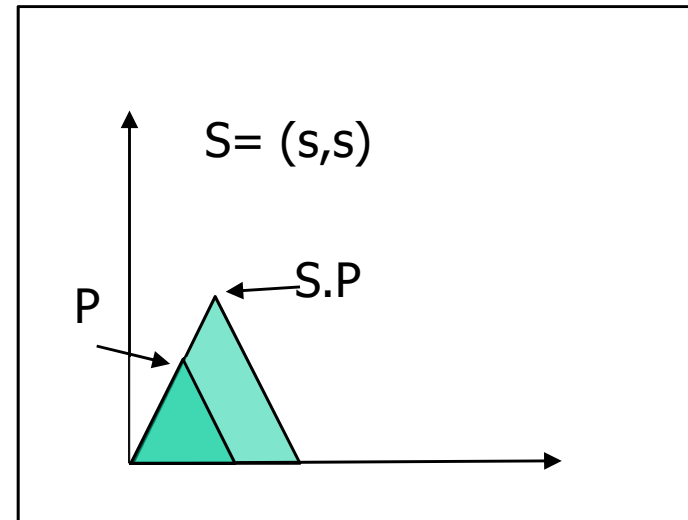
La **escala** es **uniforme** si
 $s_x = s_y = s_z$



❑ Escala con factor $S=(s, s, s)$:



Objeto no centrado



```
glm::dmat4 m = glm::scale(dmat4, dvec3);
```

```
mS = scale(dmat4(1), dvec3(sx, sy, sz));
```

$$mS = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

mS: matriz de
escala

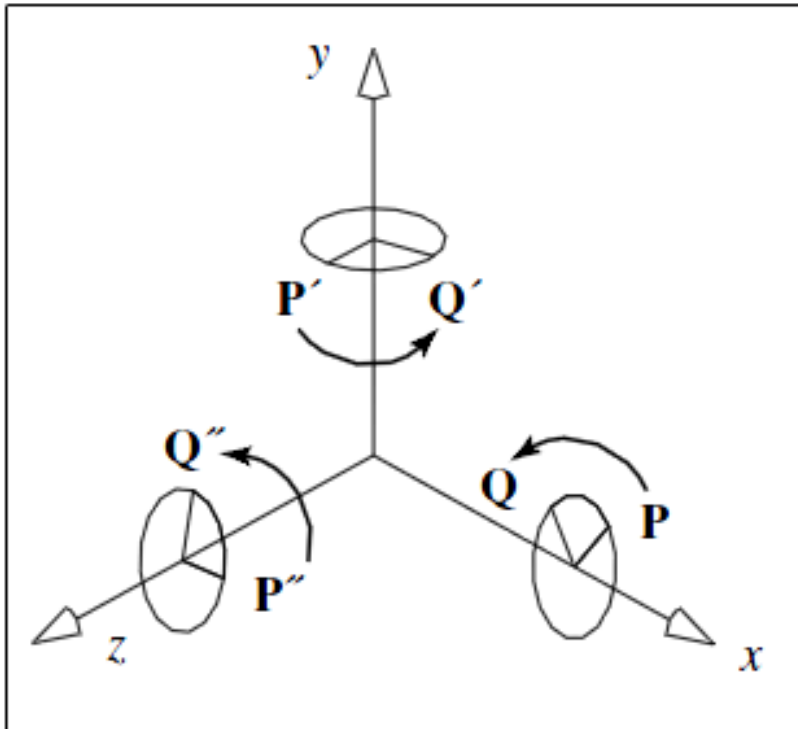
Composición:

```
m = scale(mat, dvec3(sx, sy, sz)); → m = mat * mS
```

$$m * V = (mat * mS) * V = mat * (mS * V)$$

❑ Rotaciones elementales sobre los ejes:

`glm::dmat4 m = glm::rotate(dmat4, β , dvec3);` // β en radianes



ángulo positivo -> giro **CCW**
(antihorario)

Rotación sobre el eje Z (Z-Roll)

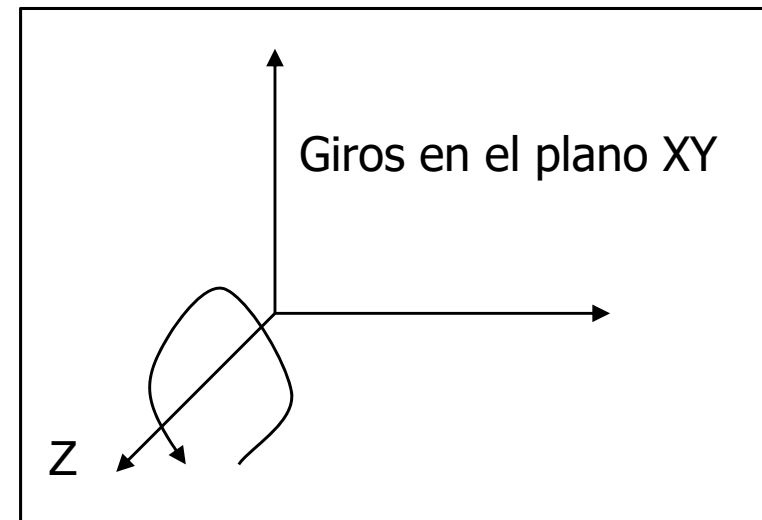
- Una rotación sobre el eje Z de θ radianes:

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta)Px - \sin(\theta)Py \\ \sin(\theta)Px + \cos(\theta)Py \\ Pz \\ 1 \end{pmatrix}$$

↑
Coordenadas del punto P una vez rotado

↑
Coordenadas de un punto P

ángulo positivo -> giro CCW
(antihorario)



❑ `glm::dmat4 m = glm::rotate(dmat4, β, dvec3);`
`// β en radianes`

mR: matriz de rotación

Z-Roll: `m = rotate(dmat4, β, dvec3(0, 0, 1));`
`mR = rotate(dmat4(1), β, dvec3(0, 0, 1)); →`

$$\begin{pmatrix} \cos \beta & -\operatorname{sen} \beta & 0 & 0 \\ \operatorname{sen} \beta & \cos \beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Y-Roll: `m = rotate(dmat4, β, dvec3(0, 1, 0));`
`mY = rotate(dmat4(1), β, dvec3(0, 1, 0)); →`

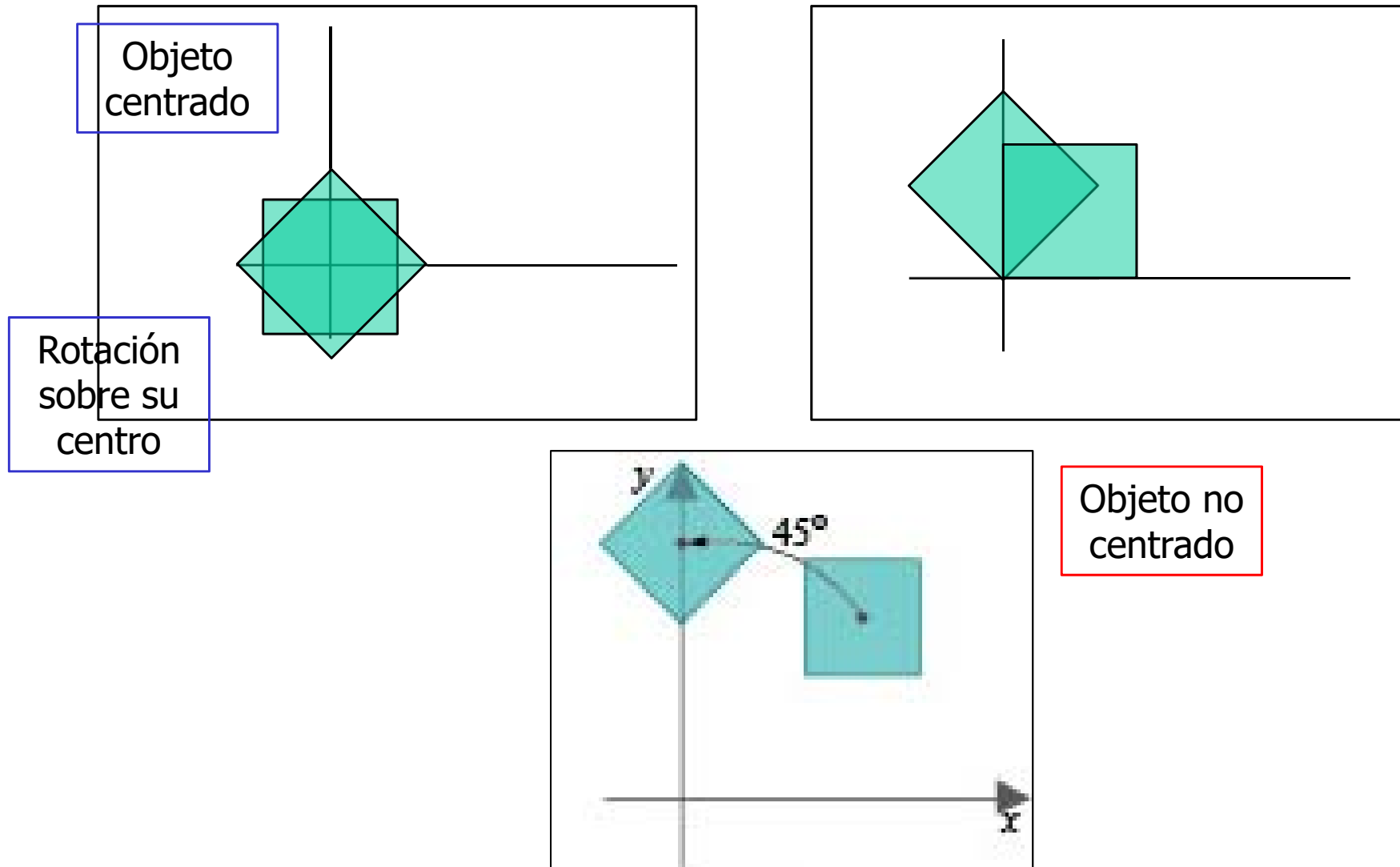
$$\begin{pmatrix} \cos \beta & 0 & \operatorname{sen} \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\operatorname{sen} \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

X-Roll: `m = rotate(dmat4, β, dvec3(1, 0, 0));`
`mP = rotate(dmat4(1), β, dvec3(1, 0, 0)); →`

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \beta & -\operatorname{sen} \beta & 0 \\ 0 & \operatorname{sen} \beta & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotación sobre el eje Z (Z-Roll)

- ❑ Una rotación sobre el eje Z de 45 grados :



Composición de transformaciones

- En OpenGL (GLM) las transformaciones se componen postmultiplicando las matrices:

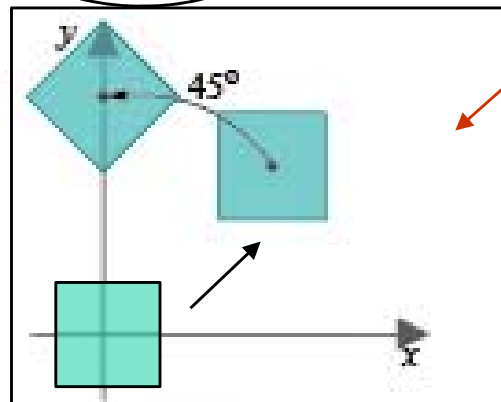
$$mC = \text{transformar}(m, \dots); \rightarrow mC = m * mA$$

Al aplicar mC a un vértice: $mC * V = (m * mA) * V = m * (mA * V)$

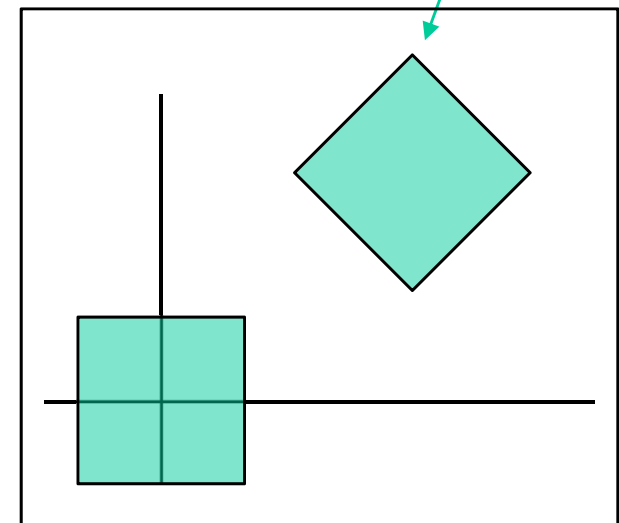
Ejemplo:

Tenemos un cuadrado centrado y alineado con los ejes, y queremos situarlo en el punto (7.5, 7.5, 0) girado 45 grados sobre su centro

```
m = rotate(mI, radians(45.0), dvec3(0,0,1));  
m = translate(m, dvec3(7.5,7.5,0));  
→ m = mI * mR * mT
```



**CUIDADO!!!
MAL**



Composición de transformaciones

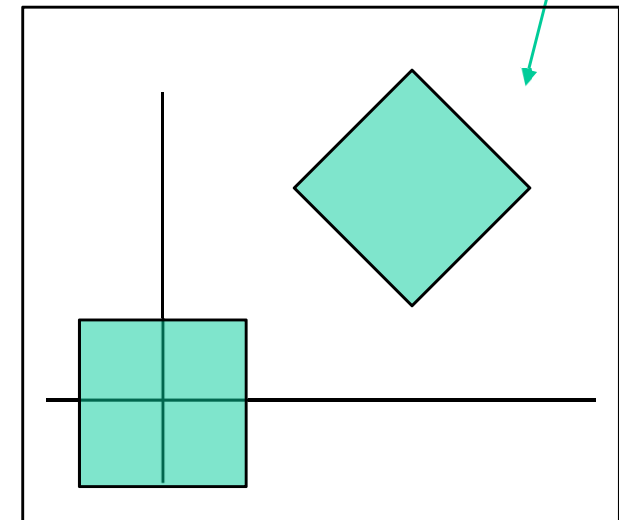
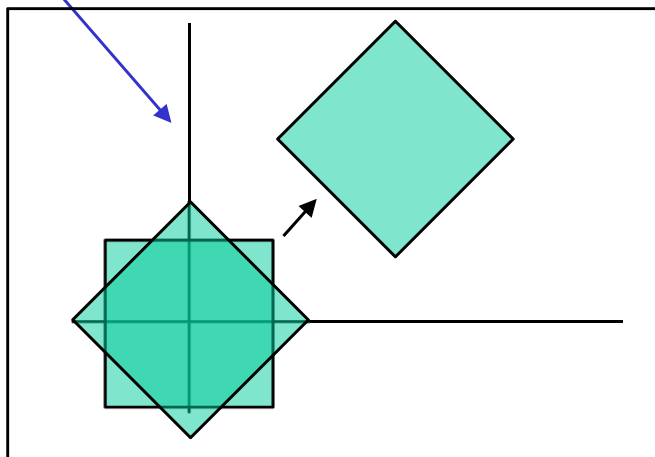
- En OpenGL (GLM) las transformaciones se componen postmultiplicando las matrices:

Al aplicar mC a un vértice: $mC * V = (m * mA) * V = m * (mA * V)$

Ejemplo:

Tenemos un cuadrado centrado y alineado con los ejes, y queremos situarlo en el punto (7.5, 7.5, 0) girado 45 grados sobre su centro

```
m = translate(mI, dvec3(7.5,7.5,0));  
m = rotate(m, radians(45.0), dvec3(0,0,1));  
→ m = mI * mT * mR
```



```
mT = translate(mI, dvec3(7.5,7.5,0));  
mR = rotate(mI, radians(45.0), dvec3(0,0,1));  
m = mT * mR ;
```

Composición de transformaciones

- ❑ **Animación de objetos:** es habitual trabajar con una ruta por la que se **desplaza** el objeto (**mT**) y la **orientación** del objeto (**mR**) mientras se desplaza. Se sitúa y orienta al objeto desde sus coordenadas originales.

Estando el **objeto centrado en coordenadas locales**

$$\text{matriz de modelado del objeto} = mT * mR$$

Si es necesario **escalar** el objeto (**mS**):

$$\text{matriz de modelado del objeto} = mT * mR * mS$$

- ❑ Y por último la **matriz de vista**:

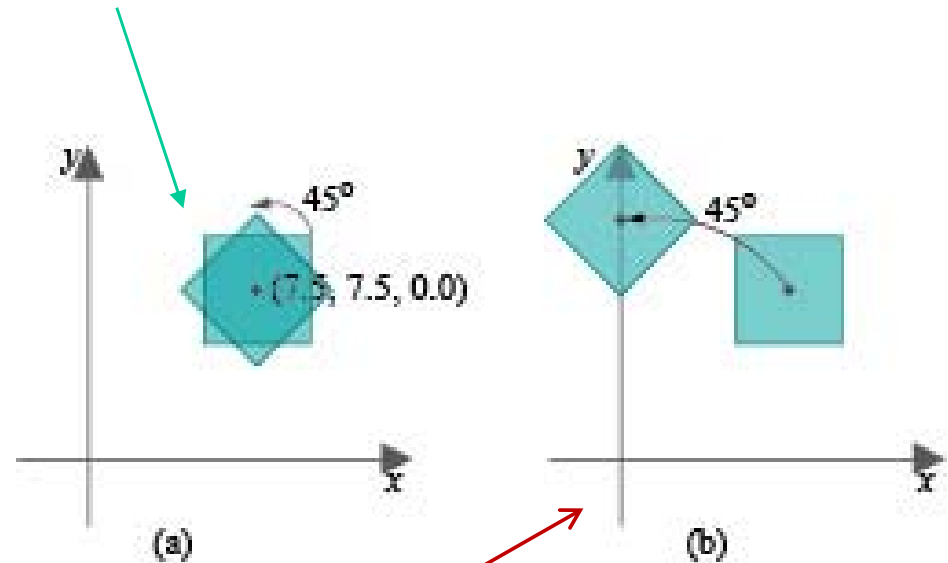
$$\text{matriz de modelado y vista} = \text{matriz de vista} * \text{matriz de modelado}$$

La **matriz de vista** es la inversa de la **matriz de modelado de la cámara**: en lugar de colocar la cámara en la escena (matriz de la cámara) coloca los objetos de la escena con respecto a la cámara.

Composición de transformaciones

- ❑ **Ejemplo:** Tenemos un cuadrado alineado con los ejes con centro en $(7.5, 7.5, 0.0)$

Queremos rotarlo 45 grados sobre su centro (a):




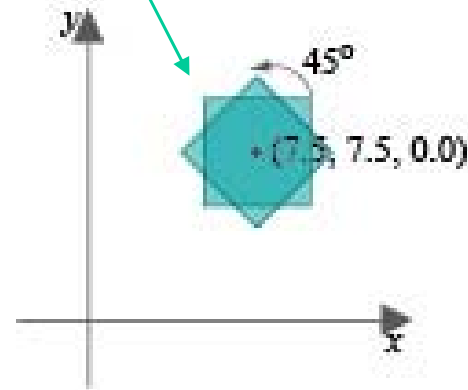
```
m = rotate(mI, radians(45.0), dvec3(0,0,1));  
→ m = mI * mR
```

**CUIDADO!!!
MAL**

Composición de transformaciones

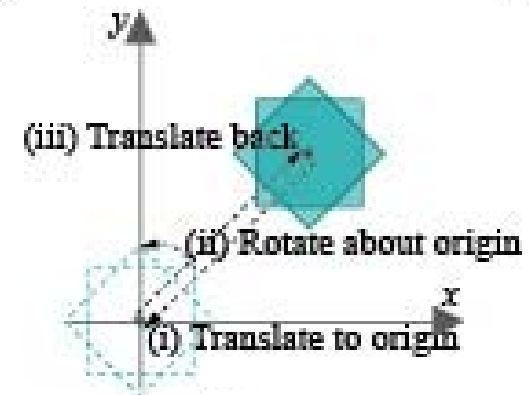
- ❑ **Ejemplo:** Tenemos un cuadrado, alineado con los ejes, con centro en $(7.5, 7.5, 0.0)$

Queremos rotarlo 45 grados sobre su centro: 



```
m = translate(mI, dvec3(7.5,7.5,0));  
m = rotate(m, radians(45.0), dvec3(0,0,1));  
m = translate(m, dvec3(-7.5,-7.5,0));  
→ m = mI * mT * mR * mT-1
```

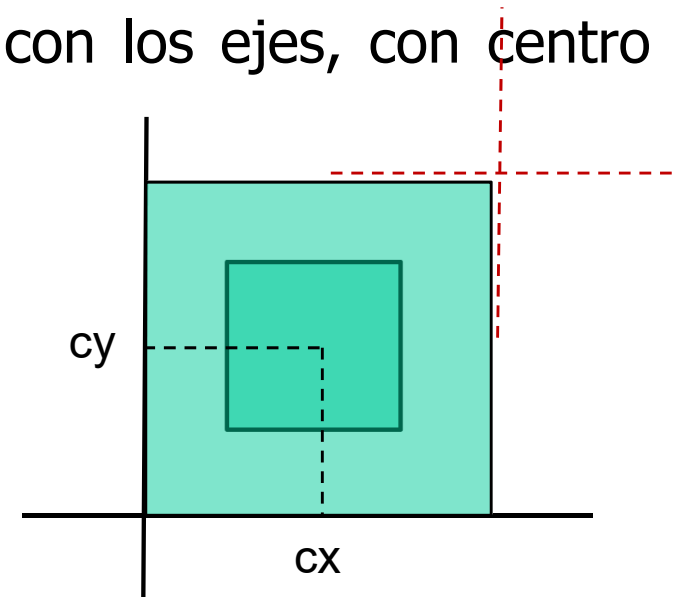
```
mT = translate(mI, dvec3(7.5,7.5,0));  
mR = rotate(mI, radians(45.0), dvec3(0,0,1));  
mTi = translate(mI, dvec3(-7.5,-7.5,0));  
m = mT * mR * mTi ;
```



Composición de transformaciones

- ❑ **Ejemplo:** Tenemos un cuadrado, alineado con los ejes, con centro en $(cx, cy, 0.0)$.

Queremos escalarlo sobre su centro
(sin modificar el centro)



```
m = scale(mI, dvec3(2,2,2));  
→ m = mI * mS  CUIDADO!!!  
MAL
```

```
m = translate(mI, dvec3(cx,cy,0));  
m = scale(m, dvec3(2,2,2));  
m = translate(m, dvec3(-cx,-cy,0));  
→ m = mI * mT * mS * mT-1
```

```
mT = translate(mI, dvec3(cx,cy,0));  
mS = scale(mI, dvec3(2,2,2));  
mTi = translate(mI, dvec3(-cx,-cy,0));  
m = mT * mS * mTi ;
```

$$\text{ModelView MATRIX} = \text{VIEW MATRIX} * \text{MODEL MATRIX}$$

