

Introducción a OpenGL

Informática Gráfica I

Material de:

Ana Gil Luezas

Adaptado por:

Elena Gómez y Rubén Rubio

{mariaelena.gomez, rubenrub}@ucm.es



Contenido

1 Open Graphics Library

2 Primitivas gráficas

- Puntos
- Líneas
- Triángulos
- Cuadriláteros
- Cuadriláteros
- Polígonos

3 Sistema cartesiano

- Traslaciones
- Escalas
- Rotaciones
- Composición

Open Graphics Library

- **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.

OpenGL SDK

- 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 biblioteca **GLUT (OpenGL Utility ToolKit)**: básica y portable.
- Para las operaciones matemáticas utilizamos la biblioteca **GLM (OpenGL Mathematics)**: especializada para la programación gráfica.
- Utilizamos el entorno de desarrollo **VS 2022 C++20** con **FreeGLUT** y **GLM** (plantilla en el campus).
- Otras utilidades: **GL Image**, **GL Load**, ...

Sintaxis de los comandos OpenGL

- 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

Tipos básicos de OpenGL

- OpenGL trabaja internamente con tipos básicos específicos que son compatibles con los de C/C++. Además de `GLboolean` (`GL_TRUE` / `GL_FALSE`),

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

Tipos de GLM

- **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: *, +,

Ventana y frame buffer

- 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:

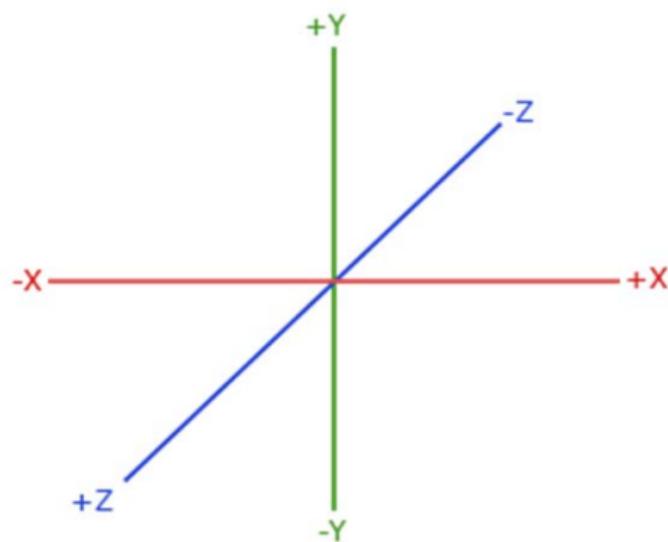
```
glClearColor(0.0, 0.0, 0.0, 0.0); // valores por defecto
```

- 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 fijado con glClearColor()
// El DepthBuffer queda a 1 (máxima distancia)
scene.render();    // dibuja los objetos en el BackColorBuffer
glutSwapBuffers(); // intercambia los buffers (Back/Front)
```

Sistema cartesiano en OpenGL

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



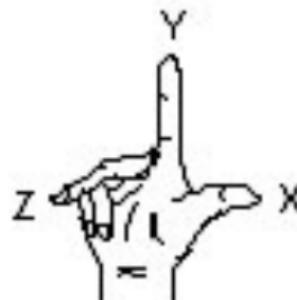
Right-handed system:

Right = X positivo

Up = Y positivo

Backwards = Z positivo

Forwards = Z negativo



Primitivas gráficas: Puntos

- 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  
// número y tipo de las componentes, paso entre valores,  
// puntero al primer elemento  
glDrawArrays(GL_POINTS, 0, numVertices);  
glDisableClientState(GL_VERTEX_ARRAY);
```

Primitivas gráficas: Atributos

- ¿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
```

Primitivas gráficas: Líneas

- Si utilizamos la constante `GL_LINES`:

Para vértices = $\{v_0, v_1, v_2, v_3, \dots, v_{n-1}, v_n\}$

```
glDrawArrays(GL_LINES, 0, numVertices);
```

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



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

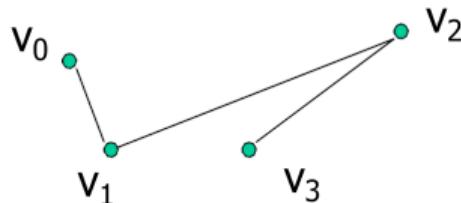
Primitivas gráficas: líneas

Para vértices = $\{v_0, v_1, v_2, v_3, \dots, v_{n-1}, v_n\}$

- Si utilizamos la constante `GL_LINE_STRIP`, las líneas se conectan, i.e., se dibujan las líneas

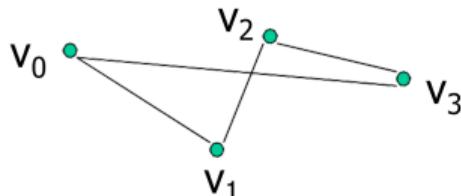
$v_0v_1, v_2v_3, \dots, v_{n-1}v_n$.

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 $v_0v_1, v_1v_2, \dots, v_{n-1}v_n, v_nv_0$



Ejemplo: Ejes RGB

- La malla EjesRGB.

```
generaEjesRGB(GLdouble l);

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

}

```

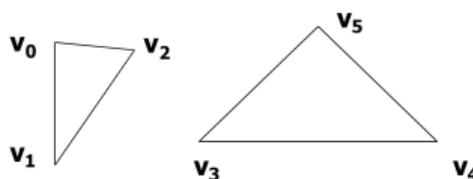
    numVertices = 6;
    vertices.reserve(numVertices);
    vertices.emplace_back(0, 0, 0);
    vertices.emplace_back(1, 0, 0);
    vertices.emplace_back(0, 0, 0);
    vertices.emplace_back(0, 1, 0);
    vertices.emplace_back(0, 0, 0);
    vertices.emplace_back(0, 0, 1);
    colores.reserve(numVertices);
    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`: vértices= $\{v_0, v_1, v_2, v_3, v_4, v_5\}$

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

Dibuja triángulos independientes: $v_0v_1v_2$, $v_3v_4v_5$



- Los vértices de un triángulo $v_0v_1v_2$ deben estar ordenados en sentido **anti-horario** (**Counter-Clock Wise**). Determina la **cara exterior**.

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

Especifica el modo en el cuál se rasterizará el polígono.

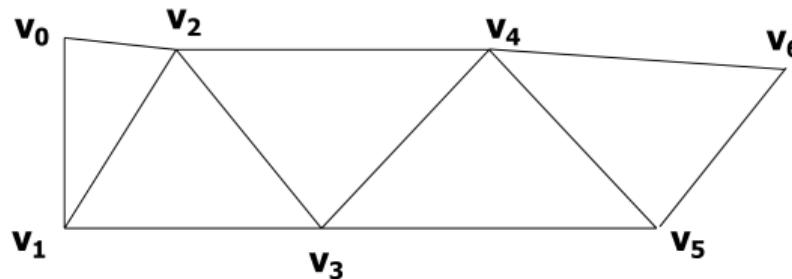
- **face** puede ser: `GL_FRONT_AND_BACK`, `GL_FRONT` o `GL_BACK`.
- **mode** puede ser: `GL_FILL`, `GL_LINE` o `GL_POINT`.

Primitivas gráficas: Triángulos

- **GL_TRIANGLE_STRIP:** vértices= $\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$

Dibuja los triángulos: $v_0v_1v_2$, $v_1v_2v_3$, $v_2v_3v_4$, $v_3v_4v_5$, $v_4v_5v_6$ uniformizando el sentido CCW con el del primer triángulo.

Por tanto, dibuja los triángulos: $v_0v_1v_2$, $v_2v_1v_3$, $v_2v_3v_4$, $v_4v_3v_5$, $v_4v_5v_6$.



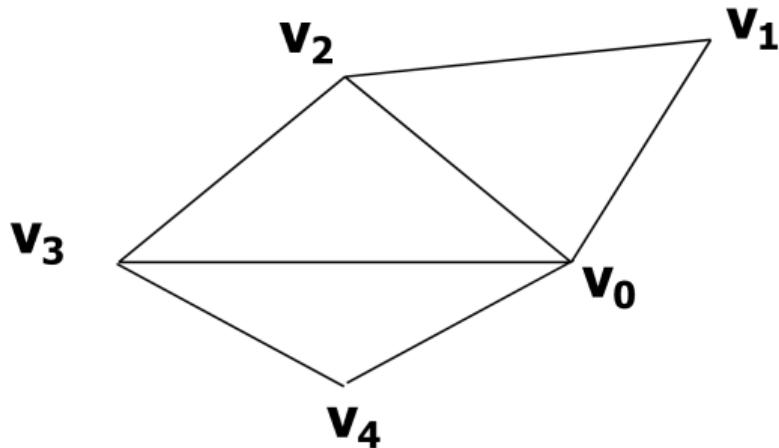
- El número de vértices tiene que ser al menos 3.

Primitivas gráficas: Triángulos

- **GL_TRIANGLE_FAN:** vértices= $\{v_0, v_1, v_2, v_3, v_4\}$

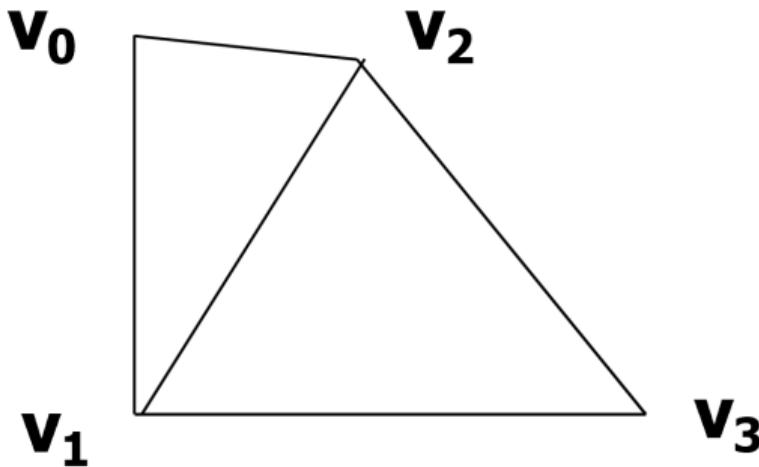
Dibuja los triángulos: $v_0v_1v_2$, $v_0v_2v_3$, $v_0v_3v_4$.

Todos los triángulos comparten un vértice común: v_0 .



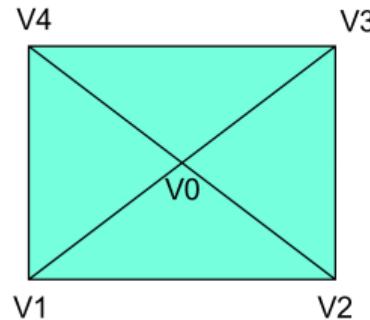
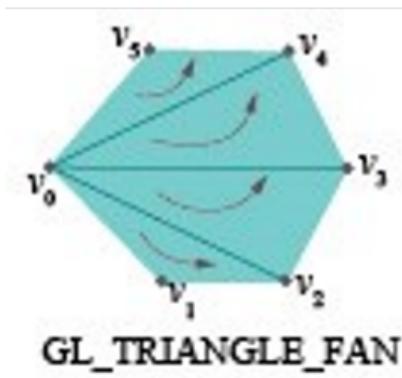
Primitivas gráficas: Cuadriláteros

- Para cuadriláteros utilizamos `GL_TRIANGLE_STRIP`. Para los cuatro vértices del cuadrilátero $v_0v_1v_2v_3$, dados en el orden $v_0v_1v_2v_3$, dibuja el cuadrilátero con 2 triángulos: $v_0v_1v_2$ y $v_2v_1v_3$.



Primitivas gráficas: Polígonos

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



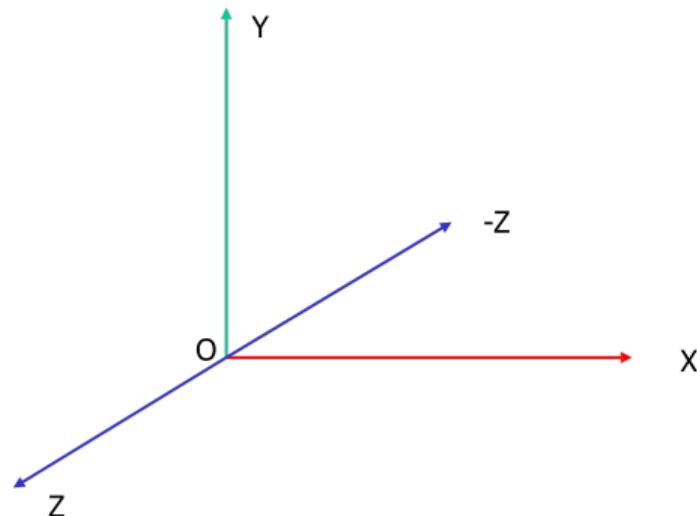
`V0, V1, V2, V3, V4, V1`

Sistema cartesiano de OpenGL

Matriz del marco cartesiano

Matriz identidad

$$\begin{pmatrix} x & y & z & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Matrices 4×4 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 4×4 **column-major**.

Transformaciones afines

- 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:

$$(M_1 \times M_2) \times V = M_1 \times (M_2 \times V), \text{ pero } M_1 \times M_2 \neq M_2 \times M_1$$

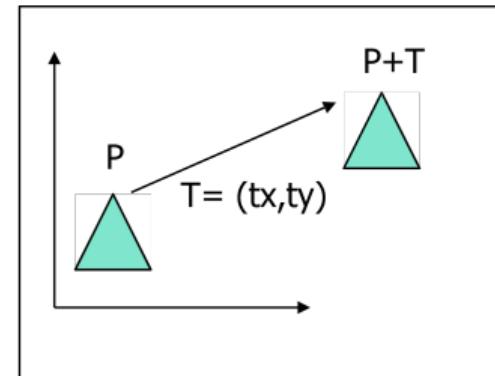
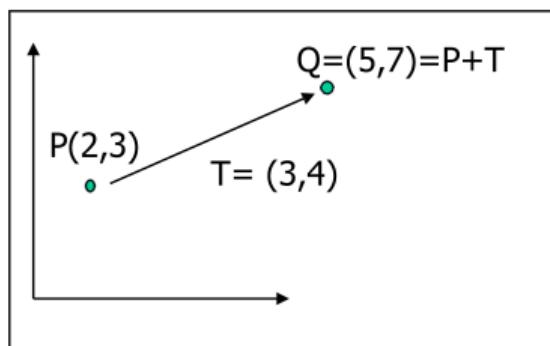
- Las escalas uniformes cambian el tamaño del objeto, las traslaciones cambian la posición del objeto y las rotaciones la orientación, sin deformar el objeto (transformaciones rígidas). La escala no uniforme puede deformar el objeto.

Traslaciones

- **Traslación con vector $T = (t_x, t_y, t_z)$:**

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix}$$

siendo $Q = (x', y', z')$ las coordenadas del punto $P = (x, y, z)$ una vez trasladado



Traslaciones con GLM

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

`mT = translate (dmat4(1), dvec3(tx, ty, tz));` dmat4(1): la matriz identidad ([ml](#))

$$mT = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`mT`: matriz de traslación

- Composición:

`m = translate(mat, dvec3(tx, ty, tz));`

$$m = mat \times mT$$

$$m \times V = (mat \times mT) \times V = mat \times (mT \times V)$$

Escalas

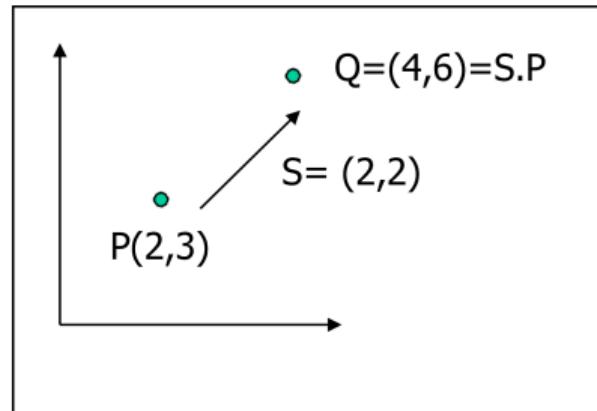
- **Escala** con **factor** $S = (s_x, s_y, s_z)$:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \\ s_z \cdot z \\ 1 \end{pmatrix}$$

siendo $Q = (x', y', z')$ las coordenadas del punto $P = (x, y, z)$ una vez escalado

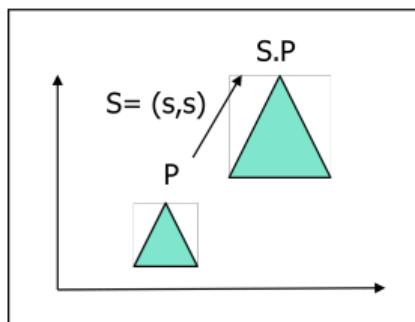
La escala es **uniforme** si

$$s_x = s_y = s_z$$

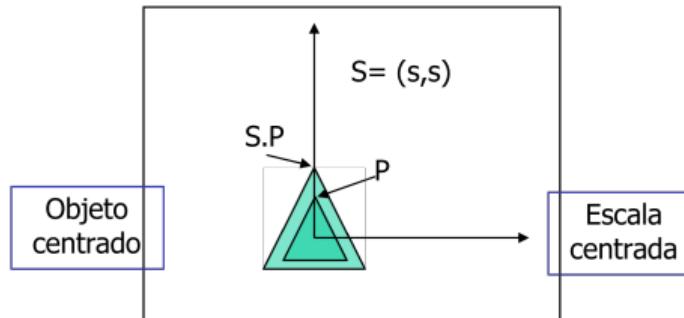
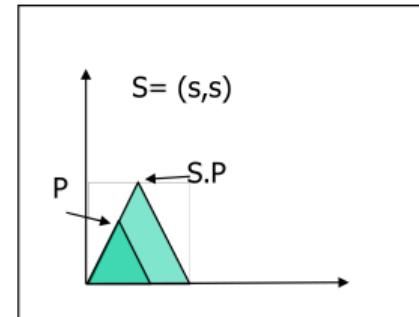


Escalas

- **Escala** con factor $S = (s, s, s)$:



Objeto no centrado



Escalas con GLM

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

$mS = \text{scale}(\text{dmat4}(1), \text{dvec3}(sx, sy, sz));$

$$mS = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

mS : matriz de escala

- Composición:

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

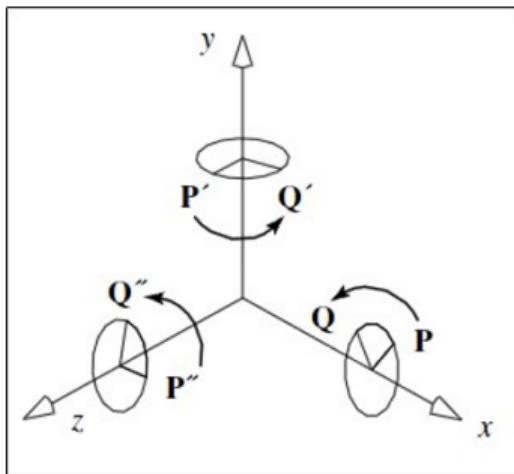
$$m = mat \times mS$$

$$m \times V = (mat \times mS) \times V = mat \times (mS \times V)$$

Rotaciones

- **Rotaciones elementales** sobre los ejes:

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



Ángulo positivo → giro **CCW**
(antihorario)

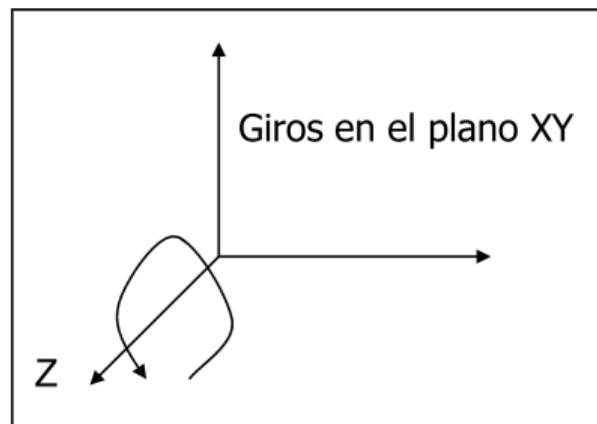
Rotaciones sobre el eje Z (Z-Roll)

- Una **rotación sobre el eje Z de θ radianes**:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) \cdot x - \sin(\theta) \cdot y \\ \sin(\theta) \cdot x + \cos(\theta) \cdot y \\ z \\ 1 \end{pmatrix}$$

siendo $Q = (x', y', z')$ las coordenadas del punto $P = (x, y, z)$ una vez rotado

Ángulo positivo → giro
CCW (antihorario)



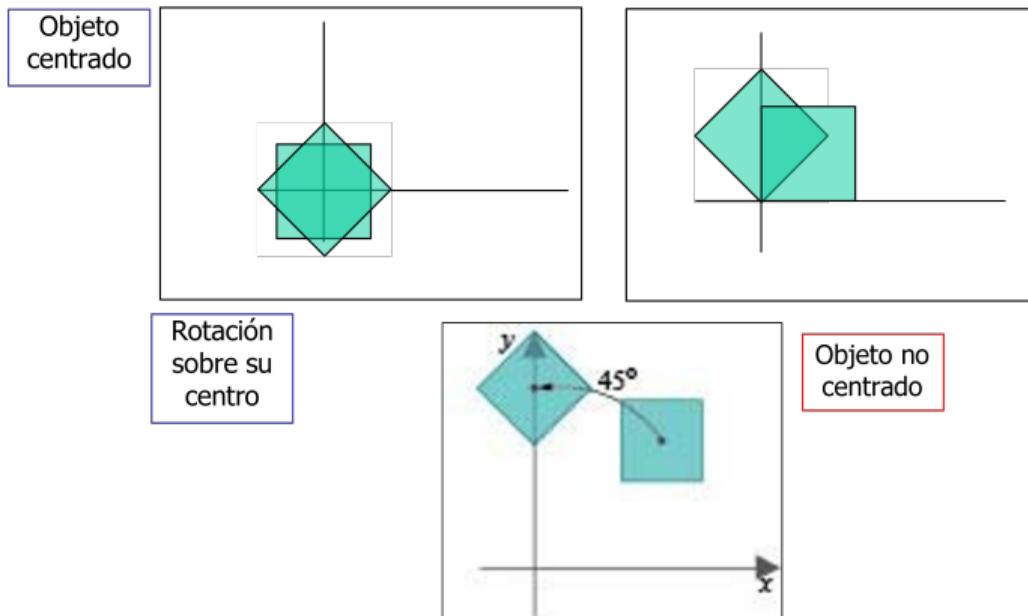
Rotaciones con GLM

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

	mR: Matriz de Rotación
Z-Roll: <code>m = rotate(dmat4, β, dvec3(0, 0, 1));</code> <code>mR = rotate(dmat4(1), β, dvec3(0, 0, 1));</code>	$\begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 & 0 \\ \sin(\beta) & \cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Y-Yaw: <code>m = rotate(dmat4, β, dvec3(0, 1, 0));</code> <code>mY = rotate(dmat4(1), β, dvec3(0, 1, 0));</code>	$\begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
X-Pitch: <code>m = rotate(dmat4, β, dvec3(1, 0, 0));</code> <code>mP = rotate(dmat4(1), β, dvec3(1, 0, 0));</code>	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) & 0 \\ 0 & \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Rotación sobre el eje Z (Z-Roll)

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



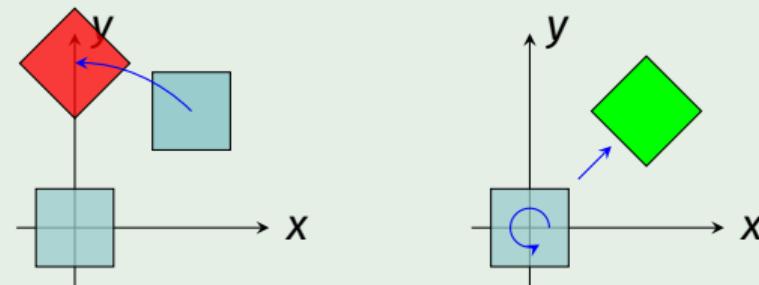
Composición de transformaciones

- En OpenGL (GLM) las transformaciones se componen post-multiplicando las matrices:
 $mC = \text{transformar}(m, ...); \rightarrow mC = m \times mA$
Al aplicar mC a un vértice: $mC \times V = (m \times mA) \times V = m \times (mA \times V)$

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
```



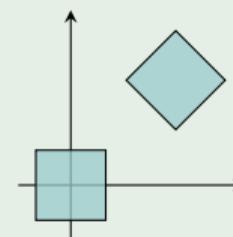
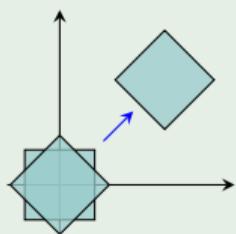
Composición de transformaciones

- En OpenGL (GLM) las transformaciones se componen post-multiplicando las matrices:
Al aplicar mC a un vértice: $mC \times V = (m \times mA) \times V = m \times (mA \times V)$

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:

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

Si es necesario escalar el objeto (mS):

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

- Y por último la matriz de vista:

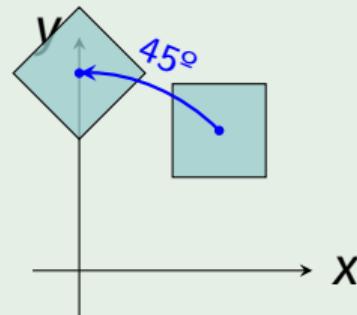
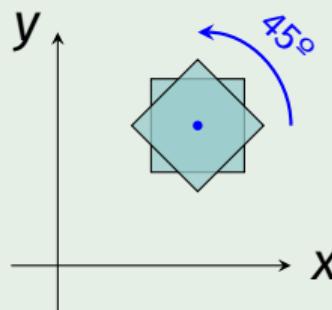
$$\text{matriz de modelado y vista} = \text{matriz de vista} \times \text{matriz de modelado}$$

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.



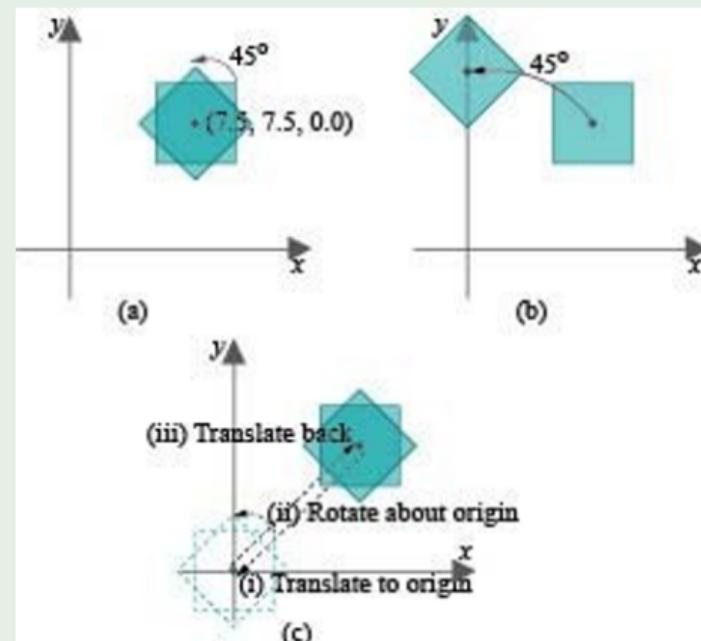
Composición de transformaciones

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

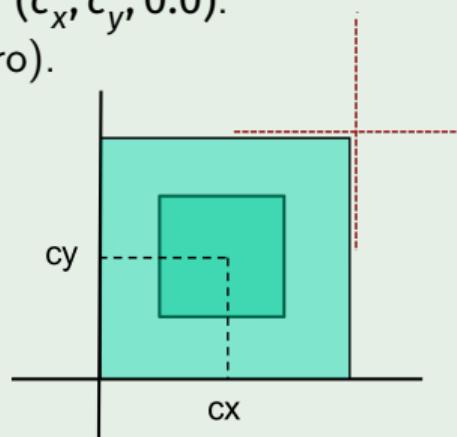
Tenemos un cuadrado alineado con los ejes con centro en $(c_x, c_y, 0.0)$.

Queremos escalarlo sobre su centro (sin modificar el centro).

CUIDADO!!! MAL

```
m = scale(mI, dvec3(2,2,2));  
→ m = mI × mS
```

```
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;
```

Transformaciones

ModelView MATRIX = VIEW MATRIX \times MODEL MATRIX

