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 HW gráfico de la máquina (GPU: Graphics Processing Unit).
- OpenGL es una máquina de estados. Tenemos una colección de variables de estado a las que vamos cambiando su valor.
- No dispone de comandos de alto nivel para describir escenas 3D. Tampoco dispone de comandos para gestionar ventanas ni para interactuar con el usuario.
- Está diseñada para trabajar en red. En un ordenador (servidor) podemos ejecutar nuestro programa OpenGL y en otro (cliente) podemos mostrar el gráfico obtenido.

- Para la gestión de ventanas y E/S puede utilizarse la librería GLUT (OpenGL Utility ToolKit), que tiene la ventaja de ser portable.
- Para las operaciones matemáticas puede utilizarse la librería GLM (OpenGL Mathematics), especializada para la programación gráfica.
- Nosotros utilizaremos el entorno de desarrollo VS 2017 C++14 con FreeGLUT y GLM.
- 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.

```
glClearColor(....)

glEnable(...)
```

- Las constantes (y variables de estado) en OpenGL 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_DEPTH_TEST

GL_COLOR_BUFFER_BIT
```

- 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, …)  // RGBA unsigned byte
glColor3d(GLdouble red, …)  // RGB double
glColor4fv(GLfloat *)       // 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)

Tipos de GLM

- GLM ofrece tipos, clases y funciones compatibles con OpenGL 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::dmat
  glm::mat3,  glm::dmat3
  ```

- Para las coordenadas de los vértices de las primitivas gráficas usaremos arrays de glm::dvec3.
- Para las componentes de los colores RGBA usaremos también glm::dvec3 y glm::dvec4.

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

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

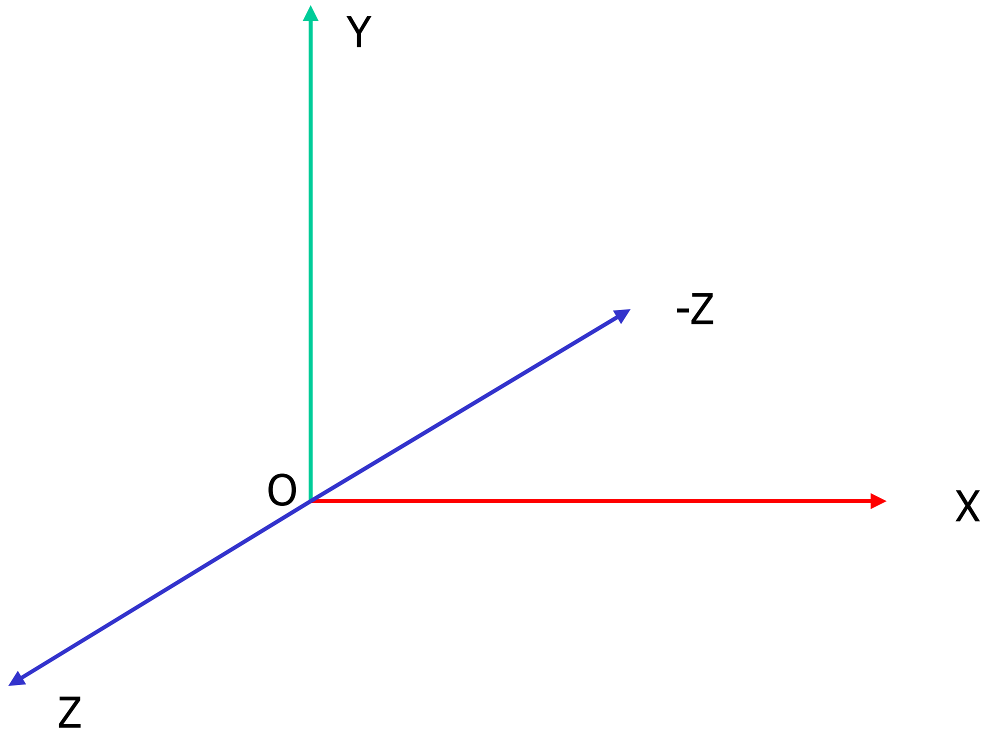
Valores de los argumentos en [0,1].

`glClearColor(0.0, 0.0, 0.0, 1.0)` // (alfa=1 -> opaco)

- Función `display()` de la ventana con doble [frame] buffer: Front y Back:

```
glClear(GL_COLOR_BUFFER_BIT);  // El BackColorBuffer queda
                               del color establecido con glClearColor()
scene.render();  // se dibujan los objetos en el BackColorBuffer
glutSwapBuffers();  // se intercambian los buffers (back/front)
```

Sistema cartesiano en OpenGL



- Por ejemplo, para definir las coordenadas de 4 puntos:

```
GLuint numVertices = 4;
glm::dvec3 * vertices = new dvec3[numVertices];
vertices[0] = dvec3(10.0, 0.0, 0.0);
vertices[1] = dvec3(0.0, 10.0, 0);
vertices[2] = dvec3(0.0, 0.0, 10.0);
vertices[3] = dvec3(0.0, 0.0, 0.0);
```

- Para dibujar puntos:

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_DOUBLE, 0, vertices); // 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*()

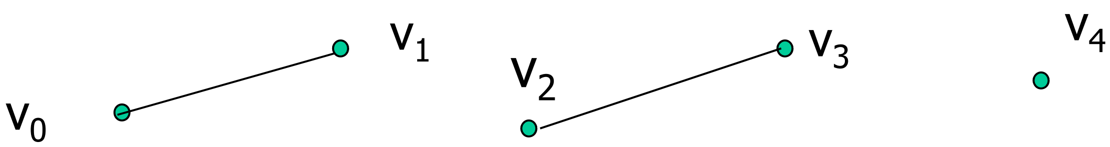
```
glPointSize(3);
glColor3d(0.5, 1, 0.25);  // alpha 1
mesh->render();
glPointSize(1);                // valores por defecto
glColor4d(1, 1, 1, 1);        // valores por defecto
```

- Para dibujar líneas:

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

Dibuja las líneas $\mathbf{v_0v_1}$, $\mathbf{v_2v_3}$, …, $\mathbf{v_{n-1}v_n}$.

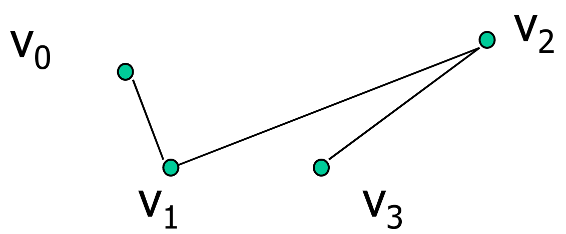
Si el número de vértices es impar, el último vértice se ignora.



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

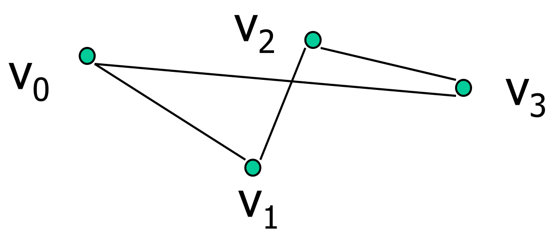
- Si utilizamos la constante `GL_LINE_STRIP`, las líneas se conectan, i.e., se dibujan las líneas $\mathbf{v_0v_1}$, $\mathbf{v_1v_2}$, $\mathbf{v_2v_3}$ etc.

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



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

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

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

- La malla EjesRGB.

```
generaEjesRGB(GLdouble l);

 int numVertices;
 glm::dvec3 * vertices;
 glm::dvec4 * colores;
```

Los arrays tienen que constar del mismo número de datos, los vectores no

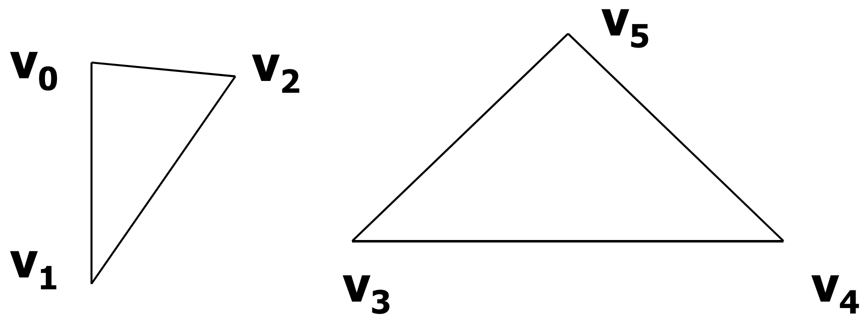
```
{
  numVertices = 6;
  vertices = new dvec3[numVertices];
  vertices[0]= dvec3(0, 0, 0);
  vertices[1]= dvec3(l, 0, 0);
  vertices[2]= dvec3(0, 0, 0);
  vertices[3]= dvec3(0, l, 0);
  vertices[4]= dvec3(0, 0, 0);
  vertices[5]= dvec3(0, 0, l);
  colores = new dvec4[numVertices];
  colores[0] = dvec4(1, 0, 0);
  colores[1] = dvec4(1, 0, 0);
  colores[2] = dvec4(0, 1, 0);
  colores[3] = dvec4(0, 1, 0);
  colores[4] = dvec4(0, 0, 1);
  colores[5] = dvec4(0, 0, 1);
}
```

- **GL_TRIANGLES**: $\mathbf{v_0}$, $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$, $\mathbf{v_4}$, $\mathbf{v_5}$

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

Dibuja triángulos independientes:

$\mathbf{v_0\ v_1\ v_2,\quad v_3\ v_4\ v_5}$



- Los vértices de un triángulo $\mathbf{v_0 v_1 v_2}$ deben estar ordenados en sentido antihorario (Counter-Clock Wise). Determina la cara exterior.

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

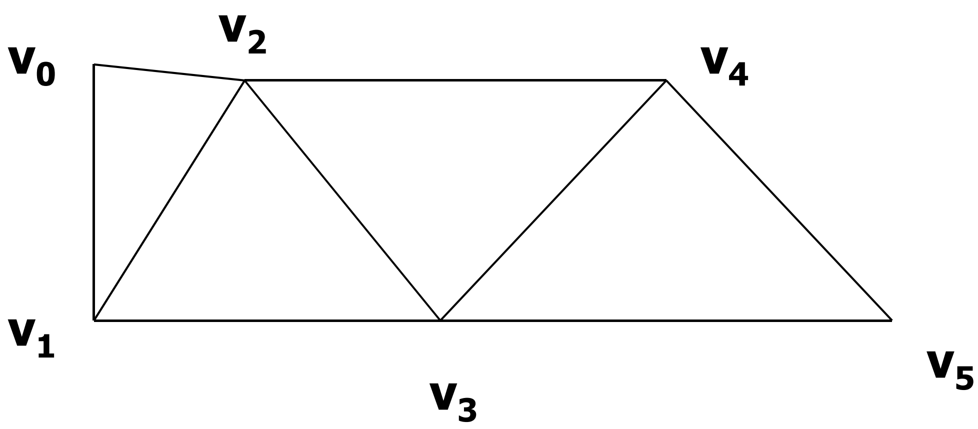
Especifica el modo en el cuál se rasterizará el polígono.

face puede ser: GL_FRONT_AND_BACK, GL_FRONT, GL_BACK

mode puede ser: GL_FILL, GL_LINE, GL_POINT

- **GL_TRIANGLE_STRIP**: $\mathbf{v_0}$, $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$, $\mathbf{v_4}$, $\mathbf{v_5}$

  Dibuja los triángulos $\mathbf{v_0v_1v_2}$, $\mathbf{v_2v_1v_3}$, $\mathbf{v_2v_3v_4}$, $\mathbf{v_4v_3v_5}$

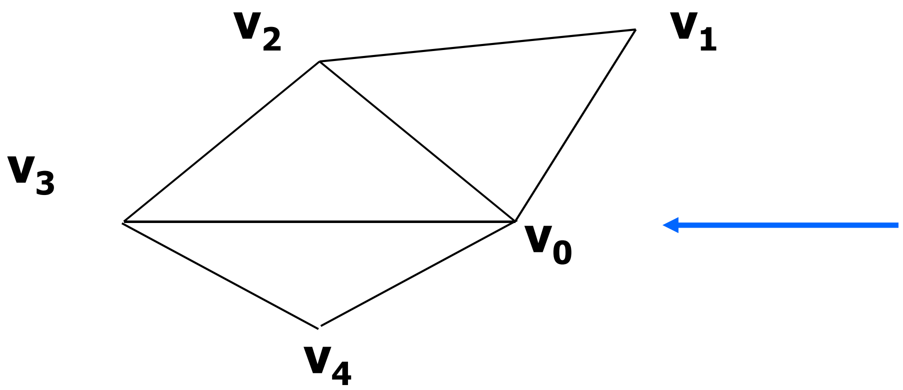


El número de vértices ha de ser al menos 3

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

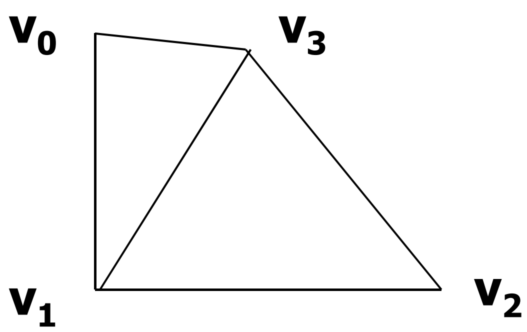
  Dibuja los triángulos $\mathbf{v_0 v_1 v_2}$, $\mathbf{v_0 v_2 v_3}$, $\mathbf{v_0 v_3 v_4}$

  Todos los triángulos comparten un vértice común: $\mathbf{v_0}$

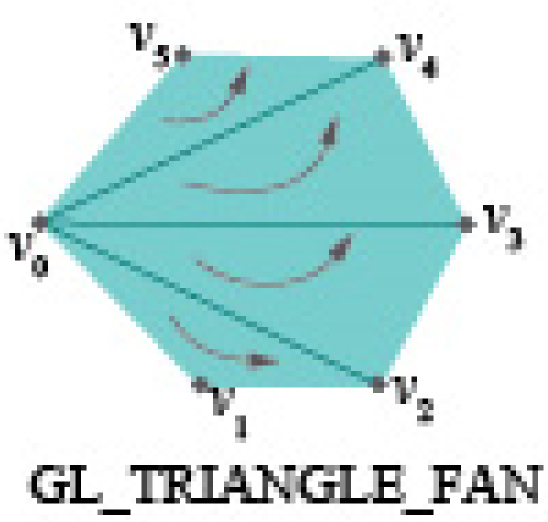


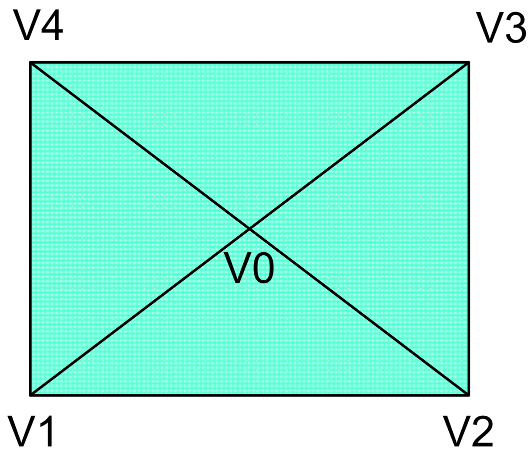
- Para cuadriláteros utilizamos GL_TRIANGLE_STRIP con los cuatro vértices del cuadrilátero $\mathbf{v_0 v_1 v_2 v_3}$ dados en el orden $\mathbf{v_0 v_1 v_3 v_2}$.

  Dibuja el cuadrilátero con 2 triángulos: $\mathbf{v_0 v_1 v_3}$ y $\mathbf{v_3 v_1 v_2}$



- Para polígonos utilizamos **GL_TRIANGLE_FAN** con los vértices del polígono $\mathbf{v_0 v_1 v_2 v_3} \ldots \mathbf{v_n}$ en orden contrario a las agujas del reloj

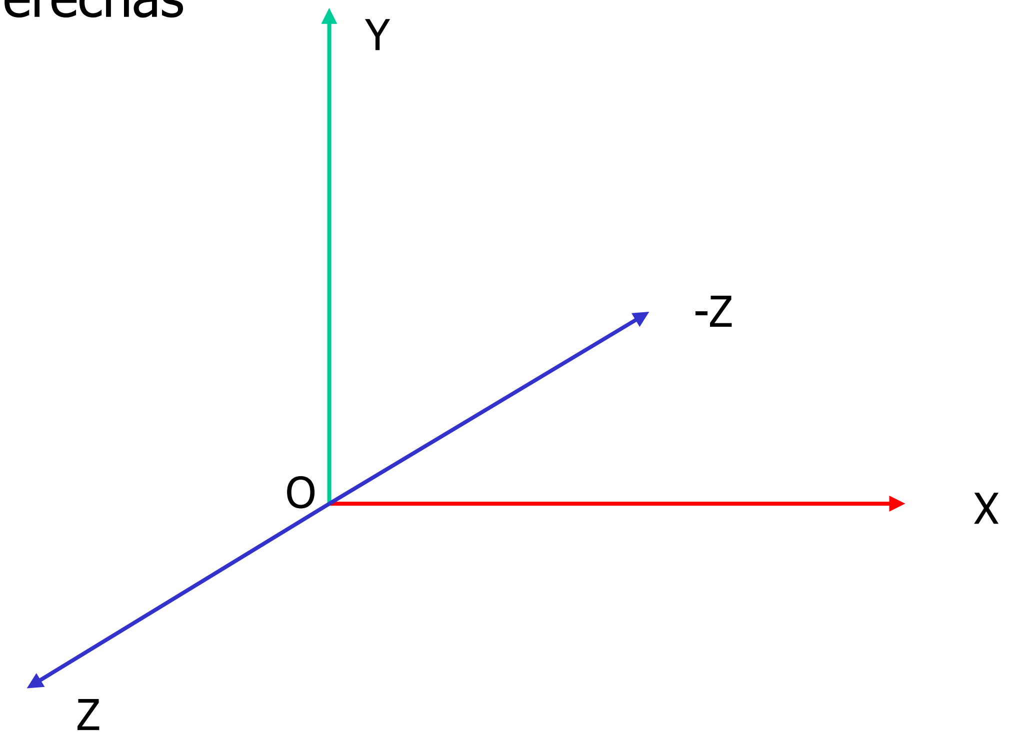




V0, V1, V2, V3, V4, V1

Matriz del marco cartesiano a derechas

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



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

(x, y, z, w)

w = 1 -> punto

w = 0 -> vector

En openGL las matrices son 4x4 **column-major**

- **Traslaciones**, **rotaciones** y **escalas**

  Se expresan mediante matrices 4x4 que representan un marco de coordenadas.

$$\begin{pmatrix} Ax & Bx & Cx & Ox \\ Ay & By & Cy & Oy \\ Az & Bz & Cz & Oz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- La escala puede deformar el objeto. Las traslaciones cambian la posición del objeto y las rotaciones la orientación, sin deformar el objeto (transformaciones rígidas).
- Las transformaciones se pueden componer (multiplicando las matrices) para obtener distintos efectos.

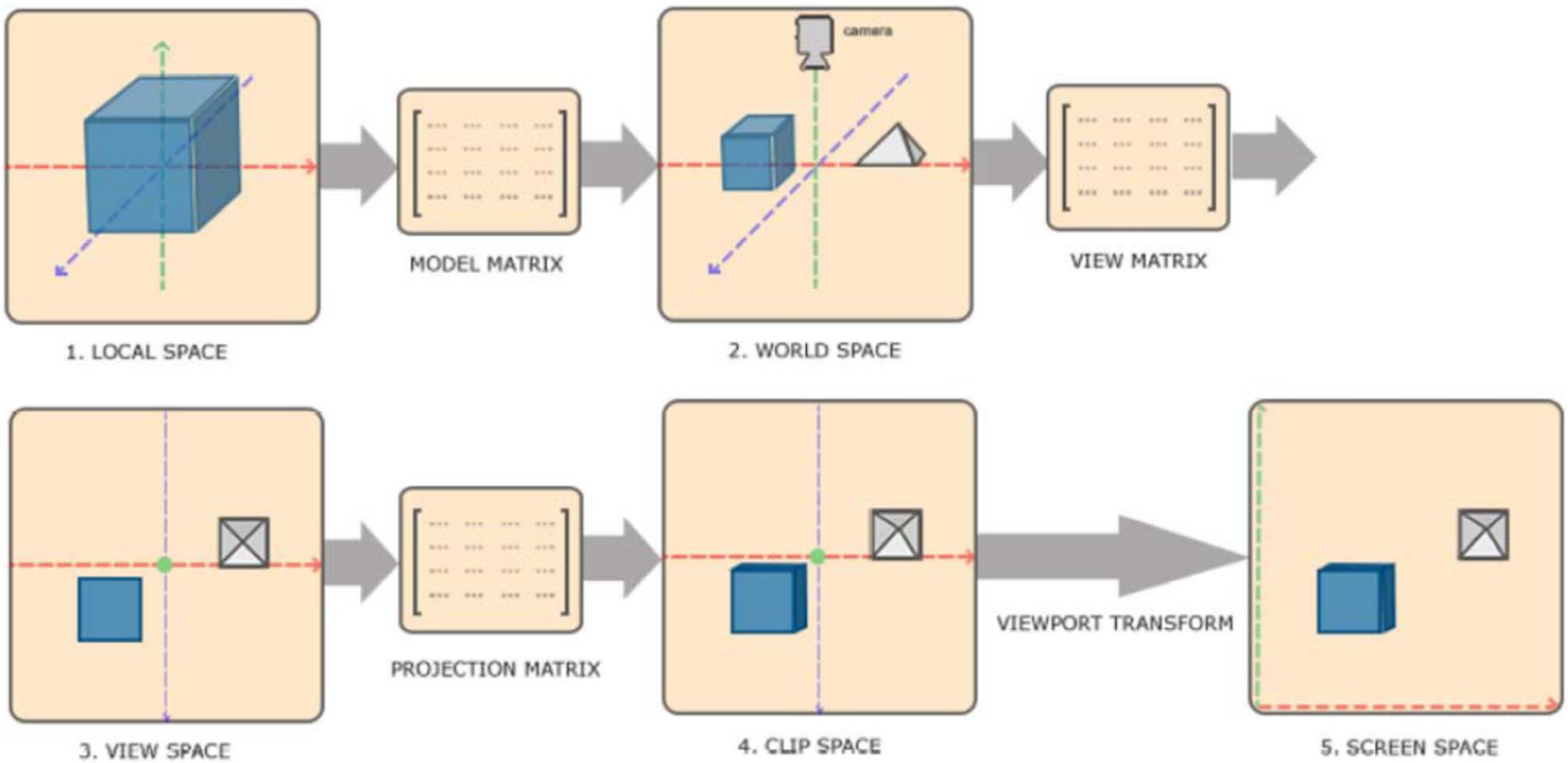
- Dada una matriz de transformación afín (marco de coordenadas), un punto P=(p1, p2, p3, 1) o un vector V=(v1, v2, v3, 0), las coordenadas transformadas de P y V se obtienen:

$$\begin{pmatrix} Ax & Bx & Cx & Ox \\ Ay & By & Cy & Oy \\ Az & Bz & Cz & Oz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 Ax + p_2 Bx + p_3 Cx + Ox \\ p_1 Ay + p_2 By + p_3 Cy + Oy \\ p_1 Az + p_2 Bz + p_3 Cz + Oz \\ 1 \end{pmatrix} = O + p_1 A + p_2 B + p_3 C$$

punto

$$\begin{pmatrix} Ax & Bx & Cx & Ox \\ Ay & By & Cy & Oy \\ Az & Bz & Cz & Oz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix} = v_1 A + v_2 B + v_3 C$$

vector

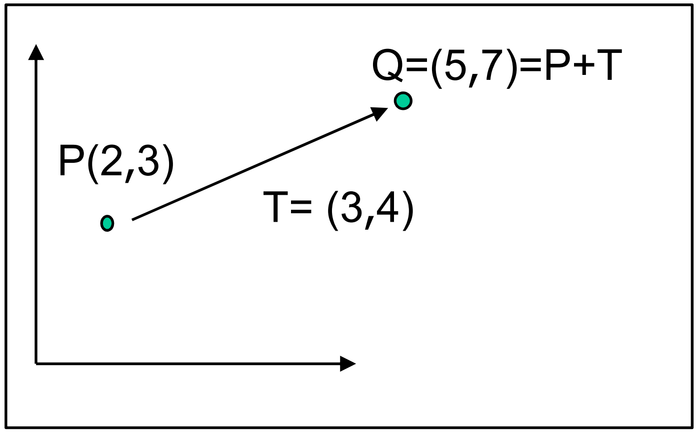

camera
MODEL MATRIX
VIEW MATRIX
1. LOCAL SPACE
2. WORLD SPACE
PROJECTION MATRIX
VIEWPORT TRANSFORM
3. VIEW SPACE
4. CLIP SPACE
5. SCREEN SPACE

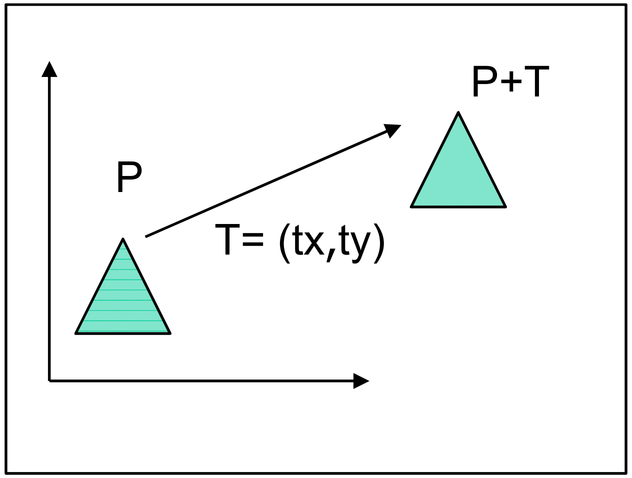
- **Traslación** con vector (tx, ty, tz):

Coordenadas de un punto P

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} Px+tx \\ Py+ty \\ Pz+tz \\ 1 \end{pmatrix}$$

Coordenadas del punto P una vez trasladado





- glm::mat4 m = glm::translate(mat4, vec3);

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

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

Composición: m = translate(mat, vT); $\longrightarrow$ m = mat * mT

Escalas

- **Escala** con factor S=(sx, sy, sz):

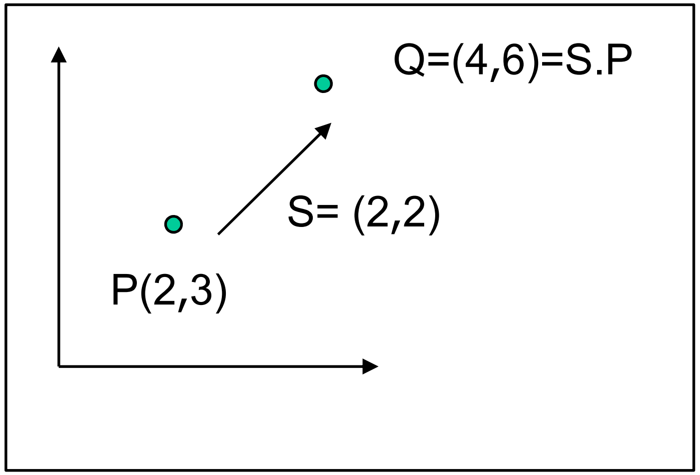
Coordenadas de un punto P

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} sxPx \\ syPy \\ szPz \\ 1 \end{pmatrix}$$

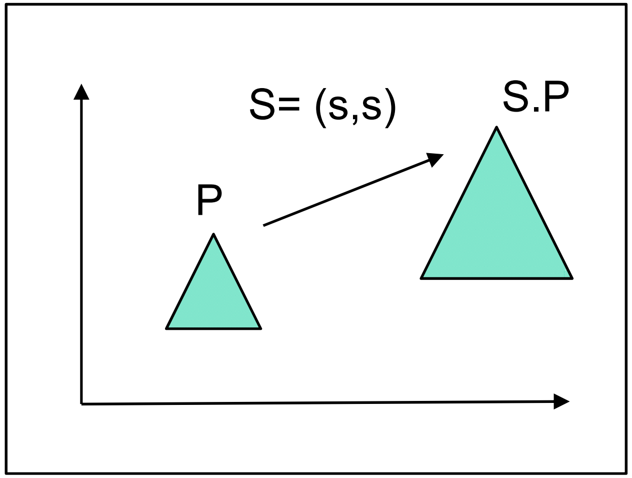
Coordenadas del punto P una vez escalado

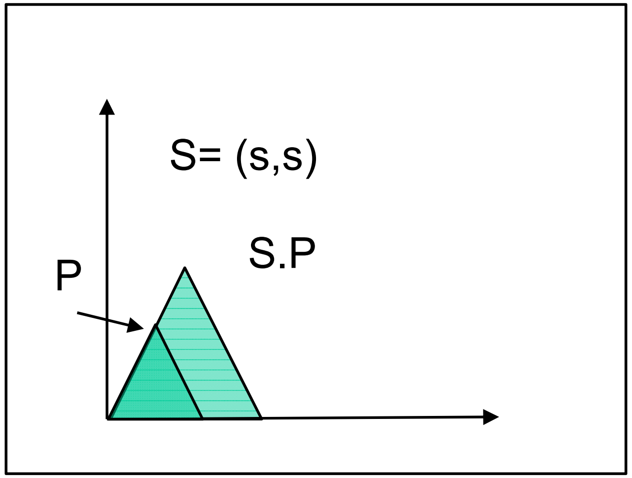
La **escala** es **uniforme** si

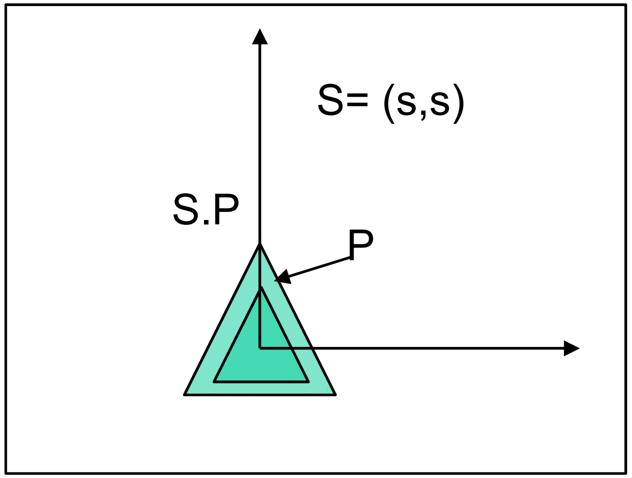
sx=sy=sz



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







- glm::mat4 m = glm::scale(mat4, vec3);

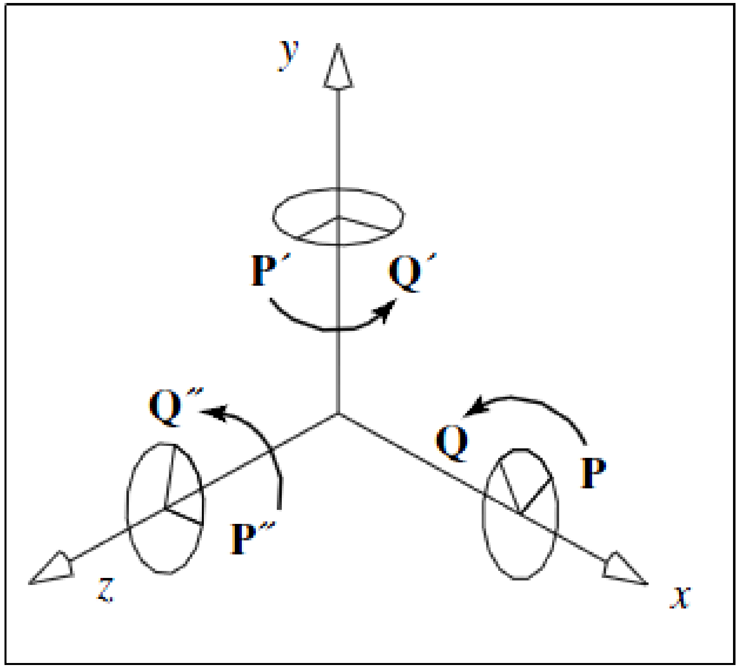
mS = scale(dmat4(1), dvec3(sx, sy, sz));

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

Composición: m = scale(mat, vS); $\longrightarrow$ m = mat * mS

- **Rotaciones elementales** sobre los ejes:

  glm::mat4 m = glm::rotate(mat4, ß, vec3); // ß en radianes



ángulo positivo -> giro antihorario CCW

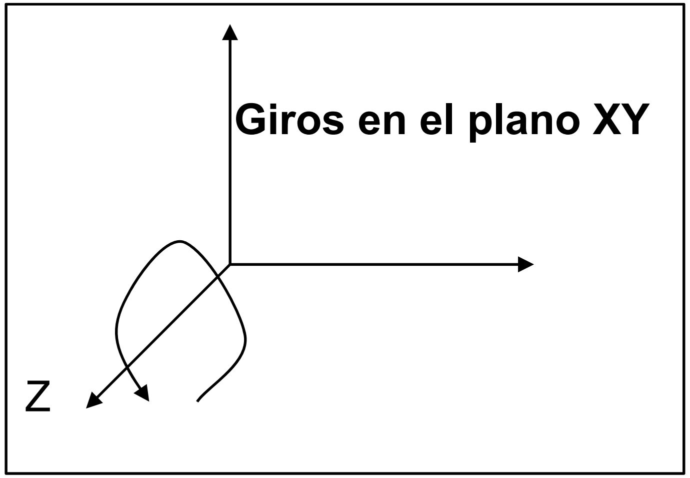
Rotación sobre el eje Z (Z-Roll)

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

Coordenadas de un punto P

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

Coordenadas del punto P una vez rotado



ángulo positivo -> giro antihorario CCW

- glm::mat4 m = glm::rotate(mat4, ß, vec3); // ß en radianes

Z-Roll: m = rotate(mat4, ß, dvec3(0, 0, 1));

mR = rotate(dmat4(1), ß, dvec3(0, 0, 1)); $\longrightarrow$ $\begin{pmatrix} \cos\beta & -sen\beta & 0 & 0 \\ sen\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

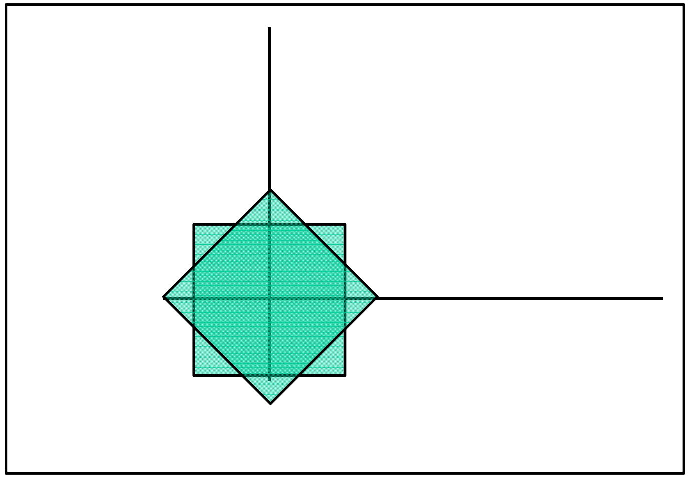
Y-Roll: m = rotate(mat4, ß, dvec3(0, 1, 0));

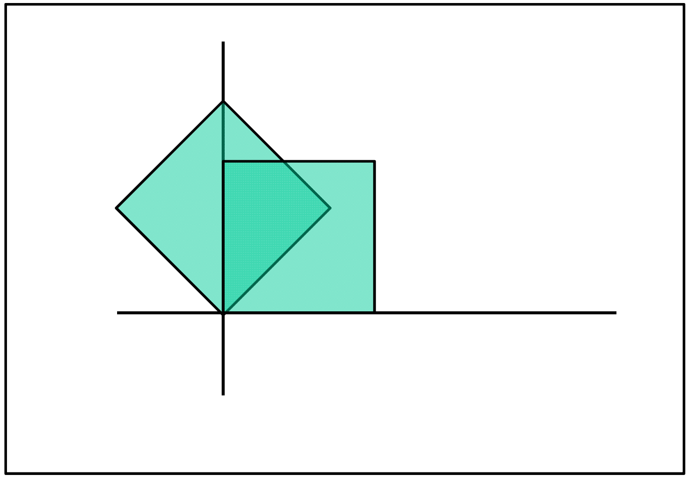
mY = rotate(dmat4(1), ß, dvec3(0, 1, 0)); $\longrightarrow$ $\begin{pmatrix} \cos\beta & 0 & sen\beta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

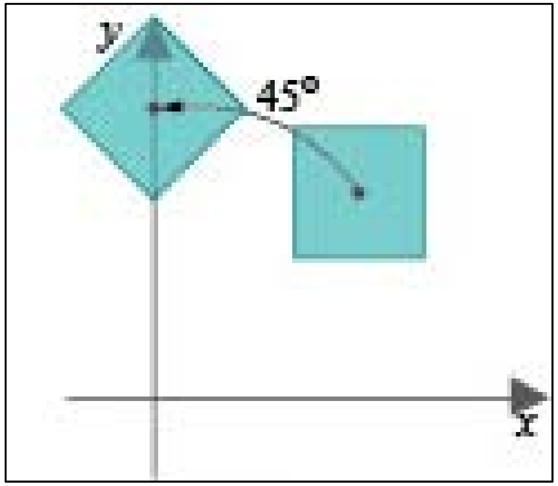
X-Roll: m = rotate(mat4, ß, dvec3(1, 0, 0));

mP = rotate(dmat4(1), ß, dvec3(1, 0, 0)); $\longrightarrow$ $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & -sen\beta & 0 \\ 0 & sen\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- Una **rotación** sobre el eje Z de 25 grados :







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

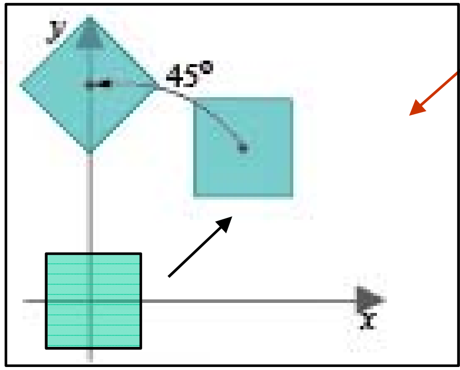
  mr = transformar(m, ...); -> mr = m * mT

  Al aplicar mr a un vértice: mr P = (m * mT) P = m * (mT P)

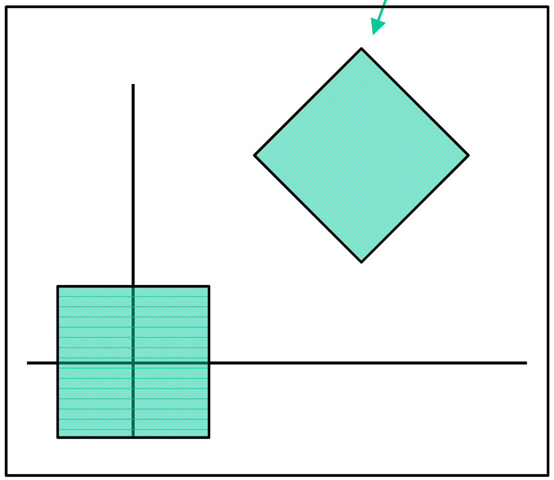
  Ejemplo:

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



CUIDADO!!!



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

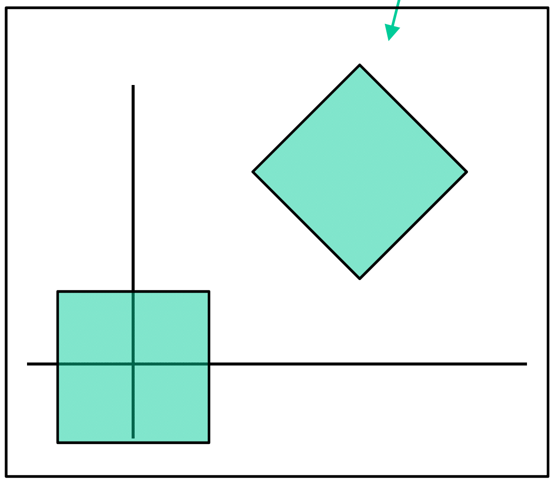
  mr = transformar(m, ...); -> mr = m * mT

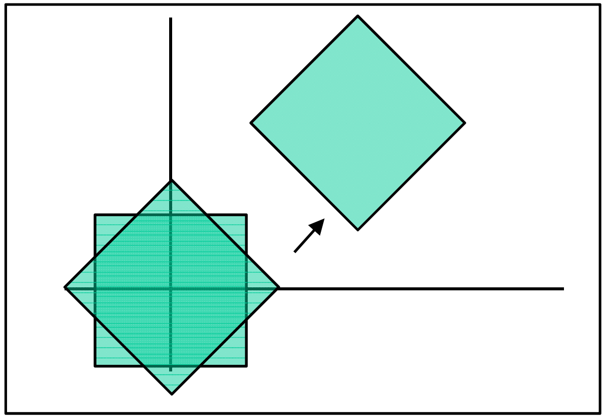
  Al aplicar mr a un vértice: mr P = (m * mT) P = m * (mT P)

  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

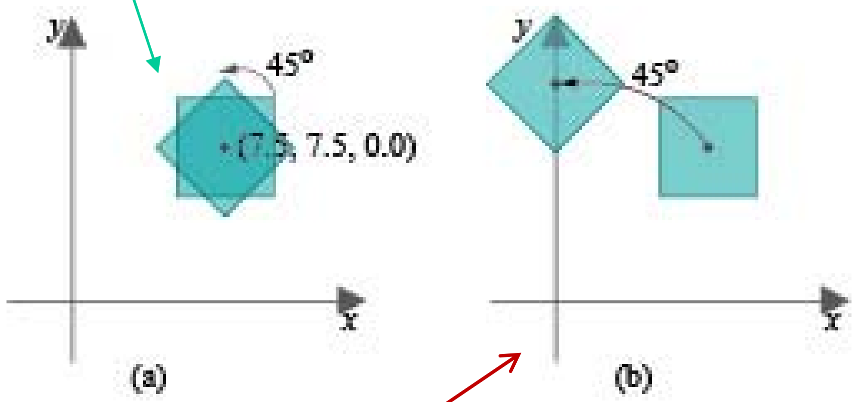
```
m = translate(mi, dvec3(7.5,7.5,0));
m = rotate(m, 45.0, dvec3(0,0,1));
m = mi * mT * mR
```





- 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):

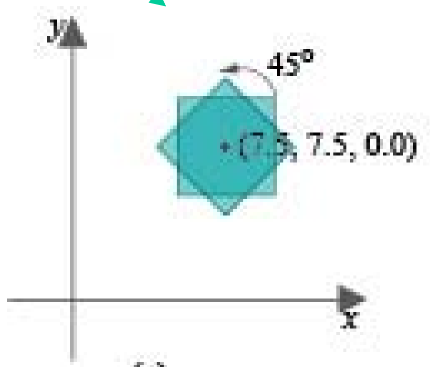



```
m = rotate(mi, 45.0, dvec3(0,0,1));
m = mi * mR
```

CUIDADO!!!

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

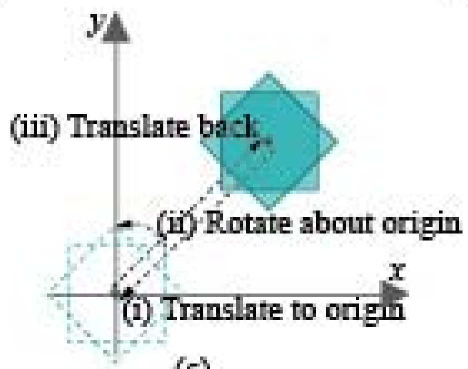
Queremos rotarlo 45 grados sobre su centro (a):



(a)

```
m = translate(mi, dvec3(7.5,7.5,0));
m = rotate(m, 45.0, dvec3(0,0,1));
m = translate(m, dvec3(-7.5,-7.5,0));
```

$m = mi * mT * mR * mT^{-1}$

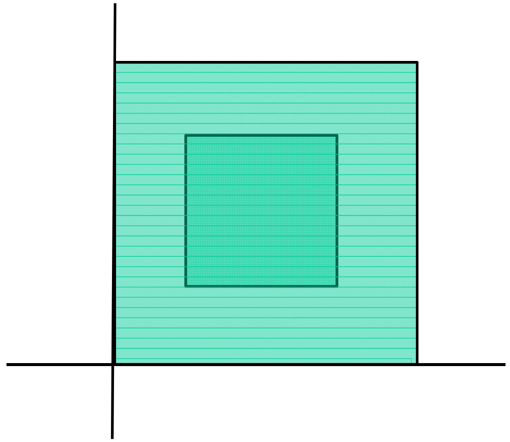


(c)

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!!!

```
m = translate(mi, dvec3(cx,cy,0));
m = scale(m, dvec3(2,2,2));
m = translate(m, dvec3(-cx,-cy,0));
```

$m = mi * mT * mS * mT^{-1}$

- Las rotaciones, escalas y traslaciones son matrices de la forma:

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

donde T=(Tx, Ty, Tz) es un vector de traslación.

- La transformación inversa es:

$$F^{-1} = \left(\begin{array}{c|c} M^{-1} & -M^{-1}T \\ \hline 0 & 1 \end{array} \right)$$

Si M (3x3) es ortonormal (vectores ortogonales y de magnitud 1), entonces: $M^{-1} = M^T$.

- Para colocar la cámara podemos establecer, en coordenadas cartesianas, un punto para su posición (**eye**), el punto al que mira (**look**) y la inclinación (**upward**):

```
glm::dvec3 eye, look, up;
```

```
glm::dmat4 viewMat = glm::lookAt(eye, look, up);
```

  Con lo que se define la **matriz de vista** (inversa de la matriz de modelado de la cámara).

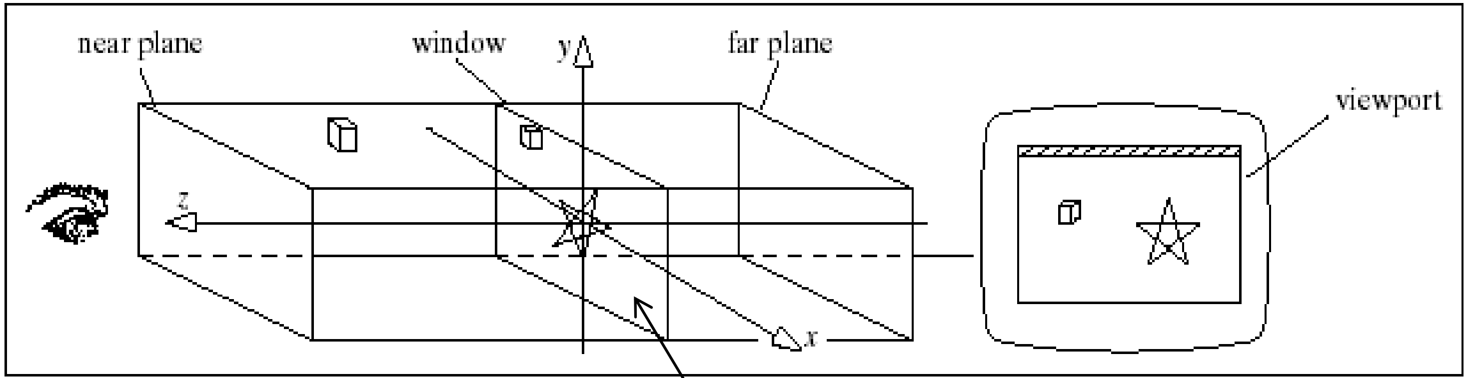
- Para colocar la cámara en el eje Z, vertical, mirando al centro del sistema (proyección frontal):

```
lookAt(dvec3(0,0,500), dvec3(0,0,0), dvec3(0,1,0));
```

- Para colocar la cámara vertical, mirando al centro del sistema, en las coordenadas 100 de los tres ejes (proyección isométrica):

```
lookAt(dvec3(100,100,100), dvec3(0,0,0), dvec3(0,1,0));
```

- El **volumen de vista** se establece con respecto a la cámara.

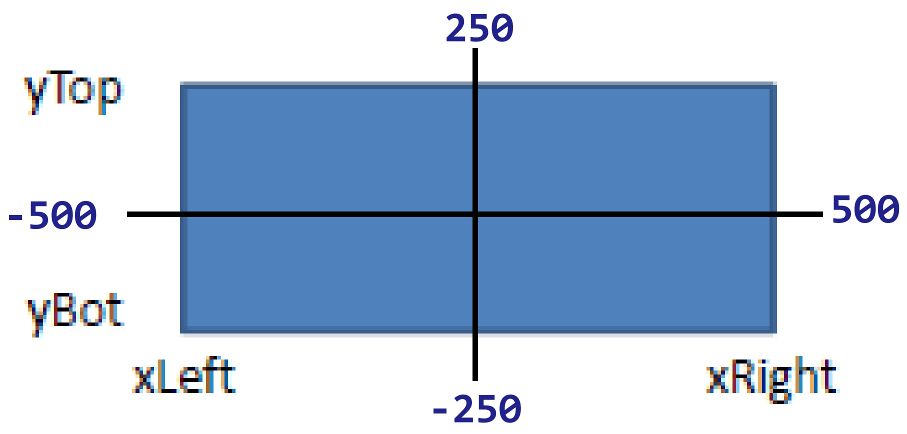


Área visible, ventana o plano de vista

- En el puerto de vista se mostrarán los objetos que quedan dentro del volumen de vista una vez proyectados sobre el plano de vista.

- El **área visible de la escena (ventana o plano de vista)** es un rectángulo perpendicular a la dirección de vista de la cámara, que corresponde a la región del **plano de proyección** de la escena que se muestra en el puerto de vista.
- Para fijar el puerto de vista se usan cuatro valores (GLdouble):

```
xLeft = -500, xRight = 500, yBot = -250, yTop = 250
```



- Podemos colocar primitivas fuera del volumen de vista de la escena, aunque no se verán por completo (se eliminan o recortan).

Volumen de vista ortogonal (VV)

- Para fijar el área visible de la escena, establecemos la **matriz de proyección del volumen de vista**:

```
glMatrixMode(GL_PROJECTION);
glm::dmat4 projMat=glm::ortho(xLeft, xRight,
                              yBot, yTop,
                              zNear, zFar);
glLoadMatrixd(value_ptr(projMat));
```

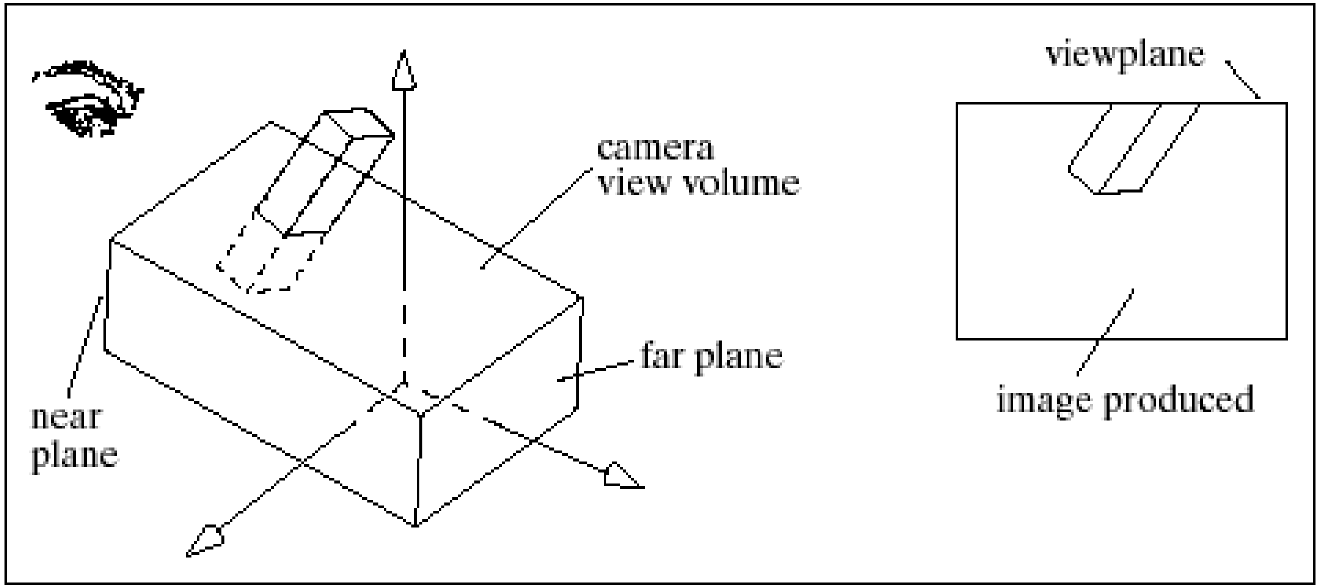
Con respecto a la cámara

- Para fijar el área visible de la escena centrada en la posición de la cámara: `xLeft = -xRight` y `yBot  = -yTop`

- zNear y zFar delimitan la profundidad del volume de vista, ambos valores son distancias a la cámara.

ortho(xLeft, xRight, yBot, yTop, zNear, zFar);

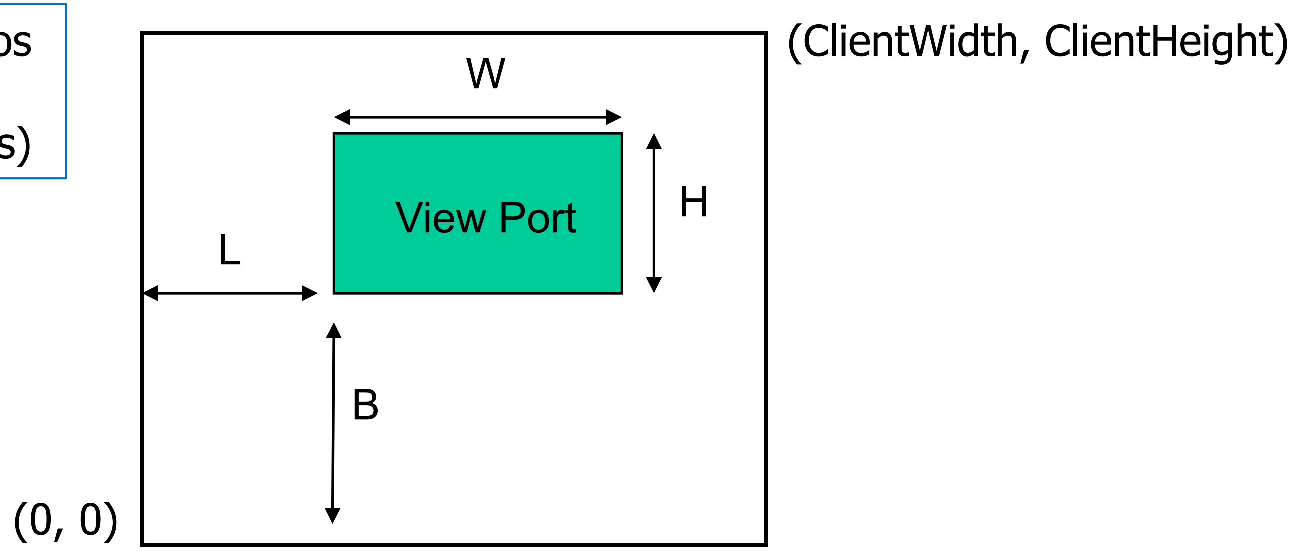
Con respecto a la cámara



- El **puerto de vista** es un rectángulo del área cliente de la ventana alineado con los ejes. Para fijar el puerto de vista:

```
glViewport(left, bottom, width, height);
```

Los parámetros son de tipo entero (píxeles)



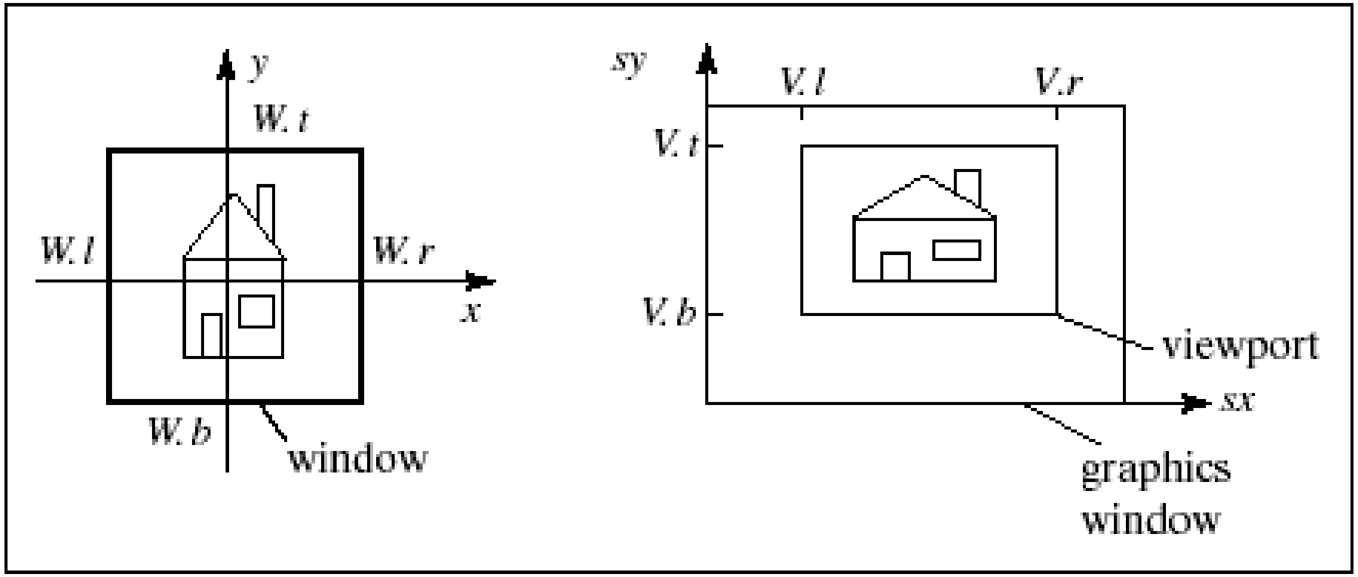
- Puerto de vista ocupando toda el área cliente de la ventana:

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

Los parámetros son de tipo entero (en píxeles)

- La relación entre el puerto de vista y el plano de vista establece una escala y una traslación.

  Esta escala determina las unidades abstractas del plano de vista.



Relación entre el plano de vista y el puerto de vista

```
glm::ortho(xLeft, xRight, yBot, yTop, zNear, zFar);

glViewport(left, bottom, width, height);
```

- Para fijar el área visible de la escena, centrada en la posición de la cámara, con una escala de relación **1:1** con un puerto de vista:

```
xRight = width/2.0, xLeft = -xRight
yTop = height/2.0,  yBot  = -yTop
```

Con respecto a la cámara

- Para fijar el área visible de la escena, centrada en la posición de la cámara, con una escala de relación **n:1** con un puerto de vista (**n** en Vp equivale a **1** en VV) :

```
xRight = width/(2.0*n), xLeft = -xRight
yTop = height/(2.0*n),  yBot  = -yTop
```

Con respecto a la cámara

Relación entre el plano de vista y el puerto de vista

- En una relación **1:1**, deben coincidir los anchos y los altos:

  xRight – xLeft = width

  yTop - yBot = height

- En una relación de **n:1** (**n** en Vp equivale a **1** en VV) uniforme:

  xRight – xLeft = width / n

  Para xLeft = -xRight -> xRight = width / 2n

  yTop - yBot = height / n

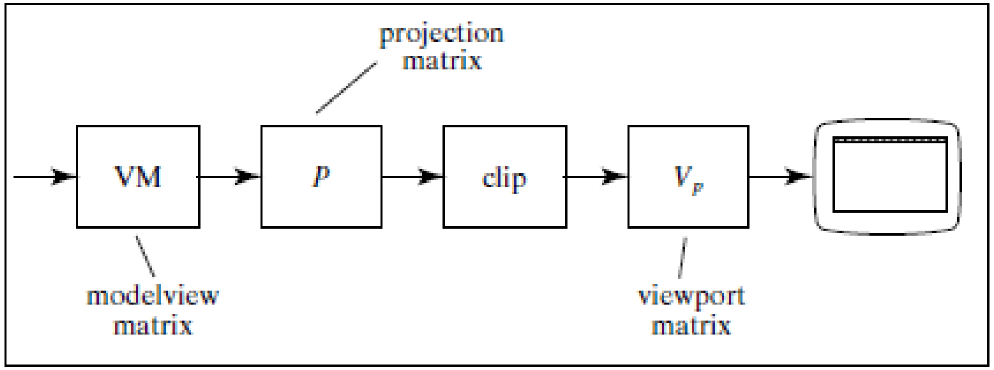
  Para yBot = -yTop -> yTop = height / 2n

n>1 -> ampliación -> el área visible disminuye

n<1 -> reducción -> el área visible aumenta

n=1 -> no cambia

- Las matrices son transformaciones que se aplican a las coordenadas de los vértices
- Matrices: GL_MODELVIEW, GL_PROJECTION y Viewport



Son matrices 4x4 que se aplican a puntos y vectores en coordenadas homogéneas: (x, y, z, w)

W determina si las coordenadas (x, y, z) corresponden a un punto (w=1) o a un vector (w=0)