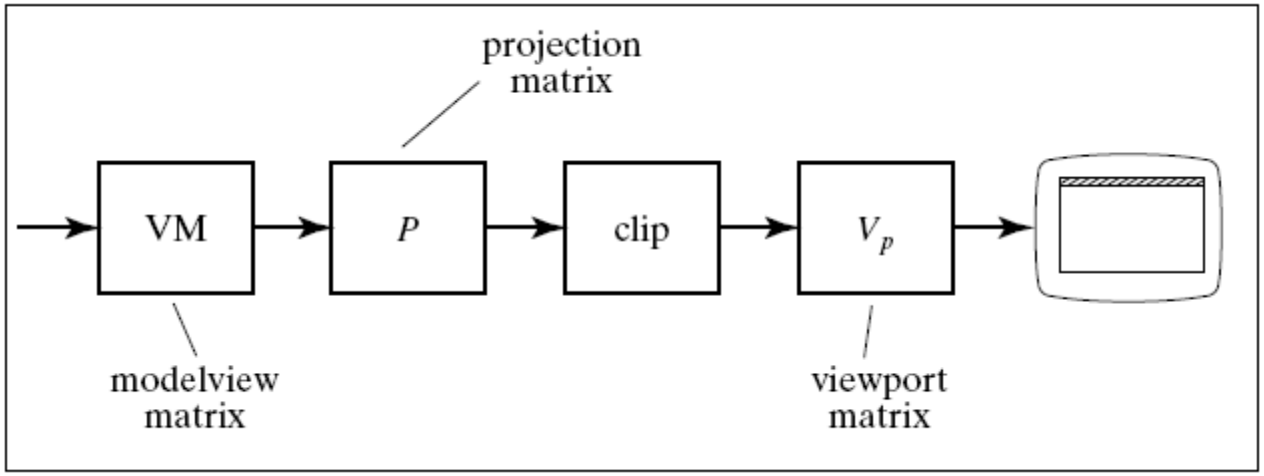
Transformaciones afines

A. Gavilanes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Transformaciones

- Imágenes virtuales vs modelado físico:
  - La matriz de modelado se aplica a los vértices.
- Aplicación de las transformaciones:
  - Replicar imágenes virtuales de un mismo modelo físico: escenas complejas.
  - Explorar la escena: mover objetos vs mover la cámara.
  - Animar objetos: se mueve la imagen virtual.
- En OpenGL: GL_MODELVIEW, GL_PROJECTION, matriz del puerto de vista.



- Están definidas por matrices de la forma:

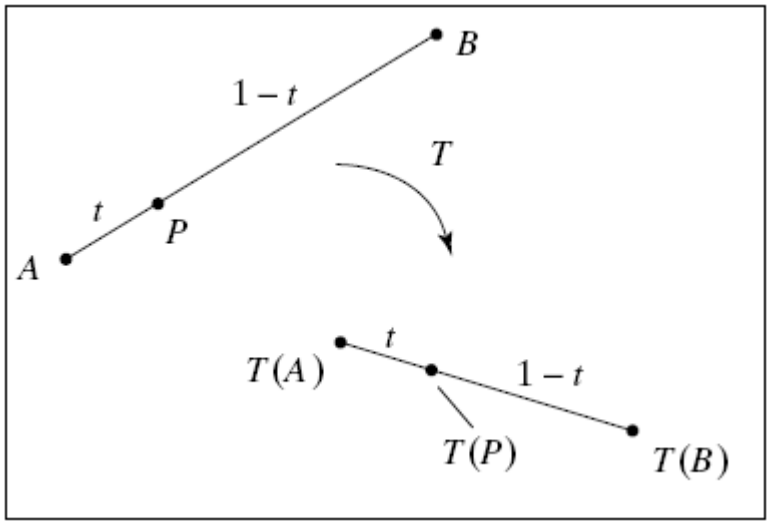
$$\begin{pmatrix} x' \\ y' \\ z' \\ \blacksquare \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ \blacksquare \end{pmatrix}$$

Coordenada homogénea: $\blacksquare \in \{0, 1\}$
Puntos (1) y vectores (0) se transforman de forma análoga!

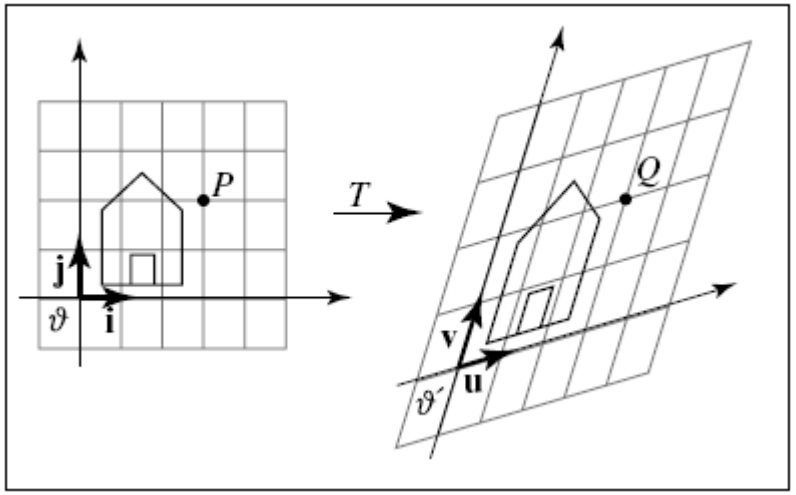
- Propiedades de las transformaciones afines:
  - Las combinaciones afines de puntos se transforman en combinaciones afines de los puntos transformados
  - Las líneas y planos se transforman en líneas y planos, respectivamente
  - Líneas y planos paralelos se transforman en líneas y planos paralelos

Transformaciones afines

- Las transformaciones afines preservan las proporciones, pero no los ángulos.



- Aplicación de una transformación afín T a una rejilla.



Transformaciones elementales 3D en OpenGL

- Traslaciones

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+T_x \\ y+T_y \\ z+T_z \\ 1 \end{pmatrix}$$

- El vector de traslación es **t** = (Tx, Ty, Tz, 0).
- El comando de OpenGL:

```
glTranslated(Tx, Ty, Tz);
```

  post-multiplica la matriz de modelado-vista por la matriz de traslación asociada al vector determinado por los tres parámetros.

- Cuando Tz=0.0 se obtienen las traslaciones 2D.

Transformaciones elementales 3D en OpenGL

- Escalaciones

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot S_x \\ y \cdot S_y \\ z \cdot S_z \\ 1 \end{pmatrix}$$

- Los factores de escalación son Sx, Sy, Sz en los ejes X, Y, Z, respectivamente.
- El comando de OpenGL:

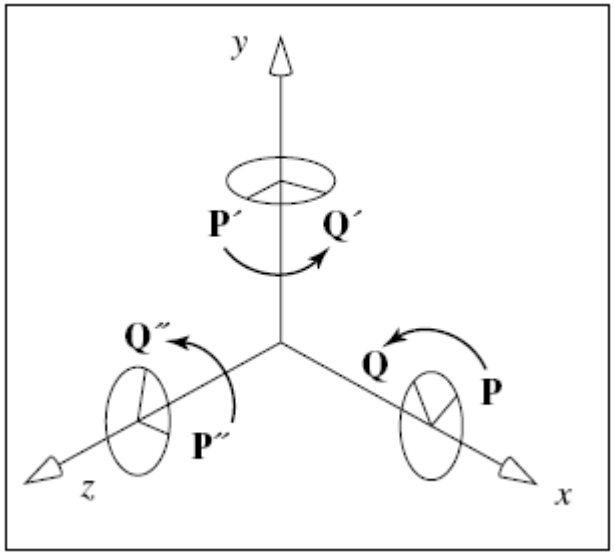
```
glScaled(Sx, Sy, Sz);
```

  post-multiplica la matriz de modelado-vista por la matriz de escalación asociada a los factores determinados por los parámetros.

- Cuando Sz=1.0 se obtienen las escalaciones 2D.

Transformaciones elementales 3D en OpenGL

- Rotaciones elementales alrededor de los ejes



- Las rotaciones se miden en sentido anti-horario, cuando se mira el origen desde la parte positiva del eje respectivo.
- Rotaciones elementales Rx(θ), Ry(θ), Rz(θ).

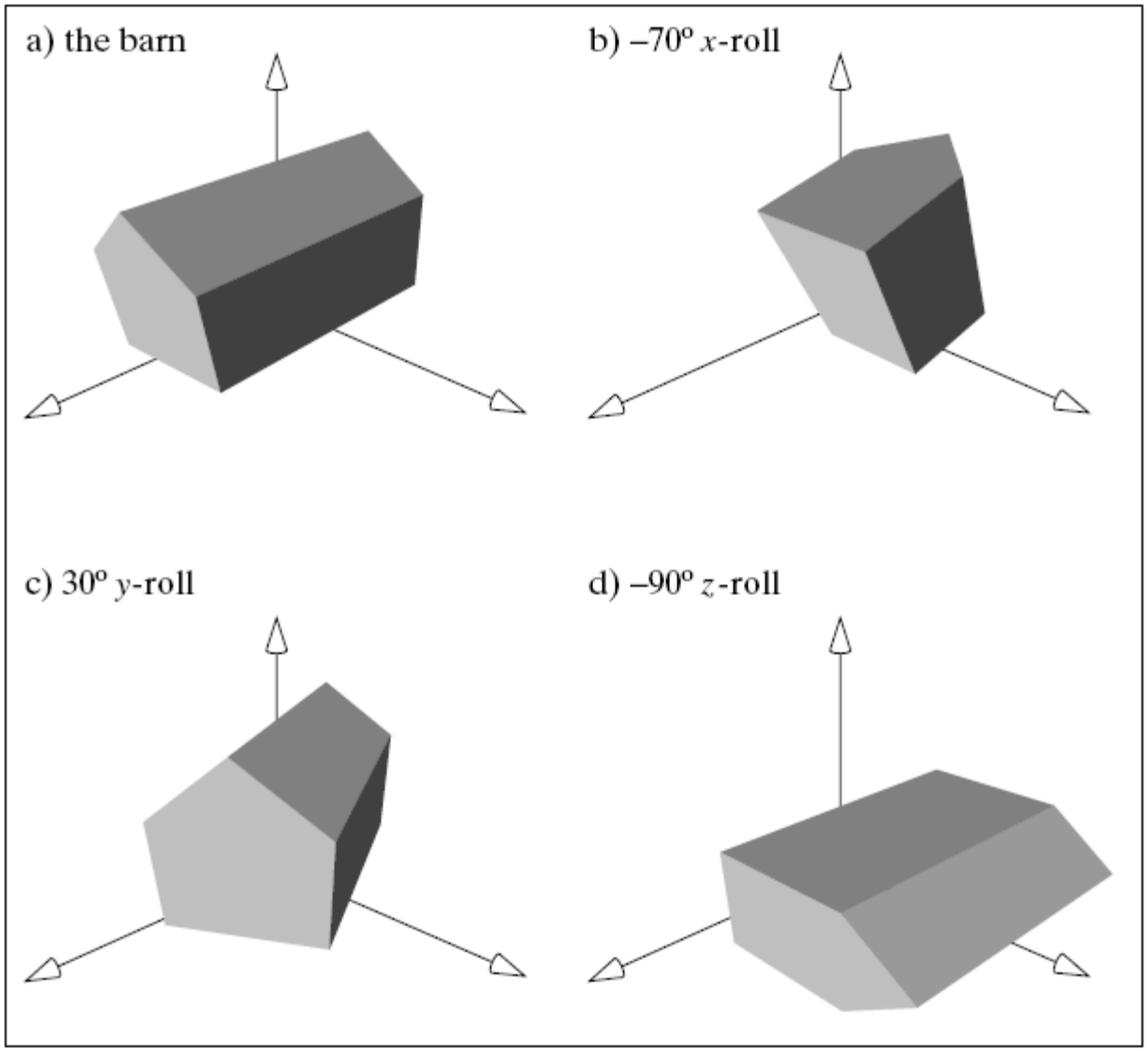
Transformaciones elementales 3D en OpenGL

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

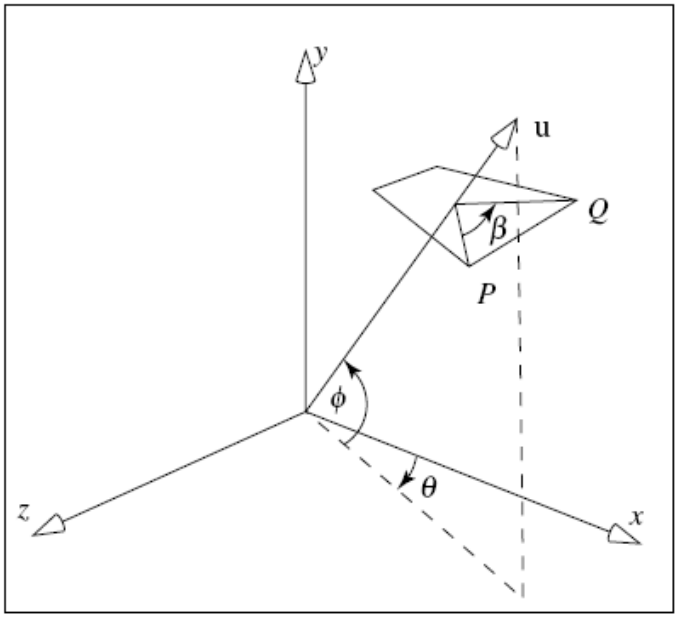
Transformaciones elementales 3D en OpenGL



Transformaciones elementales 3D en OpenGL

- Rotación alrededor de un eje que pasa por el origen
- Una rotación alrededor de una recta que pasa por el origen y tiene vector director **u**, de β grados medidos en sentido anti-horario, tomado éste cuando se mira desde el punto señalado por **u** al origen, se puede expresar como una composición de 5 rotaciones elementales alrededor de los ejes coordenados

$$R\mathbf{u}(\beta) = Ry(-\theta)\cdot Rz(\phi)\cdot Rx(\beta)\cdot Rz(-\phi)\cdot Ry(\theta)$$



Transformaciones elementales 3D en OpenGL

- El comando de OpenGL:

```
glRotated(β, x, y, z);
```

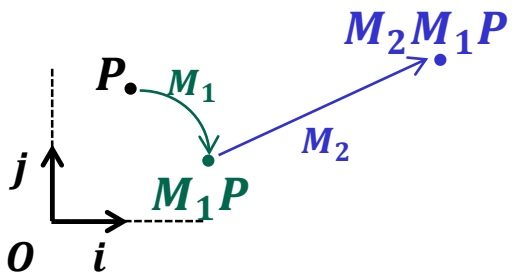
post-multiplica la matriz de modelado-vista por la matriz correspondiente a una rotación con respecto a una recta que pasa por el origen y tiene vector director **u**=(x, y, z, 0), de β grados en sentido anti-horario, tomado éste cuando se mira desde el punto (x, y, z, 1), al origen.

- La matriz correspondiente a esta rotación es:

donde c=cos(β), s=sen(β), y **u**=(x, y, z).

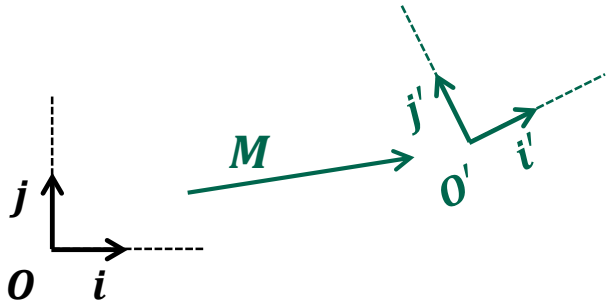
- Cuando la rotación es con respecto al eje Z de coordenadas se obtienen las rotaciones 2D.

- Las transformaciones afines vistas como transformaciones de objetos transforman puntos y vectores en un marco de coordenadas fijo.
- $M\begin{pmatrix} x \\ y \\ z \\ \blacksquare \end{pmatrix}$ son las coordenadas del punto/vector ($\blacksquare \in \{0,1\}$) transformado.
- Composición de transformaciones: primero M_1 y luego M_2



  - La transformación afín resultante es M_2M_1
  - Pre-multiplicación de matrices.
  - El producto de matrices no es conmutativo ⟹ ¡el orden en una sucesión de transformaciones importa!

- Las transformaciones afines vistas como transformaciones de marcos transforman marcos de coordenadas.



Simplificación en 2D!!

- El marco $\langle \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{0} \rangle$ se transforma en el marco $\langle \boldsymbol{i'}, \boldsymbol{j'}, \boldsymbol{O'} \rangle$

  - $\boldsymbol{i'} = \boldsymbol{M}\begin{pmatrix}\mathbf{1}\\\mathbf{0}\\\mathbf{0}\end{pmatrix} = \begin{pmatrix}\boldsymbol{m_{11}}\\\boldsymbol{m_{21}}\\\mathbf{0}\end{pmatrix}$ es el eje x transformado (primera columna de $\boldsymbol{M}$).
  - $\boldsymbol{O'} = \boldsymbol{M}\begin{pmatrix}\mathbf{0}\\\mathbf{0}\\\mathbf{1}\end{pmatrix} = \begin{pmatrix}\boldsymbol{m_{13}}\\\boldsymbol{m_{23}}\\\mathbf{1}\end{pmatrix}$ es el origen transformado (última columna de $\boldsymbol{M}$).

Coordenadas de → en el marco ↓	$\boldsymbol{i'}$	$\boldsymbol{j'}$	$\boldsymbol{O'}$
$\langle \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{O} \rangle$	$\begin{pmatrix}\boldsymbol{m_{11}}\\\boldsymbol{m_{21}}\\\mathbf{0}\end{pmatrix}$	$\begin{pmatrix}\boldsymbol{m_{12}}\\\boldsymbol{m_{22}}\\\mathbf{0}\end{pmatrix}$	$\begin{pmatrix}\boldsymbol{m_{13}}\\\boldsymbol{m_{23}}\\\mathbf{1}\end{pmatrix}$
$\langle \boldsymbol{i'}, \boldsymbol{j'}, \boldsymbol{O'} \rangle$	$\begin{pmatrix}\mathbf{1}\\\mathbf{0}\\\mathbf{0}\end{pmatrix}$	$\begin{pmatrix}\mathbf{0}\\\mathbf{1}\\\mathbf{0}\end{pmatrix}$	$\begin{pmatrix}\mathbf{0}\\\mathbf{0}\\\mathbf{1}\end{pmatrix}$

- Dado un punto/vector de coordenadas $\begin{pmatrix} x \\ y \\ \blacksquare \end{pmatrix}$ expresadas en el marco transformado $\langle i', j', O' \rangle$, $M \begin{pmatrix} x \\ y \\ \blacksquare \end{pmatrix}$ calcula las coordenadas de ese punto/vector en el marco original $\langle i, j, O \rangle$:

$$\begin{pmatrix} x \\ y \\ \blacksquare \end{pmatrix}_{\langle i', j', O' \rangle} = x\, i' + y\, j' + \blacksquare\, O' = x(M\, i) + y\, (M\, j) + \blacksquare\, (M\, O) =$$

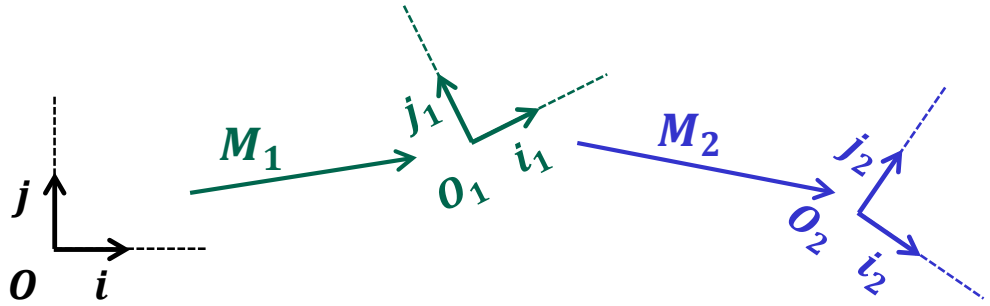
$$= M\, (x\, i) + M\, (y\, j) + \blacksquare\, (M\, O) = M\, (x\, i + y\, j + \blacksquare\, O) = \left[M \begin{pmatrix} x \\ y \\ \blacksquare \end{pmatrix} \right]_{\langle i, j, O \rangle}$$

M es la matriz del cambio de coordenadas desde el marco transformado al marco original: $M: \langle i', j', O' \rangle \longrightarrow \langle i, j, O \rangle$

- Consecuencia en el modelado:
  1. Modelamos localmente.
  2. Colocamos el marco local en el marco global.
  3. M resuelve el cambio de coordenadas de locales a globales!

OpenGL trabaja así!!

- Composición de transformaciones: primero M_1 y luego M_2



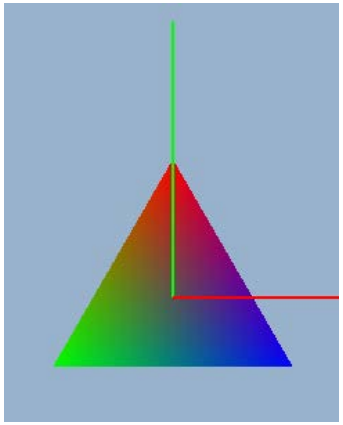
- $\boldsymbol{P}_{\langle i_2, j_2, o_2\rangle} =$
$= (\boldsymbol{M_2 P})_{\langle i_1, j_1, o_1\rangle}$
$= (\boldsymbol{M_1 M_2 P})_{\langle i, j, o\rangle}$

- La transformación afín resultante es $\boldsymbol{M_1 M_2}$
- Post-multiplicación de matrices.
- El producto de matrices no es conmutativo $\Longrightarrow$ ¡el orden en una sucesión de transformaciones importa!

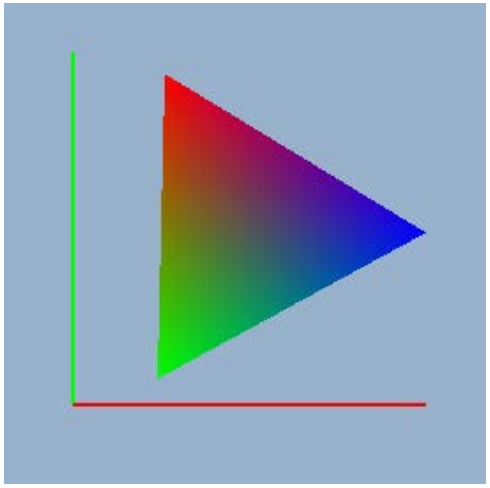
Orden en las transformaciones

- Las transformaciones afines no conmutan. ¿En qué orden se aplican?
- Ejemplo:

```
t = new TriangleRGB(10);
dmat4 m(1.0);
t->setModelMat(m);
grObjects.push_back(t);
```



- Se quiere dejar el triángulo en esta posición realizando una traslación sobre la diagonal y una rotación.
- ¿En qué orden?



Orden en las transformaciones

- Primera posibilidad:

```
m = rotate(m, 0.5, dvec3(0, 0, 1));
m = translate(m, dvec3(10, 10, 0));
```

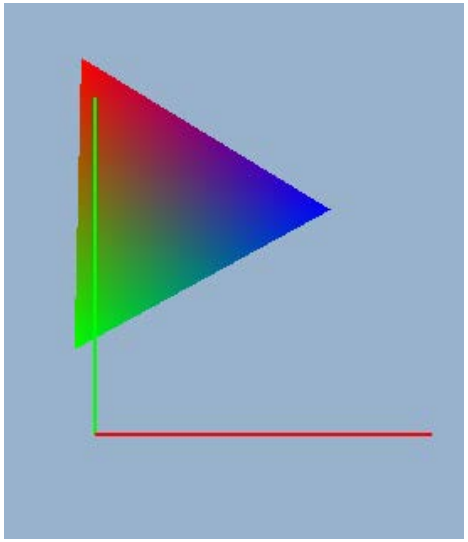
- Segunda posibilidad:

```
m = translate(m, dvec3(10, 10, 0));
m = rotate(m, 0.5, dvec3(0, 0, 1));
```

Orden en las transformaciones

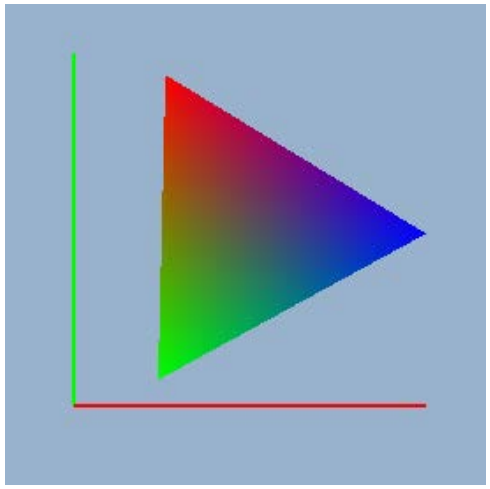
- Primera posibilidad (incorrecta):

```
m = rotate(m, 0.5, dvec3(0, 0, 1));
m = translate(m, dvec3(10, 10, 0));
```



- Segunda posibilidad (correcta):

```
m = translate(m, dvec3(10, 10, 0));
m = rotate(m, 0.5, dvec3(0, 0, 1));
```



Orden en las transformaciones

- Regla:

LA TRANSFORMACIÓN QUE PRIMERO SE QUIERE APLICAR ES LA ÚLTIMA QUE APARECE

- En el ejemplo:
  - Primero se gira el triángulo
  - Después se traslada
- Explicación. La matriz de modelado-vista que se aplica es:
  - `V*(T*R)` (por postmultiplicación de matrices, en el caso correcto)
  - `V*(R*T)` (en el "incorrecto")

  donde `V` es la matriz de vista
- Ejercicio. Explica por qué el caso incorrecto lo es.

Orden en las transformaciones

- En el caso del triángulo anterior, si lo queremos dejar como se dice, pero haciendo una traslación sobre el eje X y después una rotación, ¿cuál de las siguientes es la opción correcta?
- Primera posibilidad:

```
m = rotate(m, 0.5, dvec3(0, 0, 1));
m = translate(m, dvec3(10, 0, 0));
```

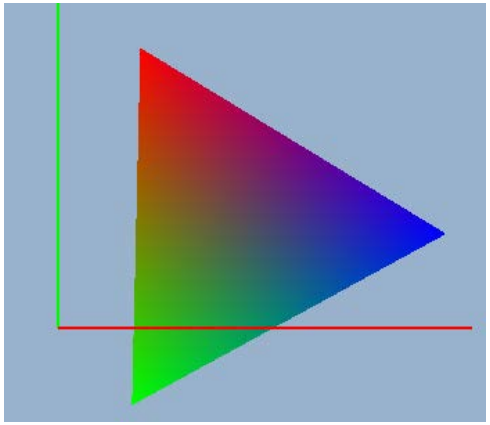
- Segunda posibilidad:

```
m = translate(m, dvec3(10, 0, 0));
m = rotate(m, 0.5, dvec3(0, 0, 1));
```

Orden en las transformaciones

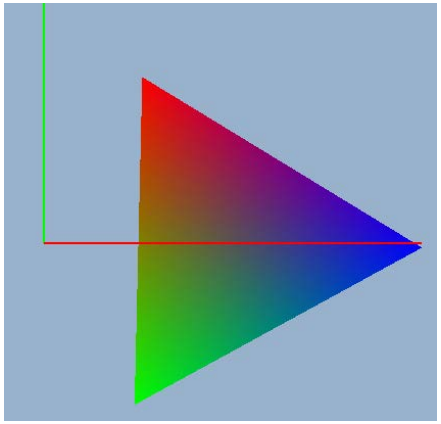
- En el caso del triángulo anterior, si lo queremos dejar como se dice, pero haciendo una traslación sobre el eje X y después una rotación, ¿cuál de las siguientes es la opción correcta?
- Primera posibilidad:

```
m = rotate(m, 0.5, dvec3(0, 0, 1));
m = translate(m, dvec3(10, 0, 0));
```

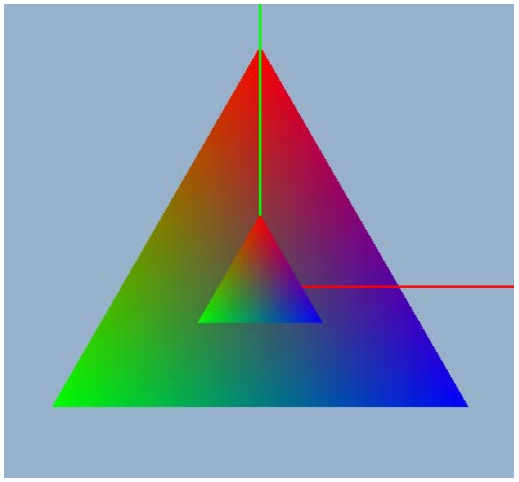


- Segunda posibilidad:

```
m = translate(m, dvec3(10, 0, 0));
m = rotate(m, 0.5, dvec3(0, 0, 1));
```



- Triángulo posicionado con respecto a otro triángulo. Se muestra con **`scene.render(camera.getViewMat());`** en **`display`**(). Antes, **`init()`** de **`Scene`** se ha definido así:

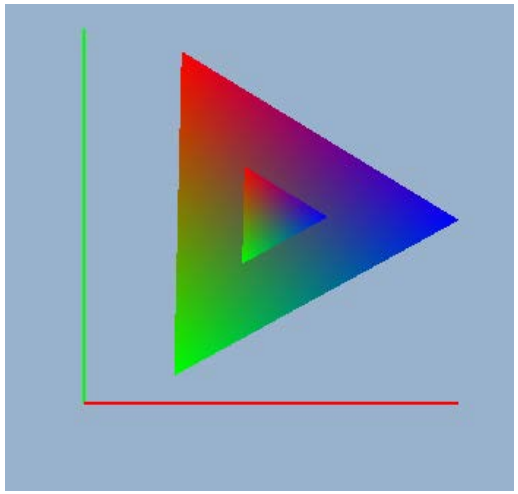


```
void init() {
    (new EjesRGB(200.0))->render(cam);  // Ejes
    Entity* t = new TriangleRGB(10);  // Triángulo grande
    dmat4 m(1.0);
    m = scale(m, dvec3(10, 10, 1));
    t->setModelMat(m);
    t->render(cam);
    Entity* tt = new TriangleRGB(3); // Triángulo pequeño
    dmat4 mm(1.0);
    mm = scale(mm, dvec3(10, 10, 1));
    mm = translate(mm, dvec3(0, 0, 10));
    tt->setModelMat(mm);
    tt->render(cam);
}
```

- Supongamos que se quiere que toda transformación que se haga al triángulo grande se le haga también al triángulo pequeño (la escalación, la rotación y traslación, ...)
- Opción primera:

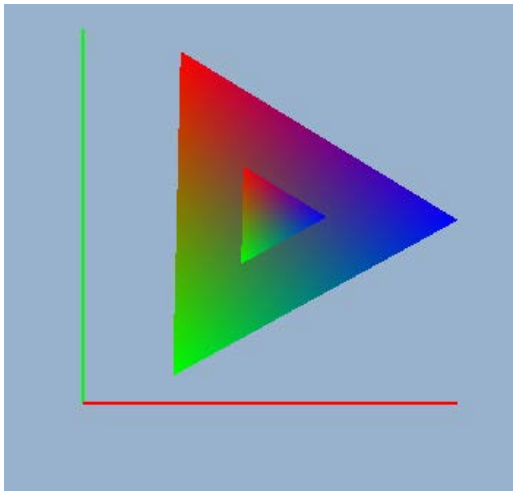
```
void posRel(Camera cam) { ...
    m = scale(m, dvec3(10, 10, 1));
    t->setModelMat(m);
    m = translate(m, dvec3(10, 10, 0));
    m = rotate(m, 0.5, dvec3(0, 0, 1));
    t->render(cam);

    Entity* tt = new TriangleRGB(3);
    dmat4 mm(1.0);
    mm = translate(t->getModelMat(), dvec3(0, 0, 10));
    tt->setModelMat(mm);
    tt->render(cam);
}
```



- Ventaja
  - Desaparece la escalación del triángulo pequeño, así como la traslación y rotación, porque se heredan del grande gracias a que la matriz de modelado del triángulo grande se incorpora como matriz inicial de modelado del triángulo pequeño
- Inconveniente
  - Es necesario conocer la **`modelMat`** del triángulo grande con respecto a la cual se posiciona y renderiza el triángulo pequeño. Una forma de hacerlo es añadir un parámetro más al **`render()`** de las entidades, pasando a haber dos parámetros. Uno es el que había, para conocer la matriz de vista (**`cam`**), y otro es nuevo, para conocer la matriz de modelado (**`t->getModelMat()`**)

- Opción segunda:



```
void posRel(dmat4 modelViewMat) { ...
    m = scale(m, dvec3(10, 10, 1));
    t->setModelMat(m);
    m = translate(m, dvec3(10, 10, 0));
    m = rotate(m, 0.5, dvec3(0, 0, 1));
    t->render(modelViewMat);

    Entity* tt = new TriangleRGB(3);
    dmat4 mm(1.0);
    mm = translate(mm, dvec3(0, 0, 10));
    tt->setModelMat(mm);
    tt->render(modelViewMat*t->getModelMat()); // (V*(S*T*R))*T
}
```

- Ventaja
  - Desaparecen la escalación, traslación y rotación del triángulo pequeño porque se heredan del grande pues la matriz de modelado del triángulo grande **se incorpora en la matriz con respecto a la cual se renderiza el triángulo pequeño**
- Inconveniente desaparece
  - No se necesitan dos parámetros al renderizar. Sencillamente el método **`render(m)`** se hace con respecto a una matriz `m` que incluye las matrices de vista y modelado, postmultiplicadas en este orden

- Conclusión:

Tenéis que modificar el método `render(Camera cam)` de `Scene` y de `Entity` del proyecto de forma que sea de la forma:

```
render(dmat4 const& modelViewMat)
```

- La llamada de **`display()`** pasa a ser de la forma:

```
scene.render(camera.getViewMat());
```