UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño
y programación de videojuegos

UNLVIRTUAL

MANIPULACION DE OBJETOS EN 2D

Unidad 2: Transformaciones. Implementación en OpenGL

Docente
Walter Sotil

Tutores
Emmanuel Rojas Fredini, Cristian Yones

CONTENIDOS

INTRODUCCIÓN

Para manipular objetos, ya sea en 2D (dos dimensiones), en 3D (tres dimensiones) o en cualquier otra dimensión, se necesita una herramienta básica que cumple la única función de llevar los puntos de un objeto de cierta posición a otra, deformando posiblemente el objeto.

Esta herramienta se llama transformación, función o mapeo.

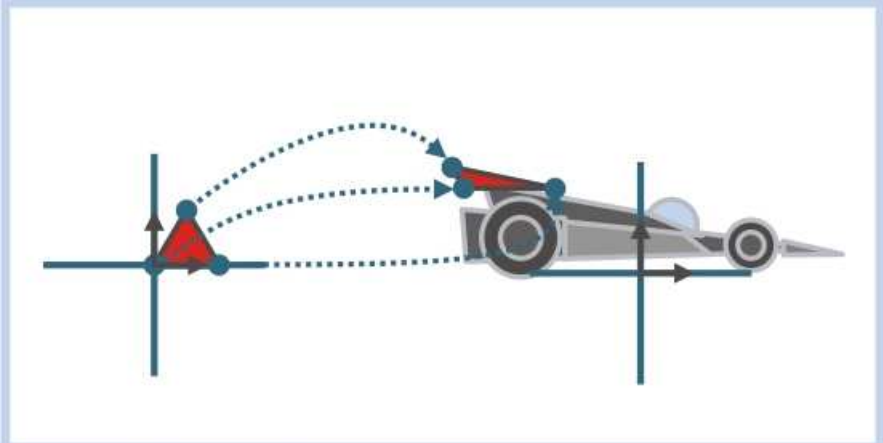

Fig. 1: Definición de transformación

En principio, se podría realizar cualquier transformación sobre todos los puntos de un objeto, una continua (botella → botella con un nudo en el cuello) o una discontinua (botella → botella rota), pero aquí se abordará un conjunto limitado solamente a unas pocas transformaciones continuas: trasladar, escalar, rotar y proyectar un objeto.

Estas operaciones tienen la ventaja de que, usando cierta abstracción del espacio, se pueden representar con una matriz y combinar o componer multiplicando de antemano las matrices de transformación, para evitar así la miríada de operaciones que llevaría hacerlas una por una.

Cualquier transformación ad-hoc (es decir, para un problema particular), que no sea representable mediante matrices, también puede realizarse, pero en un procedimiento aparte que deberá ser programado para tal fin. Aquí se omitirá este tipo de transformaciones, ya que son innecesarias para los propósitos del presente curso.

OpenGL está diseñando para realizar muy rápidamente operaciones con matrices y vectores. Multiplicando matrices de transformación por vectores que representan los puntos del modelo, se obtienen los puntos transformados en dos y tres dimensiones. Dicho de otro modo, se pueden obtener vectores transformados a partir de la multiplicación de las matrices de traslación, escalamiento y rotación por el vector original, aprovechando con ello la principal característica de las placas gráficas que es su gran eficiencia para resolver operaciones matriciales.

Transformación
También llamada función o mapeo, es una herramienta básica que lleva los puntos de un objeto de cierta posición a otra.
En términos matemáticos, consiste en hacer corresponder el dominio de la transformación con la imagen de la transformación. Por lo tanto, es un conjunto de pares donde, si se le aplica la transformación al primer elemento del par, éste se transforma en el segundo elemento.

2.1. Concepto de transformación

En términos matemáticos, una transformación consiste en hacer corresponder cada punto de un espacio, que se llama *dominio de la transformación* (***A*** en el esquema que se muestra a continuación), con otro punto de un (mismo) espacio, que se denomina *imagen de la transformación* (***B*** en el esquema).

Entonces, podemos ver a una transformación como un conjunto de pares donde, si se le aplica la transformación $\boldsymbol{T}$ al primer elemento del par, éste se transforma en el segundo elemento.

El gráfico 2 es un esquema de un mapeo. Como vemos, el objeto a se transforma en el g, es decir, el conjunto par ordenado (a, g) pertenece a la transformación $\boldsymbol{T}$.

Una transformación posee una condición elemental: que todo elemento del dominio se transforma en *exactamente un elemento* de la imagen. Puede haber más de un punto del dominio que se corresponda con un punto de la imagen, por ejemplo, al proyectar *b* y *e* en el esquema, pero todos y cada uno de los puntos del dominio tienen un solo punto correspondiente en la imagen.

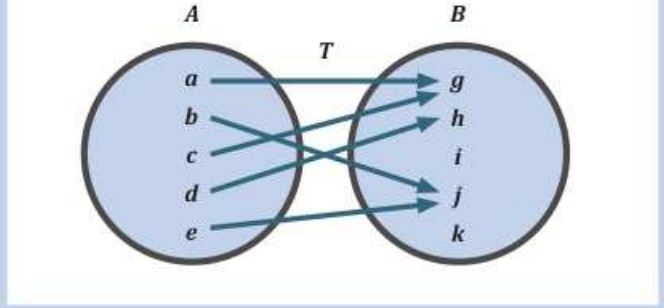

Fig. 2: Concepto de transformación

Como se desprende de la condición anterior, podría suceder que ningún elemento del espacio $\boldsymbol{A}$ se mapee en algún elemento del espacio $\boldsymbol{B}$. Por ejemplo, en el gráfico anterior, el elemento i no está en la imagen de la transformación.

Resumiendo:

Transformación:

Si $\boldsymbol{A}$ *y* $\boldsymbol{B}$ *son dos espacios con elementos* $\{a_1, a_2, \dots, a_n\}$ *y* $\{b_1, b_2, \dots, b_m\}$
Una transformación $\boldsymbol{T}$ *de* $\boldsymbol{A}$ *hacia* $\boldsymbol{B}$ *se escribe:*

$$\boldsymbol{A} \overset{T}{\Rightarrow} \boldsymbol{B}$$

Ésta se utiliza:

$$\boldsymbol{b_i} = \boldsymbol{T}(\boldsymbol{a_j})$$

Y tiene la propiedad de cada elemento a_j *sólo se transforma en 1 elemento de* B.

2.1.1. Composición de transformaciones

Composición de transformaciones
Consiste en la aplicación de sucesivas transformaciones a los puntos originales.

La composición consiste en la aplicación de sucesivas transformaciones a los puntos originales.

Como veremos luego, esto es muy aprovechado en computación gráfica, debido a que en lugar de calcular una matriz complicada, se efectúan varias transformaciones simples y entendibles.

Suponiendo que tenemos una transformación $\boldsymbol{A} \overset{\boldsymbol{T_1}}{\Rightarrow} \boldsymbol{B}$ y otra $\boldsymbol{B} \overset{\boldsymbol{T_2}}{\Rightarrow} \boldsymbol{C}$, como vemos, $\boldsymbol{T_1}$ mapea desde el espacio $\boldsymbol{A}$ al espacio $\boldsymbol{B}$, y $\boldsymbol{T_2}$ mapea desde el espacio $\boldsymbol{B}$ al espacio $\boldsymbol{C}$. Asimismo, es posible observar que si aplicamos las dos transformaciones a un elemento a perteneciente a $\boldsymbol{A}$, resulta una especie de tubería donde ingresa a como entrada a T_1 y la imagen de ésta es un elemento de $\boldsymbol{B}$.

Luego, aplicamos T_2 a ese elemento y la imagen de ésta es un elemento de $\boldsymbol{C}$. La composición es una operación que combina las dos transformaciones $\boldsymbol{T_1}$ y $\boldsymbol{T_2}$ en una nueva transformación equivalente, que será un mapeo desde $\boldsymbol{A}$ hacia $\boldsymbol{C}$, igual que al aplicar las dos transformaciones de forma secuencial, como se ve en la siguiente figura:

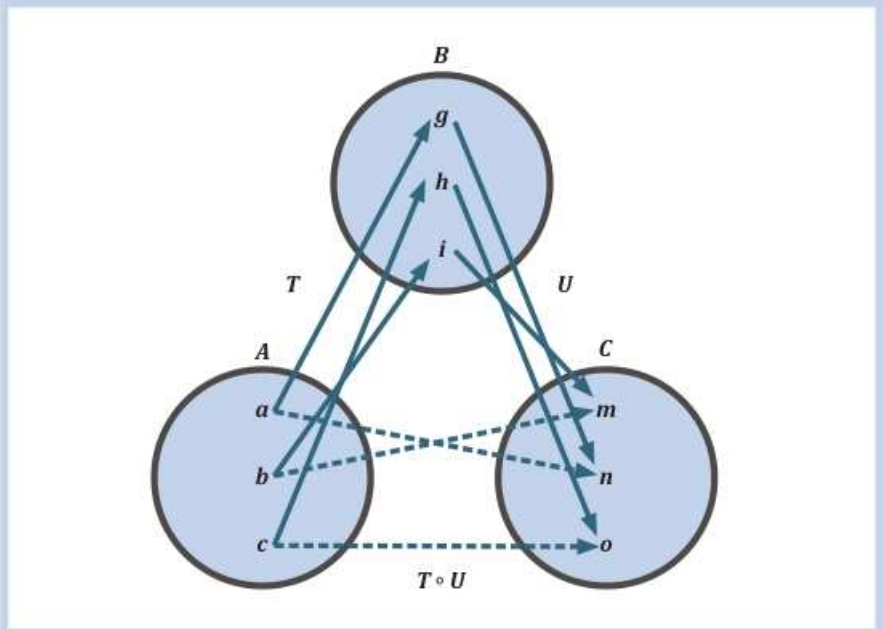

Fig. 3: Concepto de composición

Entonces, al aplicar las dos transformaciones tenemos:

$$\boldsymbol{T_2}(\boldsymbol{T_1}(a)) = (\boldsymbol{T_2}\circ\boldsymbol{T_1})(a) = c$$

El operador ° es el *operador composición.* Más adelante veremos que si representamos las transformaciones como matrices, el operador composición será la multiplicación matricial.

2.2. Transformaciones lineales

Constituyen un subconjunto de transformaciones que cumplen con la condición de linealidad. Por definición, una transformación o mapeo lineal es aquel que, siendo $\boldsymbol{V}$ y $\boldsymbol{W}$ dos espacios vectoriales, asigna a cada vector $\bar{\boldsymbol{x}}$ perteneciente a $\boldsymbol{V}$ un vector $T(\bar{\boldsymbol{x}})$ perteneciente a $\boldsymbol{W}$, y suponiendo que $\bar{\boldsymbol{u}}$ y $\bar{\boldsymbol{v}}$ son vectores pertenecientes a $\boldsymbol{V}$ y α un escalar, luego satisface las dos siguientes condiciones:

1. $T(\bar{u} + \bar{v}) = T(\bar{u}) + T(\bar{v})$
2. $T(\alpha\bar{v}) = \alpha T(\bar{v})$

Si se considera que trabajamos en un espacio vectorial de dimensiones finitas, lo cual se cumplirá ya que en el transcurso de la materia trabajaremos en 2D y 3D, podemos concluir que *toda transformación lineal puede ser representada por una matriz,* y es aquí donde éstas cobran valor para el contenido de la presente materia.

Transformaciones lineales
Constituyen un subconjunto de transformaciones que cumplen con la condición de linealidad. Permiten representar fácilmente las transformaciones, como el escalado, la rotación y la reflexión de objetos, mediante matrices.

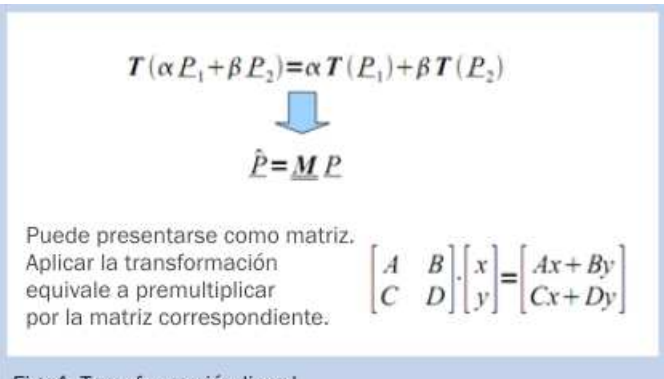

Fig. 4: Transformación lineal

Las transformaciones lineales de un vector pueden realizarse premultiplicándolo por una matriz cualquiera de NxN en N dimensiones.

Una característica primordial para mencionar es que las transformaciones lineales conservan el vector nulo:

$$T(\overline{\mathbf{0}}) = \overline{\mathbf{0}}$$ (donde $\overline{\mathbf{0}}$ es el vector de coordenadas $\{0,0\}$)

Entonces ¿qué significa esta operación?

$$\bar{\bar{\boldsymbol{T}}}\,\bar{\boldsymbol{v}} = \bar{\boldsymbol{x}} \qquad (1)$$

Con una operación como la de arriba, utilizando la multiplicación de una matriz por un vector, se puede realizar cualquier transformación lineal, como se la definió al comienzo.

Enfocándonos en lo que interesa a esta materia, ello significa que usando matrices se podrán realizar transformaciones sobre objetos para manipularlos en 2D o en 3D (o en *N* dimensiones, si se desea).

Entonces, las transformaciones lineales permiten representar fácilmente las transformaciones, como el *escalado,* la *rotación* y la *reflexión de objetos,* mediante matrices.

2.2.1. Transformación de escalado

Esta transformación altera el tamaño del objeto. Los elementos de la diagonal de la matriz se suelen llamar *factores de escala,* por su efecto sobre las componentes de los vectores, en tanto que los off-diagonal se denominan *factores de shear, corte* o *deslizamientos.*

Para cada vértice tenemos que aplicar los factores de escala S_x y S_y. Si S_x y S_y toman el mismo valor, será un escalado uniforme. Si el factor de escala es mayor a 1, se producirá un aumento del tamaño. En cambio, si el factor de escala es igual a 1, el tamaño no se modificará. Ahora bien, si el factor de escala es menor a 1, habrá una disminución del tamaño.

El vector transformado es el $\{x', y'\}$.

$$x' = x \,.S_x$$

$$y' = y \,.S_y$$

Luego, la versión matricial es:

Matriz de escalado:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Donde:

- S_x *es el escalado en la dirección* x
- S_y *es el escalado en la dirección* y

2.2.2. Transformación de reflexión

Para el espejado o reflexión se alteran los factores de escala S_x y S_y del siguiente modo:

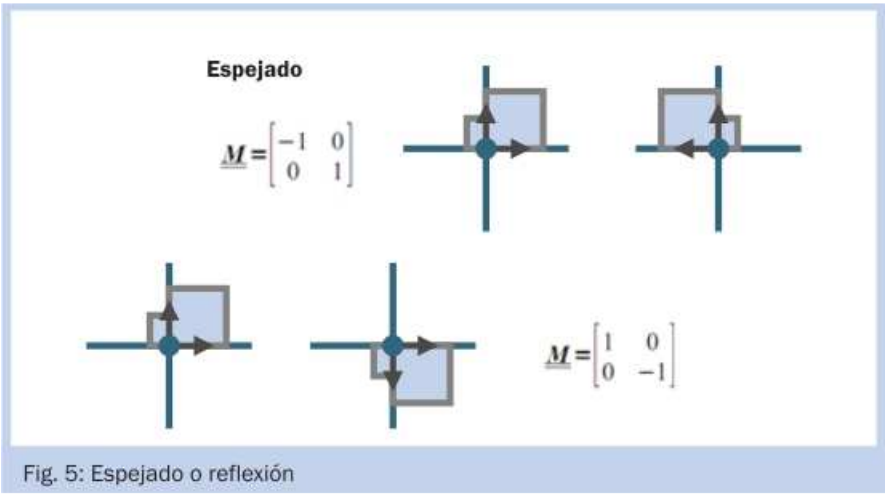

Fig. 5: Espejado o reflexión

2.2.3. Transformación de rotación

Ahora veremos la rotación de un punto, que se produce a lo largo de un arco de círculo:

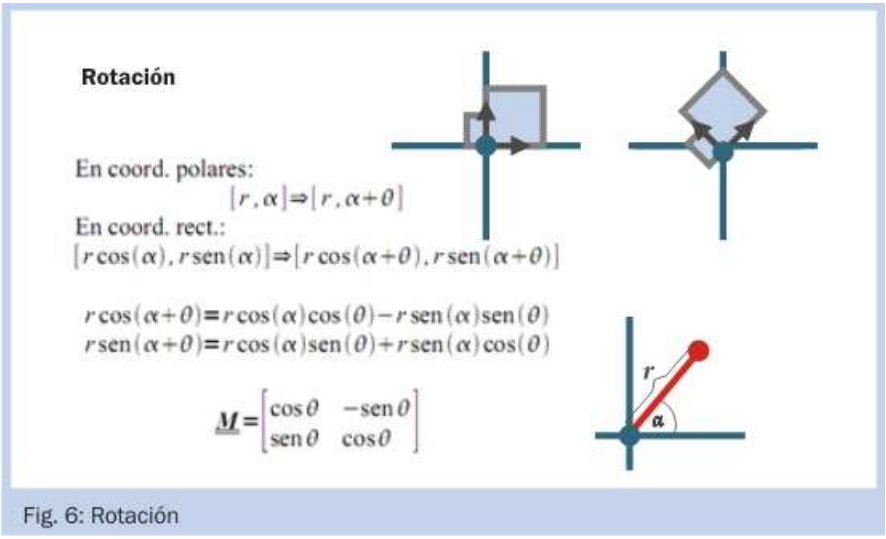

Fig. 6: Rotación

Donde *r cos(α)* es *x* ; y *r sen(α)* es *y*.

Utilizamos una identidad trigonométrica que dice:

$$\cos(\alpha \pm \beta) = \cos(\alpha)\cos(\beta) \mp sen(\alpha)sen(\beta)$$

2.3. ¿Cómo se ve una matriz?

"Lamentablemente nadie puede decir qué es la matrix, debes verla por ti mismo..." Ésta no sólo es una frase de una buena película, sino que en matemáticas también es cierto. No obstante, podemos ver una matriz como la transformación que ésta produce en los ejes coordenados, es decir, en los versores $\bar{x}$ e $\bar{y}$.

Por ejemplo, si tenemos la matriz:

$$\bar{\bar{M}} = \begin{pmatrix} 2 & -1 \\ 1 & 2 \end{pmatrix}$$

Entonces:

$$\bar{x}' = \bar{\bar{M}}\bar{x} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

$$\bar{y}' = \bar{\bar{M}}\bar{y} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

Por lo tanto, podemos representar la transformación como:

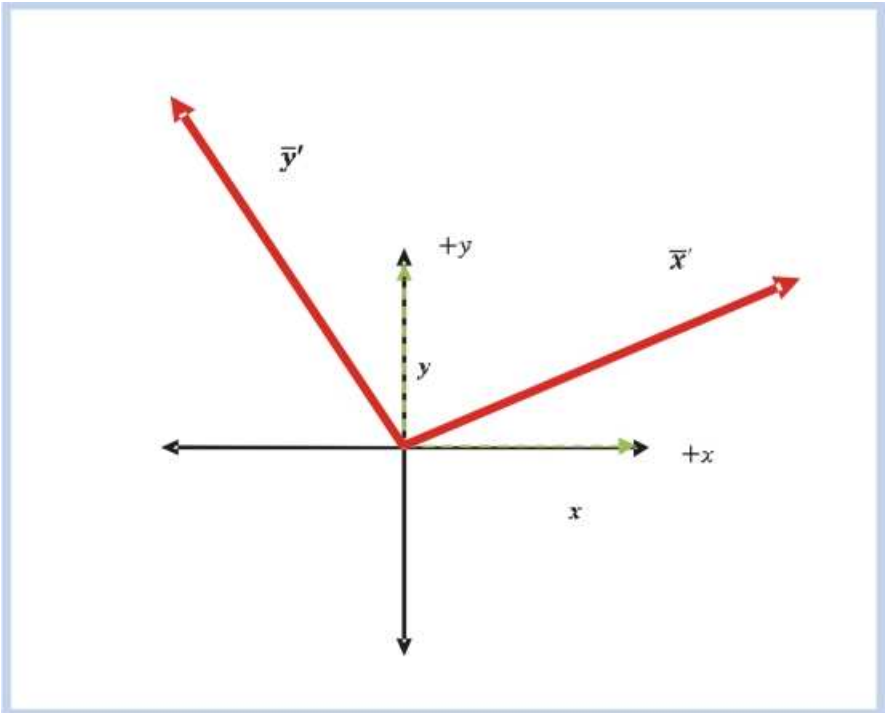

Fig. 7: Representación de los ejes transformados

La matriz anterior, entonces, representa una transformación de rotación y escalado.

Una transformación lineal está totalmente definida por la transformación de los versores. Es más, para deducir una transformación, la forma más simple de hacerlo es ver cómo lograr la transformación deseada sobre los versores. *Las columnas de la matriz representan los ejes transformados.*

2.4. Limitaciones de las transformaciones lineales

Lamentablemente, no se pueden realizar todas las transformaciones que necesitamos para manipular objetos usando transformaciones lineales: *una transformación lineal no puede representar una traslación.*

Geométricamente, una transformación lineal implica:

1. Una línea se transforma en una línea o en un punto, pero jamás en una curva.
2. Las líneas paralelas continúan siendo paralelas.
3. No se pueden realizar traslaciones.

Esta última limitación se deduce de las condiciones de transformación lineal, ya que la misma requiere que el vector nulo se transforme siempre al vector nulo, y en una traslación esto no se cumple. Dicho de otro modo, esta limitación es la más importante que tienen las transformaciones lineales, a saber, de no poder emplearse para realizar traslaciones.

En otras palabras, una transformación lineal puede solucionar el problema si en un juego deseamos rotar un tanque 45° , pero no podríamos desplazarlo.

2.5. Coordenadas homogéneas

La rotación y el escalado implican productos. En cambio, la traslación implica una suma.

La forma de solucionar este problema de las traslaciones es mediante la incorporación de las *coordenadas homogéneas.* Esto significa que por medio de las mismas podemos expresar todo como producto.

Así, la matriz de transformación para 2D, que era de 2x2, ahora pasa a ser de 3x3. Haciendo la analogía a 3D, la matriz de transformación ahora pasa a ser de 4x4 (como se verá luego, la matriz utilizada por OpenGL posee 16 números).

Las coordenadas en R^{n+1} {x,y,z...,w} se denominan coordenadas homogéneas del punto de P^n de coordenadas {x/w,y/w,z/w,...}.

El punto (x, y) pasa a convertirse en (x_w, y_w, w), donde $x = \frac{x_w}{w}$ e $y = \frac{y_w}{w}$.

En general, en los textos de CG (Computación Gráfica) se señala que $\{x, y, z, 1\}$ es un punto, mientras que $\{x, y, z, 0\}$ es un vector o dirección.

Por ahora, basta con hacer $w = 1$. De este modo, el punto (x, y) pasa a convertirse en $(x_w, y_w, 1)$.

La incorporación de w mediante las coordenadas homogéneas permite solucionar, además del problema de las traslaciones, otras cuestiones. Posibilita, por ejemplo, como veremos en la unidad de iluminación, configurar la fuente de luz en el infinito o localmente, seteando el parámetro de iluminación GL_POSITION con cero o uno, respectivamente (para agilizar o no los cálculos de iluminación).

Por lo tanto, la traslación ahora queda expresada del siguiente modo:

Matriz de traslación en coordenadas homogéneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Donde:

- t_x *es la traslación en el eje x*
- t_y *es la traslación en el eje y*

El escalado queda enunciado del siguiente modo:

Matriz de escalado en coordenadas homogéneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Donde:

- S_x *es el escalado en la dirección* x
- S_y *es el escalado en la dirección* y

La rotación queda definida del siguiente modo:

Matriz de rotación anti-horaria en coordenadas homogéneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sen(\theta) & 0 \\ sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Donde:

- θ *es el ángulo de rotación anti-horaria*

Matriz de rotación horaria en coordenadas homogéneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & sen(\theta) & 0 \\ -sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Donde:

- θ *es el ángulo de rotación horaria*

Como se puede ver, la matriz de rotación horaria es la transpuesta de la rotación anti-horaria y vice-versa.

2.6. Transformaciones rígidas y de similaridad

Transformaciones rígidas y de similaridad
Son las rotaciones junto con las traslaciones; las que no deforman el objeto.

Las *transformaciones rígidas* o *euclidianas* son las rotaciones junto con las traslaciones; las que no deforman el objeto.

E = R T Transformación rígida o euclideana.

Se suelen llamar *transformaciones de similaridad* (**S**) a las que no cambian la forma del objeto, admitiéndose sí un cambio en el tamaño y en la posición.

S = (α I) R T Transformación de similaridad (α = factor de escala; ***I*** = identidad).

Las transformaciones de similaridad conservan ángulos pero no distancias. Son las rígidas combinadas con un escalado uniforme.

2.7. Composición de transformaciones y combinación

Composición y combinación
La composición consiste en la aplicación de sucesivas transformaciones a los puntos originales. La combinación es una sola transformación que produce el mismo resultado que una composición.

Haremos aquí una extensión del concepto genérico de composición que vimos anteriormente, aplicado al uso en OpenGL.

En general, tanto en CAD (Diseño Asistido por Computadora) como en CG, lo que se hace es: en lugar de calcular una matriz complicada, se usan varias transformaciones simples y entendibles.

La composición consiste en la aplicación de sucesivas transformaciones a los puntos originales. La combinación es una sola transformación que produce el mismo resultado que una composición.

Aplicación de sucesivas transformaciones. Ejemplo:

1. Desplazar ($\boldsymbol{T}^1$): $\hat{\underline{P}}^1 = \boldsymbol{T}^1(\underline{P})$
2. Escalar ($\boldsymbol{T}^2$): $\hat{\underline{P}}^2 = \boldsymbol{T}^2(\hat{\underline{P}}^1) = \boldsymbol{T}^2(\boldsymbol{T}^1(\underline{P}))$
3. Rotar ($\boldsymbol{T}^3$): $\hat{\underline{P}}^3 = \boldsymbol{T}^3(\hat{\underline{P}}^2) = \boldsymbol{T}^3(\boldsymbol{T}^2(\boldsymbol{T}^1(\underline{P})))$

Combinación: Utilizando las matrices asociadas:

$$\begin{aligned} \underline{P}^3 &= \boldsymbol{T}^3(\boldsymbol{T}^2(\boldsymbol{T}^1(\underline{P}))) = \\ &= \underline{\boldsymbol{M}}^3 \cdot (\underline{\boldsymbol{M}}^2 \cdot (\underline{\boldsymbol{M}}^1 \cdot \underline{P})) = \\ &= \underbrace{\underline{\boldsymbol{M}}^3 \cdot \underline{\boldsymbol{M}}^2 \cdot \underline{\boldsymbol{M}}^1}_{\underline{\boldsymbol{M}}^*} \cdot \underline{P} = \underline{\boldsymbol{M}}^*(\underline{P}) \end{aligned}$$

Notar que el orden altera el resultado

Aquí vemos el efecto de dos traslaciones sucesivas mediante la combinación en una sola transformación:

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1}+t_{x2} \\ 0 & 1 & t_{y1}+t_{y2} \\ 0 & 0 & 1 \end{bmatrix}$$

Es fundamental recordar que la multiplicación de matrices no es conmutativa, por lo que el orden en cómo se apliquen las operaciones es muy importante. Esto significa que se obtendrán diferentes resultados, de acuerdo a la transformación que se aplique primero.

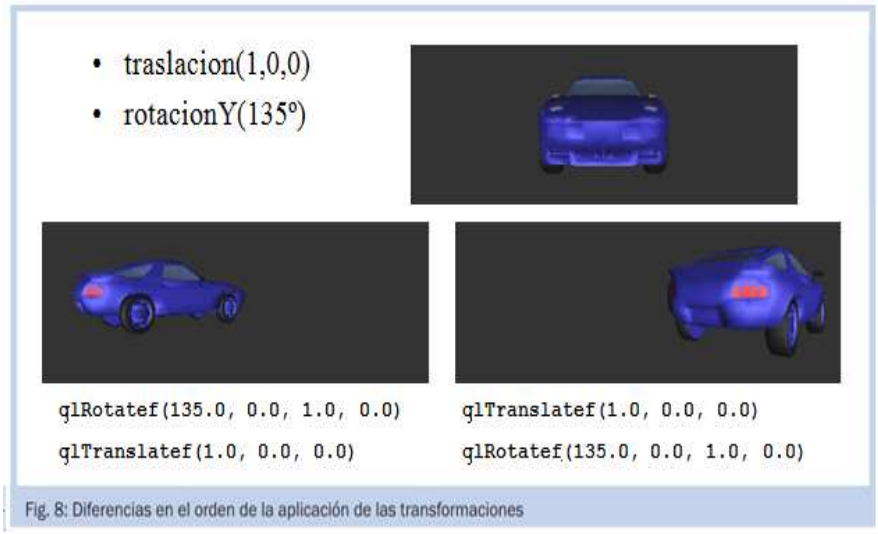

Fig. 8: Diferencias en el orden de la aplicación de las transformaciones

La principal ventaja que tenemos al usar las matrices de transformación es que si deseamos aplicar diferentes transformaciones a un mismo vector, podemos multiplicar antes las matrices correspondientes, consiguiendo el mismo resultado que si la aplicáramos de a una.

2.8. Transformaciones en OpenGL

OpenGL
Junto con las bibliotecas asociadas, compone las transformaciones usuales: las del modelo, las de proyección y las de texturas.

OpenGL y las bibliotecas asociadas componen las transformaciones usuales: las del modelo, las de proyección y las de texturas. Aquí analizaremos las dos primeras, que componen las transformaciones necesarias para posicionar los objetos, las luces y la cámara en la escena, proyectarlos y armar la imagen.

A continuación, veremos en detalle las funciones de OpenGL asociadas a dichas transformaciones.

Las transformaciones se ven en el orden en que aparecen en el programa, exactamente inverso al orden en que son realizadas.

Primero, las coordenadas de un elemento del modelo se mueven y deforman; luego, se mapean a un cubo unitario y, finalmente, se aplasta todo a una imagen plana que se acomoda en el monitor.

Las transformaciones se realizan mediante matrices activas. En determinado *momento* del programa hay una matriz activa y todas las coordenadas que se definan o calculen después serán multiplicadas por esta matriz. Por eso aparece esa confusión en el orden: cada transformación que se define se aplica a todo lo que venga después.

2.8.1. Dispositivo de salida y ventana

La ventana del programa o del dispositivo de salida se define al inicializar el subprograma gráfico, cuando se solicita una ventana al sistema operativo.

Como vimos anteriormente, con GLUT se efectúa de esta manera:

```
glutInitWindowSize(Wp, Hp);
glutInitWindowPosition(xp, yp);
```

Los parámetros son coordenadas *(x,y)* y tamaños (ancho, alto) de valores enteros en el sistema de coordenadas del dispositivo (denominado *device coordinates*), donde se vaya a renderizar.

Para GLUT y el SO, el punto de coordenadas (0,0) es el borde superior izquierdo (*upper,left*). Esta primera llamada es la última transformación que sufre el modelo en el pipeline, la cual mapea la ventana del programa en el dispositivo de salida.

La penúltima transformación es la que aplasta todo al plano y mapea en un viewport o ventana de dibujo con sus coordenadas de ventana (window coordinates). La ventana de dibujo es una región dentro de la ventana del programa:

```
glViewport(xv, yv, Wv, Hv)
```

Siendo los argumentos de esta función números enteros que definen el rectángulo en píxeles referidos a la ventana del programa, donde se dibujarán los objetos. Para OpenGL, a diferencia de GLUT y el SO, el origen es *lower, left*, es decir, debajo y a la izquierda.

El viewport se mapea y discretiza en los píxeles de la pantalla: una grilla regular de W_v columnas y H_v filas de píxeles, cada uno representando un rayo que atraviesa el modelo y pasa por el ojo:

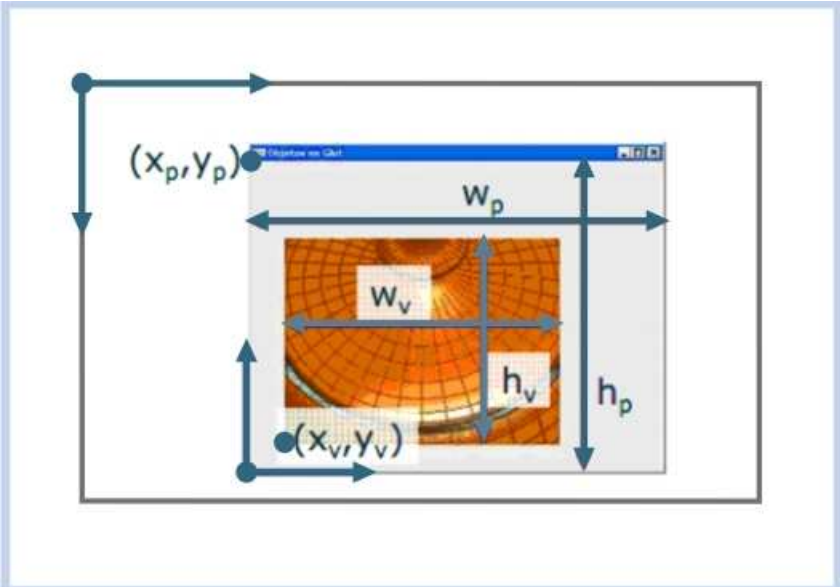

Fig. 9: Ventana del programa y viewport

Si glViewport no está presente, se considera que dicho rectángulo es toda la ventana del programa.

El tamaño del viewport cambia junto con el tamaño de la ventana del programa, por lo que sólo aparece en la rutina que maneja el resize. Es el programador el que decide cómo cambiar la(s) ventana(s) de dibujo cuando el usuario modifica la ventana del programa.

También es posible incluir varios viewports en la misma ventana, como se observa a continuación:

Fig. 10: Una ventana con varios viewports

En realidad, un programa puede definir varias ventanas y cada una de ellas puede tener varios viewports, pero no incluimos aquí los detalles de cómo realizarlo y cómo activar cada viewport para el redibujo.

Los detalles pueden verse en ejemplos y en el manual de especificación de GLUT, una vez que se comprenda el caso sencillo de una sola ventana con un solo viewport.

2.8.2. Stacks de matrices

A partir de aquí tenemos que definir la matriz activa con una llamada a la función glMatrixMode:

```
glMatrixMode(GLenum mode)
```

Esta función hace que las operaciones subsiguientes se apliquen sobre la matriz adecuada, indicada por el parámetro *mode,* que debe ser una de las siguientes tres constantes:

```
GL_MODELVIEW
GL_PROJECTION
GL_TEXTURE
```

OpenGL mantiene tres stacks o pilas de matrices homogéneas (4D). Como sus nombres indican, GL_MODELVIEW se utiliza para las transformaciones del modelo al sistema de la cámara, con una profundidad garantizada de 32 niveles; GL_PROJECTION para las proyecciones, que puede albergar al menos dos matrices, y la tercera, para la modificación de texturas (que no trataremos aquí).

Todas las transformaciones que ya se definieron en la teoría (traslación, escala y rotación), determinan la matriz de modelado GL_MODELVIEW y se aplica a cada uno de los vértices que definamos con glVertex.

En cada stack se parte de una matriz (normalmente, la *identidad*), que se va multiplicando por las que aparecen para obtener la matriz actual. La matriz *identidad* no ejerce ninguna modificación cuando se multiplica por un vector o por otra matriz. Esto significa que si $\bar{\bar{I}}$ es la matriz identidad, $\bar{\bar{M}}$ es cualquier otra y $\bar{v}$ es un vector cualquiera, tenemos que: $\bar{\bar{I}}\bar{v} = \bar{v}$ y $\bar{\bar{I}}\ \bar{\bar{M}}\ \bar{v} = \bar{\bar{M}}\,\bar{v}$.

Hay tres operaciones básicas que alteran la matriz actual:

- Reemplazo por pop
- Reemplazo por carga
- Multiplicación

Push y pop: son funciones de manejo del stack:

```
glPushMatrix() y glPopMatrix()
```

La llamada al *push* guarda (salva) una copia de la matriz actual en el stack, sin modificarla; la altura del stack aumenta una unidad. La llamada al *pop* reemplaza la matriz actual por la última que se puso en el stack (*lifo* o last-in, first-out); la altura del stack decrece una unidad.

Si se desea aplicar una transformación sólo a un determinado cuerpo de nuestra escena, es necesario definir –primero- las trasformaciones que afectarán a toda la escena y los vértices que no se verán modificados por la transformación nueva. Luego, se deberá salvar la matriz en una pila con la instrucción glPushMatrix(). Posteriormente, habrá que cargar la transformación nueva y definir los vértices de las figuras afectadas. Ahora bien, si nos interesa seguir definiendo vértices, que no deberían ser afectados por esta transformación, debemos volver al estado anterior, recuperando la matriz desde la pila con glPopMatrix().

La función:

```
glLoadMatrixd(GLdouble *M)
```

carga el conjunto de 16 doubles apuntados por *M* y lo pone en lugar de la matriz actual; la que estaba no se guarda, se perdió. Esta llamada es para introducir alguna matriz especial calculada por software:

```
glLoadIdentity()
```

Como dijimos anteriormente, se trata de una llamada especial que carga la matriz *identidad* y es habitual para empezar el proceso, el punto de partida.

La combinación de transformaciones se realiza con operaciones automáticas de multiplicación, que puede hacerse con matrices generadas automáticamente por OpenGL o multiplicando por una matriz guardada en un array:

```
glMultMatrixd(const GLdouble *M)
```

Como vemos, multiplica *M* por la matriz actual y pone el resultado como matriz activa actual.

Nota: OpenGL usa las matrices ordenadas por columnas (transpuestas):

1) Si *M* se define como double[16], el segundo elemento es el de la columna 0 y fila 1.
2) Si se define como double[4][4], *M*[0][1] es el elemento en la columna 0 y fila 1, *M*[2] es la columna 2.

Las operaciones habituales como translación, rotación y escalado se pueden aplicar en forma automática, usando alguna de las siguientes llamadas:

```
glTranslated(tx, ty, tz)
glRotated(ang, rx, ry, rz)
glScaled(sx, sy, sz)
```

La operación de Traslación se especifica con glTranslate*, esta recibe tres parámetros que son el número de unidades que deseamos trasladar la figura en cada uno de los tres ejes. También podemos pasar como parámetros valores negativos y así obtener traslaciones en sentidos contrarios.

La rotación se define con la función glRotate*, la cual recibe cuatro parámetros. El primero es el número de grados que nos interesa rotar el cuerpo o la escena y los otros tres determinan el nivel en que se verá afectada la operación respecto a cada uno de los 3 ejes X, Y y Z. Por ejemplo, si nos interesara hacer una rotación de 30º solo sobre el eje X, pasaríamos un 30 en el primer parámetro, un 1 en el segundo, y 0 en los restantes. El sentido de la rotación está dado por el signo del ángulo y la regla de la mano derecha sobre el vector.

El escalamiento se define con glScale*. Esta función recibe tres parámetros que definen el grado de escalamiento que deseamos aplicar a cada uno de los tres ejes.

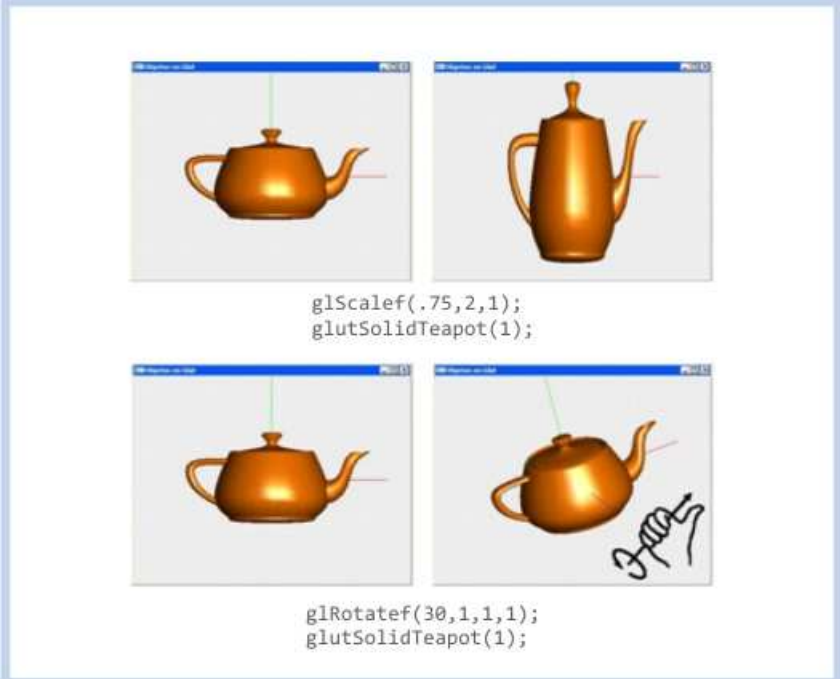

Fig. 11: Transformaciones utilizando funciones de OpenGL

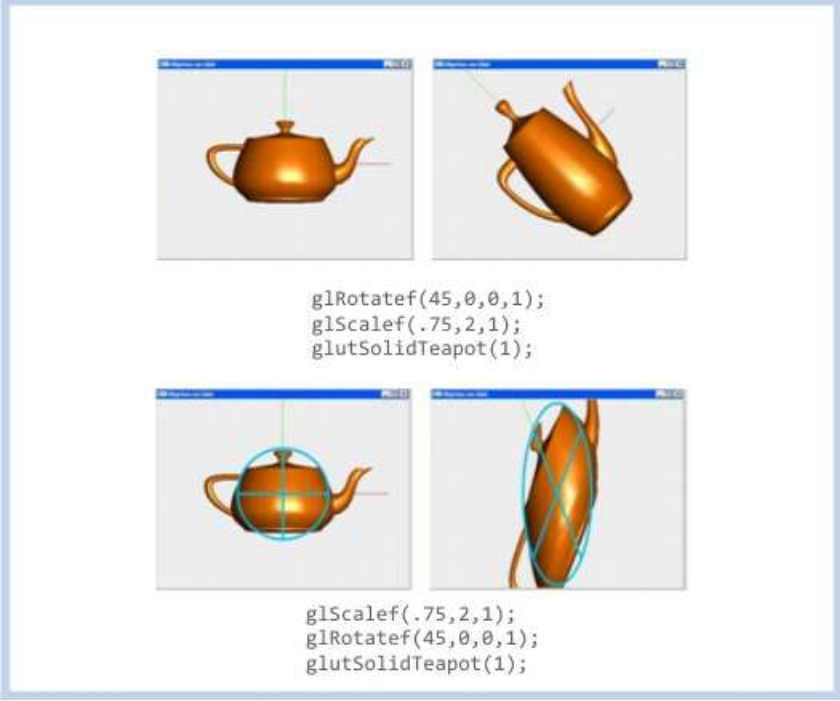

Figura 12: Composición: Arriba: rotación y escala. Abajo: escala y rotación.

En todos los casos de composición, la matriz actual $\overline{\overline{\boldsymbol{A}}}$ es reemplazada por una que consiste en el producto de $\overline{\overline{\boldsymbol{A}}}$ por la matriz que entra, $\overline{\overline{\boldsymbol{M}}}$: $A^i_j \leftarrow A^i_k \, M^k_j$. Las operaciones las hace OpenGL en forma automática y normalmente en la placa gráfica, lo que logra una aceleración impresionante de los cálculos, dado que es un procesador diseñado como calculadora de matrices.

2.8.3. Proyección

Pese al nombre, la matriz de proyección no proyecta, pero sí define cómo se realizará la proyección.

En primer lugar se llama a:

```
glMatrixMode(GL_PROJECTION); glLoadIdentity();
```

OpenGL ofrece dos modos de proyección o *modelos de cámara:* ortogonal y perspectiva central, cuyas matrices se pueden generar automáticamente, a partir de algunos datos relevantes.

Proyección ortogonal (paralelepípedo):

```
glOrtho(left, right, bottom, top, near, far)
```

Proyección perspectiva (frustum o tronco de pirámide):

```
glFrustum(left, right, bottom, top, near, far)
```

Estos datos se definen en el sistema de coordenadas de la cámara o del observador. En ambos casos se especifican las coordenadas que limitan el volumen a visualizar. En este modelo, la cámara no ve desde cero hasta el infinito; sólo será visible lo que quede entre dos planos, uno cercano al ojo, a distancia *near,* y el otro, más alejado del punto de vista, a distancia *far.*

Para la proyección ortogonal, el campo visual es un paralelepípedo. Se definen las posiciones de los seis planos que recortan el espacio (*clipping planes*). Aquí, el ojo puede considerarse en el infinito y los *clipping planes* como laterales paralelos, pero aun así los valores de near y far deben ser finitos para calcular oclusiones y obtener un valor finito de profundidad a fin de realizar las conversiones de coordenadas.

Para definir los planos near y far, una opción válida es utilizar una bola envolvente del modelo o *bounding ball.*

En el caso de la proyección perspectiva, el campo visual es un tronco de pirámide o frustum (en Latín), definido también por seis planos:

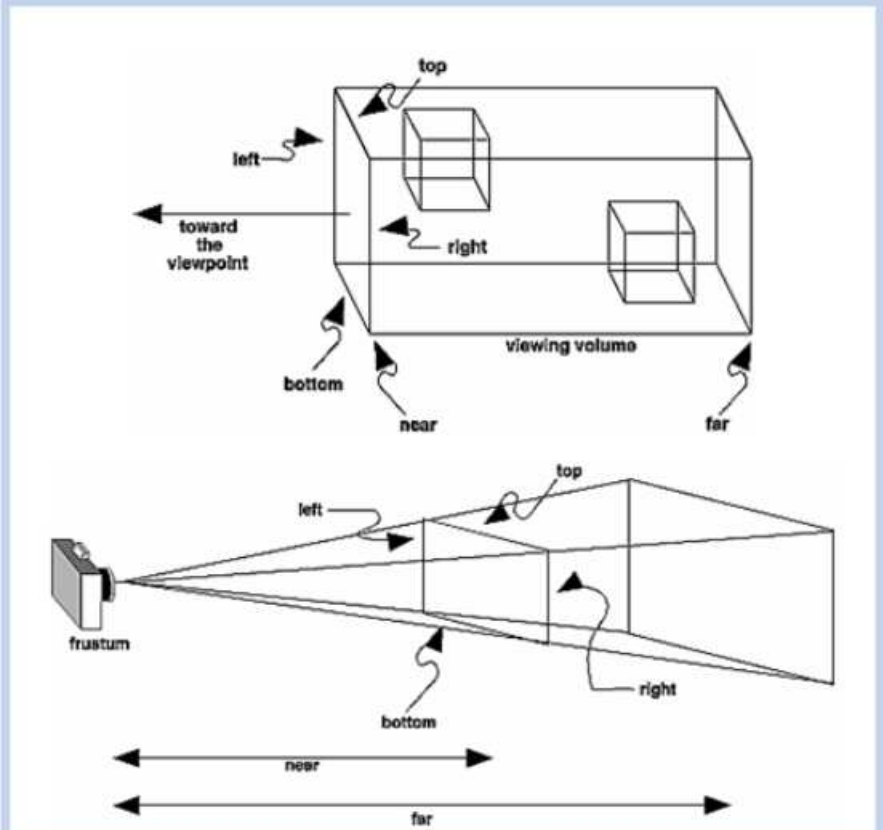

Fig. 13: Izquierda: modo de proyección ortogonal.
Derecha: modo de proyección perspectiva central.
(Figuras extraídas del capítulo 3 –"Viewing"- del Red Book).

Todo lo que esté por fuera de ese recinto no se visualiza (*culling* = descarte) y los objetos que lo atraviesan serán recortados (*clipping* = recorte). Sólo se verá la parte interior. Es decir que la matriz de proyección sólo define los seis planos principales de recorte o *clipping* de la escena.

En perspectiva central, los planos *near* y *far* deben estar delante del ojo (far > near > 0). Dado que las bases del frustum tienen distinto tamaño, los parámetros que definen los planos laterales están fijados, por convención, en el plano *near*. La distancia del ojo al plano *near* es, entonces, la que define el ángulo del campo visual (*field of view* o *fov*).

Otra opción para definir la proyección perspectiva es utilizando una función de la biblioteca GLU (del inglés *OpenGL Utility Library*):

```
gluPerspective(fovy, aspect, zNear, zFar),
```

fovy es el ángulo de visión vertical en grados y *aspect,* la relación ancho / alto.

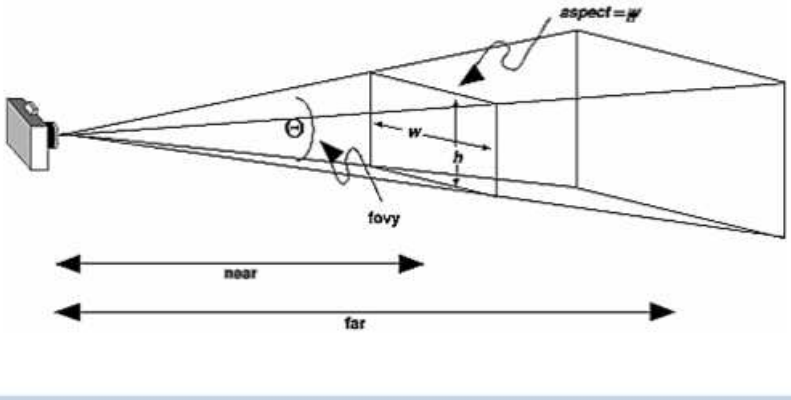

Fig. 14: modo de proyección perspectiva, utilizando la función gluPerspective. (Figuras extraídas del capítulo 3 –"Viewing"- del Red Book).

Notas

Luego de definir el modelo de cámara, se realiza la división por *w* (división perspectiva), y el volumen de visualización tridimensional o *clip space* se mapea internamente en un cubo $[-1,1]^3$ (Normalized Device Coordinates), conservando la profundidad para poder realizar el ocultamiento de líneas. Este cubo es el que se mapea al viewport, definiendo la escala visual en píxeles/unidad, para luego rasterizar las figuras y descartar los fragmentos ocultos por otras entidades.

Este volumen ya está definido en unidades espaciales, pero aún no está ubicado en la escena.

Aquí aparecen dos detalles importantes de la perspectiva:

1) al mapear el frustum en un cubo, los objetos alejados se comprimen más que los cercanos y se verán más pequeños;
2) la coordenada *z* no se mapea linealmente; 1/z sí, de modo que hay más precisión cerca del ojo. En el modelo ortogonal, *z* sí se mapea linealmente y por lo tanto con precisión independiente de la profundidad. En ambos casos, *z* vale -1 en el plano *near* y 1 en el plano *far*, por lo que el sistema NDC es izquierdo.

GLU (The OpenGL Graphics System Utility Library) es un conjunto de rutinas diseñadas para complementar a OpenGL. Ofrece funciones de dibujo de alto nivel, basadas en primitivas de OpenGL. Las funciones de GLU se reconocen fácilmente debido a que todas empiezan con el prefijo *glu.*

2.8.4. Composición de la escena

Todos los objetos se dibujan y posicionan en un sistema de coordenadas arbitrario que se suele denominar *world coordinate system* (sistema de coordenadas global) o *espacio del modelo*. Este sistema es arbitrario, determinado por el usuario o el programador.

Los objetos individuales se suelen definir en un sistema propio del objeto, pero luego son ubicados en el espacio (intermediario) del modelo y, finalmente, todas las coordenadas son transformadas al espacio visual de la cámara.

Se comienza con la llamada de definición del estado inicial de la matriz model-view:

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

La identidad nos coloca, a partir de aquí, en el sistema de coordenadas visual o de la cámara, que tiene el eje *x* positivo hacia la derecha, el *y* hacia arriba y el *z* hacia atrás del ojo.

Una vez definido el tipo de cámara, hay que ubicarla en la escena, asignarle posición y orientación (*camera aim* = hacia donde apunta). Hay dos formas de hacerlo: una consiste en ubicar la cámara (el sistema de coordenadas del ojo) en el espacio del modelo con gluLookAt(), mientras que la otra manera es ubicando el espacio del modelo en el sistema del ojo, con traslaciones y rotaciones. La primera es mucho más intuitiva:

```
gluLookAt(eyex,eyey,eyez,centerx,centery,centerz,upx,upy,upz)
```

Los parámetros son las coordenadas del ojo (*eye*) o cámara, el punto al que se mira (*center* o *target*) y un vector *up* que indica la dirección de la cabeza o parte superior de la cámara, para fijar la rotación de la cámara alrededor del eje ojo-centro.

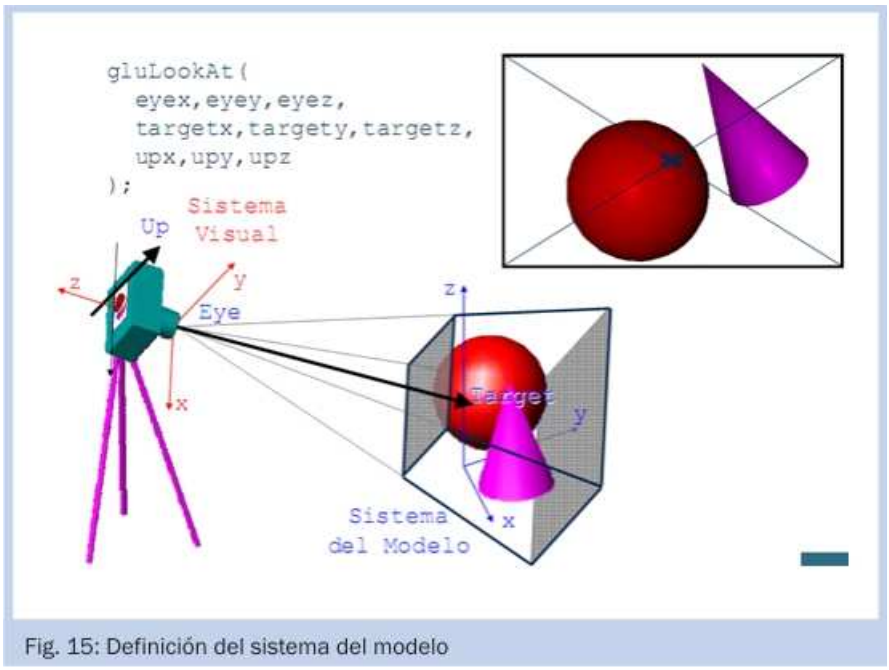

Fig. 15: Definición del sistema del modelo

Ahora estamos en el sistema de coordenadas global o espacio del modelo.

Aquí suelen posicionarse las luces fijas respecto del modelo, luces en el techo, el cielo o lámparas. Pero la luz *tipo flash* y los objetos que se mueven con la cámara se definen antes que la posición de la cámara (antes del LookAt), directamente en el sistema de coordenadas visual; por ejemplo, el frente y las luces de un auto desde el cual se está mirando o la mano y la linterna del protagonista que ve la escena.

Ahora sólo resta ubicar los objetos, utilizando la *forma jerárquica de las transformaciones.*

El stack de matrices permite armar modelos complejos, utilizando partes preensambladas. Cuando se preensambla una pieza en una rutina, se cargan y se mueven algunos objetos (probablemente preensamblados en otra rutina).

Un ejemplo es la construcción de un automóvil:

```
Posicionar el auto en el modelo
         Dibujar auto: {
                  Push (matriz auto)
                  Mover el sistema de coordenadas al centro de rueda delantera
izquierda
                  Dibujar rueda: {
                           Push (matriz rueda di)
                           Posicionar bulón 1
                           Dibujar bulon 1: {Push…….Pop}
                           Pop (matriz rueda di)
                           …..
                           …..
                           Push (matriz rueda di)
                           Posicionar bulón 4
                           Dibujar bulon 4: {….}
                           Pop (matriz rueda di)
                           Dibujar cubierta
                           Dibujar llanta
                  }
                  Pop (matriz auto)
                  ….
                  ….
                  Mover al centro de rueda trasera izquierda
                  Dibujar rueda: {….}
                  Dibujar otras partes del auto....
                  Pop (matriz auto)
         }
```

La rutina para dibujar el auto se puede llamar tantas veces como se quiera, armando una rutina dibujar_auto() y definiendo las transformaciones necesarias para llevar cada auto a la posición que le corresponde en la escena. Lo mismo sucede con las piezas. Por eso es indispensable que cada rutina de dibujo *deje las cosas como estaban* antes de salir, esto es, las matrices y el resto del estado (mediante glPushAttrib y glPopAttrib).

El modelado en sí suele realizarse en programas de CAD o cualquier otro software para luego dejar que OpenGL posicione las primitivas leídas. También puede desarrollarse directamente en OpenGL, pero con muchísimas limitaciones, componiendo primitivas básicas con algunas piezas de alto nivel provistas por GLU o GLUT. No obstante, ninguno de ellos tan *amigable* como un CAD para dibujar los objetos.

2.8.5. Pipeline

La secuencia de operaciones es la siguiente:

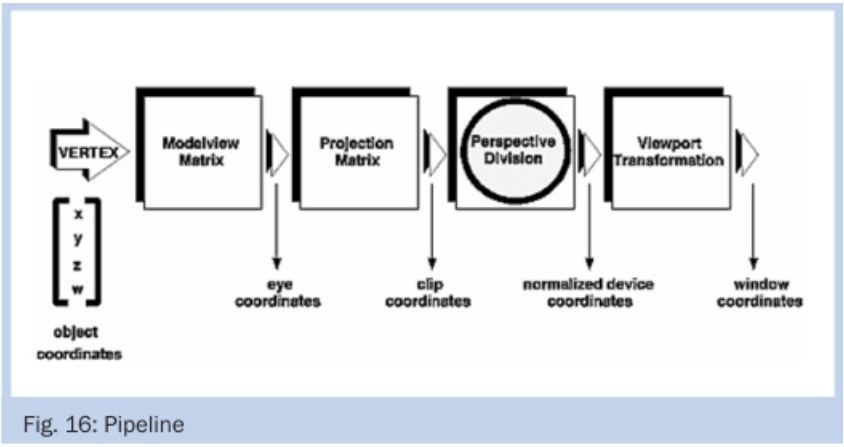

Fig. 16: Pipeline

Toda la escena: modelo, luces y cámara, se transforma mediante la matriz modelview (model→view) hacia el sistema visual.

La escena, vista desde el ojo, es transformada luego mediante la matriz de proyección, que recorta (clip) el espacio visible, aunque conservando el *w* del espacio proyectivo (4D).

Después, se procede a la división perspectiva, que simplemente divide las coordenadas por *w* y *normaliza* a un cubo $[-1,1]^3$, conservando un *z* no-lineal, que aún sirve para hacer ocultamiento.

Posteriormente, se utiliza la definición del glViewport para acomodar los valores a un dispositivo de salida específico (conservando un depth buffer).

Finalmente se realizan las operaciones *por píxel,* rasterizando, ocultando y rellenando con color o textura.

BIBLIOGRAFÍA

Cátedra Computación Gráfica, FICH - UNL-, 2010 [en línea] [CIMEC] www.cimec.org.ar/cg

Hearn, D.; Baker, P. *Computer Graphics, C version.* Second Edition. Pearson Prentice Hall, 1997.

Neider J.; Davis,T.; Woo M. "OpenGl Programming Guide: The Official Guide to Learning OpenGL". Addison-Wesley Publishing Company, 1997 [en línea] [OpenGL] http://www.opengl.org/documentation/red_book/

Kilgard, M. J. *The OpenGL Utiliy Toolkit (GLUT) Programming Interface.* Silicon Graphics, Inc, 1996 [en línea] [OpenGL] http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf

Chin, N.; Frazier, C.; Ho, P.; Liu, Z.; Smith, K. *The OpenGL Graphics System Utility Library (GLU).* 1998 [en línea] [OpenGL] http://www.opengl.org/documentation/specs/glu/glu1_3.pdf