Transformaciones

Seminario ISGI (S3)

R. Vivó

R. Vivó

Transformaciones

Seminario ISGI (S3)

Este seminario explica cómo trasladar, escalar y girar los objetos para que tengan el tamaño, posición y orientación que se desea. Se van a introducir los conceptos de matriz de transformación y encapsulamiento de transformaciones usando una pila.

Transformaciones afines

Las transformaciones son operaciones matemáticas sobre las coordenadas de los puntos del espacio 3D. Para cambiar su posición de un punto usaremos una traslación, es decir, sumaremos un vector de desplazamiento al punto para obtener el desplazado. Para escalar un punto multiplicaremos sus coordenadas por unos factores de escala en cada dirección principal. Para girar un punto alrededor de un eje multiplicaremos sus coordenadas por senos y cosenos del ángulo que se quiere girar y después las sumaremos convenientemente.

Cada una de estas tres transformaciones -traslación, escalado y rotación- puede representarse matemáticamente como una matriz 4x4 de números reales que al multiplicar las coordenadas del punto original dan el punto transformado:

$$P_t = M \cdot P$$

donde P es el punto original, M la matriz de transformación y P_t el punto transformado. El punto P tiene como coordenadas $(x,y,z,1)$, siendo necesaria la cuarta para poder hacer la multiplicación de una matriz 4x4 por una matriz columna 4x1. La cuarta coordenada, a la que se nombra como w, vale siempre 1 excepto cuando tratamos con la proyección en perspectiva, cosa que aquí no preocupa. Si queremos aplicar una transformación a un objeto basta con aplicarla a todos los vértices que lo definen y volverlo a dibujar.

Los valores de los elementos de M dependen, como es natural, de los valores que tenga la transformación a aplicar. Por ejemplo, si se quieren aplicar los factores de escala sx, sy y sz en las direcciones principales, basta con construir una matriz cuya diagonal principal sea $sx,sy,sz,1$ y el resto ceros.

Hay que hacer notar que aquí se siguen los dos mismos convenios que en OpenGL. El primero es que el sistema 3D de coordenadas es dextrógiro con el eje $\boldsymbol{X}$ hacia la derecha, el eje $\boldsymbol{Y}$ hacia arriba y el eje $\boldsymbol{Z}$ saliendo del papel, como orientación por defecto. De esto se deduce que los giros son positivos cuando se ven en el sentido contrario a las agujas del reloj. El segundo es que las coordenadas de los puntos se representan en columna, por lo que las matrices de transformación se aplican por la izquierda como ya se ha visto.

Composición de transformaciones

Si se quiere cambiar el tamaño de un objeto (escalado) y después moverlo a otro sitio (traslación) no es eficiente transformarlo dos veces. Afortunadamente, podemos componer el escalado y la traslación, multiplicando las matrices que los representan para dar una única matriz compuesta.

Es muy importante acertar en el orden de los factores, pues la multiplicación de matrices no es conmutativa y conduce a resultados diferentes. Debemos pues tener presente esta regla: **las matrices de transformación se componen por la izquierda, es decir, la primera que se aplica es la**

de más a la derecha. Así, en el ejemplo del párrafo anterior, la primera que se aplica es la matriz de escalado (S) y la segunda la matriz de desplazamiento (T), por lo que la composición será T·S.

No hay límite en el número de transformaciones a aplicar pues al final todas acaban siendo una compuesta, que es la que se aplicará a las coordenadas de los vértices del objeto a transformar. A esta matriz se la conoce como **matriz del modelo**. La función que cumple esta matriz es convertir las coordenadas de un objeto, que están en un sistema propio donde se ha modelado más fácil, a su tamaño, posición y orientación en la escena en el sistema de coordenadas general, también llamado del mundo real. Así, por ejemplo, podemos modelar un libro como una caja de lados paralelos a los planos principales y después, mediante transformaciones, crear la escena de una biblioteca con libros de diferentes tamaños en diferentes posiciones y orientaciones.

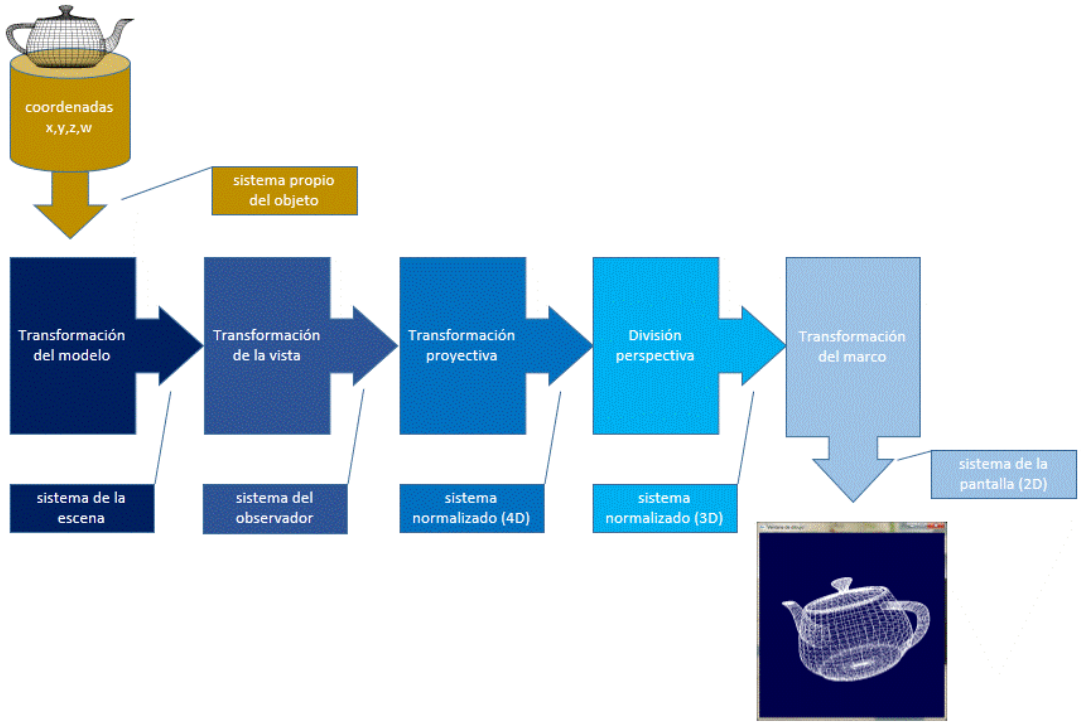


Figura 1. Cadena de transformaciones

Cambio de sistemas de coordenadas

La figura 1 muestra cómo fluyen las coordenadas de un vértice desde el sistema propio del modelo hasta las coordenadas de los pixeles en la pantalla. Cada caja atravesada representa un cambio de sistema. El cambio se consigue aplicando una matriz de transformación, en general compuesta. La matriz del modelo es la primera.

OpenGL funciona como una máquina de estados, incluidas las transformaciones. Esto quiere decir que cuando se ordene el dibujo de un punto, con `glVertex` por ejemplo, se aplicarán en secuencia las matrices de transformación corrientes de cada caja de la Figura 1. A este proceder se lo conoce como cadena de transformaciones o **tubería gráfica**.

De hecho, en OpenGL, las matrices de la primera y segunda caja que corresponden respectivamente a los cambios de sistemas modelo->mundo y mundo->observador, se almacenan en una única, resultante de multiplicar la **matriz de la vista** (que se aplica después) por la matriz del modelo (que se aplica

primero) siguiendo la regla. A esta matriz compuesta e inseparable en OpenGL se la conoce como GL_MODELVIEW.

La consecuencia de aplicar la matriz *modelview* a un vértice en el sistema del modelo es que sus coordenadas pasan a tener como origen la posición de la cámara y estar referidas a los ejes de ese sistema. A estos ejes nos referimos habitualmente como *u*, *v* y *w*. Si pensamos en una fotografía, el eje *u* correspondería a la dirección horizontal de la foto, el *v* a la vertical de la foto y el *w* perpendicular a la foto saliendo hacia nosotros. El origen se sitúa en el centro de la foto.

Para que las transformaciones afecten a la matriz *modelview* hay que seleccionarla primero. Esta se hace en OpenGL usando la orden siguiente:

```
glMatrixMode(GL_MODELVIEW)
```

A partir de ese momento todas las transformaciones **se acumularan por la derecha según las vayamos escribiendo en el código**. Es muy importante tener en cuenta esta nueva regla.

Traslación

Para desplazar un punto desde su posición las distancias *dx*, *dy* y *dz* en las direcciones principales ***X***, ***Y*** y ***Z*** respectivamente se usa la orden de OpenGL:

```
glTranslatef(dx,dy,dz)
```

donde los parámetros son de tipo `GLfloat` (por ello el sufijo f).

Escalado

Para escalar un punto respecto al origen de coordenadas con factores de escala *sx*, *sy* y *sz* se usa la orden de OpenGL:

```
glScalef(sx, sy, sz)
```

donde *sx*, *sy* y *sz* son de tipo `GLfloat`. Así, si el punto es *P(px, py, pz)*, el punto transformado será el *P'(sx*px, sy*py, sz*pz)*.

Rotación

La rotación en 3D se determina por un ángulo (positivo contrario a la agujas del reloj), y un eje de giro. La orden en OpenGL que gira un punto un ángulo α respecto a un eje que pasa por el origen y lleva la dirección ***R****(rx, ry, rz)* es:

```
glRotatef(α, rx, ry, rz)
```

donde α se da en grados y es de tipo `GLfloat`, al igual que el resto de parámetros.

Composición de transformaciones

Las matrices de transformación que generan las órdenes de translación escalado y rotación se van acumulando en la matriz *modelview*, de manera que en cada momento haya una matriz *modelview* corriente. La matriz *modelview* corriente es la que se aplica en cada llamada `glVertex` de tal forma que,

si se hacen modificaciones a la *modelview* después, el vértice ya dibujado no queda afectado por estas nuevas transformaciones.

Para asegurar que se comienza con la matriz *modelview* neutra, es decir la matriz identidad, es necesario hacer la llamada a la función OpenGL:

```
glLoadIdentity()
```

esta función sustituye la matriz *modelview* corriente por la matriz identidad.

Si por algún motivo quisiéramos sustituir la matriz *modelview* por otra definida por nosotros hay que utilizar la función:

```
glLoadMatrixf(m)
```

donde `m` es un *array* de 16 `GLfloat` que representa una matriz 4x4 construida por columnas `{m00,m10,…,m23,m33}`.

Ejercicio S3E01: Construir una matriz que haga lo mismo que la función glTranslatef(0.5,0.6,0.0). Aplicar a un cubo de lado 1 centrado en el origen.

Para manejar la composición de transformaciones es muy importante tener presente las dos reglas mencionadas:

1. El orden de aplicación de las transformaciones es de derecha a izquierda
2. Las matrices se acumulan por la derecha según se escriben en el código

Estas dos reglas se resumen en una que hay que recordar: **el orden de aplicación de las transformaciones que están por encima en el código a la llamada a glVertex es de abajo hacia arriba**.

Ejercicio S3E02: Aplicar primero un giro de 45º respecto al eje Z y una traslación de 0.5 unidades en X después a un cubo de lado 1 centrado en el origen. ¿Qué pasa si después de dibujar el cubo se modifica la modelview cargando la matriz identidad?

Encapsulamiento de transformaciones

La mayor parte de las veces los objetos van ligados a otros en una jerarquía. Imaginemos un tren con pasajeros. El desplazamiento del tren está referido a un sistema fijo. La forma más sencilla de mover los pasajeros es respecto al sistema móvil del tren. Cada pasajero tendrá un desplazamiento diferente, pero todos están afectados del desplazamiento del tren. Lo mismo sucede con el sistema sol-tierra-luna, con un brazo articulado de un robot, etc.

Volviendo al ejemplo del tren, sería muy interesante ir cambiando la matriz de transformación de cada pasajero manteniendo la misma para el tren, siguiendo esta secuencia de órdenes:

1. Construir la *modelview* con la matriz del tren
2. Dibujar el tren
3. Acumular la matriz del pasajero “i” a la *modelview*
4. Dibujar el pasajero “i”
5. Des-acumular la matriz del pasajero “i”
6. Hacer `i=i+1` y volver al 3

Para poder acumular y des-acumular matrices a voluntad OpenGL mantiene una pila de matrices. La *modelview* corriente siempre es la superior de la pila. Si se quiere acumular (para después quitarla) una secuencia de transformaciones, llamaremos a la función:

```
glPushMatrix()
```

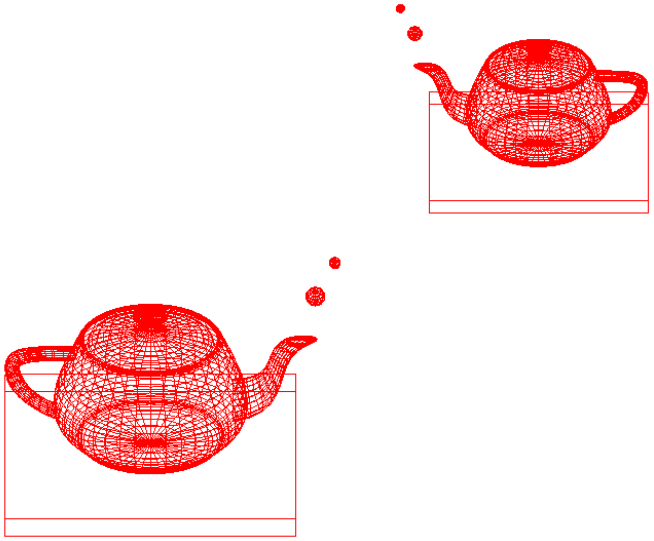
Con esta orden creamos una copia de la matriz superior de la pila (la del tren) y la ponemos encima, de manera que ahora hay dos matrices iguales, la superior y la que está inmediatamente debajo. Todas las transformaciones que se indiquen a partir de ahora se acumularan en la matriz superior.

Una vez hecho el dibujo (del pasajero) usando la matriz superior que es la *modelview* actual, podemos retirarla de la pila para restaurar el estado anterior (la del tren) con la orden:

```
glPopMatrix()
```

Ejercicio S3E03: Representar un sol y un planeta con dos lunas todos en el mismo plano (XY). El planeta está a los 45º de su órbita completa. El primer satélite está a los 100º de su órbita completa y el segundo a los 20º. Todas las órbitas comienzan en el eje X del astro superior.

Ejercicio S3E04: Representar una escena como la de la figura usando glutWireTeapot(), glutWireCube() y glutWireSphere() con el encapsulamiento de transformaciones.



Ejercicio S3E01: Construir una matriz que haga lo mismo que la función glTranslatef(0.5,0.6,0.0). Aplicar a un cubo de lado 1 centrado en el origen.

```
/*************************************************
ISGI::Traslacion Cubo
Roberto Vivo', 2013 (v1.0)

Traslada un cubo usando una matriz construida al efecto

Dependencias:
+GLUT
*************************************************/
#define PROYECTO "ISGI::S3E01::Traslacion Cubo"

#include <iostream>                             // Biblioteca de entrada salida
#include <gl\freeglut.h>                        // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
  // Matriz de traslacion por columnas
  static const GLfloat traslacion[]={1,0,0,0, 0,1,0,0, 0,0,1,0, 0.5,0.6,0,1};

  glClear(GL_COLOR_BUFFER_BIT);                 // Borra la pantalla
  glMatrixMode(GL_MODELVIEW);                   // Selecciona la modelview
  glLoadMatrixf(traslacion);                    // Carga la matriz sobreescribiendo
  glColor3f(1,1,1);                             // Dibuja en color blanco
  glutWireCube(1.0);                            // Dibuja el cubo
  glFlush();                                    // Finaliza el dibujo
}

void reshape(GLint w, GLint h)
// Funcion de atencion al redimensionamiento
{
}

void main(int argc, char** argv)
// Programa principal
{
  glutInit(&argc, argv);                        // Inicializacion de GLUT
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);    // Alta de buffers a usar
  glutInitWindowSize(400,400);                  // Tamanyo inicial de la ventana
  glutCreateWindow(PROYECTO);                   // Creacion de la ventana con su titulo
  std::cout << PROYECTO << " running" << std::endl;      // Mensaje por consola
  glutDisplayFunc(display);                     // Alta de la funcion de atencion a display
  glutReshapeFunc(reshape);                     // Alta de la funcion de atencion a reshape
  glutMainLoop();                               // Puesta en marcha del programa
}
```

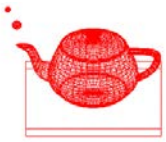
Ejercicio S3E02: Aplicar primero un giro de 45º respecto al eje Z y una traslación de 0.5 unidades en X después a un cubo de lado 1 centrado en el origen. ¿Qué pasa si después de dibujar el cubo se modifica la modelview cargando la matriz identidad?

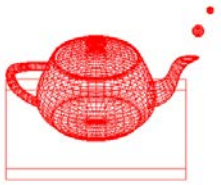
```
/*************************************************
ISGI::Giro y Traslacion
Roberto Vivo', 2013 (v1.0)

Gira y Traslada un cubo usando la composicion de OpenGL

Dependencias:
+GLUT
*************************************************/
#define PROYECTO "ISGI::S3E02::Giro+Traslacion Cubo"

#include <iostream>                             // Biblioteca de entrada salida
#include <gl\freeglut.h>                        // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
  glClear(GL_COLOR_BUFFER_BIT);                 // Borra la pantalla
  glMatrixMode(GL_MODELVIEW);                   // Selecciona la modelview
  glLoadIdentity();                             // Carga la matriz identidad
  glTranslatef(0.5,0.0,0.0);                    // modelview=I*T
  glRotatef(45,0,0,1);                          // modelview=I*T*R
  glColor3f(1,1,1);                             // Dibuja en color blanco
  glutWireCube(1.0);                            // Dibuja el cubo
  glFlush();                                    // Finaliza el dibujo
}
```

Ejercicio S3E03: Representar un sol y un planeta con dos lunas todos en el mismo plano (XY). El planeta está a los 45º de su órbita completa. El primer satélite está a los 100º de su órbita completa y el segundo a los 20º. Todas las órbitas comienzan en el eje X del astro superior.

```
/************************************************
ISGI::Dos lunas
Roberto Vivo', 2013 (v1.0)

Representa dos lunas y un planeta vistos desde arriba

Dependencias:
+GLUT
************************************************/
#define PROYECTO "ISGI::S3E03::Planeta y lunas"

#include <iostream>                             // Biblioteca de entrada salida
#include <cmath>                                // Biblioteca matematica de C
#include <gl\freeglut.h>                        // Biblioteca grafica

void display()
// Funcion de atencion al dibujo
{
  glClear(GL_COLOR_BUFFER_BIT);                 // Borra la pantalla
  glMatrixMode(GL_MODELVIEW);                   // Selecciona la modelview
  glLoadIdentity();                             // Carga la matriz identidad

  glScalef(0.2,0.2,0.2);                        // Escalado general

  // Dibuja el sol
  glutSolidSphere(1.0,10,10);                   // El sol mide 1 de radio

  glPushMatrix();                               // Salva el estado de la transformación
/* Suponiendo la orbita traslacional
  glTranslatef(  3.0*cos(45*3.1415926/180),
                 3.0*sin(45*3.1415926/180),0);*/  // Posicion en orbita


  // Dibuja el planeta en coordenadas del sol -orbita rotacional-
  glRotatef(45,0,0,1);                          // Giro alrededor del sol
  glTranslatef(3.0,0.0,0.0);                    // Puesto en Orbita
  glutWireSphere(0.5,10,10);                    // El planeta mide 0.5

  // Dibuja la primera luna en coordenadas del planeta
  glPushMatrix();                               // Salva la transformacion del planeta
  glRotatef(100,0,0,1);                         // Gira alrededor del planeta
  glTranslatef(1.0,0.0,0.0);                    // Orbita de la primera luna
  glutWireSphere(0.1,5,5);                      // La luna mide 0.1
  glPopMatrix();                                // Restaura la transformacion del planeta

  // Dibuja la segunda luna en ccordenadas del planeta
  glPushMatrix();                               // Salva la transformacion del planeta
  glRotatef(20,0,0,1);                          // Gira alrededor del planeta
  glTranslatef(1.2,0.0,0.0);                    // Orbita de la segunda luna
  glutWireSphere(0.1,5,5);                      // La segunda luna mide 0.1
  glPopMatrix();                                // Restaura la trasformacion del planeta

  // Aquí se pueden segur dibujando más lunas

  glPopMatrix();                                // Restaura la trasformacion del sol

  // Aquí se pueden seguir dibujando más planetas

  glFlush();                                    // Finaliza el dibujo
}
/* Si el movimiento del planeta alrededor del sol fuera de traslación sobre la orbita circular,
deberíamos cambiar la transformacion del planeta por una traslación al punto de la orbita con
glTraslatef(radioorbita*cos(anguloorbital),radioorbita*sin(anguloorbital),0); */
```

Ejercicio S304: Representar una escena como la de la figura usando glutWireTeapot(), glutWireCube() y glutWireSphere() con el encapsulamiento de transformaciones.





```
/*************************************************
ISGI::Bandeja con Tetera
Roberto Vivo', 2013 (v1.0)

Representa dos bandejas con teteras humeantes usando
encpsulamiento de trasformaciones

Dependencias:
+GLUT
*************************************************/
#define PROYECTO "ISGI::S3E04::Bandejas y teteras"

#include <iostream>                             // Biblioteca de entrada salida
#include <gl\freeglut.h>                        // Biblioteca grafica

void tetera()
// Dibuja una tetera humeante
{
        glPushMatrix();                         // Se salva el estado
        glutWireTeapot(1.0);                    // Dibujo de la tetera

        // Burbuja gorda
        glPushMatrix();
        glTranslatef(1.7,1,0);
        glScalef(0.1,0.1,0.1);
        glutWireSphere(1.0,10,10);
        glPopMatrix();

        // Burbuja pequeña
        glPushMatrix();
        glTranslatef(1.9,1.4,0);
        glScalef(0.06,0.06,0.06);
        glutWireSphere(1.0,10,10);
        glPopMatrix();

        glPopMatrix();                          // Restaura la modelview
}

void bandejaytetera()
// Dibuja el conjunto bandeja-tetera
{
        glPushMatrix();                         // Salva la modelview

        // Dibujo de la bandeja
        glPushMatrix();
        glTranslatef(0,-0.1,0);
        glScalef(3,0.2,3);
        glutWireCube(1.0);
        glPopMatrix();

        // Dibujo de la tetera humeante
        glPushMatrix();
        glTranslatef(0,0.8,0);
        tetera();
        glPopMatrix();

        glPopMatrix();                          // Restaura la modelview
}

void display()
// Funcion de atencion al dibujo
{
  glClear(GL_COLOR_BUFFER_BIT);                 // Borra la pantalla
  glMatrixMode(GL_MODELVIEW);                   // Selecciona la modelview
  glLoadIdentity();                             // Carga la matriz identidad
```

```
  glRotatef(30,1,0,0);                          // Inclinacion general

  // Dibujo del conjunto de la derecha
  glPushMatrix();
  glTranslatef(0.4,0,-0.5);
  glRotatef(180,0,1,0);
  glScalef(0.15,0.15,0.15);
  bandejaytetera();
  glPopMatrix();

  // Dibujo del conjunto de la izquierda
  glPushMatrix();
  glTranslatef(-0.4,-0.2,0.4);
  glScalef(0.2,0.2,0.2);
  bandejaytetera();
  glPopMatrix();

  glFlush();                                    // Finaliza el dibujo
}
```