

3. OPERACIONES CON MATRICES: DIBUJANDO EL CAMINO

La transformación de las coordenadas se realiza internamente en OpenGL a partir de las matrices de transformación y de las coordenadas de modelado del objeto. Sin embargo, para representar el rastro que dibuja la tortuga al desplazarse, se realizará explícitamente la transformación de la coordenada en la que se encuentra la tortuga. De esta forma se actúa directamente en la matriz de transformación.

3.1 LA PILA DE MATRICES

En la función **display()** se encuentran las llamadas a dos funciones de matrices que todavía no han sido comentadas. Se trata de **glPushMatrix()** y **glPopMatrix()**. Para comprender su funcionamiento, primero se va a experimentar que es lo que ocurre cuando no están dichas llamadas. Para ello se comentan en la función **display()** ambas llamadas:

```
void display(void) {
    ...
    // glPushMatrix();
    ...
    glTranslatef(0.0, 0.0, .5);
    ...
    // glPopMatrix();
    glutSwapBuffers();
}
```

Al ejecutar de nuevo la aplicación, primeramente tiene el mismo aspecto que sin comentar las llamadas, pero si obligamos a que se llame varias veces a la función **display()**, por ejemplo pulsando la tecla “c” (que activa y desactiva los polígonos posteriores del objeto), vemos que además de producirse el efecto de cambiar el modo *GL_CULL_FACE*, el objeto se va moviendo progresivamente a lo largo de eje “Z”.

La razón de este movimiento es que en la función **display** está incluida una llamada a **glTranslatef()** que se utiliza para posicionar uno de los objetos. Como se ha explicado anteriormente, las funciones de traslación multiplican la matriz actual por una matriz de traslación creada con los argumentos que se le pasan, por tanto, sucesivas llamadas a la función **display()** provocan sucesivas multiplicaciones de la matriz actual con el efecto que se observa de incrementar la traslación.

Para solucionar este problema OpenGL dispone de unos stacks o pilas de matrices, que permiten almacenar y recuperar una matriz anterior. Aunque OpenGL dispone de pilas para las matrices *GL_MODELVIEW* y *GL_PROJECTION*, sólo se suele utilizar la pila de *GL_MODELVIEW*.

Una pila es un almacén con funcionamiento LIFO, el último en entrar es el primero en salir, por lo que suele compararse a una pila de platos en la que sólo se puede dejar uno encima de la pila o coger el superior que es el último depositado. La pila de matrices tiene el mismo funcionamiento sustituyendo los platos por matrices. La matriz superior de la pila es sobre la que se aplican las distintas transformaciones, multiplicándola por la matriz que generan las distintas funciones.

Para poder guardar una determinada matriz y posteriormente recuperarla OpenGL dispone de las dos funciones comentadas: **glPushMatrix()** y **glPopMatrix()**.

La función **glPushMatrix()** realiza una copia de la matriz superior y la pone encima de la pila, de tal forma que las dos matrices superiores son iguales. En la figura 1 se observa la pila en la situación inicial con una sola matriz, al llamar a la función **glPushMatrix()** se duplica la matriz superior. Las siguientes transformaciones que se realizan se aplican sólo a la matriz superior de la pila, quedando la anterior con los valores que tenía en el momento de llamar a la función **glPushMatrix()**.

La función **glPopMatrix()** elimina la matriz superior, quedando en la parte superior de la pila la matriz que estaba en el momento de llamar a la función **glPushMatrix()**.

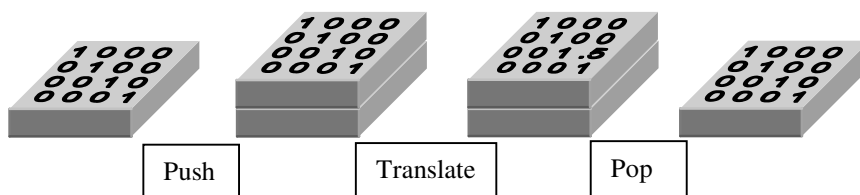


Figura 1 PushMatrix y PopMatrix

En la función **display()** al llamar a la función **glPushMatrix()** se realiza una copia de la matriz actual. La traslación en el eje Z se realiza en la matriz superior de la pila, es decir, en la copia de la matriz, de tal forma que al llamar a la función **glPopMatrix()**, como se muestra en la figura 1, se elimina la matriz superior, que es la que tenía el efecto de esta transformación, quedando la matriz que estaba en el momento de llamar a **glPushMatrix()**.

Al descomentar las llamadas a las funciones **glPushMatrix()** y **glPopMatrix()** las transformaciones realizadas entre ambas no afectan al resto de la aplicación.

3.2 DIBUJANDO UN RASTRO

Una característica de Logo es que la tortuga al avanzar va dibujando el camino por el que ha pasado. Hasta ahora la aplicación va transformando las coordenadas del objeto para situarlo en la nueva posición según las instrucciones introducidas pero no muestra la ruta seguida.

Para mostrar la ruta es necesario almacenar los puntos por los que pasa la tortuga. El rastro consistirá en una línea que una estos puntos.

Necesitaremos realizar tres operaciones: calcular la coordenada donde se encuentra la tortuga, almacenar dicha coordenada y dibujar el rastro.

Para almacenar los puntos se utiliza una variable para indicar el número de puntos y tres vectores para las coordenadas x, y, z.

```
int np = 0;
float px [10000];
float py [10000];
float pz [10000];
```

Para calcular las coordenadas de la tortuga es necesario conocer la matriz de transformación de modelado. Debido a que en OpenGL, la matriz de modelado se almacena junto con la de visualización en la matriz *GL_MODELVIEW*, es necesario guardar de modo independiente esta matriz de modelado. Para ello definimos la variable *mModel*, como una variable global, ya que va a ser accedida en distintos puntos de la aplicación:

```
GLdouble mModel[16];
```

Para obtener la matriz actual de una de las pilas se dispone de la función de OpenGL **glGetDoublev()** a la que se indica que matriz se quiere obtener, *GL_MODELVIEW* en nuestro caso, y un puntero a un vector de 16 posiciones donde se rellenarán los valores de la matriz.

Hay que tener en cuenta que OpenGL almacena esta matriz por columnas de modo que los elementos son:

$$M = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

Para operar con la matriz *mModel*, se cargará en la pila de matrices de *GL_MODELVIEW* después de realizar un **PushMatrix()**, de modo que no altere la matriz actual. En la función **main()** se inicializa esta matriz con el código:

```
glMatrixMode (GL_MODELVIEW) ;
glPushMatrix() ;
glLoadIdentity() ;
glGetDoublev (GL_MODELVIEW_MATRIX, mModel) ;
glPopMatrix() ;
```

En este código se realizan las siguientes operaciones:

- se indica primeramente sobre que matriz se van a realizar las operaciones con **glMatrixMode()**;
- se crea una nueva matriz con **glPushMatrix()**;
- se carga la matriz identidad con **glLoadIdentity()**;
- se almacena la matriz superior de la pila en el vector *mModel* con la función **glGetDoublev()**;
- y finalmente se elimina la matriz superior de la pila con **glPopMatrix()** para dejar la que estaba antes de este proceso.

En realidad todo este proceso lo que ha hecho ha sido inicializar la matriz que representa *mModel* con la matriz identidad.

Para guardar las distintas transformaciones que se realizan con las instrucciones de logo (**FORWARD**, **RIGHT**, ...), se carga la matriz *mModel* en la pila, antes del bloque de “if” que determinan que instrucción se va a realizar:

```
glPushMatrix() ;
glLoadIdentity() ;
glMultMatrixd(mModel) ;
```

La función **glMultMatrixd()** multiplica la matriz superior de la pila por la matriz que tiene como argumento. Al multiplicar en este caso por la matriz identidad, la matriz que queda en la posición superior de la pila es *mModel*.

A continuación se realiza la operación correspondiente a la instrucción dada (**glTranslate()**, **glRotate()**, ...) que actuará sobre esta matriz.

Al finalizar el bloque “if” se recupera la matriz *mModel* y se restaura la que estaba previamente en la pila con el código:

```
glGetDoublev (GL_MODELVIEW_MATRIX, mModel) ;
glPopMatrix() ;
```

Para conocer las coordenadas de la tortuga en la situación actual, el proceso que se realiza es obtener la matriz de modelado despues de la transformación y aplicar dicha matriz de transformación sobre la coordenada original de la tortuga. Esto se realiza en la función **addPointToTrace()**, que añade un nuevo punto a la lista de coordenadas.

```
void addPointToTrace() {
    int i;
    GLdouble m[16];
    glGetDoublev (GL_MODELVIEW_MATRIX, m);
    // print the matrix
    printf ("\nMatrix:\n");
    for (i = 0; i < 4; i++) {
        printf ("Row %i: %f \t%f \t%f \t%f \n",
            i+1, m[i+0],m[i+4],m[i+8],m[i+12]);
    }
    // if is the first point
    if (np == 0) { // add the first point
        px [0] = 0;
        py [0] = 0;
        pz [0] = 0;
        np++;
    }
    px [np] = m[0] * px [0] + m[4] * py [0] + m[8] * pz [0] + m[12];
    py [np] = m[1] * px [0] + m[5] * py [0] + m[9] * pz [0] + m[13];
    pz [np] = m[2] * px [0] + m[6] * py [0] + m[10] * pz [0] + m[14];
    printf ("Point %i: %f \t%f \t%f \n",
        np, px[np],py[np],pz[np]);
    np++;
}
```

La matriz se obtiene de la pila en lugar de ser `mModel`, porque esta última todavía no tiene incorporada la ultima transformación realizada.

El proceso de la función **addPointToTrace()** consiste en:

- obtener la matriz,
- la imprime a efectos informativos, la matriz se imprime de la forma habitual que es con la traslación como última columna de la matriz, por ello en la primera fila se imprimen los elementos 0, 4 ,8 y 12.
- si es el primer punto de la lista lo introduce directamente, en este caso es el origen,
- calcula las coordenadas del nuevo punto como producto de la matriz por el vector de coordenadas del primer punto
- y finalmente imprime estas coordendas.

En este caso al ser las coordenadas del punto inicial igual al origen, bastaría con sumar los términos de traslación. Se ha dejado como un procedimiento general porque así se pueden cambiar las coordenadas del punto inicial.

Para dibujar la ruta se implementa la función **displayTrace()**, que consiste en dibujar una polilínea (*GL_LINE_STRIP*) con todos los puntos almacenados. A continuación se muestra la función **displayTrace()**:

```
void displayTrace() {
    int i;
    glColor3f(1.0,1.0,1.0) ;
    glBegin(GL_LINE_STRIP);
    for (i = 0; i < np; i++) {
        glVertex3f (px[i],py[i],pz[i]);
    }
}
```

```

    }
    glEnd();
}

```

Para que se realice la representación de la ruta es necesario invocar estas dos funciones en el código.

La llamada a **addPointToTrace()** se introduce después de las llamadas a **glTranslatef()** en las instrucciones correspondientes al FORWARD y BACK.

```
addPointToTrace();
```

La llamada a **displayTrace()** se realiza en la función **display()**.

El dibujo del objeto que representa la tortuga se debe realizar después de multiplicar la matriz actual (la matriz de visualización) por la matriz de modelado que se almacena en **mModel**. Sin embargo, el rastro solo debe multiplicarse por la matriz de visualización.

A continuación se muestra la función **display()**:

```

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glMultMatrixd(mModel);
    glColor3f(1.0, 1.0, 0.0);
    drawTurtle();
    glPopMatrix();
    displayTrace();
    glutSwapBuffers();
}

```

La llamada a **PushMatrix()** realiza una copia de la matriz actual, a continuación se multiplica por la matriz de modelado y se realiza el dibujo del objeto.

Se restaura la matriz de visualización y se dibuja el rastro. La última llamada realiza el intercambio de buffers.

En la figura 2 se muestra el rastro tras realizar "fd 5 rt 90 fd 2".

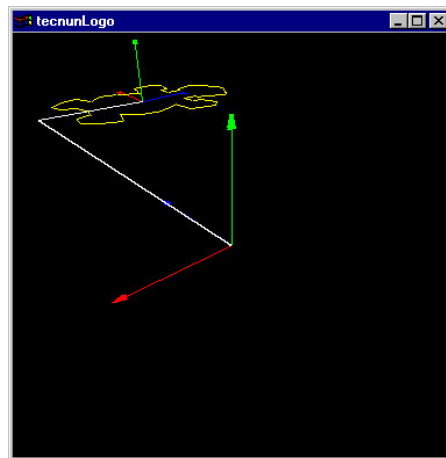


Figura 2 Dibujo del rastro

Dibujar un rastro que consista en una superficie en lugar de una línea. Para ello se puede utilizar `glBegin(GL_QUAD_STRIP)` que dibuja una sucesión de rectángulos para cada pareja de puntos que recibe, como se muestra en la figura 3.

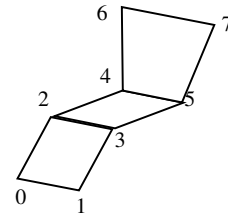


Figura 3 GL_QUAD_STRIP

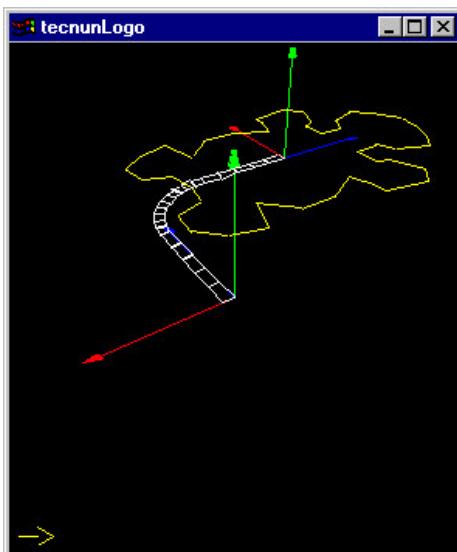


Figura 4 Dibujo del rastro

Ya no bastará con tener un solo punto inicial, el origen, sino que será necesario disponer además de otro perpendicular a él, por ejemplo el (0.1, 0.0, 0.0). En

cada nuevo movimiento será necesario añadir estos dos puntos transformados. El resultado es el que se muestra en la figura 4, en la que se dibuja el rastro después de haber realizado 5 “fd .2”, 10 “rt 9 fd .2” y 5 “fd .2”.

3.3 MOSTRAR TEXTO

Las instrucciones introducidas se muestran en la ventana MSDOS. Se pueden mostrar en la ventana gráfica. Para ello es necesario cambiar las matrices de transformación. La siguiente función realiza la representación del texto:

```
void text(GLuint x, GLuint y, GLfloat scale, char* format, ...) {
    va_list args;
    char buffer[255], *p;
    GLfloat font_scale = 119.05f + 33.33f;

    va_start(args, format);
    vsprintf(buffer, format, args);
    va_end(args);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0, glutGet(GLUT_WINDOW_WIDTH), 0,
        glutGet(GLUT_WINDOW_HEIGHT));

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glPushAttrib(GL_ENABLE_BIT);
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_DEPTH_TEST);
    glTranslatef(x, y, 0.0);

    glScalef(scale/font_scale, scale/font_scale, scale/font_scale);

    for(p = buffer; *p; p++)
        glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);

    glPopAttrib();

    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
}
```

Para mostrar el texto se llama desde la función **display()** con la sentencia:

```
if (command) {
    glColor3f(1.0, 1., 0.0) ;
    text(5, 5, 20, "->%s", strCommand);
}
```

3.4 PUNTOS A REALIZAR

Utilizando los comandos de logo representar una esfera compuesta por un conjunto de circunferencias en el espacio.

Utilizando los comandos de logo realizar la representación de una helicoidal. En la figura 5 se muestra el resultado de dicho comando.

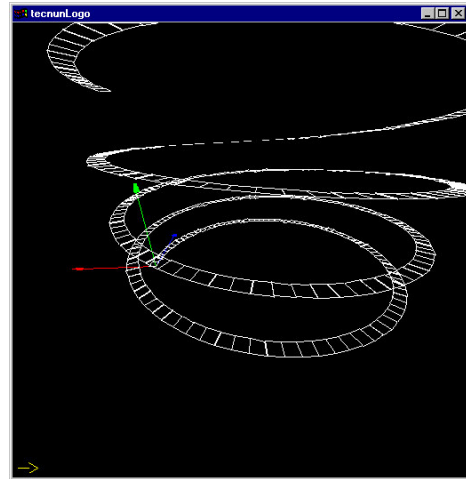


Figura 5 Helicoidal