eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea

Control de camara y iluminación en OpenGL

Ignacio Belzunegui y Aitor Domec

December 24, 2018

1 Objetivos de nuestra aplicación

Nuestra aplicación tiene que ser capaz de controlar una cámara y la iluminación. Tienen que existir varias cámaras, debemos de poder ver lo que un objeto, entrar en modo análisis/vuelo, modificar el volumen de vision, trasladar/rotar la cámara y cambiar el tipo de proyección entre modo paralelo o perspectiva. En el apartado de iluminación, tenemos que crear mínimo tres tipos de fuentes de luz (un sol, una bombilla y un foco relacionado a un objeto), tenemos que poder transformarlas y cambiar de tipo de iluminación (flat o smooth).
Hemos dividido este trabajo en dos partes, cámara y iluminación (cada una tendrá sus propios sub-apartados).

1. **Cámara.**
   (a) **Camara / Estructura de la cámara.**
      i. **Estructura.**
      ii. **Cambio de cámara.**
      iii. **Cámara del objeto.**
   (b) **Modificaciones sobre la cámara.**
      i. **Transformaciones.**

A. **Cambio volumen de visión.**
B. **Modo análisis / Modo vuelo.**
C. **Rotación.**
D. **Traslación.**

ii. **Cambio de tipo de proyección.**

2. **Iluminación.**

(a) **Explicación.**

i. **Luz.**
ii. **Materiales.**
iii. **Fuentes de luz.**

(b) **Vectores normales.**

(c) **Creación de luces.**

(d) **Modificación de luces.**

2 Manual de usuario

Para compilar el programa debemos de compilar main.c, display.c, io.c y load_obj_joseba.c por supuesto teniendo en cuenta que usamos OpenGL y librerias de lglu, el comando para compilar es el siguiente:
gcc main.c display.c io.c load_obj_joseba.c -lGL -lGLU -lglut -lm
Este comando generará el ejecutable **a.out**. Para ejecutarlo usaremos el comando: *./a.out* . Una vez ejecutado el programa el usuario verá en el terminal la guía de ayuda que explica los comandos existentes y sus utilidades. También verá una ventana emergente donde se visualizarán todos los objetos y donde realizará las transformaciones pertinentes.
Primero se deberá de cargar un objeto, de lo contrario no nos permitirá realizar ninguna operación. Para cargar un objeto deberá de escribir el path al objeto, dándole a la "f" antes; nosotros recomendamos el uso del siguiente comando para ahorrarnos el tener que escribir el path cada vez que cargamos un objeto: **cp objects/abioia.obj ./a** donde objects/abioia.obj es el path al objeto deseado. De esta forma para cargar un objeto bastará con escribir únicamente "a".
Una vez cargado el objeto tenemos acceso completo a todos los comandos, de forma predefinida los cambios se realizarán respecto al objeto y comenzará con el modo de rotación activado, pero esto podrá ser cambiado a voluntad en cualquier momento. Si deseamos utilizar la camara podremos acceder a

sus comandos si pulsamos la tecla 'k', del mismo modo si deaseamos activar/desactivar la iluminaciónn deberemos de pulsar la tecla 'f9'.
Los comandos permitidos son los siguientes:

1. "?" Visualizar ayuda.
2. "ESC" Salir del programa.
3. "f / F" Cargar un objeto.
4. "TAB" Cambiar la seleccion de un objeto cargado.
5. "DEL" Borrar el objeto seleccionado.
6. "CTRL + -" Aumentar el zoom / aumentar el volumen de visión / aumentar el ángulo de apertura de un foco
7. "CTRL + +" Reducir el zoom / reducir el volumen de visión / reducir el ángulo de apertura de un foco
8. "o / O" Activar transformaciones sobre objetos
9. "l / L" Activar transformaciones respecto al objeto.
10. "g / G" Activar transformaciones respecto al mundo.
11. "m / M" Activar la traslación.
12. "b / B" Activar la rotación.
13. "t / T" Activar el escalado.
14. "k / K" Activar transformaciones sobre cámaras.
15. "c" Cambiar de cámara.
16. "C" Cambiar a la cámara del objeto.
17. "g / C" Modo análisis.
18. "l / L" Modo vuelo.
19. "p / P" Cambio de tipo de perspectiva.
20. "F9" Activar/Desactivar el modo iluminación.
21. "F1-F4" Activar/desactivar fuentes de luz.

22. "1-4" Seleccionar fuente de luz.

Para realizar las tranformaciones deseadas debemos de pulsar las flechas del teclado u las teclas 'PageUp' o 'PageDown'.

3 Cambios realizados

1. **Cámara.**

   (a) **Estructura.** Nuestra cámara va a ser una pila doble que contendrá, una lista de matrices, un identificador y varios parametros que nos serviran para modificar el volumen de visión de la misma. Cada cámara apunta a la cámara anterior y a la siguiente, creando así un circuito cerrado.

```
typedef struct camara_l{
    int numero;                 /*Identificador de la camara*/
    GLdouble left;              /**/
    GLdouble right;             /**/
    GLdouble bottom;            /**/
    GLdouble top;               /**/
    GLdouble near;              /*Desde como de cerca vemos*/
    GLdouble far;               /*Como de lejos llegamos a ver*/

    GLdouble ortho_x_min;       /* Parametros usados*/
    GLdouble ortho_x_max;       /* para el modo     */
    GLdouble ortho_y_min;       /* de proyeccion    */
    GLdouble ortho_y_max;       /* ortogonal        */

    struct matrix_l *matrix;    /*Lista de matrices*/
    struct camara_l *next;      /*Siguiente camara */
    struct camara_l *previous;  /*Camara anterior */
} camara_l;
```

   (b) **Cambio de cámara.** Existen 4 cámaras creadas en nuestro proyecto. Si activamos el modo cámara (pulsando las tecla 'k' o 'K') y pulsamos la 'c' cambiaremos de la cámara actual a la siguiente en la lista; se nos indicará por terminal en cual nos encontramos.
   Cabe destacar que la cámara inicial es simplemente la de identidad y que comenzaremos viendo todo desde un modo de perspectica ortogonal, es decir, en **paralelo**. Estos significa que todos los objetos se verán como si se encontraran a la misma distancia, independientemente de su posición real.

Para conseguir el efecto de cambiar de cámara, simplemente guardaremos la cámara actual en 'camara_auxiliar' y haremos que la principal ('camaraG') apunte a la siguiente en la lista. Si queremos volver desde la del objeto a las cámaras predefinidas en el aplicación es importante tener un "backup" de la cámara global en el momento en el que se ha seleccionado la del objeto. Por eso mismo, hemos definido una *matrix_l* llamada camara_auxiliar (nótese que es una variable global) que tiene como objetivo guardar el valor de la cámara global cuando se ha seleccionado la del objeto.

```
case 'c'://Cambiamos de camara
  if(camaraG==_selected_object->camara){
    camaraG=camara_auxiliar;
  }
  else{
    camara_auxiliar=camaraG;
    camaraG=camaraG->next;
  }
    printf("Usted acaba de cambiar a
    la camara: %i\n",camaraG->numero);
break;
```

(c) **Cámara del objeto.** Si pulsamos la tecla 'C' la cámara actual se trasladará al objeto, viendo asi lo que el objeto "ve". Una vez activado el modo de cámara del objeto no seremos capaces de modificarla, si queremos cambiar nuestro punto de vista deberemos de transformar el objeto selecionado o desactivar el modo de cámara del objeto.
'modog' es una variable que dice si están activadas las transformaciones en los objetos (modog=0), cámara (modog=1) o iluminación (modog=2). 'modoCamara' indica si la cámara se ha colocado encima del objeto o no (Activado modoCamara=1, desactivado modoCamara=0), ya que si está activado no podremos modificar la cámara.
Cabe mencionar que si queremos trasladarnos o rotar, tendremos que activar el modo objeto y trasladarnos o rotar desde ahí. Eventualmente podremos observar que el objeto y la cámara se mueven a la par. Para desactivar el modo cámara del objeto debemos activar el modo cámara pulsando K y pulsando C otra vez.

```
case 'C':
  if(modog==1) {
    if(modoCamara==0){
        modoCamara=1;
        printf("Usted acaba de cambiar a la camara
        del objeto\n");
      }
      else {
        modoCamara=0;
```

```
        printf("Usted acaba de desactivar
        la camara del objeto\n");
      }
  }
  else {
    printf("No esta activado el modo de camara\n");
  }
break;
```

Una vez activado el modoCamara deberemos de tener en cuenta que cada vez que se modifique el objeto deberemos de modificar también la cámara. Para ello hemos creado una función llamada 'camera_update()' que, una vez activado el modoCamara, se asegurará de actualizar la matriz de la cámara cada vez que se realize una transformación en el objeto. Cada vez que llamamos a 'camera_update' realizaremos un gluLookAt con los siguientes valores: Eye=Posición del objeto, Center=Posición del objeto-Vector Z del objeto, Up= Vector Y del objeto

.

```
void camera_update() {
  if(modoCamara==1) {
    glLoadIdentity();
    gluLookAt(_selected_object->matrix->matrix[12],
    _selected_object->matrix->matrix[13],
    _selected_object->matrix->matrix[14],
    _selected_object->matrix->matrix[12]-
    _selected_object->matrix->matrix[8],
    _selected_object->matrix->matrix[13]-
    _selected_object->matrix->matrix[9],
    _selected_object->matrix->matrix[14]-
    _selected_object->matrix->matrix[10],
    _selected_object->matrix->matrix[4],
    _selected_object->matrix->matrix[5],
    _selected_object->matrix->matrix[6]);
    glGetDoublev(GL_MODELVIEW_MATRIX,
    camaraG->matrix->matrix);

  }
}
```

Esta función es llamada cada vez que pulsamos una tecla, por lo que va a aparecer al final de tanto la función de teclado normal como la especial (encargada de leer las flechas).

2. **Modificaciones sobre la cámara.**

   (a) **Transformaciones.**

      i. **Cambio volumen de visión.** Si la cámara está activada (tecla 'k') y pulsamos la 't/T' vamos a activar el cambio en volumen de visión.

Dentro de cada cámara guardamos los valores left, right, top, bottom, near y far. Los dos últimos son los encargados de decir como de lejos vemos (far) y desde como de cerca empezamos a ver (near). Los otros cuatro valores son los encargados de especificar la 'ventana' desde la que vemos en el modo perspectiva, si los alteramos veremos el escenario deformado. Es decir, nos muestran las "dimensiones" de nuestro volumen de visión.
Visualizaremos la implementación del cambio de volumen de visión mediante el código implementado en el botón UP, ya que el resto de botones van a ser idénticos pero con argumentos diferentes.

```
case GLUT_KEY_PAGE_UP:
  if(modoV == 0){//Modo vuelo
        if(modo==1) {
  (...)
        }
        if(modo==2) {
    (...)
        }
        if(modo==3) {
          if(modog==0){
  (...)
            }
            if (modog==1)
             {
            camaraG->near=camaraG->near*1.1;
            camaraG->far=camaraG->far*1.1;
             }
        }
     }
     else {//Modo analisis
      if(modo==2){
        (...)
      }
      if(modo==1){
        (...)
      }
     }
        (...)
break;
```

Los parámetros near y far aumentan en un 10% (multiplicamos por 1.1) cuando le damos al botón UP y disminuyen un 10% (multiplicamos por 0.9) cuando es pulsado el botón DOWN, y los parámetros right y left aumentan un 10% cuando RIGHT es pulsado y disminuyen un 10% cuando LEFT es pulsado.

ii. **Modo Análisis/Vuelo.** El modo vuelo nos permite "volar" por el escenario; la cámara podrá ser movida libremente por el mundo dependiendo de lo que queramos hacer (como trasladarnos

o rotar).
El modo análisis nos permite "analizar" (como indica su nombre) un objeto, situándolo en el centro de la cámara y permitiéndonos verlo desde cualquier ángulo a cualquier distancia. La variable modoV define el modo en el que nos encontramos; modoV = 0 modo VUELO y modoV = 1 modo ANÁLISIS. Cambiaremos entre modo vuelo y modo análisis pulsando la tecla G/g. Por defecto estará activado el modo vuelo.

iii. **Rotación.**
La cámara puede rotar sobre si misma en el MODO VUELO (modoV = 0) y alrededor de un objeto concreto en el MODO ANÁLISIS (modoV=1).

```
void rotate(float angle,float a, float b, float c)
{
    //Hacemos el cambio respecto al objeto
    if(_selected_object!=NULL) {
        if(modog==0) {//Sobre el objeto
            if(modo2==0){
        (...)
            }
        //Hacemos el cambio respecto al mundo
            else {
        (...)
            }
        }
        else {//Cambio respecto a la camara
            if (modoV==0)//Modo vuelo
            {
              glMatrixMode(GL_MODELVIEW);
              glLoadIdentity();
              glRotatef(angle,a, b, c);
              glMatrixMode(GL_MODELVIEW);
              glMultMatrixd(camaraG->matrix->matrix);
              guardar_estado();
              representar_matriz();
            }
            else{//modo analisis

              matrix_l *matrizaux;
            matrizaux = (matrix_l *)
            malloc(sizeof (matrix_l));

            //Matriz inversa de la MCSR
            matrizaux->matrix[0] =
            camaraG->matrix->matrix[0];
        matrizaux->matrix[1] =
        camaraG->matrix->matrix[4];
        matrizaux->matrix[2] =
        camaraG->matrix->matrix[8];
        matrizaux->matrix[3] = 0.0;


        matrizaux->matrix[4] =
        camaraG->matrix->matrix[1];
```

```
        matrizaux->matrix[5] =
        camaraG->matrix->matrix[5];
        matrizaux->matrix[6] =
        camaraG->matrix->matrix[9];
        matrizaux->matrix[7] = 0.0;


        matrizaux->matrix[8] =
        camaraG->matrix->matrix[2];
        matrizaux->matrix[9] =
        camaraG->matrix->matrix[6];
        matrizaux->matrix[10] =
        camaraG->matrix->matrix[10];
        matrizaux->matrix[11] = 0.0;
        matrizaux->matrix[12] =
        camaraG->posicionC.x;
            matrizaux->matrix[13] =
            camaraG->posicionC.y;
            matrizaux->matrix[14] =
            camaraG->posicionC.z;
        matrizaux->matrix[15] = 1.0;


            glLoadIdentity();
            glMatrixMode(GL_MODELVIEW);
            glTranslatef(_selected_object-
            >matrix->matrix[12],
            _selected_object->matrix->matrix[13],
            _selected_object->matrix->matrix[14]);
          glRotatef(angle,a,b,c);
            glTranslatef(-_selected_object->
            matrix->matrix[12],
            -_selected_object->matrix->matrix[13],
            -_selected_object->matrix->matrix[14]);
            glMultMatrixd(matrizaux->matrix);
            glMatrixMode(GL_MODELVIEW);
            glGetDoublev
            (GL_MODELVIEW_MATRIX,matrizaux
            ->matrix);
            camaraG->posicionC.x =
            matrizaux->matrix[12];
            camaraG->posicionC.y =
             matrizaux->matrix[13];
            camaraG->posicionC.z =
            matrizaux->matrix[14];

        //Generamos la Mcsr a partir de la inversa
        camaraG->matrix->matrix[0] =
        matrizaux->matrix[0];
        camaraG->matrix->matrix[1] =
        matrizaux->matrix[4];
        camaraG->matrix->matrix[2] =
        matrizaux->matrix[8];
        camaraG->matrix->matrix[3] = 0.0;


        camaraG->matrix->matrix[4] =
        matrizaux->matrix[1];
        camaraG->matrix->matrix[5] =
        matrizaux->matrix[5];
        camaraG->matrix->matrix[6] =
        matrizaux->matrix[9];
```

```
        camaraG->matrix->matrix[7] = 0.0;


        camaraG->matrix->matrix[8] =
        matrizaux->matrix[2];
        camaraG->matrix->matrix[9] =
        matrizaux->matrix[6];
        camaraG->matrix->matrix[10] =
        matrizaux->matrix[10];
        camaraG->matrix->matrix[11] = 0.0;


        camaraG->matrix->matrix[12] =
        -((matrizaux->matrix[0]*matrizaux-
        >matrix[12])+
        (matrizaux->matrix[1]*matrizaux-
        >matrix[13]) + (matrizaux-
        >matrix[2]*matrizaux->matrix[14]));


        camaraG->matrix->matrix[13] =
        -((matrizaux->matrix[4]*matrizaux
        ->matrix[12])+(matrizaux-
        >matrix[5]*matrizaux->matrix[13]) +
        (matrizaux->matrix[6]*matrizaux-
        >matrix[14]));

        camaraG->matrix->matrix[14] = -
        ((matrizaux->matrix[8]*matrizaux-
        >matrix[12])+(matrizaux-
        >matrix[9]*matrizaux->matrix[13]) +
        (matrizaux->matrix[10]*matrizaux-
        >matrix[14]));
        camaraG->matrix->matrix[15] = 1.0;

        glLoadMatrixd(camaraG->matrix->matrix);
            guardar_estado();
            representar_matriz();
            }
        }
    }
    else {
      (...)
    }
}
```

Hemos implementado la rotación en modo vuelo y modo análisis en la función rotate. Si el modo cámara está activado y el modoVuelo está activado, entonces la cámara rotará en el ángulo definido por angle en el vector definido por a,b,c.
Si nos encontramos en modo vuelo (modoV = 0), se realizará la siguiente operación:

```
            glMatrixMode(GL_MODELVIEW);
            glLoadIdentity();
            glRotatef(angle,a, b, c);
            glMatrixMode(GL_MODELVIEW);
            glMultMatrixd(camaraG->matrix->matrix);
            guardar_estado();
            representar_matriz();
```

Primero cargamos la matriz de identidad, y de esta forma "limpiamos" la matriz activa de openGL. Despues, aplicamos la matriz de rotación sobre esta matriz de identidad (la cámara rotará en un ángulo angle, en el vector definido por a, b y c). Posteriormente, multiplicamos a la matriz activa de openGL la matriz de la cámara (también llamada la matriz de cambio de sistema de referencia). Las dos siguientes líneas de código se encargan de guardar en la pila de matrices de la cámara el estado actual de la cámara y de representar la matriz en el terminal.

Si nos encontramos en modo análisis (modoV=1) la cámara rota inscribiendo una órbita alrededor del objeto seleccionado, por lo que dicho objeto siempre estará en el centro de la pantalla cuando este modo esté activado, y estaremos mirándolo en todo momento. El procedimiento para rotar cambia bastante respecto al método anterior:

```
            matrix_l *matrizaux;
          matrizaux = (matrix_l *) malloc(sizeof (matrix_l));


          //Matriz inversa de la MCSR
          matrizaux->matrix[0] = camaraG->matrix->matrix[0];
      matrizaux->matrix[1] = camaraG->matrix->matrix[4];
      matrizaux->matrix[2] = camaraG->matrix->matrix[8];
      matrizaux->matrix[3] = 0.0;


      matrizaux->matrix[4] = camaraG->matrix->matrix[1];
      matrizaux->matrix[5] = camaraG->matrix->matrix[5];
      matrizaux->matrix[6] = camaraG->matrix->matrix[9];
      matrizaux->matrix[7] = 0.0;


      matrizaux->matrix[8] = camaraG->matrix->matrix[2];
      matrizaux->matrix[9] = camaraG->matrix->matrix[6];
      matrizaux->matrix[10] = camaraG->matrix->matrix[10];
      matrizaux->matrix[11] = 0.0;
      matrizaux->matrix[12] = camaraG->posicionC.x;
          matrizaux->matrix[13] = camaraG->posicionC.y;
          matrizaux->matrix[14] = camaraG->posicionC.z;
      matrizaux->matrix[15] = 1.0;


          glLoadIdentity();
          glMatrixMode(GL_MODELVIEW);
          glTranslatef(_selected_object->matrix->matrix[12],
          _selected_object->matrix->matrix[13],
          _selected_object->matrix->matrix[14]);
        glRotatef(angle,a,b,c);
          glTranslatef(-_selected_object->matrix->matrix[12],
          -_selected_object->matrix->matrix[13],
```

```
            -_selected_object->matrix->matrix[14]);
            glMultMatrixd(matrizaux->matrix);


            glMatrixMode(GL_MODELVIEW);
            glGetDoublev(GL_MODELVIEW_MATRIX,matrizaux->matrix);
            camaraG->posicionC.x = matrizaux->matrix[12];
            camaraG->posicionC.y = matrizaux->matrix[13];
            camaraG->posicionC.z = matrizaux->matrix[14];

        //Generamos la Mcsr a partir de la inversa
        camaraG->matrix->matrix[0] = matrizaux->matrix[0];
        camaraG->matrix->matrix[1] = matrizaux->matrix[4];
        camaraG->matrix->matrix[2] = matrizaux->matrix[8];
        camaraG->matrix->matrix[3] = 0.0;


        camaraG->matrix->matrix[4] = matrizaux->matrix[1];
        camaraG->matrix->matrix[5] = matrizaux->matrix[5];
        camaraG->matrix->matrix[6] = matrizaux->matrix[9];
        camaraG->matrix->matrix[7] = 0.0;


        camaraG->matrix->matrix[8] = matrizaux->matrix[2];
        camaraG->matrix->matrix[9] = matrizaux->matrix[6];
        camaraG->matrix->matrix[10] = matrizaux->matrix[10];
        camaraG->matrix->matrix[11] = 0.0;


        camaraG->matrix->matrix[12] =
        -((matrizaux->matrix[0]*matrizaux->matrix[12])
        +(matrizaux->matrix[1]*matrizaux->matrix[13])
        +(matrizaux->matrix[2]*matrizaux->matrix[14]));

        camaraG->matrix->matrix[13] =
        -((matrizaux->matrix[4]*matrizaux->matrix[12])
        +(matrizaux->matrix[5]*matrizaux->matrix[13])
        + (matrizaux->matrix[6]*matrizaux->matrix[14]));

        camaraG->matrix->matrix[14] =
        -((matrizaux->matrix[8]*matrizaux->matrix[12])
        +(matrizaux->matrix[9]*matrizaux->matrix[13])
        +(matrizaux->matrix[10]*matrizaux->matrix[14]));

        camaraG->matrix->matrix[15] = 1.0;

        glLoadMatrixd(camaraG->matrix->matrix);
            guardar_estado();
            representar_matriz();
```

Primero definimos una matriz auxiliar llamada matrizaux de tipo matrix_l y le asignamos el espacio de memoria correspondiente al tamaño de su tipo de dato. Esta matriz será la encargada de modificar la posición a la matriz de la cámara. Antes de continuar con la explicación, es necesario saber qué datos contiene la matriz de la cámara:

$$\mathbf{Mcsr} = \begin{pmatrix} Xcx & Xcy & Xcz & -Ex * Xc \\ Ycx & Ycy & Ycz & -Ey * Yc \\ Zcx & Zcy & Zcz & -Ez * Zc \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Xc, Yc, Zc, 0 define los ejes del sistema de referencia de la cámara en el sistema de referencia del mundo. Donde Xcx muestra a las coordenadas X de la cámara en el eje X del mundo, Ycx a las coordenadas X de la cámara en el eje Y del mundo, y Zcx a las coordenadas X de la cámara en el eje Z del mundo. La primera columna hace referencia a la X de la cámara, la segunda a la Y de la cámara, y la tercera a la Z de la cámara. La última fila se encarga de definir estos datos como coordenadas homogéneas. Si el valor de la última fila en una columna determinada es 0 entonces la columna es un vector, si es 1. entonces es un punto.
La última columna, como podemos observar, es un producto escalar entre la posición de la cámara en el mundo y un número E que desconocemos.
Para obtener la matriz que nos permita modificar la posición de la cámara, necesitamos obtener la matriz inversa de la cámara. Para rotar (y trasladar) la cámara en el modo vuelo nos vale con aplicar la función glTranslatef sobre la matriz de cambio de sistema de referencia, pero en este caso no nos vale. Necesitamos obtener la matriz inversa de la matriz de cambio de sistema de referencia, la cual contiene la posición de la cámara en su cuarta columna.

Primero transmutamos los valores de las tres primeras columnas de la matriz de la cámara en la matriz auxiliar, es decir, los componentes de la i-ésima fila en la matriz de la cámara serán los componentes de la i-ésima columna en la matriz auxiliar *matrizaux* (la que posteriormente será la matriz inversa de la matriz de la cámara. Después situamos la posición de la cámara en la cuarta columna de la matriz auxiliar.
Posteriormente cargamos la matriz de identidad en la matriz activa de openGL. Realizamos un *glTranslatef* a la posición del objeto, realizamos la rotación, y después volvemos a la posición original haciendo *glTranslatef* a la posición negativa del objeto. Despúes de realizar estas operaciones, multiplicamos la matriz auxiliar a la matriz cargada en openGL. Esta

nueva matriz es clave, porque ahora poseemos la posición actual de la cámara, y porque al invertirla como hemos hecho anteriormente, obtenemos la matriz de cambio de sistema de referencia completamente actualizada. Guardamos en *matrizaux* mediante *glGetDoublev* la matriz cargada en openGL, y procedemos a realizar exactamente el mismo proceso de inversión que hemos llevado a cabo antes. Sin embargo, como hemos visto, la matriz de cambio de sistema de referencia tenemos que colocar, en la i-ésima fila de la última columna el producto entre el vector de la i-ésima columna y la posición de la i-ésima fila en *matrizaux*. Cabe mencionar que el valor de estos productos debe ser negativo.

$$\mathbf{Mcsr(INVERSA)} = \begin{pmatrix} Xcx & Ycx & Zcx & Ex \\ Xcy & Ycy & Zcy & Ey \\ Xcz & Ycz & Zcz & Ez \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esta explicación puede ser visualizada mejor mediante el código que lo implementa.

```
camaraG->matrix->matrix[12] =
-((matrizaux->matrix[0]*matrizaux->matrix[12])
+(matrizaux->matrix[1]*matrizaux->matrix[13])
+ (matrizaux->matrix[2]*matrizaux->matrix[14]));

camaraG->matrix->matrix[13] =
 -((matrizaux->matrix[4]*matrizaux->matrix[12])
 +(matrizaux->matrix[5]*matrizaux->matrix[13])
 + (matrizaux->matrix[6]*matrizaux->matrix[14]));

camaraG->matrix->matrix[14] =
-((matrizaux->matrix[8]*matrizaux->matrix[12])
+(matrizaux->matrix[9]*matrizaux->matrix[13])
+(matrizaux->matrix[10]*matrizaux->matrix[14]));
camaraG->matrix->matrix[15] = 1.0;
```

Finalmente cargamos la matriz de cambio de sistema de referencia en la matriz activa de openGL.

iv. **Traslación.** La traslación, al igual que la rotación, puede llevarse a cabo tanto en modo vuelo como en modo análisis, y dependiendo del modo en el que estemos, se realizará de manera distinta.

En el modo vuelo la traslación se realiza llevando a cabo un proceso homólogo al de la rotación, siguiendo el siguiente código:

```
glMatrixMode(GL_MODELVIEW);
```

```
glLoadIdentity();
glTranslatef(a,b,c);

camaraG->posicionC.x += a;
camaraG->posicionC.y += b;
camaraG->posicionC.z += c;

glMatrixMode(GL_MODELVIEW);
glMultMatrixd(camaraG->matrix->matrix);
guardar_estado();
representar_matriz();
```

Primero definimos que la matriz activa sea el modelview. Después cargamos la matriz de identidad para "limpiar" la matriz activa de openGL. Posteriormente hacemos la traslación mediante la que movemos todos los elementos visibles por la cámara en la dirección y sentido declarados por el vector compuesto por los parámetros a,b,c. Cabe mencionar que los elementos no son trasladados en el sistema de referencia global (es decir, si tenemos un avión en el centro, y movemos la cámara a la derecha el avión no se moverá hacia la izquierda en el sistema de referencia global), si no en el sistema de referencia local de la cámara.
Después mediante la función *glMultMatrixd* multiplicamos a la matriz activa de openGL la matriz de cambio de sistema de referencia (o matriz de la cámara) mediante la cual hacemos efectiva la traslación de todos los puntos visibles por la cámara. Es decir, trasladamos los puntos visibles por la cámara, pero porque es la cámara la cual se mueve y no los puntos que ella ve.
Como en cualquier transformación realizada sobre cualquier objeto, guardamos el estado mediante guardar_estado() y representar_matriz().

Al igual que en la rotación, en el modo análisis también se presentan severas diferencias respecto a la del modo vuelo.

```
double distanciaZ=absoluto(-camaraG->posicionC.z
- _selected_object->matrix->matrix[14]);

if(distanciaZ>c){
  glLoadIdentity();
  glTranslatef(a,b,c);
  camaraG->posicionC.z += c;
  glMatrixMode(GL_MODELVIEW);
  glMultMatrixd(camaraG->matrix->matrix);
  guardar_estado();
  representar_matriz();
}
else{
```

```
  printf("No nos podemos mover mas adelante\n");
  printf("DistanciaZ:%f <= Movimiento:%f\n",distanciaZ,c);
}
```

En primer lugar, cabe mencionar que únicamente nos trasladaremos en el eje z cuando esté activado el modo análisis. La principal razón por la que esto es así es sencilla: el objeto a analizar SIEMPRE HA DE HALLARSE EN EL CENTRO DE LA CÁMARA. Si nos pudiesemos desplazar en los ejes x e y esto ya no sería así.
Cabe mencionar también que una función llamada *absoluto* se ha realizado para que devuelva el valor absoluto del número introducido como parámetro de entrada.

```
double absoluto(double a) {
  if(a>0) return a;
  else  return -a;
}
```

Comenzamos hallando el valor absoluto de la diferencia entre el valor negativo de la cámara en z y la componente z de la posición del objeto. Si está diferencia no es mayor que el número de unidades en z en la que nos trasladamos cada vez que pulsamos *GLUT_KEY_PAGE_UP*, ya no podremos avanzar más. Si la diferencia sí es mayor, primero cargaremos la matriz de identidad para "limpiar" la matriz activa de openGL y después usaremos glTranslatef para trasladar los puntos en el vector definido por a,b y c. Como hemos trasladado la cámara, actualizamos la variable componentes posicionC.z de camaraG aumentando su valor en c unidades. Posteriormente multiplicaremos a la matriz activada la matriz de cambio de sistema de referencia mediante *glMultMatrixd*, para que los cambios se produzcan sobre el sistema de referencia de la cámara. Después guardamos el estado y representamos la matriz en el terminal.

(b) **Cambio de tipo de proyección.**

Existen dos tipos de proyecciones: La perspectiva y la paralela, con grandes diferencias entre ambas. La variable encargada de establecer el modo de proyección es *modop*, el cual puede tomar dos valores: modop = 0 (modo paralelo) y modop = 1 (modo perspectiva)
La proyección en perspectiva nos permite visualizar un objeto en relación a su posición dentro del volumen de visión de la cámara. EL volumen de visión hace referencia a la totalidad del espacio

visible por una cámara dentro del mundo, y se define como una píramide cuya cúspide se halla en la posición de la cámara y que se extiende tanto como el objeto pueda llegar a ver. La implementación de este tipo de perspectiva se realizó en los anteriormente mencionados left, right, far, near, bottom y top.
Top y bottom establecen la extensión vertical de nuestro volumen de visión, right y left la extensión horizontal, y far y near cuanto de lejos y cerca respectivamente podemos llegar a ver. Conforme más se acerque un objeto a la cámara, más grande se verá, y conforme más se aleje, más pequeño. El objeto dejará de ser visible cuando sobrepase los límites definidos por near y far. La función encargada de realizar esta transformación ha sido *glFrustrum* y se ha realizado de la siguiente manera en *display.c*

```
if(modop==0) {
 (...)
 }
 else {
     glFrustum(camaraG->left,
     camaraG->right,camaraG->bottom,
     camaraG->top,camaraG->near,camaraG->far);
 }

 /* Now we start drawing the object */
 glMatrixMode(GL_MODELVIEW);
 glLoadMatrixd(camaraG->matrix->matrix);
```

La proyección en paralelo no contempla que la dimensionalidad de los objetos varíe en relación a su posición. Un objeto se verá con las mismas dimensiones independientemente de su posición en el eje z, del mismo modo si lo trasladamos a un lateral lo seguiremos viendo de la misma forma. Si antes hemos definido el volumen de visión en la proyección en perspectiva como una pirámide cuyo cúspide estaba en la cámara, podemos definir el volumen de visión de la proyección en paralelo como un rectángulo. La implementación de la proyección en modo paralelo también se ha realizado en el fichero *display.c*, y ha sido realizada de la siguiente manera:

```
if(modop==0) {
    /*When the window is wider than
     our original projection plane we
     extend the plane in the X axis*/
      if ((_ortho_x_max - _ortho_x_min)
      / (_ortho_y_max - _ortho_y_min)
      < _window_ratio) {
        /* New width */
          GLdouble wd = (_ortho_y_max
          - _ortho_y_min) * _window_ratio;
```

```
            /* Midpoint in the X axis */
              GLdouble midpt = (_ortho_x_min
              + _ortho_x_max) / 2;
            /*Definition of the projection*/
              glOrtho(midpt - (wd / 2),
              midpt + (wd / 2), _ortho_y_min,
              _ortho_y_max, _ortho_z_min, _ortho_z_max);
          }
          else {/* In the opposite situation
          we extend the Y axis */
            /* New height */
              GLdouble he = (_ortho_x_max
              - _ortho_x_min) / _window_ratio;
            /* Midpoint in the Y axis */
              GLdouble midpt = (_ortho_y_min
              + _ortho_y_max) / 2;
              /*Definition of the projection*/
              //glOrtho(_ortho_x_min,
              _ortho_x_max, midpt - (he / 2),
              midpt + (he / 2), _ortho_z_min,
              _ortho_z_max);
              glOrtho(camaraG->
              ortho_x_min,camaraG->
              ortho_x_max,camaraG->ortho_y_min,
              camaraG->ortho_y_max,camaraG->near,
              camaraG->far);
          }
      else {
      (...)
      }

      /* Now we start drawing the object */
      glMatrixMode(GL_MODELVIEW);
      glLoadMatrixd(camaraG->matrix->matrix);
```

Iluminación

1. **Explicación.** A continuación se explicará el funcionamiento de la luz y los materiales en OpenGL. Para empezar es importante saber que OpenGL usa la composición de colores RGB (RedGreenBlue). Esto significa que los colores primarios son el rojo, el verde y el azul y el resto de colores los generará con una mezcla de estos. El blanco sería la unión perfecta de los tres colores en la misma proporción y el negro la ausencia de estos.

   (a) **Luz.** Existen 4 tipos de luz en OpenGL, estos son, la luz ambiental, la difusa, la especular y la emisiva.

      i. **Luz ambiental:** Esta es la luz que ha sido tan esparcida despues de rebotar tanto que es imposible determinar su origen. Viene de todas direcciones, es simplemente la luz que hay en el ambiente que no proviene de un origen concreto.

Cuando la luz ambiental golpea una superficie, esta se esparce de forma equitativa en todas direcciones.

ii. **Luz difusa:** Esta luz, al contrario que la ambiental, proviene de una dirección concreta, pero de forma difusa. La luz del sol sería un buen ejemplo de una luz difusa.
Esta luz es más brillante si golpea directamente una superficie que si lo hiciera de forma parcial. Al igual que la ambiental al golpear un objeto se esparce equitativamente en el espacio.

iii. **Luz especular:** Esta es la luz que proviene de una dirección particular y tiende a rebotar hacia otra concreta dirección. Piensa en ella como el brillo de la superficie, dependiendo del brillo rebotará con más o menos intensidad y hacia una dirección concreta. Un espejo o un metal, por ejemplo, tendran mucha luz especular ya que reflejan muy bién la luz.

Simplificando, podríamos decir que la luz está compuesta por la intensidad de luz, roja, verde o azul que emite. Cabe destacar que estos 3 números toman valores entre 0 y 1.

(b) **Material.** Un material se verá de X color dependiendo de la cantidad y tipo de luz que emita de los rayos de luz que lo golpean. Si simplificamos, la luz está compuesta por 3 valores, que son la cantidad de luz roja, verde o azul que emiten. El material (simplificando) también estará compuesto por estos tres valores, pero en este caso simbolizan la cantidad de luz que emitirá en función de la cantidad de rayos de luz que lo golpeen. Esto quiere decir que si iluminamos con una luz blanca a un material que queremos que sea rojo, el valor de luz roja del material deberá de ser mayor al de los otros dos valores. Si la luz es blanca (Rojo=1,Verde=1,Azul=1) y el material tiene estos valores, Rojo=1,Verde=0,Azul=0, el objeto se verá rojo.
Cabe destacar que esto es una explicación simplificada. Al igual que la luz, los materiales también tienen distinción entre el tipo de luz que reflejan (emisiva, especular, difusa y ambiental).

(c) **Fuentes de luz.** Nosotros trabajamos con 3 tipos diferentes de fuentes de luz, soles, bombillas y focos.

i. **Soles.** El sol es una fuente de luz que proviene desde una dirección a una distancia infinita. Sus rayos de luz vienen en paralelo.
Este tipo de fuente no puede ser trasladado ya que no tiene una posición concreta pero si que puede ser rotado para mod-

ificar la dirección en la que viene.

ii. **Bombillas.** Este tipo de fuente de luz tiene una posición fija en el espacio y ilumina en todas direcciones por igual (aunque esto podría ser modificado a placer).
Puede ser trasladada porque tiene una posición finita, pero no tiene sentido rotarla porque ilumina a todas direcciones por igual.

iii. **Focos.** Al igual que la bombilla, el foco tiene una posición fija en el mundo pero no ilumina en todas direcciones por igual, ilumina en una dirección concreta y de forma limitada. Un foco, al igual que una linterna real, ilumina en una dirección y con un haz de luz de forma cónica, este cono va definido por el ángulo de apertura que le especifiquemos, cuanto mayor sea, mas extenso será la base del cono y iluminará mas cosas cercanas a él, cuanto mas pequeño, más le costará iluminar objetos cercanos y mas concentrada estará la luz en el punto al que mira.
El foco puede ser tanto rotado, como trasladado sin problema alguno.

2. **Vectores normales.** Un vector normal es un vector perpendicular a la superficie del polígono al que pertenece, su distancia es de 1. Con este vector podemos calcular la cantidad de luz que llegará al objeto, dependiendo de como se usen tendremos 2 formas de iluminación distintas (FLAT o SMOOTH).

   (a) **FLAT.** Esta forma de iluminar usa únicamente los vectores normales de los polígonos del objeto, asi que cada superficie estará compuesta por un único tono de iluminación.

   (b) **SMOOTH.** Esta forma de iluminar usa tanto los vectores normales de los polígonos como los de los vértices. Al usar ambos vectores, el objeto tendra sombras más gradientes, no terminarán de forma tan abrupta como con el modo FLAT.

   Para almacenar los vectores normales de los vértices y de las caras hemos tenido que modificar la estructura de las caras y los vértices de los objetos, quedando de la siguiente manera.

```
typedef struct {
    point3 coord;
    GLint num\_faces;
    vector3 normalV;      /*Vector normal del vertice*/
```

```
} vertex;

typedef struct {
    GLint num_vertices;
    vector3 normalF;      /*Vector normal del poligono*/
    GLint *vertex\_table;
} face;
```

Para calcular el vector normal de una cara necesitamos tres de sus puntos (a,b y c), una vez obtenidos obtenemos 2 vectores (vector1 = 3-1,vector2 = 2-1). Una vez obtenidos estos dos vectores realizaremos su producto vectorial para obtener el vector perpendicular a la cara.
Ahora solo queda normalizar el vector, es decir, hacer que su distancia sea de 1. Esto lo haremos calculando su distancia (raiz cuadrada de la suma de cuadrados) y dividiendo sus coordenadas por esta recién calculada distancia. De esta forma ya tenemos el vector final de ese polígono concreto.
Para calcular el vector normal de los vértices, primero les sumaremos los vectores normales de todas las superficies de las que formen parte y una vez hecho esto, normalizaremos dichos vectores, al igual que hemos explciado anteriormente.
Cabe destacar, que en el código estos cálculos tendrán que ser realizados una única vez al cargar los objetos.

```
/*Esta parte de codigo es muy extensa*/
/*por lo que si desea verlo debera de*/
/*dirigirese al codigo donde lo hemos*/
/*explicado en detalle, esta situado */
/*en load_obj_joseba.c al cerrar el  */
/*archivo tras la primera pasada y   */
/*antes de la segunda.               */
```

Ahora ya tenemos los vectores normales calculados, pero nos falta decirle a OpenGL cuando y cual va a ser el vector normal de los vértices. Esto lo haremos a la hora de dibujar el objeto, es decir, en el 'display.c', justo antes de que dibuje los vértices.

```
/* Draw the object; for each face create a new polygon
with the corresponding vertices */
for (f = 0; f < aux_obj->num_faces; f++) {
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glBegin(GL_POLYGON);
    for (v = 0; v < aux_obj->face_table[f].
    num_vertices; v++) {
        v_index = aux_obj->face_table[f].
        vertex_table[v];

        glNormal3d(aux_obj->vertex_table[aux_obj->
```

```
        face_table[f].vertex_table[v]].normalV.x,
            aux_obj->vertex_table[aux_obj->
            face_table[f].vertex_table[v]].
            normalV.y,
            aux_obj->vertex_table[aux_obj->
            face_table[f].vertex_table[v]].normalV.z);

        glVertex3d(aux_obj->vertex_table[v_index].
        coord.x,
                aux_obj->vertex_table[v_index].
                coord.y,
                aux_obj->vertex_table[v_index].
                coord.z);

    }
```

Con esto ya habriamos terminado de implementar los vectores normales que son necesarios a la hora de especificar el tipo de iluminación que queremos (flat o smooth).
Cabe destacar que en este mismo bucle le asignamos el material deseado a las caras del objeto (también podriamos asignárselo únicamente al objeto). En nuestro caso, el material de todos nuestros objetos oro (esto puede ser cambiado a voluntad modificando los valores utilizados).

3. **Creación de luces.** Nuestra aplicación dispone de 4 fuentes de luz. 1:Un sol, 2: Una bombilla, 3:Un foco en el objeto y 4:Un foco externo. Las luces han sido creadas en el 'display.c'. La iluminación se apaga y enciende pulsando la tecla 'f9', pero si deseamos apagar/encender una fuente de luz en concreto tendremos que pulsar las teclas del 'f1' al 'f4', cada una apaga/enciende la luz correspondiente a su número.
Cabe destacar la diferencia entre una fuente de luz posicional y no. El sol, por ejemplo, es una fuente no posicional, esto significa que no tiene una posición fija en el espacio, esto se define con la siguiente llamada: 'glLightfv(GL_LIGHT0, GL_POSITION, sun_position)' donde el vector "sun_position" está compuesto por 4 elementos, si el cuarto elemento es un 0, significa que la fuente será un sol y sus 3 valores restantes definiran de donde proviene la luz. En caso de que el cuarto valor valga 1, los tres primero definirán la posición en el mundo de la fuente.
Sabiendo esto, solo queda aclarar que nuestra bombilla está en la posición (1,1,1) y ilumina en todas direcciones. Nuestro foco relacionado al objeto se encuentra en la posición del objeto seleccionado, apunta hacia el mismo objeto y el ángulo de su foco es de 25º. Y el foco externo está colocado en la posicion (-1,-1,-1), apunta a (0,0,-1) y su ángulo de foco es de 15º.
Nuestra aplicación dispone de una variable 'iluminacion' que indica

si la iluminación general está activada (iluminacion=1) o no (iluminacion=0). También existe un array de integers de longitud igual al número de luces (4), este array representará si esa luz en concreto está apagada o no y también será la forma que tendremos de seleccionar una luz concreta a través de un puntero, esto se explicará en el siguiente apartado.
Cabe destacar que para desactivar la iluminación basta con desactivarla en general, no hace falta desactivar todas las fuentes de luz de forma individual.

```
case GLUT_KEY_F9:
    if(iluminacion==0) {
        printf("Usted acaba de activar el modo de
        iluminacion.\n");
        iluminacion=1;
        luces[0]=1; luces[1]=1; luces[2]=1; luces[3]=1;
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHT1);
        glEnable(GL_LIGHT2);
        glEnable(GL_LIGHT3);
        glEnable(GL_DEPTH_TEST);
    }
    else{
        iluminacion=0;
        glDisable(GL_LIGHTING);
        luces[0]=0; luces[1]=0;
        luces[2]=0; luces[3]=0;
    }
break;

case GLUT_KEY_F12:
    if(iluminacion==1) {
        iluminacion=2;
        printf("Usted acaba de activar
        el modo SMOOTH.\n");
        glShadeModel(GL_SMOOTH);
    }
    else{
        iluminacion=1;
        printf("Usted acaba de activar
        el modo FLAT.\n");
        glShadeModel(GL_FLAT);
    }
break;

case GLUT_KEY_F1:
    if(iluminacion!=0) {
        if(luces[0]==1) {//Luz 0 activada, la desactivamos
            luces[0]=0;
            printf("Se ha DESACTIVADO la fuente de luz 0,
            el sol.\n");
            glDisable(GL_LIGHT0);
        }
        else{//Luz 0 desactivada, la activamos
            luces[0]=1;
            printf("Se ha ACTIVADO la fuente de luz 0,
            el sol.\n");
```

```
            glEnable(GL_LIGHT0);
        }
    }
    else {
        printf("Activa la iluminacion para encender/apagar
        fuentes de luz.\n");
    }
break;
//f3 y f4 son identicos a f1 o f2
...
...
```

4. **Modificación de luces.** Nuestra aplicación solo permite trasladar fuentes de luz y aumentar o reducir el ángulo de apertura de un foco. Dado las características de las fuentes, solo seremos capaces de trasladar la bombilla y el foco externo, pero podremos modificar el ángulo de apertura de tanto el foco del objeto como el externo.
   Para seleccionar la fuente que deseamos modificar deberemos de pulsar las teclas del '1' al '4', cada una seleccionará la fuente correspondiente a su número. Aunque solo podremos trasladar la 2 y 4 y modificar el ángulo de apertura de los focos 3 y 4.
   Este es un fragmento de la función 'traslate(a,b,c)' que hemos creado para trasladar los objetos, la variable 'moverluces' representa si debemos de modificar una luz o no. 'a','b' y 'c' son los valores en los que nos trasladamos, por lo que únicamente deberemos de volver a designar la posición de la luz modificándola con los valores anteriormente mencionados.
   Para aumentar/reducir el ángulo de apertura de un foco usaremos un proceso simiar, simplemente volveremos a definir ese ángulo aumentandolo o reduciéndolo en una proporcion del 10 por ciento.

```
//Funcion traslate(a,b,c)
...
if (moverluces==1)
  {
    if(luzseleccionada==luces[1]){
      printf("Trasladamos la bombilla\n");
      pbombilla[0]+=a;
      pbombilla[1]+=b;
      pbombilla[2]+=c;
      glLightfv(GL_LIGHT1, GL_POSITION, pbombilla);
      pbombilla[1],pbombilla[2]);
    }
    else if(luzseleccionada==luces[3]){
      printf("Trasladamos el foco externo\n");
      pfoco2[0]+=a;
      pfoco2[1]+=b;
      pfoco2[2]+=c;
      glLightfv(GL_LIGHT3, GL_POSITION, pfoco2);
      pfoco2[1],pfoco2[2]);
    }
    else {
```

```
      printf("La fuente de luz seleccionada no
      puede ser traladada.\n");
    }
...
//-------------------------------------------------------
//Funcion de teclado
...
case '-':
      printf("Acaba de pulsar '-'\n");
      if(iluminacion=!0 && luces[2]==1
      && luzseleccionada==luces[2]){
        glLightf(GL_LIGHT2, GL_SPOT_CUTOFF,
        angulofoco*0.9);
        angulofoco=angulofoco*0.9;
      }
      else if(iluminacion=!0 && luces[3]==1
      && luzseleccionada==luces[3]){
        glLightf(GL_LIGHT3, GL_SPOT_CUTOFF,
        angulofoco2*0.9);
        angulofoco2=angulofoco2*0.9;
      }
      else {
        printf("No pasa nada\n");
      }
      ...

case '+':
      printf("Acaba de pulsar '+'\n");
      if(iluminacion=!0 && luces[2]==1
      && luzseleccionada==luces[2]){
        glLightf(GL_LIGHT2, GL_SPOT_CUTOFF,
        angulofoco*1.1);
        angulofoco=angulofoco*1.1;
      }
      else if(iluminacion=!0 && luces[3]==1
      && luzseleccionada==luces[3]){
        glLightf(GL_LIGHT3, GL_SPOT_CUTOFF,
        angulofoco2*1.1);
        angulofoco2=angulofoco2*1.1;
      }
      else {
        printf("No pasa nada\n");
      }
      ...
```

Cabe destacar que tanto en el caso de pulsar '-' o '+' existe más código, pero al no tener nada que ver con esta parte, hemos decidido no representarlo aqui.