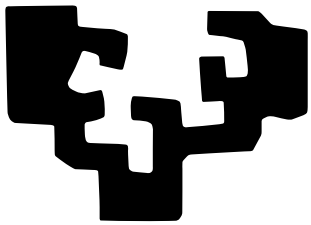
eman ta zabal zazu



Universidad del País Vasco

Euskal Herriko Unibertsitatea

Gráficos por Computador : Transformaciones geométricas

Finalizado el Sábado, 31 de diciembre, 2016

Cristina Mayor y Sergio Tobal

Contents

1.- Objetivos de la práctica **3**
1.1. Transformaciones de los objetos . . . 3
1.2. Transformaciones de la cámara . . . 3
1.3. Transformaciones de las luces . . . 3

2.- Métodos de resolución **4**
2.1. Transformaciones de los objetos . . . 4
2.2. Transformaciones de la cámara . . . 5
2.3. Transformaciones de las luces . . . 5

3.- Solución elegida **6**
3.1. Transformaciones de los objetos . . . 6
3.2. Transformaciones de la cámara . . . 7
3.3. Transformaciones de las luces . . . 8

4.- Explicación del código **9**
4.1. Transformaciones de los objetos . . . 9
4.2. Transformaciones de la cámara . . . 16
4.3. Transformaciones de las luces . . . 21

5.- Ampliación a lo pedido **33**
5.1. Transformaciones de los objetos . . . 33
5.2. Transformaciones de la cámara . . . 33
5.3. Transformaciones de las luces . . . 33

6.- Conclusiones **35**
6.1. Transformaciones a los objetos . . . 35
6.2. Transformaciones a la cámara . . . 35
6.3. Transformaciones de las luces . . . 35

Referencias **36**

Anexos **37**

Anexo I: Manual del usuario - Transformaciones de los objetos **37**

Anexo II: Manual del usuario - Transformaciones de la cámara **41**

Anexo III: Manual del usuario - Transformaciones de las luces **42**

1.- Objetivos de la práctica

En esta entrega podemos apreciar dos diferentes objetivos ya que tenemos un archivo con los contenidos correspondientes a la práctica de cámaras y a la práctica de objetos.

1.1. Transformaciones de los objetos

En esta parte aprenderemos cómo utiliza OpengGL las matrices internamente para poder hacer transformaciones a los objetos.

Durante este proceso, también veremos la pila que usa OpenGL internamente, y como la usa para empilar y desempilar las distintas matrices de transformaciones que hemos ido realizando.

1.2. Transformaciones de la cámara

En esta parte de la entrega aprenderemos a gestionar diferentes cámaras para poder observar un mismo entorno desde la perspectiva de una cámara o desde un objeto que se ha convertido a cámara. También aprenderemos a cómo aplicar transformaciones a las cámaras.

En esta parte aprenderemos cómo utiliza OpenGL las perspectivas (la matriz de proyección) para así dejarnos tener diferentes perspectivas en las cámaras.

1.3. Transformaciones de las luces

En esta parte de la entrega aprenderemos a gestionar diferentes tipos de luces para poder iluminar las secciones del entorno que queramos visualizar. También aprenderemos como se realizan las transformaciones a las luces.

Durante este proceso aprenderemos cómo utiliza OpenGL las luces, los parámetros necesarios para inicializarlas y cómo se gestiona cada tipo de luz.

2.- Métodos de resolución

Dentro de este apartado también apreciamos dos sub-secciones ya que tanto en la práctica de los objetos como en la práctica como en la práctica de las transformaciones de la cámara hay diferentes maneras de llegar a una solución.

2.1. Transformaciones de los objetos

Hemos visto 3 posibles maneras de realizar esta practica, que detallamos a continuación:

1. **Calculando nosotros las matrices**, y las transformaciones. Este método es el mas complicado de los tres pero es el que mas versatilidad nos ofrece, ya que podremos aplicar algoritmos o mejoras que no estén en OpenGL. Por otro lado, es el mas proclive a errores debido a que somos nosotros los que deberemos calcular todas las matrices y sus transformaciones. La manera de crear la matriz sería la siguiente:

   char[16] identityMatrix = {1.0f,0.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,0.0f,0.0f,1.0f}

2. **Usando la manera de OpenGL 4** en adelante, es decir, usando una librería matemática como GLM, usada debido a que en esas versiones se entiende que se van a usar shaders para programar las transformaciones. En caso contrario, se debe usar una librería externa a OpenGL para hacer los cálculos. De este modo, nos olvidamos de definir las matrices pero nos sigue permitiendo versatilidad en cuanto a las maneras de utilizarlas. El problema reside al tener que acordarnos de cómo funcionan las matrices en OpenGL, que se guarda siempre la traspuesta de la matriz usada.

   glm::mat4 myIdentityMatrix = glm::mat4(1.0f);

3. **Depender de OpenGL** para generar las transformaciones. Esta es la manera mas cómoda ya que no debemos saber el código que se efecúa detrás de las funciones a las que llamamos. Nosotros conoceremos qué devuelven las funciones dependiendo de los parámetros de entrada pero no necesitamos conocer la matemática que hay detrás de los mismos. En este caso utilizamos los métodos que OpenGL nos ofrece para efectuar el cálculo matricial, es la forma mas sencilla pero que esta en estado *deprecated* ya que en las versiones mas nuevas no existen estos métodos.

   glLoadIdentity();

   Los ejemplos vistos en los 3 casos eran para crear una matriz identidad, es decir, la matriz que está formada por unos en la diagonal principal y por ceros en el resto de la matriz. Como podemos ver, cada caso es mas simple que el anterior, siendo el más sencillo el depender de OpenGL.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1*x+0*y+0*z+0*w \\ 0*x+1*y+0*z+0*w \\ 0*x+0*y+1*z+0*w \\ 0*x+0*y+0*z+1*w \end{bmatrix} = \begin{bmatrix} x+0+0+0 \\ 0+y+0+0 \\ 0+0+z+0 \\ 0+0+0+w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

2.2. Transformaciones de la cámara

Al igual que con las transformaciones al objeto, las transformaciones de la cámara de al menos 2 maneras diferentes.

1. **Calculando nosotros las matrices**. Las cámaras se representan como matrices 4x4, que almacenan 3 valores, el punto donde esta la cámara, el punto al que mira la cámara y la verticalidad de la misma. De este modo, para realizar una transformación nosotros deberíamos hacer todo el cálculo matemático que hay tras cada acción con las matrices aplicando los conocimientos adquiridos en las clases teóricas de la asignatura. Deberíamos calcular la matriz de transformación y actualizar con ella nuestra matriz cámara.

2. **Usando OpenGL Mathematics**. Este método es muy similar al anterior, tendremos que calcular todas las matrices de las cámaras. Pero, usando GLM, tenemos ciertas ventajas. Podríamos nombrar, por ejemplo, el cálculo de una matriz inversa. En vez de calcularla paso a paso como se nos enseña en los primeros cursos de la carrera, podemos usar la función

   glm::inverse();

   Esto nos sirve de gran ayuda ya que muchos fallos surgen a raíz de pequeñas alteraciones en números o errores de operaciones. Gracias a esta función podríamos solventar algún error. Ademas, para usar OpenGL tendríamos que calcular la traspuesta de las matrices dado que es con lo que trabaja.

3. **Usando las funciones de OpenGL**. Gracias a las funciones que OpenGL incluye, se puede hacer todo el cálculo con las transformaciones que hemos aprendido en la práctica anterior, es decir, a la hora de querer aplicar una transformación sobre una cámara utilizando las funciones

   glTranslate();

   glRotate();

   y a la hora de generar el campo de visión de la cámara utilizar la funciones:

   gluLookAt();

   glFrustum();

   Esta solución podría generar un problema si quisiéramos portar la aplicación a otro motor gráfico ya que si no tiene las opciones de OpenGL no se podría usar este código.

2.3. Transformaciones de las luces

En esta entrega no hemos visto demasiadas maneras de hacerlo, ya que en clase se nos ha explicado una que es la que hemos visto en la mayor parte de Internet. Por supuesto siempre nos quedaba la opción de hacerlo de manera matemática, calculando nosotros las matrices o dejando a OpenGL que las calculara usando los métodos glTranslate() y demás métodos vistos en el anterior punto.

El problema de usar los métodos de OpenGL es que deberíamos pasar los vectores de los tipos de luz, ambiental, difusa y especular a matrices para poder realizar las operaciones mediante glRotate() y glTranslate(), y terminar sacando los valores de la nueva matriz que genere OpenGL.

Mientras que si lo hacemos sin usar los métodos de OpenGL y sabiendo como se multiplica una matriz por un vector son métodos triviales en el que con 2 lineas hemos calculado la multiplicación y guardado su nuevo en el vector, esto lo explicaremos y veremos mas a fondo en siguientes puntos.

3.- Solución elegida

3.1. Transformaciones de los objetos

Para efectuar esta práctica hemos decidido depender de las funciones que OpenGL nos ofrece para el cálculo matricial que hay que hacer para las transformaciones.

Al inicio del proyecto se nos ha entregado un programa a medio realizar (un código fuente) en el que podemos cargar distintos objetos 3D. Para realizar las pruebas necesarias y asegurarnos de que nuestro programa funcionaba correctamente hemos utilizado los ficheros abioia.obj y logoehu_ona.obj ya que estos dos tienen una dimensión similar y están bien definidos. Con este programa también podemos hacer zoom al entorno virtual donde cargamos los distintos objetos.

Se nos pide aplicar las transformaciones de escalado, rotación y traslación, la tecla TAB para movernos entre los distintos objetos, las tecla CTRL Z para deshacer las transformaciones realizadas sobre el objeto y, como suplemento, hemos añadido el CTRL Y rehacer los cambios de nuevo. Con el uso de las teclas de dirección AVPAG, REPAG, '+' y '-' podremos hacer las transformaciones de los objetos.

Para empezar, tenemos que saber como funciona OpenGL; tiene 3 matrices, una para modelos, otra para proyecciones, y dependiendo de la versión otra para la vista, aunque en algunas versiones es la misma que la de proyección.

En esta practica usaremos la matriz del modelo, cuyo nombre es GL_MODELVIEW_MATRIX, en ella se guardan todos los cambios que se han hecho a un objeto, como por ejemplo un escalado que se calcula usando la función glScale().

Internamente glScale() se encarga de multiplicar los valores que le pasemos, por la matriz usada por ultima vez. Así que es recomendable seleccionar la matriz del modelview para no tener problemas, esto lo haremos usando glMatrixMode(GL_MODELVIEW). De esta manera cuando se haga la multiplicación y se llame otra vez al dibujado, la matriz del modelview o modelado tendrá ya las coordenadas transformadas.

De esta manera, se genera una nueva matriz con los resultados de la multiplicación, que se coloca en la cima de la pila interna que tiene OpenGL, y que se encarga de guardar todos los cambios realizados a los objetos.

Tenemos también una pila propia dentro de cada objeto. En esta pila se almacena los cambios que se han ido realizando sobre el objeto seleccionado. Esta pila tiene una estructura de lista doblemente enlazada (DoubleLinkedList) , permitiendo así que el usuario pueda deshacer y rehacer sus cambios, cargando la matriz de un paso concreto y devolviendo el objeto al aspecto en ese paso.

3.2. Transformaciones de la cámara

Hemos decidido optar por el tercer método de resolución planteado, es decir, utilizando las opciones que OpenGL nos brinda para facilitar nuestro trabajo.

Para generar esta segunda fase hemos partido del código final de la segunda entrega de la asignatura. Con ese fichero ya podíamos transformar los objetos pero no podíamos cambiar la perspectiva con la que los observábamos ni podíamos cambiar el punto de vista. Esto es lo que se nos pide en esta tercera entrega.

De este modo se nos pide crear diferentes cámaras (tanto cámaras libres como cámaras ligadas a objetos) y poder realizar las transformaciones a dichas cámaras. Las transformaciones que se nos piden son las siguientes: trasladar la cámara, rotar la cámara y cambiar el volumen de visión de la cámara. Con las teclas de dirección, AVPAG y REPAG podremos observar las transformaciones que hacemos a la cámara.

Para empezar tenemos que conocer cómo gestiona OpenGL las cámaras. Vamos a tener que aprender a usar las diferentes proyecciones que se pueden usar (en nuestro caso en paralela y en perspectiva) y como transformar las mismas. Por otro lado, deberemos saber cómo realizar los movimientos en las cámaras.

Para poder cambiar la perspectiva tendremos que decir que vamos a usar la matriz de proyección y para ello utilizaremos glMatrixMode(GL_PROJECTION). Una vez seleccionado que vamos a trabajar con la matriz de proyección tendremos que decidir si queremos trabajar con una proyección en perspectiva (utilizar la función glFrustum()) o queremos trabajar con una proyección en paralelo (glOrtho()).

Por otro lado se nos pide poder tener una cámara en modo análisis, es decir, gestionar una cámara para que todas las rotaciones que haga las realice sobre el objeto seleccionado. Además, deberemos de tener una lista de cámaras, gestionada otra vez como una DoubleLinkedList ya que se nos pide poder pasar entre cámaras.

Con todo lo aprendido para esta práctica y utilizando los conocimientos adquiridos en la práctica anterior se procederá a la programación de esta segunda parte del proyecto.

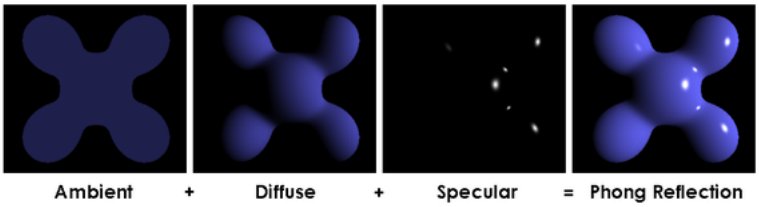
3.3. Transformaciones de las luces

Las luces que hemos creado se componen principalmente de 3 componentes, cuya teoria también nos ha servido para definir los coeficientes de los materiales:

- Luz ambiental: Es la luz que entra en un espacio y va rebotando antes de iluminar cualquier objeto que haya en ese espacio, la contribución de la luz ambiental depende del color de la luz y del color ambiente del material. El color de la luz representa el color y la intensidad de la luz, mientras que el color del material representa la luz general ambiente que el material refleja.

- Luz difusa: Esta luz representa la luz direccional que golpea a una superficie, su contribución a la iluminación general del espacio depende de su angulo de incidencia. Depende de muchos factores, como son el color del material, el color de la luz, su dirección o el vector normal del objeto donde incide.

- Luz especular: Esta es la luz que vemos como un brillo blanco en superficies muy pulidas o cristales, y por tanto depende de la dirección de la luz, de las normales y de donde este situación el observador de la escena, en nuestro caso la cámara.



En nuestro trabajo existen 3 tipos de luces. Todas ellas pueden tener una atenuación, una luz difusa, ambiental o especular para definir como son, pero lo que las hace ser lo que son, es lo siguiente:

1. Luz direccional: Es la luz que viene de una dirección y sus "rayos" son paralelos porque vienen de una fuente de luz muy lejana. Esta luz afecta a todos los objetos por igual y decimos que es de tipo sol ya que en el mundo real es la luz que más se le parece.

2. Luz puntual: Es una luz que viene de un punto concreto e ilumina en todas las direcciones. Esta luz, a diferencia de la luz direccional, no afecta a todos los objetos por igual. Decimos que es de tipo bombilla ya que en el mundo real es la luz que más se le asemeja.

3. Luz focal: es una luz emitida por un foco que solo ilumina en la dirección a partir de la luz y un grado de apertura seleccionado por el usuario. En el mundo real la podríamos identificar como una linterna o un flexo. En cuanto a las anteriores esta es la más diferente

El otro elemento importante del trabajo son los materiales de los objetos, por lo que hemos conseguido una lista de coeficientes de distintos materiales, que si se quieren consultar hemos dejado el enlace se puede encontrar en la bibliográfica. La importancia de los materiales vienen de que los coeficientes de sus vectores dicen cuanto luz de cada tipo va a devolver, y tambien dice si el material es mas o menos brillante. Y aunque sabemos que hay algunos materiales que emiten luz propia, hemos decidido no incluirlos ya que lo normal son materiales que no emiten luz.

4.- Explicación del código

4.1. Transformaciones de los objetos

Con el primer código que se nos dio hemos construido nuestro código. A lo largo de la práctica hemos ido cambiando el código hasta llegar al código final que presentamos en esta entrega. De este modo podemos ver los siguientes cambios:

1. Dado que utilizamos sistemas operativos diferentes las librerías que cada uno de nosotros utiliza son diferentes, por lo que necesitamos imports condicionales, de manera que dependiendo del sistema operativo del que se quiera ejecutar se llamen a unas librerías o a otras. Estos imports los podemos encontrar en la mayoría de nuestros archivos y tienen la forma de:

```
#ifdef __linux__
    #include <GL/glut.h>
    #include <GL/gl.h>
    #include <GL/glu.h>
#elif __APPLE__
    #include <GLUT/glut.h>
    #include <OpenGL/gl.h>
    #include <OpenGL/glu.h>
#endif
```

   Este ejemplo está tomado del archivo display.c. Podemos observar que cuando estamos con sistema operativo linux solo se importará la librería <GL/glut.h>, <GL/gl.h> y <GL/glu.h>, en cambio, cuando estamos con el sistema operativo OS X necesitamos las librerías <GLUT/glut.h>, <OpenGL/glu.h> y <OpenGL/gl.h>.

   Del mismo modo que las librerías son diferentes en los sistemas operativos, el comando en la terminal para compilar los archivos también son diferentes:

   - Para linux: gcc -lGL -lGL -lGLU -lglut *.c -I./ -o test.out
   - Para OS X: gcc -w *.c -I./ -framework OpenGL -framework GLUT -o test

2. En el archivo **definitions.h** hemos creado una nueva estructura para poder tener en el objeto una pila de las matrices de transformaciones por las que ha pasado. Esta pila será una lista doblemente enlazada que tiene la siguiente estructura:

```
struct typeNode {
    float m[16];
    struct typeNode *next;
    struct typeNode *prev;
};
typedef struct typeNode typeNode;
```

   En esta nueva estructura, como podemos observar, tenemos una matriz 4x4 (16 valores), y dos apuntadores, uno al siguiente (next) typeNode y otro al anterior (prev). De este modo, podremos tener guardadas todas las matrices de transformación de los objetos. En el momento de crear el struct se carga la identidad en el atributo m y tanto prev como next serán NULL.

3. Dentro de **io.h** hemos incluido la definición del método *void SpecialKeys(int key, int x, int y);* ya que este método lo hemos incluido en io.c.

4. En **main.c**, dada la utilización de las teclas especiales, hemos incluido la definición del método que gestiona las teclas especiales *glutSpecialFunc(SpecialKeys)* y ampliado la ayuda que venia por defecto incluyendo las nuevas funciones.

5. Para poder gestionar la pila de matrices de un objeto hemos creado la clase **stack2.c**. Esta clase se encarga de gestionar las opciones de la pila; insertar al comienzo, borrar un elemento, liberar la memoria, etc.

   - *isEmpty()*: nos dice si la pila del objeto está vacía o no.

   - *length()*: devuelve la longitud de la pila.

   - *insertFirst(typeNode *v)*: este método tiene dos funciones dado que utilizamos también el ctrl + y. Por un lado libera todos los elementos tipo typeNode que quede antes de lo que nosotros consideramos el primer elemento de la pila (_selected_object->pila), y por el otro inserta en la primera posición de la pila el elemento que pasamos por parámetro.

   - *deleteFirst()*: se usa para poner en el atributo pila del objeto seleccionado el siguiente elemento que hubiera en la lista, es decir, la transformación anterior.

   - *deleteMiddle()*: se encarga de poner en el atributo pila del objeto seleccionado el elemento anterior que hubiera en la lista, es decir, rehacer la transformación que anteriormente se hubiera deshecho.

6. Hemos creado el archivo cabecera **stack2.h** y en él hemos incluido el nombre de los métodos que forma stack2.c.

7. En la clase **display.c** hemos hecho las siguientes modificaciones:

   - *Eliminación de loadIdentity()*: inicialmente al cargar cada objeto se iniciaba con una matriz identidad, para que al hacer multiplicaciones sobre ella se fueran produciendo las distintas transformaciones.

     Ahora realizamos la carga de la matriz identidad en la pila de cada objeto, en el momento de la carga del fichero, y el resto de transformaciones según se van pulsando las teclas de dirección.

   - *Carga de la matriz del objeto*: relacionado a lo anterior hemos definido la matriz que se va dibujar, la cual sera inicialmente la matriz identidad, aquella que tiene 1 en la diagonal principal y 0 en el resto de elementos. Pero ahora tenemos una pila interna con todos los cambios, que usaremos para mostrar los distintos cambios que le ocurran al objeto.

8. En la clase **io.c** hemos hecho los siguientes cambios y hemos añadido el siguiente código:

   - En primer lugar, tuvimos que modificar las funciones principales del archivo fuente que nos dieron para que en ejecución se ajustaran a lo que se supone que hacían.

– *TAB*: para movernos entre los distintos objetos que hemos cargado utilizamos la tecla TAB. Con ella pasaremos al siguiente objeto cargado. En un principio (en el código fuente) si nos encontrábamos en el último elemento de la lista de objetos cargados, no podíamos pasar al primero, por eso incluimos una condición. Esta condición consiste en que si nos encontramos en el último elemento, el siguiente elemento sería el primer elemento de la lista. El código sería el siguiente:

```
if (_first_object != 0){
     _selected_object = _selected_object->next;
     if (_selected_object == 0)_selected_object = _first_object;
}
```

– *F, f*: Se nos indica que al presionar la tecla f o F se debería cargar un fichero *.obj. Este fichero va a estar dado mediante la ruta que especifiquemos nosotros, en el caso de que la ruta sea correcta y se proceda a dibujar el objeto, el código resultante sería:

```
/*Read OK*/
case 0:
     auxiliar_object->next = _first_object;
     _first_object = auxiliar_object;
     _selected_object = _first_object;
     _selected_object->pila = (typeNode *) malloc(sizeof (typeNode));
     /* insertar la identidad en la pila */
     glGetFloatv(GL_MODELVIEW_MATRIX, _selected_object->pila->m);
     _selected_object->pila->next = NULL;
     _selected_object->pila->prev = NULL;
     printf("%s\n",KG_MSSG_FILEREAD);
break;
```

Con esto inicializamos la pila de nuestro objeto, y creamos espacio en la memoria para almacenarlo, lo cual haremos usando la función malloc, cuyo parámetro será el indicador del espacio que deseamos reservar; una vez hemos creado el espacio, procedemos a insertar en el atributo m de la pila la matriz identidad; cuando ya hemos completado este paso, procedemos a poner tanto el puntero prev como el puntero next a NULL.

– *SUPR*: Al presionar esta tecla se nos pide que el objeto seleccionado sea eliminado. En este código hemos tenido que meter la condición de que el primer objeto sea distinto de 0, es decir, que solo se pueda eliminar un objeto en el caso de que haya al menos un objeto cargado.

```
if (_first_object != 0)
     if (_selected_object == _first_object) {
        _first_object = _first_object->next;
        free(_selected_object);
        _selected_object = _first_object;
     }else{
     auxiliar_object = _first_object;
     while (auxiliar_object->next != _selected_object)
          auxiliar_object = auxiliar_object->next;
     auxiliar_object->next = _selected_object->next;
     free(_selected_object);
     _selected_object = auxiliar_object;
}
```

– *CTRL + +*: Con la utilización de estas dos teclas conjuntamente, deberemos reducir el volumen de visualización. De este modo, tomando como referencia el código que se nos daba de CTRL + - hemos logrado escribir el siguiente código, el cual hace lo que se nos pide.

```
if (glutGetModifiers() == GLUT_ACTIVE_CTRL){
    wd=(_ortho_x_max-_ortho_x_min)*KG_STEP_ZOOM;
    he=(_ortho_y_max-_ortho_y_min)*KG_STEP_ZOOM;

    midx = (_ortho_x_max+_ortho_x_min)/2;
    midy = (_ortho_y_max+_ortho_y_min)/2;
    _ortho_x_max = midx + wd/2;
    _ortho_x_min = midx - wd/2;
    _ortho_y_max = midy + he/2;
    _ortho_y_min = midy - he/2;
}
```

- Una vez modificadas las funciones principales tuvimos que hacer las transformaciones. En este apartado se nos pedía que pudiéramos elegir una de las tres transformaciones para que se pudiera proceder a hacerla, elegir si se iba a hacer una transformación global o local y, dado que este proyecto va a ser un proyecto por fases, que se pudiera también elegir transformar el objeto, la cámara o la luz, aunque, para esta primera fase, solo se nos pedían las traslaciones del objeto.

  Para poder gestionar que traslación se iba a realizar creamos la variable *estado*. Por defecto inicializada a 0, es decir, no hay ninguna transformación elegida. Tal como se nos plantea, la transformación se puede elegir:

  – *M, m*: presionando esta tecla, lograremos cambiar el estado a 1, es decir, que se aplique la traslación.

  – *B, b*: presionando esta tecla, lograremos cambiar el estado a 2, es decir, que se aplique la rotación.

  – *T, t*: presionando esta tecla, lograremos cambiar el estado a 3, es decir, que se aplique el escalado.

  Para poder gestionar si la transformación que se va a aplicar va a ser local (en el eje de coordenadas del objeto) o global (en el eje de coordenadas global) vamos a utilizar una variable llamada *glob_loc* que inicialmente va a estar a 0, es decir, que las transformaciones que se van a realizar son locales. Para poder cambiar el valor de esta variable tenemos las siguientes posibilidades:

  – *G, g*: Presionando esta tecla haremos que la variable glob_loc tome como valor 1 y que, por tanto, las transformaciones que sufra el objeto seleccionado sean globales.

  – *L, l*: Presionando esta tecla haremos que la variable glob_loc tome como valor 0 y que, por tanto, las transformaciones que sufra el objeto seleccionado sean locales.

  Para poder gestionar si las transformaciones se van a efectuar sobre el objeto, la cámara o la luz hemos creado la variable *mov*, inicialmente vale 0 y en esta primera fase del código está en desuso ya que no tenemos otro elemento a transformar más que el objeto. Aun así, hemos definido cómo va a cambiar el valor de la variable. Esta variable va a cambiar de la siguiente manera:

- *O, o*: Presionando esta tecla, haremos que la variable mov tome como valor 1, esto nos indicará que las transformaciones se han de efectuar en el objeto.

- *K, k*: Con esta tecla, la variable mov sera 2, es decir, haremos transformaciones en la cámara.

- *A, a*: Por ultimo, con a o A, asignaremos a mov 3, lo que establecerá el hacer transformaciones de luz.

- Otro de los puntos que se nos pedían era utilizar las teclas + y - para *escalar todos los ejes a la vez*. Para hacer este paso hemos utilizado las funciones que OpenGL nos ofrece y hemos acabado escribiendo el siguiente código:

  - A la hora de hacer escalado (ampliar) los tres ejes:

```
case +:
  struct typeNode *matriz = (typeNode *) malloc(sizeof (typeNode));
  glLoadMatrixf(_selected_object->pila->m);
  glScalef(2.0f,2.0f,2.0f);
  glGetFloatv(GL_MODELVIEW_MATRIX, matriz->m);
  insertFirst(matriz);
  glPopMatrix();
  break;
```

  - A la hora de hacer escalado (disminuir) los tres ejes:

```
case -:
  struct typeNode *matriz = (typeNode *) malloc(sizeof (typeNode));
  glLoadMatrixf(_selected_object->pila->m);
  glScalef(0.5f,0.5f,0.5f);
  glGetFloatv(GL_MODELVIEW_MATRIX, matriz->m);
  insertFirst(matriz);
  glPopMatrix();
  break;
```

  Podemos observar que ambos códigos son iguales salvo en la asignación que se hace en la función glScalef(). Inicialmente crearemos el espacio para la estructura typeNode, esta estructura será introducida en la pila de matrices de nuestro objeto. Una vez hemos creado el espacio, procedemos a cargar en la modelview la matriz a la que apunta nuestra pila. Haremos el escalado sobre esa matriz, y esa nueva matriz la vamos a cargar en la estructura para la que hemos guardado memoria. Una vez finalizado este paso, procederemos a insertar esta nueva matriz en nuestra pila y quitaremos la matriz que hemos puesto en el modeview.

  Para el caso del -, el procedimiento será el mismo solo que el escalado será a la mitad en vez de al doble de las dimensiones del objeto.

- Otro de los puntos a tratar en esta primera fase del proyecto es poder hacer *transformaciones* (escalado, rotación y traslación) *globales y locales*.

  En el caso de las traslaciones locales, haremos referencia al código insertado para el + y el -.

  A la hora de hacer un escalado, la estructura que vemos en esos puntos es la que se va a llevar a cabo.

  A la hora de hacer una transformación, se seguirán todos los puntos anteriormente citados con una leve modificación. En la línea 4 hemos definido glScalef() para hacer el escalado, ahora nos encontramos ante una traslación por lo que sustituiremos esa línea de código por: glTranslatef(a.0f, b.0f, c.0f). En el caso de que queramos trasladar sobre el eje x (nosotros hemos decidido trasladar de unidad en unidad, sin utilizar los decimales) el valor de 'a' sería 1/-1 y el valor de b y c sería 0, es decir, tomará por valor 1 aquel eje sobre el que queramos trasladar.

  A la hora de hacer una rotación, seguirán todos los puntos citados en el subapartado anterior con una modificación. En vez de utilizar glScalef() utilizaremos glRotatef(a.0f,b.0f, c.0f, d.0f). En esta función el valor de a será el número de grados que se quiera rotar el elemento y, al igual que en el caso de la traslación, tomarán como valor 1/-1 (es el estándar que nosotros hemos utilizado) aquellos ejes sobre los que se quiera hacer el giro siendo 'b' el referente al eje X, 'c' el referente al eje Y y 'd' el referente al eje Z.

  En el escalado existen tres parámetros que se le pasan a la función, todos ellos tendrán que ser 1.0f exceptuando aquel que se quiera ampliar o reducir que en nuestro caso es 1.5f y 0.75f.

  En el caso de las transformaciones globales el código cambia un poco y en el caso del escalado será el siguiente:

```
struct typeNode *matriz = (typeNode *) malloc(sizeof (typeNode));
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glScalef(1.0f,1.5f,1.0f);
glMultMatrixf(_selected_object->pila->m);
glGetFloatv(GL_MODELVIEW_MATRIX, matriz->m);
insertFirst(matriz);
glutPostRedisplay();
```

  Como podemos observar, hay ciertos puntos que se repiten pero también hay variaciones en el código. Uno de los cambios más significativos es que la multiplicación de las matrices se hace por el lado contrario, de este modo el código cambia de la siguiente manera:

  Inicialmente creamos el espacio para la estructura typeNode que contiene la matriz que vamos a generar. Cargamos la identidad en la ModelviewMatriz y multiplicamos la identidad por la matriz que hacer la traslación (en nuestro caso el escalado sobre el eje Y). Procederemos a multiplicar esa matriz por la matriz de nuestra pila. El resultado obtenido lo guardaremos en la matriz de la estructura typeNode para la que hemos reservado memoria. Una vez guardada, la añadiremos a nuestra pila.

- Por ultimo tenemos el *deshacer* y *rehacer* de las *acciones* realizadas por el usuario.

  Inicialmente estas acciones se suponía que iban a ser efectuadas mediante CTRL Z (deshacer) y CTRL Y (rehacer). Dados los proglemas surgidos en el laboratorio para deshacer tendremos que presionar las teclas 'u' o 'U' y, para rehacer la teclas 'r' o 'R'.

  Para efectuar estas acciones, haremos uso de las matrices alojadas en los nodos de la pila del objeto seleccionado, seleccionando el nodo siguiente en el caso de querer deshacer un cambio, o el nodo anterior en el caso de rehacer.
  De esta manera nos olvidamos de tener que usar multiplicaciones o revertir los cambios realizados, simplemente cogemos la forma que tenia previamente el objeto y lo cargamos.
  Al final se llamara a la función *glutPostRedisplay()* que se encargara de dibujar el estado actual de la matriz.

```
case 'U':// Deshacer
case 'u':
     if (_selected_object)
          if (_selected_object->pila->next != NULL)
               _selected_object->pila = _selected_object->pila->next;
break;

case 'R':// Rehacer
case 'r':
     if (_selected_object)
          if (_selected_object->pila->prev != NULL)
               _selected_object->pila = _selected_object->pila->prev;
break;
```

4.2. Transformaciones de la cámara

Basándonos en el código de la anterior entrega hemos realizado algún cambio en las estructuras y también hemos creado nuevo código para llevar a cabo todos los puntos que se nos establecían a cumplir.

Los cambios producidos sobre el código anterior son los siguientes:

1. En la clase **definitions.h** hemos hecho los siguientes cambios y hemos añadido el siguiente código:

```
struct object3d {
    GLint num_vertices;
    vertex *vertex_table;
    GLint num_faces;
    face *face_table;
    point3 min;
    point3 max;
    struct object3d *next;
    struct camera *camara;
    struct typeNode *pila;
    float mat_rot_cam[16] ;
};

struct camera {
    int pertenece_a_objeto;
    int modo;
    GLdouble left, right, bottom, top, nearVal, farVal;
    float miCamara[16];
    float m[16];

    struct camera *next;
    struct camera *prev;
};
```

   - Como podemos observar en la línea 9 del código anterior, hemos introducido en el objeto una cámara para así poder gestionar la opción de que los objetos también pudieran ser cámaras. Por otro lado hemos creado una matriz llamada mat_rot_cam para poder hacer que la cámara del objeto sea independiente del mismo, es decir, que se pueda rotar la cámara del objeto aunque el objeto no se rote.

   - Por otro lado hemos creado la estructura camera. Tal como se puede observar, esta estructura también posee diferentes valores dentro de ella.
     - Pertenece_a_objeto: Este valor tomará 0 en el caso de ser una cámara libre y 1 en el caso de que la cámara pertenezca a un objeto. De este modo, cada vez que estemos utilizando una cámara sabremos si pertenece a un objeto o no.
     - Modo: Gracias al valor de esta variable sabremos si hay que hacer una proyección en perspectiva (toma valor 1) o si queremos hacer una proyección en paralelo (toma el valor 0).
     - En la línea 17 vemos que hay diferentes variables declaradas, estas serán las que formarán parte a la hora de realizar la proyección. Son los valores que tomarán los campos que hay que usar tanto en glOrtho() como en glFrustum().

- Por otro lado vemos que hemos creado dos matrices en esta nueva estructura. Cada matriz tiene una función determinada:
  - miCamara: Esta es la matriz que vamos a usar como referencia para saber dónde se encuentra la cámara, a qué punto está mirando y cual es su verticalidad. De este modo en los campos 0, 1 y 2 de la matriz guardamos el punto en el que se encuentra la cámara; en los campos 4, 5 y 6 el punto al que mira; y en los campos 8, 9 y 10 su verticalidad.

  - m: Esta va a ser nuestra matriz de vista, es decir, la matriz que se genera al llamar a a la función gluLookAt(). Con esta matriz obtenemos los vectores x, y, z de la cámara.
- Por último podemos observar que hay dos punteros en la cámara. Esto nos sirve para gestionar la cola de cámaras que tenemos creada. Gracias a estos dos parámetros tendremos referencia a la cámara anterior y posterior de todas las cámaras.

2. En la clase **main.c** hemos realizado diferentes cambios para así poder gestionar una cámara inicialmente:
   - En primera instancia hemos creado dos variables, first_camara y la segunda camara. Tal como se hacía en los objetos, la variable first_camara será la lista de las cámaras que se han creado y camara será la cámara actual en la que estemos.

   - Por otro lado, dado que creamos cámaras debemos inicializarlas por lo que en el método de initialization() hacemos malloc de la estructura de las cámaras y las inicializamos.

3. En la clase **io.c** es donde hemos aplicado la mayoría de los cambios de esta práctica ya que es donde se gestionan los controles del teclado. Podemos apreciar los siguientes cambios:
   - En primer lugar tenemos que decir que hemos definido dos variables externas más. Estas variables son la primera cámara (first_camara) y la cámara actual (camara).

   - En segundo lugar tenemos que comentar que hemos añadido más variables como son modo_c cuyo valor será 0 si la cámara está en modo vuelo y 1 si está en modo análisis. Por otro lado hemos creado la variable obj_cam para poder saber si la cámara con la que visualizas la escena pertenece a un objeto o no. Esta variable tomará como valor 0 en el caso de que no pertenezca al objeto y 1 en caso contrario.
   - Dentro de la función *keyboard* hemos hecho diferentes cambios. Los cambios realizados son los siguientes:
     - *F, f*: A la hora de cargar un objeto, dado que a este le hemos añadido en su estructura una cámara, alojamos memoria para la estructura de la cámara y la inicializamos. Una vez inicializada, la añadimos a la cola de cámaras que tenemos. Esto se hace mediante las siguientes funciones:
       * inicializarCamara(): este método es el encargado de inicializar la cámara según la matriz del objeto que cargamos. Le asignamos por defecto los valores que formarán parte de la proyección que creará la cámara y pondremos a NULL los punteros de esa cámara.

       * insertarCamara(): este método recorre toda la lista de cámaras para insertar al final de la misma la nueva cámara creada.

– *TAB*: Dentro de esta opción nos aseguramos de que si hemos hecho que se visualice lo que el objeto ve, una vez pasas al siguiente objeto, el campo de visión será el que este objeto nos ofrezca. Por otro lado, si la cámara con la que visualizamos está en modo análisis, al cambiar al siguiente objeto, la cámara estará en modo análisis sobre el nuevo objeto seleccionado.

– *SUPR*: A la hora de suprimir, el único cambio que hemos hecho es el de asegurarnos que al eliminar cualquier objeto se actualicen los punteros de la cámara anterior y posterior a la misma.

– *+, -*: Dado que se nos pedía que se pudiera hacer zoom, en estos dos casos hemos añadido una condición, si nos encontramos haciendo transformaciones del objeto se realizarán las accione que se hacían en la práctica anterior en cambio, si estamos realizando transformaciones en la cámara, se aumentará o disminuirá el campo de visión, es decir, se hará zoom.

```
case '+':
     camara->left *= 0.5;
     camara->right *= 0.5;
     camara->top *= 0.5;
     camara->bottom *= 0.5;
     camara->nearVal /= 0.5;
     break;

case '-':
     camara->left /= 0.5;
    camara->right /= 0.5;
    camara->top /= 0.5;
    camara->bottom /= 0.5;
    camara->nearVal *= 0.5;
     break;
```

– *P, p*: Tal como se nos pedía en esta práctica hemos creado la opción de que se pueda pasar de una proyección en paralelo a una proyección en perspectiva. Presionando la tecla P o la tecla p conseguiremos modificar el valor modo de la cámara.

– *T, t*: La funcionalidad de esta tecla será la misma que la de los objetos solo que en los objetos se interpreta que es escalado de objetos y en modo cámara se interpreta como cambio de volumen.

En este punto hemos tomado una decisión. Al inicializar la cámara, los valores top, bottom, right, left y near del campo de visión toman el valor 0.1. Dado que al ampliar o reducir el volumen en X, Y o Z y la variación de los valores es de 0.1, los objetos se podían invertir en el caso de reducir más de 0,1. De este modo hemos decidido que cuando los valores top, bottom, right, left y near sean 0,1 no se podrá disminuir el campo de visión en ninguno de los casos.

– *G, g*: Esta tecla ahora tiene dos funciones, si nos encontramos en transformaciones a los objetos su función no cambia; en cambio, si nos encontramos haciendo transformaciones a la cámara, se cambiarán los valores del punto de mira de la cámara por la posición en la que se encuentra el objeto, es decir, pondremos la cámara en modo análisis.

- *L, l*: Esta tecla ahora tiene dos funciones, si nos encontramos en transformaciones a los objetos su función no cambia; en cambio, si nos encontramos haciendo transformaciones a la cámara, volveremos a dejar que la cámara esté en modo vuelo.

- *C*: Al presionar esta tecla, tal como se nos ha pedido en la práctica, visualizaremos el entorno a través de la cámara del objeto seleccionado. Para poder llevar a cabo esta acción primero procederemos a actualizar la cámara del objeto según los valores que toma la matriz m del mismo para luego proceder a asignarle a la cámara de visualización la cámara del objeto seleccionado. De este modo, podemos ver lo que el objeto está visualizando.

- *U, u, R, r*: Estos eran los métodos de ctrl z y ctrl y que hicimos en la entrega anterior. En esta entrega tenemos que los objetos pueden ser las cámaras y dado que podemos deshacer las transformaciones que generamos en el objeto hemos cambiado estas dos opciones de modo que si la cámara con la que estamos visualizando el entorno es la del objeto, ésta también sufra el cambio que le hacemos al objeto.

- En el método specialKeys también hemos hecho ciertos cambios de modo que se pueda gestionar la cámara:
  - En primer lugar, en el código de la práctica anterior hemos hecho ciertas modificaciones de manera que si la cámara de visualización es la del objeto, ésta se transforme según las transformaciones del objeto.

  - A la hora de aplicar transformaciones a la cámara tenemos que tener en cuenta si la cámara está en modo vuelo o en modo análisis:

    En el caso de que esté en modo vuelo, tendremos que tener en cuenta si el objeto es la cámara o no. En el caso de que el objeto sea la cámara, dado que le hemos dado cierta independencia a la cámara, las transformaciones a realizar serán únicamente rotaciones y éstas se gestionarán mediante la matriz de rotación de la cámara que tiene el objeto. En caso contrario, se cargará la identidad en la modelview, haremos que el origen esté en el punto donde se encuentra la cámara, haremos las transformaciones pertinentes y devolveremos el origen a su punto inicial.

    En el caso de que esté en modo análisis la cámara solo podrá hacer rotaciones manteniendo el punto al que mira. Para llevar a cabo este proceso creamos una matriz auxiliar que guarde la referencia al punto donde el objeto se encuentra, haremos que el punto al que la cámara mira sea el origen, realizamos los cambios pertinentes a la cámara y posteriormente devolvemos el origen a su situación inicial. Si el objeto fuera la cámara, la cámara no podría realizar transformación alguna en este modo.

    Tras cualquiera de los dos pasos, multiplicamos la matriz que guarda las posiciones que tiene la cámara por la matriz resultante en la modelview y esa será nuestra nueva matriz de posiciones de la cámara.

4. En el fichero **display.c** hemos tenido que reflejar ciertos cambios para que se puedan aplicar los cambios realizados para esta segunda entrega. Estos cambios son los siguientes:

   - En primer, para hacer la proyección, deberemos tener en cuenta el modo de la cámara. En el caso de que sea 0 (nos encontramos en proyección en paralelo) se llamará a la función glOrtho() con los parámetros definidos dentro de la cámara. En el caso de que sea 1 (nos encontramos

en proyección en perspectiva) se llamará a la función glFrustum() con los parámetros definidos dentro de la cámara.

- Una vez hecha la proyección deberemos tener en cuenta, si estamos visualizando desde la cámara del objeto seleccionado. De ser así en la matriz del modelview cargaríamos la matriz de rotación de la cámara que tiene el objeto, en el caso contrario, cargamos la identidad.

- Tras realizar ambas acciones procedemos a llamar a la función gluLookAt() con los parámetros de la matriz de la cámara (miCamara) y así obtener la nueva matriz de vista.

4.3. Transformaciones de las luces

Basándonos en el código de la anterior entrega hemos tenido que realizar varios cambios para implementar la iluminacion, principalmente en **display.c** para el dibujado de los materiales e **io.c** para el control de las luces y sus transformaciones.

1. En **definitions.h** hemos creado dos nuevas estructuras para poder guardar la información de las luces y de los materiales de los objetos. Las estructuras son las siguientes:

   - La estructura para las luces es la siguiente:

```
struct light {
    int estado; // 0 - no creado / 1 - apagado / 2 - encendido

    struct lightInfo *info;
};

struct lightInfo {
    int tipo; // 0 - sol / 1 - bombilla / 2 - foco
    GLenum numero;

    GLfloat diffuseLight[4];
    GLfloat ambientLight[4];
    GLfloat specularLight[4];
    GLfloat positionLight[4];
    GLfloat spotCutoff;
    GLfloat spotDirection[3];
    GLfloat spotExponent;
    GLfloat spotAttenuation[3];
};
```

   Como podemos comprobar en la imagen, las luces están definidas mediante dos estructuras. La estructura light contiene el estado de la luz, siendo 0 cuando no esté creada, 1 cuando esté creada pero se encuentre apagada y 2 cuando ademas de estar creada esté encendida. También contiene un puntero a una estructura tipo lightInfo donde guardamos la información correspondiente a cada luz.

   Desde el primer momento todas las luces tienen reservado un espacio en la memoria pero, tal como se nos pide solo las primeras 3 luces tienen que estar creadas (un sol, una bombilla y un foco). Gracias a utilizar la estructura que enseñamos arriba, solo en el momento de inicializar una luz reservaremos la memoria para su contenido.

   De este modo, al crear una luz tendremos que reservar memoria para poder guardar su información, que como ya hemos comentado se guardará en una estructura llamada lightInfo.

   Dentro de esta estructura podemos observar diferentes campos:

(a) tipo: En este campo guardamos la información referente al tipo de luz que queremos definir. Tal como se muestra en el código anterior, se designará por 0 a la luz de tipo sol; 1 a la luz de tipo bombilla; y 2 a la luz de tipo foco.

(b) numero: Este campo es el indicador de a que luz le pertenece, es decir si la luz es GL_LIGHT0, GL_LIGHT1, ...

(c) diffuseLight: En este campo guardamos el vector de valores de intensidad difusa de la luz. La explicación de este y los dos siguientes campos están explicados en el capitulo de Solución elegida para las luces.

(d) ambientLight: En este campo guardamos el vector de valores de intensidad ambiental de la luz.

(e) specularLight: En este campo guardamos el vector de valores de intensidad especular de la luz.

(f) positionLight: En este campo guardamos el vector de valores de posición de la luz. En el caso de que el último campo del vector sea 0 estaremos determinando que la luz sera de tipo sol, es decir, no tendrá posición, sino que usaremos este vector de posición como dirección, dándonos los tres primeros valores la dirección de la luz. En el caso de que el último valor del vector sea 1 nos estará diciendo que es una luz de tipo bombilla, es decir tendrá una posición desde la que alumbrara en todas direcciones.

(g) spotCutoff: Este es un campo que solo afecta a los focos. En él guardamos el ángulo de apertura del foco. Podrá tomar valores de 0 a 90, ya que toma el valor para un lado del angulo y luego lo duplica al mostrar el foco. También se le puede dar el valor especial 180 que da lugar a una distribución uniforme de la luz.

(h) spotDirection: Este es un campo que solo afecta a los focos. En él guardamos el vector de valores de dirección del foco. Hay que tener en cuenta que el vector indica la dirección del foco y no a donde mira, es decir, si le decimos (0, -1, 0) no le estamos indicando que mire a ese punto, sino que le estamos indicando que mire hacia el eje Y negativo, es decir, hacia abajo.

(i) spotExponent: Este es un campo que solo afecta a los focos. En él guardamos el exponente, es decir la intensidad de la distribución de la luz que va de 0 a 128, si le damos un valor alto estaremos enfocando mucho la luz, sin tener en cuenta el angulo mencionado en el cutOff, y puede parecer que nuestra luz no funcione, así que suele ser bueno tener un valor bajo salvo que se sepa lo que se esta haciendo.

(j) spotAttenuation: Este es un campo que solo afecta a los focos. En él guardamos el vector de valores correspondiente a la atenuación. Es un vector de tres campos, en el primero guardaremos el valor correspondiente a la atenuación constante; en el segundo a la atenuación lineal; y en el tercero a la atenuación cuadrática. Al igual que el exponente es mejor dejar la atenuación constante a 1 y el resto a 0 para que no se modifique demasiado la intensidad de luz.

- La estructura para los materiales es la siguiente:

```
struct material {
    GLfloat matAmb[4];
    GLfloat matDiff[4];
    GLfloat matSpec[4];
    GLfloat matShininess;
};
```

Dentro de esta estructura podemos observar los siguientes campos:

(a) matAmb: En este campo guardamos el vector de valores del color ambiente del material.

(b) matDiff: En este campo guardamos el vector de valores del color difuso del material.

(c) matSpec: En este campo guardamos el vector de valores del color especular del material.

(d) matShininess: En este campo guardamos el valor del exponente especular del material.

- Por otro lado, hemos cambiado también la estructura de las caras y vértices, añadiendo las normales de cada una. Así que, las estructuras tras los cambios tienen el siguiente aspecto:

```
typedef struct {
    point3 coord;           /* coordinates,x, y, z */
    GLint num_faces;        /* number of faces that share this vertex */
    vector3 normal;
} vertex;

typedef struct {
    GLint num_vertices;     /* number of vertices in the face */
    GLint *vertex_table;    /* table with the index of each vertex */
    vector3 normal;
} face;
```

- Además hemos cambiado la estructura de los objetos, ya que al crear materiales para los objetos, cada objeto guardará la información correspondiente a su material asignado mediante el menú creado para ello. De este modo, la nueva estructura de los objetos es:

```
struct object3d {
    GLint num_vertices;         /* number of vertices in the object*/
    vertex *vertex_table;       /* table of vertices */
    GLint num_faces;            /* number of faces in the object */
    face *face_table;           /* table of faces */
    point3 min;                 /* coordinates' lower bounds */
    point3 max;                 /* coordinates' bigger bounds */
    struct object3d *next;      /* next element in the pile of objects */
    struct camera *camara;
    struct typeNode *pila;      /* pila de transformaciones */
    float mat_rot_cam[16];
    struct material *material;
};
```

2. En **main.c** hemos hecho los siguientes cambios para así poder utilizar luces en nuestro programa:

   - En primer lugar decir que hemos creado las siguientes variables globales:
     - arrayLuces: en este vector guardaremos las ocho luces que dejaremos utilizar en nuestro programa, numeradas de 0 a 7 (GL_LIGHT0 - GL_LIGHT7).
     - luzSeleccionada: este es un puntero a la luz que está seleccionada en ese instante. Inicialmente va a ser la luz 0 ya que es el sol, la única luz que encendemos al principio.
     - mat: en este vector vamos a guardar los 4 materiales que inicialmente creamos.
     - flat_or_smooth: esta variable la vamos a utilizar como flag siendo su valor 0 si el usuario decide que su shading sea flat y 1 si el usuario decide que sea smooth.
     - foco_debug: este es un flag que nos sirve para saber si tenemos el foco en modo debug o no.
   - glutInitDisplayMode(GLUT_RGB — GLUT_DEPTH). Hemos añadido el GLUT_DEPTH para que nuestro programa acepte el z-buffer, y por tanto entienda de profundidad.
   - Para poder utilizar el z-buffer hacemos uso de la instrucción glEnable(GL_DEPTH_TEST).
   - Definimos también una luz ambiental que afecta a todos los objetos definida con un vector de 4 valores llamado global_ambient. Con ella haremos que siempre haya un poco de luz, incluso cuando no haya ninguna luz encendida, al igual que con las luces de escena, esta la colocamos con glLightMode(GL_LIGHT_MODE_AMBIENT, valores).
   - Tal como se nos ha pedido, debemos tener la opción de elegir entre el sombreado flat o smooth. Nosotros por defecto lo inicializaremos con flat. Esto es importante porque si esta en modo flat las normales se calculan por polígono, mientras que si esta en modo smooth las normales se calculan por vértices, consiguiendo de esta manera una iluminación mas suave y uniforme.
   - Hemos creado un menú (apartado opcional) que se inicializa mediante la llamada al método initMenu(). En este método creamos la estructura del menú, definiendo submenus y botones, y a cada uno de ellos le asignamos un valor que gestionaremos en el método rightClickMenu(int value) que solo tiene un switch pero es el mas importante del menú, ya que dice que pasa cuando se pulsa una opción del menú, aquí nos hemos valido del trabajo que hemos ido haciendo en los trabajos anteriores, modularizando lo que teníamos antes, permitiendo así llamadas mas sencillas, y de paso consiguiendo un código mas limpio y entendible.
   - Por otro lado, en el método initializationLights() inicializamos las tres luces (sol, bombilla y foco) que se nos piden obligatoriamente, sabiendo que lo que diferencia al sol del resto de luces es el ultimo valor del vector de posición, ya que al ponerlo a 0 no tiene una posición exacta y por lo tanto es un sol, y si pusiéramos 1 seria una bombilla, y el foco que se diferencia por tener una dirección del foco y un angulo de apertura. Por defecto hemos dejado todas las demás luces en no creadas y solo damos la opción de crear en total 8 luces.
   - Aquí también tenemos un método para declarar como son los materiales, mediante el método crearMateriales() definimos unas propiedades de materiales para luego usarlos en el resto del programa, y que el usuario pueda seleccionar de los que nosotros hemos definido.

3. En **io.c** vamos a observar dos tipos de cambio.

   Como ya hemos comentado anteriormente, hemos modularizado algo el código y esto lo podemos observar en la creación de los siguientes funciones:

   - opcionSuprimir: esta función contiene el método que antes estaba en el caso de suprimir. Al haberlo sacado a un función además de modularizar el código hemos hecho que sea posible llamarlo desde nuestro nuevo menú.

   Por otro lado, como podemos observar en el código se han creado nuevas funciones y se ha ampliado el número de opciones que se podían realizar.

   Las nuevas funciones que podemos observar son:

   - calcularNormales: Por defecto sabemos que no todos los objetos tienen las normales calculadas, por tanto, antes de dibujar un objeto nos aseguramos de calcular sus normales para que así el objeto se ilumine correctamente.

```
int ind1, ind2, ind3, i, j;
double x1, y1, z1, x2, y2, z2;

for (i = 0; i < optr->num_vertices; i++) {
  //poner a 0 el vector normal del vertice 0 0 0
  optr->vertex_table->normal.x = 0;
  optr->vertex_table->normal.y = 0;
  optr->vertex_table->normal.z = 0;
}
for (i = 0; i < optr->num_faces; i++) {
  ind1 = optr->face_table[i].vertex_table[0];
  ind2 = optr->face_table[i].vertex_table[1];
  ind3 = optr->face_table[i].vertex_table[2];

  x1 = optr->vertex_table[ind2].coord.x - optr->vertex_table[ind1].coord.x;
  y1 = optr->vertex_table[ind2].coord.y - optr->vertex_table[ind1].coord.y;
  z1 = optr->vertex_table[ind2].coord.z - optr->vertex_table[ind1].coord.z;

  x2 = optr->vertex_table[ind3].coord.x - optr->vertex_table[ind1].coord.x;
  y2 = optr->vertex_table[ind3].coord.y - optr->vertex_table[ind1].coord.y;
  z2 = optr->vertex_table[ind3].coord.z - optr->vertex_table[ind1].coord.z;

  optr->face_table[i].normal.x = (y1 * z2) - (z1 * y2);
  optr->face_table[i].normal.y = (z1 * x2) - (x1 * z2);
  optr->face_table[i].normal.z = (x1 * y2) - (y1 * x2);

  for (j = 0; j < optr->face_table[i].num_vertices; j++) {
    ind1 = optr->face_table[i].vertex_table[j];

    optr->vertex_table[ind1].normal.x += optr->face_table[i].normal.x;
    optr->vertex_table[ind1].normal.y += optr->face_table[i].normal.y;
    optr->vertex_table[ind1].normal.z += optr->face_table[i].normal.z;
  }
}
```

- Por otro lado hemos creado la función actualizarFoco. La tercera luz creada (en nuestro caso GL_LIGHT2), Es un foco que pertenece al objeto seleccionado, por tanto, cada vez que se haga alguna transformación al objeto seleccionado (ya sean rotaciones, traslaciones, cambiar el objeto seleccionado o suprimir el objeto seleccionado) se actualizará el foco.

- Dentro de nuestro código también podemos observar la función copiarContenidoArray. Este método recibe como parámetro la matriz de origen, es decir, de donde queremos copiar los datos y la matriz destino, donde queremos copiarlos. Gracias a este método nos evitamos tener que ir poniendo uno a uno todos los campos de las matrices sobre las que queremos cambiar los valores.

- La función seleccionarMaterial se encarga de, dado un número entero, seleccionar el material que corresponde a ese índice en el vector de materiales y asignárselo al objeto seleccionado.

- En los puntos a realizar en esta entrega se nos pedía poder habilitar y deshabilitar la iluminación en nuestro escenario, de este modo, con la función onOffIluminacion nos encargamos de comprobar si está habilitada la iluminación. En el caso de que esté habilitada, la deshabilitamos y, si no está habilitada, la habilitaremos.

- En esta entrega, al igual que en las anteriores, tenemos también apartados de cosas extra que hemos hecho. Inicialmente entendimos que lo explicado en el punto anterior era simplemente que si había alguna luz encendida se apagarían todas las luces y, en caso contrario, se encendieran todas las que estuvieran creadas. De este modo realizamos los siguientes métodos para gestionar lo dicho:
  - existeLuzEncendida: este método recorre todo el vector de luces que hemos creado. Al encontrar una luz encendida devolverá 1, identificación que hemos utilizado para referirnos a que hay una luz encendida. En el caso de que termine de recorrer el vector y no haya encontrado ninguna luz encendida, devolverá el valor 0 que nos indica que no hay ninguna luz encendida.

  - disableLights: si la función anterior nos confirma que hay alguna luz encendida, este método se encargará de recorrer todo el vector de luces e ir apagando aquellas que están encendidas.

  - enableLights: si la función anterior nos confirma que no hay alguna luz encendida, este método se encargará de recorrer todo el vector de luces e ir encendiendo aquellas luces que están apagadas.

- La función rotarFoco recibe por parámetro los dos índices del spotDirecction de un foco que hay que cambiar, dos valores multiplicadores que nos indicarán qué sentido toma la rotación y el coseno del ángulo que queremos rotar. Dada la teoría estudiada en clase, aplicamos las multiplicaciones y sumas correspondientes para guardar en esos índices el nuevo valor tras la rotación.

- La función rotarLuz recibe por parámetro los dos índices de la positionLight de una luz que hay que cambiar, dos valores multiplicadores que nos indicarán qué sentido toma la rotación y el coseno del ángulo que queremos rotar. Dada la teoría estudiada en clase, aplicamos las multiplicaciones y sumas correspondientes para guardar en esos índices el nuevo valor tras la rotación.

- La función nuevaLuz se encarga de gestionar la creación de una nueva luz. Al intentar encender una luz, puede que está luz ni siquiera esté creada, de ser así se llamará a esta función que dado

un índice pedirá al usuario que introduzca por pantalla los valores que sean oportunos para el tipo de luz que desee crear. Suponemos que el usuario está entendido en el tema y que, por tanto, sabe los valores que puede y que no puede darle a una luz.

- La función cambiarEstadoLuz, dado un índice, se encarga de comprobar si la luz está encendida o no. En el caso de estar encendida, apagará la luz y, en caso de estar apagada, la encenderá. Si no se da ninguno de los anteriores casos querrá decir que la luz aun no está creada por lo que se llamará a la función nuevaLuz con el índice que se le ha dado a esta función para crear una nueva luz.

A parte de los métodos nuevos creados, tenemos las nuevas funcionalidades que se nos han pedido. Dichas funcionalidades son las siguientes:

- Inicialmente tenemos que comentar que hacemos uso de nuevas variables globales, estas son las nuevas variables declaradas en el fichero main.c. También hemos creado la variable transf_luz cuyo valor será 0 si queremos hacer transformaciones locales a la luz y 1 si queremos hacer transformaciones globales a la luz.

- Por otro lado hemos ampliado el método keyboard de la siguiente manera:
  - Dentro de la opción f, F (carga de fichero) hemos calculado las normales del objeto e inicializado todos los objetos con un material nulo para que no se produzcan errores inesperados.

  - Dentro de la opción del tabulador hemos añadido una llamada a la función actualizarFoco para que, cada vez que se cambie entre los objetos seleccionados el foco asociado al objeto se actualice según la información del nuevo objeto seleccionado.

  - Como ya hemos ido comentando a lo largo del informe, la opción de suprimir la hemos sacado a una función auxiliar y, en esta función hemos cambiado que al eliminar el objeto seleccionado, se actualice también el foco según el nuevo objeto que esté seleccionado.

  - Dentro de la opción '-' hemos añadido un nuevo caso, en el caso de que estemos en modo luz, y la luz seleccionada sea un foco, haremos que la apertura del foco disminuya.

```
if (mov == 3){
     if (luzSeleccionada->info->tipo == 2){ //Es un foco
          if((luzSeleccionada->info->spotCutoff -5) > 0)
          //Comprobar que nunca sea un valor negativo.
               luzSeleccionada->info->spotCutoff -= 5;
     }
}
```

  - Dentro de la opción '+' hemos añadido un nuevo caso, en el caso de que estemos en modo luz, y la luz seleccionada sea un foco, haremos que la apertura del foco aumente.

```
if (mov == 3){
     if (luzSeleccionada->info->tipo == 2){ //Es un foco
          if((luzSeleccionada->info->spotCutoff + 5 <= 90)
               luzSeleccionada->info->spotCutoff += 5;
     }
}
```

- Hemos creado como apartado extra el uso de W, w. al presionar esta tecla podremos elegir por teclado el material que le queremos asociar al objeto y, mediante el método seleccionarMaterial se lo asociaremos al objeto.

- Dentro de la opción G, g, en el caso de que estemos en modo luz, el valor de transf_luz será 1 indicando que las transformaciones que se hagan sobre la luz van a ser globales.

- Dentro de la opción L, l, en el caso de que estemos en modo luz, el valor de transf_luz será 0 indicando que las transformaciones que se hagan sobre la luz van a ser locales.

- En caso de presionar la tecla 0 se procederá a cambiar el tipo de luz que se tiene. En primer lugar se comprobará que la luz seleccionada haya sido creada en algún punto ya que, de no estar creada, no se le podrá cambiar el tipo. A continuación se comprueba que la luz seleccionada no sea ninguna de las tres que se dan al inicio ya que hemos decidido que esas no se pueden tocar. Si sabemos que no es ninguna de esas la que está seleccionada, procederemos a dejar al usuario que cambie el tipo de luz. Dependiendo del tipo de luz que el usuario seleccione se procederá a reaccionar de una manera determinad.

  * En el caso de que decida cambiar a un tipo bombilla con los datos que se tenían anteriormente se llevará a cabo el cambio de tipo de luz. Pondremos el indicador de tipo de luz como 1, indicando que es una bombilla y cambiaremos el último valor del vector posición a 1.0.

  * En el caso de que decida cambiar a un tipo sol con los datos que se tenían anteriormente se llevará a cabo el cambio de tipo de luz. Pondremos el indicador de tipo de luz como 0, indicando que es un sol y cambiaremos el último valor del vector posición a 0.0.

  * En el caso de que decida cambiar a un tipo foco con los datos que se tenían anteriormente se llevará a cabo el cambio de tipo de luz. Pondremos el indicador de tipo de luz como 2, indicando que es un foco y cambiaremos el último valor del vector posición a 1.0. Además procederemos a pedirle al usuario que introduzca el valor de apertura, el exponente y la dirección. De nuevo suponemos que el usuario conoce cómo se manejan las luces y que, por tanto, no necesitamos comprobar los datos que nos introduce.

- Los casos de las teclas del 1-8 serán para seleccionar una luz diferente. Cada vez que queramos seleccionar una luz, deberemos indicar antes presionando dichas teclas que luz queremos seleccionar.

- Dentro del caso de deshacer hemos insertado la llamada a la función actualizarFoco para que, una vez se deshaga un movimiento, el foco asociado al objeto también se cambie.

- Dentro del caso de rehacer hemos insertado la llamada a la función actualizarFoco para que, una vez se rehaga un movimiento, el foco asociado al objeto también se cambie.

- Dentro del método SpecialKeys también hemos realizado ciertos cambios para así procesar las transformaciones de la luz. Las transformaciones son las siguientes:

  - Para poder llevar a cabo esta parte primero debíamos saber cuales eran las transformaciones que debíamos realizar. En primer lugar descartamos la idea de poder hacer cualquier tipo de

escalado pero, por dejar la opción ya que el valor del modo puede cambiar, haremos la misma funcionalidad que en las opciones anteriormente dichas '+' y '-'.

Dedujimos que las traslaciones había que hacerlas y que las rotaciones también pero que no todos los tipos de luz tendrían que tener las mismas opciones. De este modo hemos separado las transformaciones de la siguiente manera:

* Las bombillas y los soles pueden hacer rotaciones en global, pero no en local, no tendría sentido que un sol rotara localmente ya que no cambiaría el modo en que esa rotación ilumina a los objetos. En el caso de que el usuario quiera hacer una rotación local de un sol o de una bombilla supondremos que se ha confundido y que en realidad quiere hacer una rotación global. En cuanto al foco, este puede rotar tanto en global como en local ya que en local la dirección de su foco cambiaría y en global se cambiaría tanto la dirección de su foco como la de su posición.

* Por otro lado, en cuanto a la traslación, no es posible aplicarla a las luces de tipo sol una traslación ya que las traslaciones se realizan sobre puntos y el sol es una luz que solo posee dirección.

* Dado que las bombillas tienen posición y el foco es también una luz posicional, pueden trasladarse. La traslación en local y en global será la misma, por lo que hemos supuesto que si el usuario quiere cambiar entre local y global en cuanto a la traslación se aplicaría el mismo proceso. No hay diferencias entre ellas. Tanto como si está en modo local como en modo global se realizará la misma traslación.

De este modo, se seguiría el siguiente proceso para llevara a cabo las transformaciones de las luces:

(a) Comprobamos que la luz está encendida y que la iluminación está activada. De no ser así no se podrá realizar ninguna transformación.

(b) Comprobamos que tipo de transformación se desea hacer:

Si es una traslación, miramos que la luz seleccionada no sea la luz 3 ya que es el foco asociado al objeto y, si no es esa luz procederemos a comprobar que la luz sea un foco o una bombilla. En caso afirmativo, realizaremos los cálculos pertinentes para llevar a cabo la traslación. Como podemos observar en el código son cálculos sencillos.

Si es una rotación miramos a ver que tipo de luz es el seleccionado.

En caso de ser una bombilla o un sol, como hemos comentado, siempre se realizará la rotación en global y por tanto, basándonos en la teoría vista en clase procederemos a calcular los nuevos valores del vector positionLight mediante la función rotarLuz.

En el caso de ser un foco, como en el paso anterior miraremos a ver si el foco seleccionado es el 3, en caso de ser así no produciríamos ningún cambio. En caso contrario, miraremos el tipo de transformación se quiere hacer, es decir, si es local o si es global. En el caso de ser una transformación local, procederemos a calcular los nuevos valores del vector spotDirecction mediante la función rotarFoco. En caso de ser una transformación local, procederemos a calcular los nuevos valores de los vectores spotDirection mediante la función dicha anteriormente y los valores de positionLight mediante la función rotarLuz.

En último lugar cambiaremos el volumen de luz que emite un foco. Primero comprobaremos si la fuente de luz seleccionada es un foco, y, en caso de ser así, procederemos a ampliar o reducir el volumen de luz que emite comprobando antes que la amplitud del foco nunca sea negativa o pase los 90º.

- Por último, nos quedan las opciones remarcadas con las teclas F1 a F12. Dentro de este bloque vamos a dividir en 4 grupos las teclas agrupandolas por funcionalidad.
  - Teclas F1-F8: estas teclas sirven para des/activar una luz, llamando a la función cambiarEstadoIlluminacion se logra gestionar el encendido o apagado de las luces. Al encender una luz esta no se selecciona como luzSeleccionada habrá que pulsar los números 1-8 para seleccionarla.
  - Tecla F9: con esta tecla logramos habilitar o deshabilitar el uso de las luces en nuestro proyecto. Esto se hace gracias a llamada a la función onOffIlluminación.
  - Tecla F10: con esta tecla (opción añadida) logramos apagar todas las luces en el caso de que haya una encendida o, en caso contrario, encender todasl las luces.
  - Tecla F11: con esta tecla (opción añadida) podemos hacer debug de la luz seleccionada para así saber cómo cómo se están realizando las transformaciones de dicha luz. Además, en el caso de que la luz seleccionada sea un foco, también observaremos su dirección.
  - Tecla F12: Con esta tecla podemos cambiar el tipo de iluminación que deseamos en la escena.

4. En **io.h** dados los cambios producidos en el fichero io.c hemos tenido que añadir las cabeceras de los nuevos métodos. De este modo, dentro de este fichero podemos encontrar el siguiente contenido:

```
#ifndef IO_H
#define IO_H
#include "definitions.h"

void print_help();
void insertarCamara(struct camera *cam);
void actualizarCamara();
void inicializarCamara();
void calcularNormales(object3d *optr);
void actualizarFoco();
void copiarContenidoArray(GLfloat matrizOrigen[4], GLfloat matrizDestino[4]);
void opcionSuprimir(object3d *auxiliar_object);
void keyboard(unsigned char key, int x, int y);
int existeLuzEncendida();
void disableLights();
void enableLights();
void rotarFoco(int index1, int index2, int valorMultipl, int valorMultipl2,
     float angle);
void rotarLuz(int index1, int index2, int valorMultipl, int valorMultipl2,
     float angle);
void nuevaLuz(int ind);
void cambiarEstadoLuz(int luz);
void seleccionarMaterial(int num);
void onOffIluminacion();
void SpecialKeys(int key, int x, int y);

#endif // IO_H
```

5. En **display.c** hemos hecho uso de variables externas al fichero y hemos añadido una variable más. Por otro lado también hemos creado diferentes funciones para poder llevar a cabo todo lo que se nos pedía hacer.

   - En la parte de variables globales, hemos hecho uso de variables externas como puede ser el array de luces creadas o la luz seleccionada. Por otro lado, también hemos creado la variable foco_debug a modo de flag para así saber si está activado el modo debug. Además de esa nueva variable, también hemos creado variables auxiliares para recorrer vectores o almacenar valores.

   - Tenemos una función nueva establecerLuces() para establecer los valores que toman los vectores de las luces. Cada vez que se haga cualquier cambio, a la hora de dibujar el objeto, se llamará a esta función para establecer las luces que están encendidas. Esta función funciona del siguiente modo:

     (a) Vamos a recorrer el vector de luces que tenemos creado. Por cada luz comprobaremos su estado.

     (b) En caso de que esté la luz encendida procederemos a la función glLightv para indicar la luz que se quiere modificar, qué componente se quiere modificar y el calor que le vamos a dar a ese componente.

         En caso de estar ante un sol o una bombilla solo actuarán los componentes GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_CONSTANT_ATTENUATION, GL_LINEAL_ATTENUATION y GL_QUADRATIC_ATTENUATION. En caso de estar ante una luz tipo foco, además de los componentes anteriores también actuarán GL_SPOT_CUTOFF, GL_SPOT_DIRECTION y GL_SPOT_EXPONENT.

   - Por ultimo, en el método de dibujado, display, tenemos varios cambios.
     - Al inicio del método se ha añadido el uso de un buffer de profundidad con la instrucción glClear(GL_DEPTH_BUFFER_BIT).

     - Como ya se ha explicado en el apartado de explicación de io.c, presionando la tecla F12 podemos cambiar entre los tipos de iluminación. Gracias al flag flat_or_smooth que hemos creado, podremos utilizarlo para establecer el tipo de iluminación deseada. Mediante la función glShadeMode(), introduciendo GL_FLAT obtendremos una iluminación de tipo flat y, mediante GL_SMOOTH tendremos una iluminación tipo smooth.

     - El siguiente cambio que observamos es que, siempre que se vaya a dibujar un objeto, comprobaremos anteriormente si se le ha asignado algún material en concreto. En caso de tener un tipo de material asignado, se le aplicará ese material al objeto.

     - Otro cambio que podemos observar en este método es el debug de la luz seleccionada. En el caso de que el modo debug esté activado, procederemos a dibujar una esfera en el punto donde se encuentre esa luz y, en el caso de que sea una luz tipo foco, procederemos a dibujar mediante una recta la dirección de su foco. Solo se realizará el dibujado de la dirección en el foco dado que las luces tipo bombilla y tipo sol aplican luz en todas las direcciones lo que nos impediría ver el entorno con claridad. La generación de la esfera y de la recta de la dirección del foco está gestionada en dos partes a lo largo de este método.

– El último cambio que podemos observar en este método es la asignación de normales creada en el fichero io.c.

En primer lugar miraremos a ver cual es el tipo sombreado deseado. En el caso de "baja calidad" o tipo flat, asignaremos las normales para cada cara del polígono, esto nos ofrece una peor calidad pero también un decremento en el número de operaciones a realizar. En cambio, al desear una mejor calidad, tipo smooth, realizaríamos las normales sobre todos los vértices del polígono. Esto nos acarrearía un mayor coste computacional pero unos mejores resultados también.

```
 for (f = 0; f < aux_obj->num_faces; f++) {
   glBegin(GL_POLYGON);

  if (!flat_or_smooth)
    glNormal3f(aux_obj->face_table[f].normal.x,
      aux_obj->face_table[f].normal.y, aux_obj->face_table[f].normal.z);

  for (v = 0; v < aux_obj->face_table[f].num_vertices; v++) {
    v_index = aux_obj->face_table[f].vertex_table[v];

    if (flat_or_smooth)
      glNormal3f(aux_obj->vertex_table[v_index].normal.x,
        aux_obj->vertex_table[v_index].normal.y,
        aux_obj->vertex_table[v_index].normal.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);
  }
  glEnd();
}
```

6. En **display.h** al igual que con los anteriores ficheros de cabeceras, hemos tenido que añadir las declaraciones de los nuevos metodos mencionados anteriormente. De este modo, el nuevo fichero display.h será:

```
#ifndef DISPLAY_H
#define DISPLAY_H

void draw_axes();
void reshape(int width, int height);
void establecerLuces();
void display(void);

#endif // DISPLAY_H
```

5.- Ampliación a lo pedido

5.1. Transformaciones de los objetos

Referente a las transformaciones de los objetos hemos añadido lo siguiente:

1. *Rehacer*: Usando las teclas 'R' o r' el usuario podrá rehacer los cambios deshechos.

2. *Conditional imports*: Al encontrarnos con el problema de que nuestro código variaba dependiendo del sistema operativo usado, hemos investigado como se hace un programa multiplataforma, ya que no estábamos por la labor de andar copiando y pegando las diferentes lineas todos los días, y hemos descubierto la existencia de #ifdef y #elif así como los descriptores __linux__, __APPLE__ y __WIN32__ o __WIN64__ dependiendo de la versión de Windows que usemos.

5.2. Transformaciones de la cámara

Referente a las transformaciones de los cámaras hemos añadido lo siguiente:

1. *Cámara desacoplada del objeto*: Gracias a la matriz que hemos creado en el objeto (mat_rot_cam) podemos rotar la cámara de los objetos independientemente a si movemos los objetos o no. Para que esto funcionara, en el fichero display hemos tenido que hacer cierto cambio para que funcionara. El cambio es el siguiente:

```
if(obj_cam == 0){
     glLoadIdentity();
}
else{
     glLoadMatrixf(_selected_object->mat_rot_cam);
}
```

   De este modo, si el objeto es la cámara, en la modelview guardaremos la matriz de rotación de la cámara y si no, en modelview guardaremos la matriz identidad.

2. *Cámara nueva*: Presionando las teclas I, i podremos crear una nueva cámara en el punto que queramos, mirando al punto que queramos y con la verticalidad que queramos. Esta cámara se guardará en la lista de las cámaras. En ningún momento podrá pertenecer a un objeto.

5.3. Transformaciones de las luces

Aunque al inicio no pensábamos incluir extras, al final hemos incluido unos cuantos debido a que hemos pensado que eran interesantes.

1. *Modo debug*: A la hora de crear los focos y crear sus transformaciones no estábamos seguros de que funcionaran bien, debido a varios factores, así que decidimos crear una esfera y una línea que apuntara en la misma dirección del foco, para así saber hacia dónde enfocaba el foco y ver porque no veíamos ninguna iluminación. Inicialmente solo iba a funcionar para los focos, pero, a modo de hacer pruebas, para saber si se estaban haciendo correctamente las transformacioens en soles y bombillas, en el caso de que la luz sea un sol o sea una bombilla, la esfera se colocará en el punto que indiquen los tres primeros valoers del vector positionLight de la luz. La opción de debug se podrá utilizar pulsando F11.

2. *Encender o apagar todas las luces*: Con la función de desactivar luces entendimos que era apagar todas las luces, cuando resultaba que era apagar el modelo de luz, así que cambiamos lo que, hacia la función normal, y mantuvimos el fallo que habíamos tenido como una característica nueva del programa, así que ahora con F10 se encienden o apagan todas las luces, mientras que con F9 se hace el funcionamiento normal de activar o desactivar el modelo de luz.

3. *Menú contextual*: Por no tener que recordar todas las teclas necesarias para hacer una acción, al final hemos implementado un menú que con el clic derecho en cualquier parte de la pantalla nos muestre algunas opciones básicas. Este menú se genera usando la API de GLUT, la cual usa el estilo actual del sistema para los colores y la estética en general del menú y del texto, por lo que, si se viera que se muestra un menú oscuro o sin letras, se deberá cambiar el estilo del sistema para que sea visible y se vea lo que se pulsa. En todo caso, en el momento de encender una luz si esta luz no está creada, habrá que introducir los valores por consola como se hace al intentar encenderla por teclado.

4. *Cada objeto tiene su propio material*: Al inicio teníamos un material para todos los objetos, pero luego hemos implementado un menú que se muestra al pulsar la tecla 'w' que muestra y solicita al usuario el material deseado para el objeto seleccionado, de entre un listado que tiene el programa. Así mismo, esta opción la hemos implementado en el menú contextual, ya que nos parecía una forma más rápida de cambiar el material del objeto seleccionado.

6.- Conclusiones

6.1. Transformaciones a los objetos

Aunque el inicio ha sido un poco complicado, en especial porque no habíamos usado C hasta ahora y no terminábamos de entender como trabajar con punteros, con ayuda de nuestros compañeros hemos conseguido superar ese bache. Aunque luego nos hemos encontrado con algún otro problema, los hemos superado y nos ha gustado llegar al final del ejercicio, ver lo que hemos creado y sentirnos orgullosos de ello.

Hemos aprendido como se pueden utilizar funciones que existen en librerías que necesitábamos, y que nos ayudan a quebrarnos menos la cabeza logrando un resultado igual (o mejor) que en el caso de intentar programar nosotros esas mismas funciones (sobre todo en el caso de las transformaciones).

6.2. Transformaciones a la cámara

En la fase de la cámara hemos observado una mayor dificultad pero, al igual que en el anterior trabajo, hemos ido poco a poco entendiendo la teoría y gracias a las tutorías y a las sesiones prácticas de laboratorio hemos entendido como aplicarla para lograr el resultado. Con ello hemos entendido la importancia de aclararnos en cuanto a lo que se pide y cómo hay que resolverlo ya que en caso de no entender alguna de esas partes la dificultad del trabajo aumenta.

Una vez nos hemos aclarado con los nuevos conceptos hemos podido apreciar como las transformaciones a realizar sobre las cámaras son similares a las transformaciones que hicimos sobre los objetos.

6.3. Transformaciones de las luces

Aunque al inicio estábamos bastante perdidos y pensábamos que no nos daría tiempo, al final metiéndole horas y yendo varias veces a resolver dudas en tutorias hemos podido solventar todas las dudas y acabarlo a tiempo, aunque luego la documentación hemos tardado un poco más terminarla.

Aun así, hemos acabado el proyecto, y la asignatura, con buen sabor de boca; nos ha gustado lo que hemos terminado haciendo y aunque en algún momento ha sido pesado o nos hemos visto estresados por la cantidad de cosas que hacer junto con otras asignaturas, al final creemos que ha merecido la pena, y tenemos una aplicación que nos gusta mucho como ha quedado.

Referencias

[vertice-pixel] Tutorial camaras en openGL 4

[espacio3D] Moviendonos en un espacio 3D

[teclado] Teclado y funciones especiales

[tutorial-camaras] Tutorial camaras en openGL 4

[teoria-cam] Teoria de camaras

[ejemplo-glm] Teoria y ejemplos con GLM

[redBook] Libro rojo de openGL v.1.1

[menuClickDerecho] Crear menu contextual

[guiaLuces] Luces - Guia general

[materiales] Lista de materiales

Anexos

Anexo I: Manual del usuario - Transformaciones de los objetos

El usuario podrá usar las siguientes teclas para manejar el programa:

- ?: Visualizar la ayuda
- ESC: Finalizar la ejecución de la aplicación
- F,f: Carga de objeto desde un fichero *.obj
- TAB: Seleccionar siguiente objeto (de entre los cargados)
- SUPR: Eliminar objeto seleccionado
- CTRL + +: Reducir el volumen de visualización
- CTRL + -: Incrementar volumen de visualización
- M,m: Activar traslación.
- B,b: Activar rotación.
- T,t: Activar escalado.
- G,g: Activar transformaciones en el sistema de referencia del mundo. (transformaciones globales)
- L,l: Activar transformaciones en el sistema de referencia local del objeto.
- U,u: Deshacer.
- R,r: Rehacer.
- UP: Trasladar +Y; Escalar + Y; Rotar +X.
- DOWN: Trasladar -Y; Escalar - Y; Rotar -X.
- RIGHT: Trasladar +X; Escalar + X; Rotar +Y.
- LEFT: Trasladar -X; Escalar - X; Rotar -Y.
- AVPAG: Trasladar +Z; Escalar + Z; Rotar +Z.
- REPAG: Trasladar -Z; Escalar - Z; Rotar -Z.
- +: Escalar + en todos los ejes. (solo objetos)
- -: Escalar - en todos los ejes. (solo objetos)

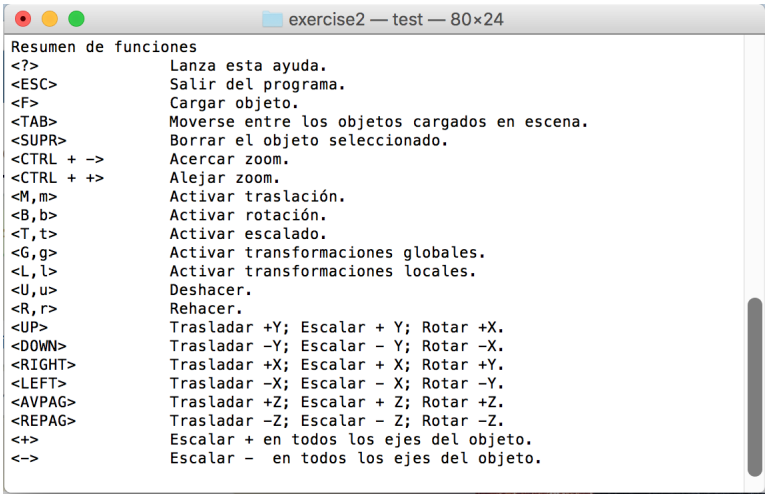
En primer lugar el usuario deberá conocer su sistema operativo para saber que comando utilizar a la hora de compilar el código.

Deberá abrir la terminal, colocarse en la carpeta que contiene los ficheros necesarios para la compilación y posterior ejecución del programa y, dependiendo de en que sistema operativo esté, deberá ejecutar los siguientes comandos:

- En Linux: gcc -lGL -lGL -lGLU -lglut *.c -I./ -o test
- En MAC OS X: gcc -w *.c -I./ -framework OpenGL -framework GLUT -o test

A la hora de ejecutar el programa habrá que ejecutar el comando ./test y podremos observar la siguiente pantalla:

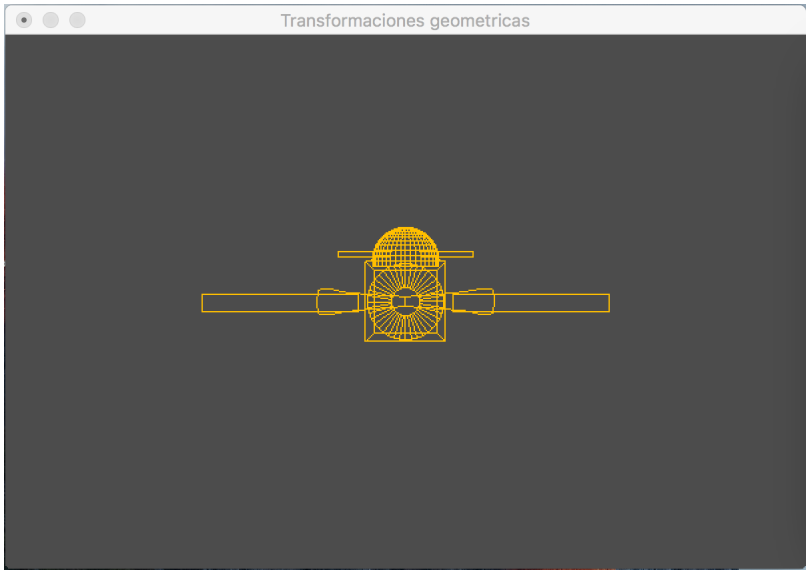
En la terminal observaremos un resumen de las funciones que tiene nuestro programa.



Para poder iniciar nuestro proceso de transformación de un objeto, es imprescindible insertar un objeto. La manera de insertar un objeto en nuestra pantalla es presionando la tecla f ó F.

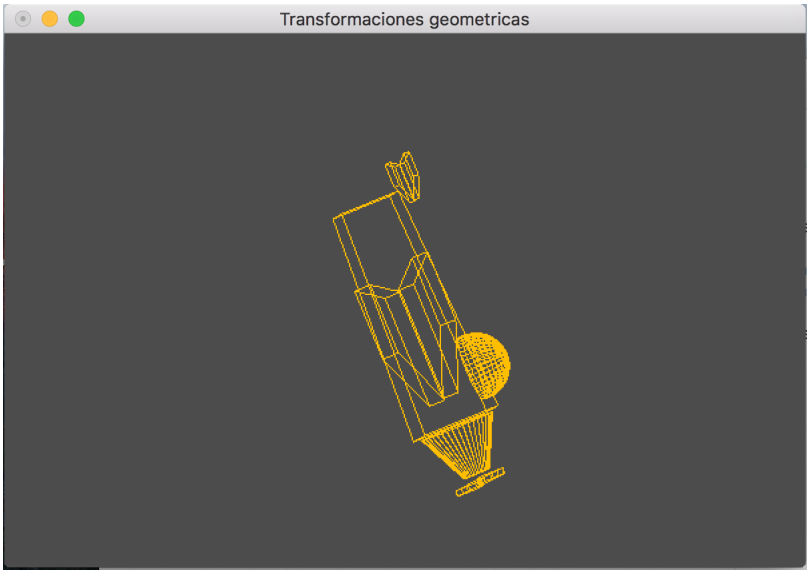
Una vez presionada esta tecla, por la terminal se nos indicará que debemos introducir un path donde haya un objeto. Esta es la ruta que tiene el objeto.

Para hacer el ejemplo nosotros vamos a utilizar el objeto abioia.obj. Al cargarlo podremos observar:

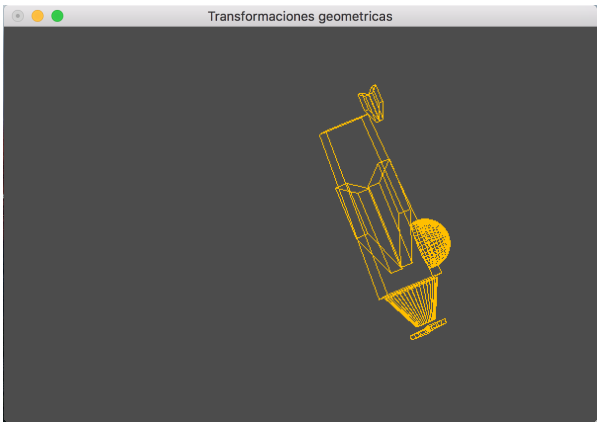


Por defecto solo se pueden utilizar las teclas '+' y '-' para escalar todos los ejes del objeto. Presionando '+' agrandaremos los objetos y presionando '-' decrementaremos los mismos. Para poder transformar el objeto de cualquier manera tendremos que presionar las teclas que hemos mostrado en las funciones anteriormente.

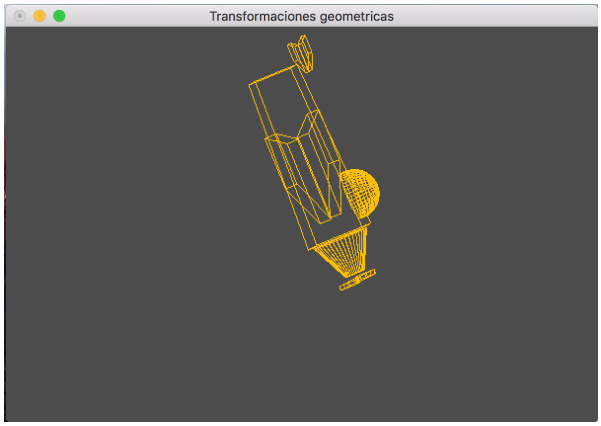
Tal como se muestra podemos hacer transformaciones locales o globales. La diferencia de las mismas es la siguiente: Si tenemos el objeto en una posición



La transformación local hacia arriba sería lo que observamos en la figura de la izquierda y la transformación global hacia arriba lo que observamos en la figura de la derecha:



(a) Tranformacion local



(b) Transformacion global

Moviéndose hacia arriba

A partir de ahora, y con el uso de las flechas de dirección del teclado, la tecla AVPAG y la tecla REPAG ya se podría mover los objetos cargados.

Del mismo modo que se ha cargado un objeto, volviendo a repetir el proceso de carga se pueden cargar tantos objetos como se deseen y, para moverse entre ellos solo habrá que presionar la tecla TAB. En el caso de que se quiera suprimir el objeto seleccionado (aquel que está en color naranja) habrá que presionar la tecla SUPR.

Anexo II: Manual del usuario - Transformaciones de la cámara

Para operaciones relacionadas con la cámara, como su movimiento o creación de múltiples cámaras tenemos las siguientes teclas.

- c: Presionando esta tecla podemos pasar a la siguiente cámara que esté en la lista de cámaras.
- C: Ver lo que el objeto seleccionado visualiza.
- K, k: Opción de transformaciones a la cámara. En el caso de no haber presionado esta tecla (o en su defecto las teclas O, o) no se podrá realizar ninguna transformación.
- G, g: Elección del modo análisis de la cámara. En este modo las transformaciones (rotación) se realizan sobre un objeto determinado, el objeto que esté seleccionado en ese momento.
- L, l: Elección del modo vuelo de la cámara. Las transformaciones se realizan con respecto a la cámara seleccionada.
- T, t: Permite cambiar el volumen de visión de la cámara, permitiendo así ampliar o reducir el volumen de visión.
- B, b: Presionando esta tecla elegiremos hacer como transformación la rotación. Por defecto las transformaciones se realizarán en modo vuelo.
- M, m: Presionando esta tecla elegiremos hacer como transformación la traslación. Por defecto las transformaciones se realizarán en modo vuelo.
- P, p: Permite cambiar el modo de proyección de la cámara, podremos elegir entre proyección en paralelo o proyección en perspectiva.
- U, u: Deshacer cambios. En el caso de que la cámara sea la del objeto seleccionado, se le aplicarán también los cambios a la misma.
- R, r: Rehacer cambios. En el caso de que la cámara sea la del objeto seleccionado, se le aplicarán también los cambios a la misma.
- I, i: Creación de una nueva cámara no relacionada a ningún objeto.
- Tecla +: Permite hacer el zoom a la cámara.
- Tecla -: Permite deshacer el zoom a la cámara.

Para poder aplicar las transformaciones hay que tener en cuenta qué transformación se hace dependiendo del modo en el que estemos y la tecla que presionemos. Para llevar a cabo las transformaciones se presionarán las flechas (arriba, abajo, derecha e izquierda) y las teclas AvPag y RePag del teclado. Los cambios que sufrirá la cámara al presionar las diferentes teclas serán los siguientes: dirección.

- Arriba: Trasladar +Y; Rotar + en X; Volumen +Y
- Abajo: Trasladar -Y; Rotar - en X; Volumen -Y
- Derecha: Trasladar +X; Rotar + en Y; Volumen +X
- Izquierda: Trasladar -X; Rotar - en Y; Volumen -X
- AvPag: Trasladar +Z; Rotar + en Z: Volumen +n,+f
- RePag: Trasladar -Z; Rotar - en Z; Volumen -n,-f

Anexo III: Manual del usuario - Transformaciones de las luces

- 1-8: Seleccionar la fuente de luz correspondiente.
- 0: Asignar tipo de fuente de luz a la fuente seleccionada (solamente luces de la 4 a la 8)
- F1-F8: Enciende o apaga la fuente de luz seleccionada. El F1, F2, F3 corresponde respectivamente con un sol, una bombilla y un foco asociado al objeto seleccionado; el resto serán espacios vacíos para que el usuario añada sus luces.
- F9: Activar/Desactivar modelo de luz.
- F10: Encender/Apagar todas las luces a la vez.
- F11: Activar modo debug de luz tipo foco, para ver donde esta el foco y a donde apunta.
- F12: Cambiar de tipo de iluminación, entre flat y smooth.
- Click derecho: Menú con opciones que se pueden hacer por teclado pero son mas rápidas mediante clicks. Hay que tener que algunas opciones pedirán información al usuario mediante la consola, como por ejemplo al seleccionar una luz que no exista, se pedirá la información para la luz nueva a través de la consola.
- a,A: Activar el uso de transformaciones a las luces.
- +: Si la luz es un foco entonces podremos aumentar el angulo de apertura del mismo.
- -: Decrementar el angulo de apertura del foco seleccionado.

Las transformaciones que se realicen a las luces con a,A siguen manteniendo las teclas que se usaban para transformaciones de la cámara.

- M,m: Activar traslación.
- B,b: Activar rotación.
- G,g: Activar transformaciones en el sistema de referencia del mundo. (transformaciones globales)
- L,l: Activar transformaciones en el sistema de referencia local del objeto.
- UP: Trasladar +Y; Rotar +X.
- DOWN: Trasladar -Y; Rotar -X.
- RIGHT: Trasladar +X; Rotar +Y.
- LEFT: Trasladar -X; Rotar -Y.
- AVPAG: Trasladar +Z; Rotar +Z.
- REPAG: Trasladar -Z; Rotar -Z.