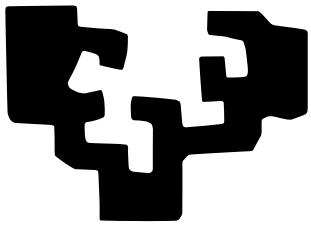
eman ta zabal zazu



Universidad del País Vasco

Euskal Herriko Unibertsitatea

Gráficos por Computador : Transformaciones geométricas

Finalizado el Domingo, 4 de diciembre, 2016

Cristina Mayor y Sergio Tobal

Contents

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.

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 camaras. 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.

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.

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 hechos, 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.

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.

10.- Conclusiones

10.1. 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).

10.2. 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.

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

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)

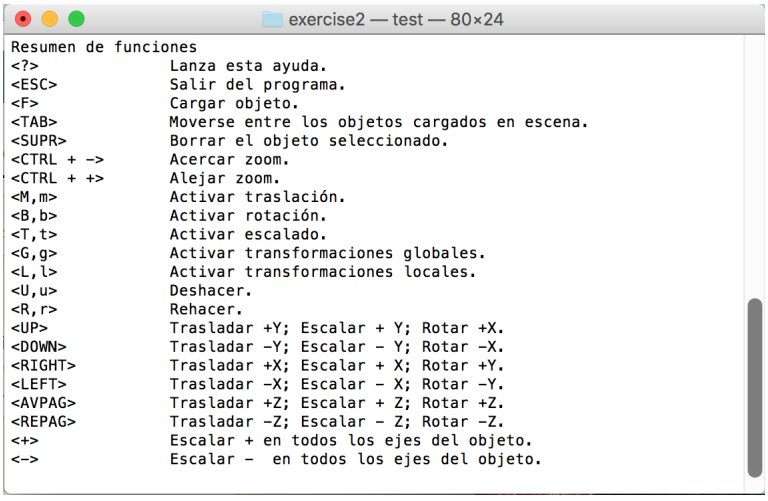
En primer lugar el usuario deberá conocer su sistema operativo para saber que comando utilizar a la hora de compilar el código.

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:

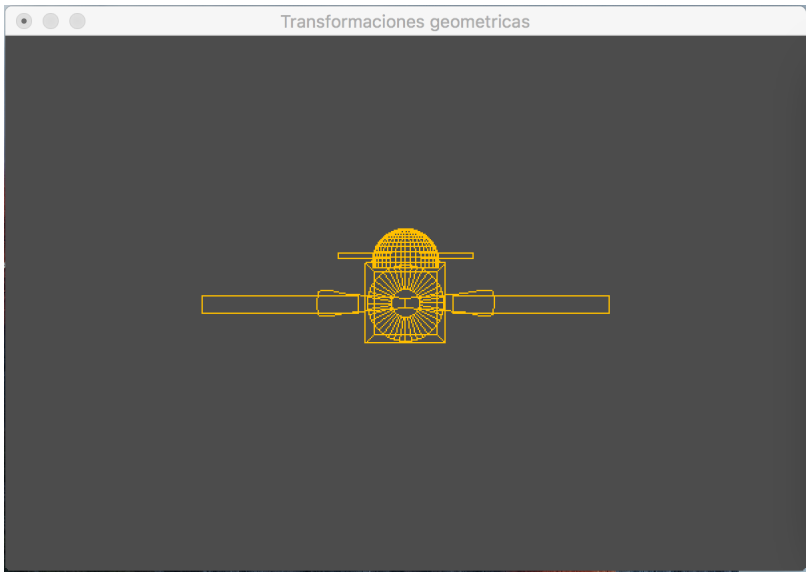
En la terminal observaremos un resumen de las funciones que tiene nuestro programa.



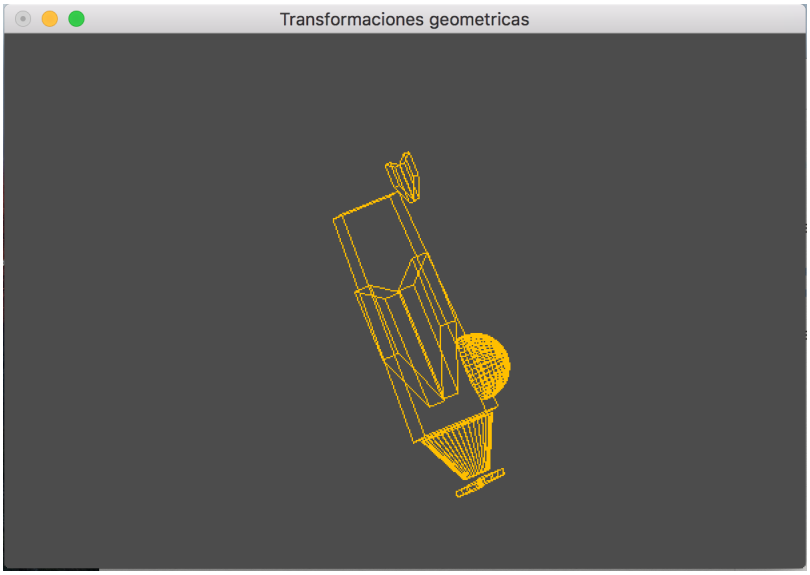
Para poder iniciar nuestro proceso de transformación de un objeto, es imprescindible insertar un objeto. La manera de insertar un objeto en nuestra pantalla es presionando la tecla f ó F.

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.

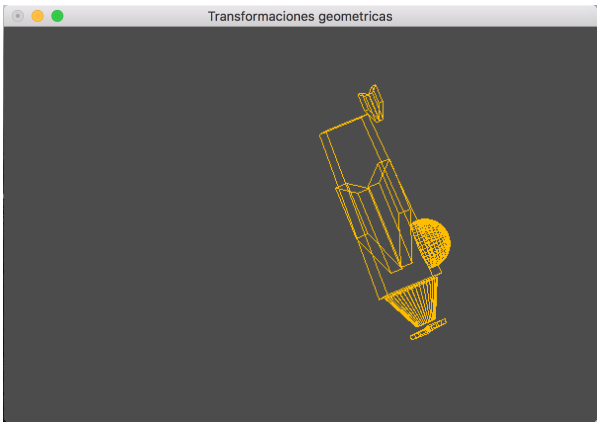
Para hacer el ejemplo nosotros vamos a utilizar el objeto abioia.obj. Al cargarlo podremos observar:



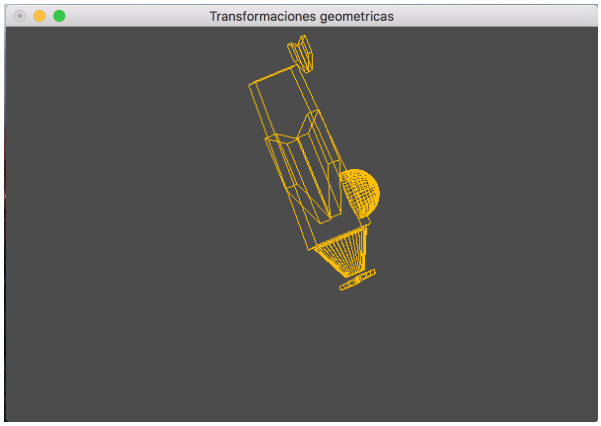
Por defecto solo se pueden utilizar las teclas '+' y '-' para escalar todos los ejes del objeto. Presionando '+' agrandaremos los objetos y presionando '-' decrementaremos los mismos. Para poder transformar el objeto de cualquier manera tendremos que presionar las teclas que hemos mostrado en las funciones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