# Transformaciones, Cámaras e Iluminación

Julen Beristain

Gráficos por Computador

**Abstract**

Documentación de una aplicación que toma objetos definidos en formato .obj y los visualiza en un entorno 3D, dando la posibilidad de transformar objetos y cámaras, cambiar la proyección y el modo (vuelo o análisis) de la cámara, y con un sistema de iluminación.

# Contents

**1 Objetivos de nuestra aplicación** **2**

**2 Interacción con el usuario** **2**

**3 Descripción de las transformaciones** **5**
3.1 Cámaras . . . . . 5
3.2 Objetos . . . . . 5
3.3 Luces . . . . . 6

**4 Ficheros de objetos** **6**

**5 Valores por defecto** **6**

**6 Código C** **8**

**7 Trabajo futuro** **19**

**References** **20**

# 1 Objetivos de nuestra aplicación

El objetivo de esta aplicación es dibujar objetos en formato .obj con materiales predefinidos en un entorno tridimensional. La aplicación incluye la posibilidad de transformar los objetos y las cámaras (traslaciones y rotaciones para ambos y adicionalmente escalado para los objetos). Con respecto a la cámara podemos modificar dos aspectos: la proyección de lo que visualiza, ortográfica o en perspectiva; y por otro lado, el modo: vuelo (libre) y análisis (orbitando alrededor del objeto seleccionado). Por último, también implementa un sistema de iluminación con luz ambiental, luz solar (puramente direccional), una bombilla (que ilumina igual en todas las direcciones) y sendos focos para el objeto y la cámara seleccionados. Los focos están sincronizados con el objeto o la cámara seleccionada (tienen la misma orientación y posición), mientras que podemos trasladar la bombilla y rotar la dirección de la luz solar. Además, también es posible modificar las intensidades (los colores rgb) de las luces.

# 2 Interacción con el usuario

Las opciones de nuestra aplicación son todas por teclado:

1. **f**: se utiliza para cargar un nuevo objeto desde el fichero indicado por línea de comandos. Pedirá el nombre del fichero desde donde se cargará el nuevo objeto, además de mostrar la lista de materiales predefinidos y hacer que elija uno. El objeto se sitúa en el centro del mundo virtual, con la orientación definida en su propio fichero, pasando a ser el seleccionado.

2. **F**: se utiliza para cargar una nueva cámara desde el fichero "Objetos/-cam.obj". La cámara se sitúa en el centro del mundo mirando hacia delante (eje z negativo), pasando a ser la seleccionada.

3. **TAB**: dependiendo de qué estemos transformando en cada momento (es decir, del valor de la variable global `transformations_to`), esta tecla nos sirve para seleccionar el siguiente elemento; es decir, cuando estemos transformando a las cámaras, pasaremos a seleccionar el siguiente. Lo mismo con los objetos y las luces. La selección es circular, por lo que al llegar al último de la lista y volver a pulsar esta tecla, se vuelve a elegir el primero de la lista.

4. **Intro**: solo sirve para el modo polígono, donde se dibuja cada polígono por separado sin considerar el objeto en su conjunto. Dibuja el polígono

indicado por la variable `indexx` del objeto seleccionado, y pasa a seleccionar el siguiente polígono. Esta selección también es circular.

5. **ESC**: finaliza la aplicación.

6. **l/L**: modifica la forma de dibujar los polígonos; los polígonos se pueden dibujar de dos formas: por un lado dibujando sólo las aristas, y por otro lado, dibujando todos los puntos internos. Esta tecla cambia de una forma de dibujo a la otra.

7. **o/O**: permite dibujar todo el objeto o solamente el polígono seleccionado del conjunto. La tecla hace cambiar de una opción a la otra.

8. **d/D**: Si estamos en el modo de dibujar el objeto entero, mediante esta tecla se puede pedir que se dibujen todos los objetos o solo el seleccionado. Si estamos en el modo de dibujar cada polígono por separado, al pulsar "Intro" los polígonos dibujados anterioremente se mantienen dibujados o se eliminan. Cada vez que pulsemos la tecla cambiaremos de una opción a la otra.

9. **t/T**: cambia la transformación a aplicar seleccionada a traslación.

10. **r/R**: cambia la transformación a aplicar seleccionada a rotación.

11. **s/S**: cambia la transformación a aplicar seleccionada a escalado. Sólo funciona cuando transformamos objetos.

12. **g/G**: cuando transformamos a las cámaras, pasamos del modo vuelo, donde la cámara seleccionada es libre, a modo análisis, donde pasa a mirar al centro de un objeto y orbita a su alrededor, y viceversa.

    Cuando transformamamos a los objetos, pasamos de tomar el sistema de referencia local del objeto a tomar el sistema de referencia global, y viceversa.

13. **x/X**: no son equivalentes, transforman en sentidos contrarios. Efectúan la transformación seleccionada en el eje de las X a la cámara, objeto o luz seleccionada (dependiendo de `transformations_to`). Mirar la sección 3.

14. **y/Y**: no son equivalentes, transforman en sentidos contrarios. Efectúan la transformación seleccionada en el eje de las Y a la cámara, objeto o luz seleccionada (dependiendo de `transformations_to`). Mirar la sección 3.

15. **z/Z**: no son equivalentes, transforman en sentidos contrarios. Efectúan la transformación seleccionada en el eje de las Z a la cámara, objeto o luz seleccionada (dependiendo de `transformations_to`). Mirar la sección 3.

16. **u/U**: deshace la última transformación realizada al objeto, cámara o luz (solo en caso de que sea el sol o la bombilla, ya que los focos están en sincronía con las transformaciones a su elemento seleccionado y la luz ambiental no acepta transformaciones) seleccionada (dependiendo de `transformations_to`).

17. **c**: cambia el sujeto de transformaciones; es decir, hace la selección circular "$camara$" $\rightarrow$ "$objeto$" $\rightarrow$ "$luz$" $\rightarrow$ "$camara$" $\rightarrow$ ...

18. **C**: cambia el observador: de cámara a objeto (los seleccionados) y viceversa, afectando al sistema de referencia.

19. **p/P**: cambia el modo de proyección: de ortográfica a en perspectiva y viceversa.

20. **b/B**: pasa de no dibujar a dibujar las caras ocultas de los objetos y las cámaras en rojo, y viceversa.

21. **n/N**: dibuja los vectores normales de las caras (los poligonos) o los deja de dibujar.

22. **m/M**: dibuja los vectores normales de los vértices de los polígonos, que se definen como el vector compuesto por las medias aritméticas de los componentes x, y, z de los vectores normales de las caras que contienen al vértice, o los deja de dibujar.

23. **0/1/2/3/4**: encienden o apagan la luz correspondiente. El array de luces tiene este orden: ambiental, sol, bombilla, objeto seleccionado, cámara seleccionada.

24. **i/I**: cambio de técnica de iluminación: al elegir los vectores normales para considerar la inclinación de los polígonos con respecto a los rayos de luz pasa de emplear los normales de los vértices a los normales de las caras, y viceversa.

25. **+/-**: en caso de estar transformando luces de tipo foco (luces 3 y 4) aumenta o disminuye el ángulo de iluminación, es decir, la apertura de los focos.

26. **q/Q**: en caso de estar transformando a las luces, aumenta o disminuye la intensidad roja de la luz seleccionada. Como la tecla "r/R" ya estaba ocupada, he elegido a "q/Q" por estar próximo a "a/A".

27. **a/A**: en caso de estar transformando a las luces, aumenta o disminuye la intensidad azul de la luz seleccionada.

28. **v/V**: en caso de estar transformando a las luces, aumenta o disminuye la intensidad verde de la luz seleccionada.

# 3 Descripción de las transformaciones

## 3.1 Cámaras

En el modo vuelo, para las transformaciones X e Y da igual que la transformación elegida sea traslación o rotación, la cámara gira horizontalmente y verticalmente el sitio donde está ("yaw" y "pitch", en el sistema local de la cámara). Por otro lado, para la transformación Z, cuando se traslada se mueve en la dirección Z (hacia atras y hacia delante) y cuando rota lo hace alrededor de dicho eje ("roll"). Las transformaciones son en el sistema de referencia local de la cámara.

En el modo análisis, de nuevo a las transformaciones X e Y les da igual que la elegida sea traslación o rotación, ya que siempre se orbita horizontal y verticalmente alrededor del objeto (sistema de referencia global, ejes paralelos a los locales de la cámara pero que pasan por el centro del objeto). En cuanto a la Z, cuando se traslada nos alejamos o acercamos al centro del objeto y cuando se rota se hace alrededor de dicho eje ("roll"), como antes (en el sistema local de la cámara).

En cuanto al escalado, no se permite aplicarlo a las cámaras.

## 3.2 Objetos

Las traslaciones desplazan el objeto una distancia predeterminada (pequeña) en el eje que toque.

Las rotaciones rotan al objeto alrededor del eje que toque (las rotaciones alrededor de cualquier eje solo se aplican a las cámaras en el modo análisis para poder orbitar alrededor de los objetos)

Los escalados son uniformes en las tres dimensiones.

Las transformaciones se pueden realizar en el sistema local de los objetos o en el global.

### 3.3 Luces

La luz ambiental no acepta transformaciones.

La luz direccional (el sol) solo acepta rotaciones. Por ello, da igual qué transformación esté elegida. Mientras estemos transformando a las luces y la elegida sea el sol la luz rotará sea cual sea la transformación elegida (teclas "t", "r" y "s" sin efecto aparente).

La luz posicional (la bombilla) solo acepta traslaciones, por lo que se aplica lo mismo que lo mencionado acerca del sol.

En cuanto a los focos, como tienen que estar al unísono con la orientación y localización del objeto o la cámara seleccionados, para rotarlos o trasladarlos es necesario aplicar las transformaciones a estos últimos.

## 4 Ficheros de objetos

Utilizamos ficheros con formato .obj para definir los objetos. El formato de archivo .obj es un formato de archivo de texto utilizado para representar objetos tridimensionales en gráficos por computadora, y es un estándar ampliamente adoptado. Un archivo .obj describe la geometría de un objeto tridimensional mediante la especificación de sus vértices y caras. Contiene información sobre la posición tridimensional de cada vértice en las líneas que empiezan por "v". En cuanto a las caras, se tienen en cuenta las líneas que empiezan por "f", que contienen índices a los vértices que componen dicho polígono. Además de la información geométrica, un archivo .obj puede incluir materiales y texturas asociados al objeto, aunque en nuestro caso no los utilizamos. Como mucho, aceptamos también líneas que empiezan por "c" con los componentes rgb que utilizamos para el color en modo "líneas" del objeto, pero para los materiales tenemos una lista predeterminada y en esta iteración de la aplicación no se consideran texturas.

## 5 Valores por defecto

1. **Número de cámaras**: en esta iteración, al inicio de la aplicación contamos con una única cámara.

2. **Posición y dirección de mira de la cámara**: la cámara está posicionada en el punto $(0, 0, 400)$, mirando en la dirección negativa del eje Z, es decir, hacia delante (donde están los objetos).

3. **Volumen de visión de la cámara**: estas son las constantes que utilizamos en "camaras.h" para definir el volumen de visión en proyección

ortográfica (cubo) y en perspectiva (piramide con la punta recortada):

```
// Matriz de proyeccion en perspectiva
#define NEAR_     1.0f
#define FAR_      2000.0f
#define LEFT     (-RIGHT)
#define RIGHT    (NEAR_ * 0.4142136f)    //(tan(PI / 8.0)), 22'5
#define BOTTOM   (-TOP)
#define TOP      (NEAR_ * 0.4142136f)    //(tan(PI / 8.0)), 22'5

//Matriz de proyeccion ortografica
#define ONEAR       0.0f
#define OFAR        2000.0f
#define OLEFT       (-ORIGHT)
#define ORIGHT      500.0f
#define OBOTTOM     (-OTOP)
#define OTOP        500.0f
```

4. **Número de objetos**: al inicio, tenemos tres objetos: una "cámara" de forma piramidal ("cam.obj"), una nave alargada ("x_wing.obj") y otra nave más aplanada y "circular" ("r_falke.obj").

5. **Posición y dirección de los objetos**: todos los objetos están mirando en la misma dirección y sentido que la cámara, hacia delante. El objeto "cam" está en la posición $(0, 0, -400)$; x_wing en $(100, 0, 0)$; y r_falke en $(-100, 0, 0)$.

6. **Materiales de los objetos**: tenemos una lista de 19 materiales predeterminados en "luces.h". Los elegidos para los tres objetos son obsidiana para "cam", esmeralda para "x_wing" y rubí para "r_falke":

```
// EMERALD
#define EMERALD_KA ((color3){0.0215, 0.1745, 0.0215})
#define EMERALD_KD ((color3){0.07568, 0.61424, 0.07568})
#define EMERALD_KS ((color3){0.633, 0.727811, 0.633})
#define EMERALD_NS 76.8

// OBSIDIAN
#define OBSIDIAN_KA ((color3){0.05375, 0.05, 0.06625})
#define OBSIDIAN_KD ((color3){0.18275, 0.17, 0.22525})
#define OBSIDIAN_KS ((color3){0.332741, 0.328634, 0.346435})
#define OBSIDIAN_NS 38.4

// RUBY
#define RUBY_KA ((color3){0.1745, 0.01175, 0.01175})
#define RUBY_KD ((color3){0.61424, 0.04136, 0.04136})
#define RUBY_KS ((color3){0.727811, 0.626959, 0.626959})
#define RUBY_NS 76.8
```

7. **Intensidades de las luces**: todas las luces son blancas (rgb = (255, 255, 255)) al inicio.

8. **Dirección del sol**: el sol empieza apuntando hacia abajo, en la dirección $(0, -1, 0)$.

9. **Posición de la bombilla**: la bombilla se posiciona en $(0, 0, -200)$.

10. **Apertura de los focos**: $22.5^{\text{o}}$ o $45^{\text{o}}$, según consideremos la apertura total de izquierda o derecha o solo la mitad, del eje del foco hacia cualquiera de los "lados" (el volumen que iluminan es cónico).

11. **Atenuación de los focos**: los focos son concentrados, cuanto más nos alejamos de la dirección del foco, menos iluminan. El factor $p = 5$.

12. **Atenuación por distancia**: cuanto más lejos están la bombilla o los focos menos iluminan. El factor de atenuación obtenido por simple prueba y error es:

$$\frac{1}{1 + 0.000005d + 0.000005d^2}$$

# 6 Código C

Me limitaré a resaltar los puntos clave del código, sin ahondar en cada una de las líneas.

El archivo principal "main.c" es el que tiene el punto de entrada al programa. En la función `main`, se inicializa OpenGL, las variables globales y la lista de objetos, la de cámaras y el array de luces. Las otras dos partes principales del archivo son las funciones "callback" que pasamos a `glut` para que se llamen cuando el usuario pulsa una tecla y cuando OpenGL tenga que dibujar la escena.

La función `teklatua` se encarga de lo primero. Fundamentalmente, gestiona la interacción con el usuario, tal y como se ha descrito en la sección 2. Toma en cuenta las interacciones entre las distintas opciones. Por ejemplo, cuando "undo" deshace la primera transformación de la cámara al entrar en modo análisis ("trasformación look at"), salimos automáticamente del modo análisis, para no quedarnos en un estado incongruente (modo análisis sin que la cámara enfoque el centro del objeto seleccionado). En cuanto a curiosidades de la implementación, como las funciones de transformación a objetos, cámaras y luces tienen el mismo tipo (mismo valor de retorno y parámetros del mismo tipo, con la misma cantidad, y el mismo orden) he escrito una función auxiliar (`aldatu`) que recibe como parámetro la función correspondiente a la dimensión, según la tecla que se haya pulsado ("x", "y" o "z"). El último detalle importante es que al final de la función se llama a `glutPostRedisplay` (que llama a `marraztu`) para que se redibuje la escena teniendo en cuenta los cambios realizados.

```
// FUNCION AUXILIAR PARA LAS TRANSFORMACIONES A CAMARAS Y OBJETOS
void aldatu(
    funcAldaketaCamaraVuelo     aldaketa_cam_vuelo,
    funcAldaketaCamaraAnalisis  aldaketa_cam_analisis,
    funcAldaketaObjeto          aldaketa_objeto,
    funcAldaketaLuzDireccional  aldaketa_luz_direccional,
    funcAldaketaLuzPosicional   aldaketa_luz_posicional,
    int dir)
{
    punto3 target;

    assert(transformations_to == CAMARA || transformations_to == OBJETO ||
    transformations_to == LUZ);
    if(transformations_to == CAMARA){
        assert(camara_mode == VUELO || camara_mode == ANALISIS);
        if(camara_mode == VUELO){
            aldaketa_cam_vuelo(sel_cam_ptr, dir);
        } else { //ANALISIS
            mxp3(&target, sel_ptr->mptr->m, sel_ptr->center);
            aldaketa_cam_analisis(sel_cam_ptr, target, dir);
            ++(sel_cam_ptr->num_transf_analisis_mode);
        }
        update_sel_ligth(sel_cam_ptr, SEL_CAM);

    } else if (transformations_to == OBJETO) {
        aldaketa_objeto(sel_ptr, dir);

        if(camara_mode == ANALISIS){
            lookAtSelectedObj();
        }
        update_sel_ligth(sel_ptr, SEL_OBJ);

    } else { // transformations_to == LUZ
        if(lights[sel_luz].type == DIRECTIONAL){
            aldaketa_luz_direccional(&(lights[sel_luz]), dir);
        } else if (lights[sel_luz].type == POSITIONAL){
            aldaketa_luz_posicional(&(lights[sel_luz]), dir);
        }
    }
}
```

La función `marraztu` se encarga de dibujar la escena. Lo primero a destacar es que para hacer la proyección simplemente se fija en el valor de la variable proyection y carga en la matriz de proyección de OpenGL la matriz que toque, con glOrtho o con glFrustum. Es la forma más sencilla y eficiente de gestionar la proyección. También había probado en la anterior iteración hacer la proyección manualmente. En modo "líneas" daba ciertos problemas (aparecían ciertas líneas que no deberían) pero el modo "polígonos" funcionaba bien. En el código siguen los comentarios (no totalmente actualizados) para llevar a cabo la proyección manualmente (en ficheros como "camara.c" y "dibujar_linea_z.c"), pero para la última iteración no me he querido complicar.

```
//ACTUALIZACION MATRIZ OPENGL DE PROYECCION
assert(proyection == PARALELO || proyection == PERSPECTIVA);
glMatrixMode(GL_PROJECTION);
```

```
glLoadIdentity();
if (proyection == PARALELO){
    glOrtho(OLEFT, ORIGHT, OBOTTOM, OTOP, ONEAR, OFAR);
} else { // if (proyection == PERSPECTIVA) {
    glFrustum(LEFT, RIGHT, BOTTOM, TOP, NEAR_, FAR_);
}
```

Lo segundo importante es el cálculo del sistema de referencia, dependiendo del observador. Recalculo la matriz `mcsr` en cada llamada, lo cual también permite que dicha matriz sea una variable local de `marraztu`. Tratar de minimizar estas operaciones me ha parecido contraproducente, ya que tiene sus contras: hacer global a `mcsr` y tener que hacer el recálculo una única vez en la parte precisa que toque, añadiendo comprobaciones (`if`s) por el camino. Esto puede llevar a bugs, porque es muy sencillo olvidarse de algún caso aislado en el que tuviera que recalcular `mcsr`, y por otra parte, teniendo que añadir condicionales al código hace que sea menos legible y no queda del todo claro si fuera a ser más eficiente. Por ello, como el cálculo de esta matriz no es la parte más costosa (en sí es simplemente mirar la matriz del observador y hacer unos pocos cálculos, comparados con tener que dibujar todos los puntos de los objetos), me he decidido por esta aproximación.

```
//CALCULO DEL SISTEMA DE REFERENCIA
assert(observer == CAMARA || observer == OBJETO);
if(observer == CAMARA){
    mobj_to_mcsr(mcsr, sel_cam_ptr->mptr->m);
} else { //if (observer == OBJETO) {
    //Desde el punto de vista del objeto
    mobj_to_mcsr(mcsr, sel_ptr->mptr->m);
}

//en camaras.c
void mobj_to_mcsr(real_t *const mcsr, const real_t *const mobj){
    vector xobj = (vector){mobj[0], mobj[4], mobj[8]};
    vector yobj = (vector){mobj[1], mobj[5], mobj[9]};
    vector zobj = (vector){mobj[2], mobj[6], mobj[10]};
    vector eobj = (vector){mobj[3], mobj[7], mobj[11]};

    mcsr[0] = xobj.x; mcsr[1] = xobj.y; mcsr[2]  = xobj.z;
    mcsr[3]  = - scalar_product(eobj, xobj);
    mcsr[4] = yobj.x; mcsr[5] = yobj.y; mcsr[6]  = yobj.z;
    mcsr[7]  = - scalar_product(eobj, yobj);
    mcsr[8] = zobj.x; mcsr[9] = zobj.y; mcsr[10] = zobj.z;
    mcsr[11] = - scalar_product(eobj, zobj);
    mcsr[12] = 0; mcsr[13] = 0; mcsr[14] = 0; mcsr[15] = 1;
}
```

En tercer lugar, pasa al sistema de referencia del observador todo lo que no dependa de los objetos a la hora de calcular las intensidades, es decir, las luces:

```
//V: la posicion del observador en su SR es el origen 0
//L_sol:
v_sol_mundo = mxv(lights[SUN].mptr->m, lights[SUN].dir);
mxp3(&origen, mcsr, (punto3){0, 0, 0});
```

```
mxp3(&p_dir_sol, mcsr, (punto3){v_sol_mundo.x, v_sol_mundo.y,
    v_sol_mundo.z});
lights[SUN].camdir = normalized(calculate_vector_AtoB3(origen, p_dir_sol));
//L_bombilla
mxp3(&p_pos_bombilla, lights[BULB].mptr->m, lights[BULB].pos);
mxp3(&lights[BULB].campos, mcsr, p_pos_bombilla);
//L_sel_obj: pos y dir ya en el SRMundo
mxp3(&lights[SEL_OBJ].campos, mcsr, lights[SEL_OBJ].pos);
mxp3(&p_dir_sel_obj, mcsr, (punto3){lights[SEL_OBJ].dir.x,
    lights[SEL_OBJ].dir.y, lights[SEL_OBJ].dir.z});
lights[SEL_OBJ].camdir = normalized(calculate_vector_AtoB3(origen,
    p_dir_sel_obj));
//L_sel_cam: pos y dir ya en el SRMundo
mxp3(&lights[SEL_CAM].campos, mcsr, lights[SEL_CAM].pos);
mxp3(&p_dir_sel_cam, mcsr, (punto3){lights[SEL_CAM].dir.x,
    lights[SEL_CAM].dir.y, lights[SEL_CAM].dir.z});
lights[SEL_CAM].camdir = normalized(calculate_vector_AtoB3(origen,
    p_dir_sel_cam));
```

En cuarto lugar, para evitar comprobar valores de booleanos globales, cuyos valores ya conocemos al inicio de `marraztu`, cada vez que dibujamos un poligono, selecciono las funciones que voy a utilizar para llevar a cabo el dibujado. `f_mostrar_normal_cara` y `f_mostrar_normal_vertice` son variables globales de "dibujar_poligono.c" que apuntan a funciones que se llaman para dibujar o no las normales. Las funciones `no_mostrar_normal_cara` y `no_mostrar_normal_vertice` son funciones vacías, sin ninguna línea, pero del mismo tipo que las que sí dibujan las normales, para que las variables mencionadas puedan apuntarlas. Utilizo un truco parecido con las funciones para comprobar que una cara es trasera. Por otro lado, en "dibujar_poligono.c" defino también un array con las direcciones de las funciones que dibujan cada polígono, y utilizando la función `indice_dibujar_poligono` selecciono la función correcta en función de los valores de `mostrar_caras_traseras`, `lineak` y `take_normals_of`. Las funciones son muy parecidas entre sí, pero para unificarlas tendría que poner condicionales comprobando los valores de las anteriores variables booleanas que definen el modo de dibujado, eso por cada polígono, que es justo lo que he pretendido evitar. Esta solución también tiene su pega, y es que con tres variables booleanas que he tenido que tener en cuenta conlleva a tener que implementar $2^3 = 8$ funciones. El número de funciones se duplica por cada variable booleana nueva que usemos para definir el estado del modo de dibujado, a menos que se puedan utilizar trucos específicos como con el dibujado de los vectores normales.

```
//Dependiendo del estado del programa, configurar las funciones de dibujado
f_mostrar_normal_cara = mostrar_normales_caras ? mostrar_normal_cara :
    no_mostrar_normal_cara;
f_mostrar_normal_vertice = mostrar_normales_vertices ?
    mostrar_normal_vertice : no_mostrar_normal_vertice;
f_is_looking_backwards = (proyection == PARALELO) ?
    is_looking_backwards_paralelo : is_looking_backwards_perspectiva;
f_dibujar_poligono = tabla_dibujar_poligono[indice_dibujar_poligono()];
```

Por último, dependiendo de los valores de `denak` y `objektuak`, dibujo todos los objetos, solo el seleccionado, o solo el triangulo seleccionado del objeto seleccionado, llamando a la función auxiliar `dibujar_objeto`. Después de eso, dibujo (siempre) todas las cámaras y también un "diamante" (dos piramides invertidas) amarillo en la posición de la bombilla, siempre que esté encendida.

En la función `dibujar_objeto` calculo una única vez el producto matricial entre la matriz del objeto y `mcsr`. Así, llamo a la función de dibujado de polígono elegida anteriormente y definida en "dibujar_poligono.c" con cada polígono del objeto. A partir de este punto, viene una cadena de llamadas a funciones definidas en "dibujar_poligono.c", "dibujar_triangulo.c" y "dibujar_linea_z.c".

```
void dibujar_objeto(
    object *optr, real_t *const mcsr_transf, const real_t *const mcsr,
    funcDibujarPoligono f_dibujar_poligono)
{
    unsigned int i;

    mxm(mcsr_transf, mcsr, optr->mptr->m);

    for(i = 0; i < optr->num_faces; ++i){
        //mcsr se sigue necesitando para los vectores normales
        f_dibujar_poligono(optr, optr->face_table + i, mcsr_transf, mcsr);
    }
}
```

En "dibujar_poligono.c" están definidas las funciones antes mencionadas de dibujado de normales, cálculo de caras traseras y dibujado de polígonos. Estas últimas, las más importantes, son muy parecidas entres si: calculan si la cara es trasera, dibujan el normal de la cara, y tomando los vértices del polígono de tres en tres (el primero siempre, y avanzando en uno los dos últimos) van dibujando los triangulos (por supuesto, teniendo en cuenta lo hablado sobre las variables globales). Cabe destacar que el cálculo de las intensidades (cuando toca llamar a la función `dibujar_triangulo_iluminado`) se hace en estas funciones, una única vez por vértice.

```
// TABLA DE VARIABLES GLOBALES BOOLEANAS QUE DEFINEN QUE FUNCION
// DE DIBUJADO DE POLIGONO UTILIZAR
//
// take_normals_of --> 0 == FACES // 1 == VERTICES
//
// mostrar_caras traseras | lineak | take_normals_of || Indice
//            0                0            0              0
//            0                0            1              1
//            0                1            0              2
//            0                1            1              3
//            1                0            0              4
//            1                0            1              5
//            1                1            0              6
//            1                1            1              7
```

```
//Por ejemplo, el primero:
// NO mostrar_caras_traseras - dibujar AREAS triangulos -
// take_normals_of FACES
void dibujar_poligono_0(
    const object *const optr, const face *const fptr,
    const real_t *const mcsr_transf, const real_t *const mcsr)
{
    unsigned int i;
    vertex *v1, *v2, *v3;
    bool es_trasera;
    punto3 center, pnormal, p1, p2, p3;
    vector N;
    color3 I1, I2, I3;

    //transformamos y pasamos el centro de la cara y el punto del
    //vector normal en SRLocal del objeto al SRCamara
    mxp3(&center, mcsr_transf, fptr->center);
    mxp3(&pnormal, mcsr_transf, fptr->N);

    //calculamos el vector normal de la cara
    N = normalized(calculate_vector_AtoB3(center, pnormal));

    //calculamos si la cara es trasera
    es_trasera = f_is_looking_backwards(center, N);
    if(es_trasera){
        return;
    }

    //dibujar vectores normales del poligono (face)
    //punto ndc_p1, ndc_pnormal;
    //calculateNDCp2(&ndc_p1, p1, &ndc_pnormal, pnormal);
    f_mostrar_normal_cara(center, pnormal, optr->rgb);

    //cada cara tiene como minimo tres vertices
    v1 = optr->vertex_table + fptr->vertex_ind_table[0];
    v2 = optr->vertex_table + fptr->vertex_ind_table[1];
    v3 = optr->vertex_table + fptr->vertex_ind_table[2];
    mxp3_mostrar_normal(&p1, v1, mcsr_transf, optr->rgb);
    I1 = calculate_intensity(p1, optr, N);
    mxp3_mostrar_normal(&p2, v2, mcsr_transf, optr->rgb);
    I2 = calculate_intensity(p2, optr, N);

    for(i = 3; ; ++i){
        mxp3_mostrar_normal(&p3, v3, mcsr_transf, optr->rgb);
        I3 = calculate_intensity(p3, optr, N);
        dibujar_triangulo_iluminado(p1, p2, p3, I1, I2, I3);
        if(i == fptr->num_vertices) break;
        p2 = p3;
        I2 = I3;
        v3 = optr->vertex_table + fptr->vertex_ind_table[i];
    }
}
```

En "dibujar_triangulo.c", está la función que dibuja solo el perímetro del triángulo y las que dibujan también su interior, ya sea con un color uniforme (anterior iteración) o con el cálculo de la intesidad según las luces y el material del objeto. La función que más nos interesa es esta última. Básicamente, he tenido que hacer una versión "iluminación" de cada función auxiliar de la

versión "color" para poder saber cual es la iluminación del punto superior, medio e inferior del triangulo y de los puntos de corte en cada altura. A partir de ahí, es muy parecido al anterior: interpolo las intensidades en los puntos de corte y llamo a `dibujar_linea_z_iluminado`. Cabe mencionar también la definición de $YCHANGE = 0.1$, la diferencia en altura entre las líneas horizontales. En las anteriores iteraciones simplemente decrementaba el valor de "y" en uno, pero para que incluso acercándonos bastante a los objetos se sigan viendo bien era necesario aumentar la densidad de puntos dibujados.

```
void dibujar_area_triangulo_iluminado(
    punto3 *const pgoiptr, punto3 *const perdiptr, punto3 *const pbeheptr,
    color3 *const Igoiptr, color3 *const Ierdiptr, color3 *const Ibeheptr)
{
    real_t y;
    punto3 c1, c2;
    color3 I1, I2;
    punto3 *pizqptr, *pderptr;
    color3 *Iizqptr, *Iderptr;

    // Dibujar todo el area del triangulo
    // Primera mitad del triangulo: del punto superior al medio
    for(y = pgoiptr->y; y > perdiptr->y; y -= YCHANGE){
        calcular_punto_corte_iluminado(y, pgoiptr, pbeheptr, Igoiptr,
            Ibeheptr, &c1, &I1);
        calcular_punto_corte_iluminado(y, pgoiptr, perdiptr, Igoiptr,
            Ierdiptr, &c2, &I2);
        calcular_punto_izq_der_iluminados(&pizqptr, &pderptr, &Iizqptr,
            &Iderptr, &c1, &c2, &I1, &I2);
        dibujar_linea_z_iluminado(y, pizqptr->x, pizqptr->z, pderptr->x,
            pderptr->z, *Iizqptr, *Iderptr);
    }
    // Segunda mitad del triangulo: del punto medio al punto inferior
    for(; y > pbeheptr->y; y -= YCHANGE){
        calcular_punto_corte_iluminado(y, pgoiptr, pbeheptr, Igoiptr,
            Ibeheptr, &c1, &I1);
        calcular_punto_corte_iluminado(y, perdiptr, pbeheptr, Ierdiptr,
            Ibeheptr, &c2, &I2);
        calcular_punto_izq_der_iluminados(&pizqptr, &pderptr, &Iizqptr,
            &Iderptr, &c1, &c2, &I1, &I2);
        dibujar_linea_z_iluminado(y, pizqptr->x, pizqptr->z, pderptr->x,
            pderptr->z, *Iizqptr, *Iderptr);
    }
    // Especialmente para el caso limite donde el punto medio y el inferior
    // esten a la misma altura es necesario para dibujar los puntos
    // (sino faltaria la recta inferior). Para los demas casos, solo
    // faltaria el punto inferior.
    if(perdiptr->y == pbeheptr->y){ //== y
        calcular_punto_izq_der_iluminados(&pizqptr, &pderptr, &Iizqptr,
            &Iderptr, perdiptr, pbeheptr, Ierdiptr, Ibeheptr);
        dibujar_linea_z_iluminado(y, pizqptr->x, pizqptr->z, pderptr->x,
            pderptr->z, *Iizqptr, *Iderptr);

    } else {    // un unico punto, del punto inferior a si mismo
        dibujar_linea_z_iluminado(y, pbeheptr->x, pbeheptr->z, pbeheptr->x,
            pbeheptr->z, *Ibeheptr, *Ibeheptr);
    }
```

```
}
```

En `dibujar_linea_z`, la versión de iluminación es calcada a la anterior, añadiendo la interpolación de la intensidad. Aquí también tengo definida la macro $XCHANGE = 0.1$, para aumentar la cantidad de los puntos dibujados. Estas dos macros juntas multiplican aproximadamente por 100 la cantidad de puntos dibujados. Naturalmente, eso ralentiza un poco la aplicación, pero es necesario para que los objetos se vean bien. Un detalle muy importante es poner un tope de 255 a las intensidades obtenidas al interpolar. Sino, al pasar esos valores (de coma flotante) a `glColor3ub`, que recibe `unsigned chars`, se castean perdiendo primero la parte fraccionaria (no es un gran problema) y a lo que queda se le aplica $mod\,256$, quedándose con el resto (un número entre 0 y 255). Si la intensidad resultante es mayor incluso que la intensidad máxima de 255, lo suyo es que se dibuje con ese valor de intensidad máxima (por ejemplo, si $I_i = 256$, no tiene sentido que ese punto pase a tener una intesidad nula, sino la máxima).

```
void  dibujar_linea_z_iluminado(
    real_t linea, real_t c1x, real_t c1z, real_t c2x, real_t c2z,
    const color3 I1, const color3 I2)
{
    real_t xkoord, zkoord;
    color3 I;
    //punto3 ndcp, p;

    const real_t difx = (c2x - c1x) / XCHANGE;
    const real_t zchange = (c2z - c1z) / difx;
    const color3 Ichange = (color3){(I2.r - I1.r)/difx, (I2.g - I1.g)/difx,
        (I2.b - I1.b)/difx};

    glBegin( GL_POINTS );
    for (xkoord = c1x, zkoord = c1z, I = I1; xkoord <= c2x;
    xkoord += XCHANGE){
        //Clamping: valor maximo para los componentes rgb == 255
        I.r = fmin(I.r, 255); I.g = fmin(I.g, 255); I.b = fmin(I.b, 255);
        glColor3ub(I.r, I.g, I.b);

        //p = (punto3){xkoord, linea, zkoord};
        //calculateNDCp1(&ndcp, p);
        //glVertex3f(ndcp.x, ndcp.y, ndcp.z);
        glVertex3f(xkoord, linea, zkoord);

        zkoord += zchange;
        I = (color3){I.r + Ichange.r, I.g + Ichange.g, I.b + Ichange.b};
    }
    glEnd();
}
```

La otra parte importante es el tema de las transformaciones. Éstas se definen en "transformations.c". Tenemos transformaciones aplicadas a los objetos, a las cámaras en modo vuelo, modo análisis y transformaciones a la dirección del sol y a la posición de la bombilla. Las más interesantes son

las transformaciones a las cámaras en modo análisis, que requieren la implementación de la rotación alrededor de cualquier eje, ya que las transformaciones las realizo en el sistema de referencia global. Para ello están definidas "rotacion_global" y "calculateRotationMatrix". La primera utiliza el método de hacer rotaciones alrededor de los ejes X y Z, rotar el ángulo deseado, y después deshacer las rotaciones iniciales. La segunda utiliza la fórmula de Rodrigues, más eficiente, y el que utilizamos en esta última iteración. Una vez tenemos la matriz de rotación necesaria para orbitar alrededor del objeto, antes de la rotación la cámara se traslada de tal forma que está con respecto al origen tal y como estaba respecto al centro del objeto. Así, rotando y deshaciendo después el cambio de origen, obtenemos la transformación deseada (se calcula y se mete una única matriz con las traslaciones y la rotación comentadas en la lista de transformaciones de la cámara).

```
//Por ejemplo, orbitar horizontalmente
void x_aldaketa_cam_analisis(camara *camptr, punto3 center, int dir){
    vector axis;
    real_t angle;
    real_t mrotation[16];

    if(aldaketa != 't' && aldaketa != 'r') return;

    real_t mtraslation[16] = initialize_traslation(-center.x, -center.y,
        -center.z);

    angle = dir ? ANG_ROT : -ANG_ROT;
    axis = normalized(mobj_to_y(camptr->mptr->m));
    //rotacion_global(mrotation, axis, angle);
    calculateRotationMatrix(mrotation, axis.x, axis. y, axis.z, angle);

    real_t mcomplete[16];
    calculate_analisis_transformation(mcomplete, mrotation, mtraslation);
    objektuari_aldaketa_sartu_ezk(camptr, mcomplete);
}
```

En cuanto a las cámaras, "camaras.c" se limita a definir la función que toma la matriz de un objeto (o cámara) y devuelve la matriz de cambio de sistema de referencia asociada a ese objeto (mostrada antes al explicar la función `marraztu`). También están comentadas las funciones para pasar las coordenadas de los puntos en el mundo a los NDC (normalized device coordinates, en el intervalo $[-1, 1]$, utilizado por OpenGL), y las definiciones de las matrices de proyección (ya que, como he mencionado antes, realizamos esto con "glOrtho" y "glFrustum").

En "lights.c" se definen las constantes de las luces, la función que las inicializa, y la función para calcular la intensidad según lo visto en teoría (modelo de Phong, modelo de Warn para focos, atenuación por distancia...). El vector normal que recibe `calculate_intensity` puede ser de una cara o de un vértice, dependiendo de la función `dibujar_poligono_i` ($0 \leq i \leq 7$ e $i \in \mathbb{N}$) que se haya utilizado. Como los cálculos los realizamos en el sistema

de referencia del observador, para calcular el vector $V$ que va del punto al observador no hace falta más que calcular el vector (normalizado) que va del punto al origen.

```
//
// FUNCION PARA CALCULAR LA INTENSIDAD DE ILUMINACION EN UN PUNTO
// Precondiciones:  p y N estan en el SRObservador
//                  lights[...].campos/camdir estan en el SRObservador
color3 calculate_intensity(
    const punto3 p, const object *const material_info_ptr, const vector N)
{
    color3 I = {0, 0, 0};
    vector F;
    real_t dist_atten, dist;

    const color3 ka = material_info_ptr->ka;
    const color3 kd = material_info_ptr->kd;
    const color3 ks = material_info_ptr->ks;
    const real_t ns = material_info_ptr->ns;

    //Vector del punto al observador: en el SRObservador, V == p->O
    const vector V = normalized(opposite((vector){p.x, p.y, p.z}));
    const vector Lsun = opposite(lights[SUN].camdir);
    const vector Lbulb = normalized(calculate_vector_AtoB3(p,
        lights[BULB].campos));
    const vector Lselobj = normalized(calculate_vector_AtoB3(p,
        lights[SEL_OBJ].campos));
    const vector Lselcam = normalized(calculate_vector_AtoB3(p,
        lights[SEL_CAM].campos));

    //luz ambiental
    if(lights[ENVIRONMENT].onoff){
        accumulate_intensity(&I, 1, lights[ENVIRONMENT].I, ka);
    }

    //sol
    if(lights[SUN].onoff){
        accumulate_difused_intensity(&I, N, Lsun, lights[SUN].I, kd, 1);
        accumulate_specular_intensity(&I, N, V, Lsun, lights[SUN].I,
            ks, ns, 1);
    }

    //bombilla
    if(lights[BULB].onoff){
        dist = calculate_vector_module(calculate_vector_AtoB3(p,
            lights[BULB].campos));
        dist_atten = distance_attenuation(dist);
        accumulate_difused_intensity(&I, N, Lbulb, lights[BULB].I,
            kd, dist_atten);
        accumulate_specular_intensity(&I, N, V, Lsun, lights[BULB].I,
            ks, ns, dist_atten);
    }

    //objeto seleccionado
    if(lights[SEL_OBJ].onoff){
        F = lights[SEL_OBJ].camdir;
        dist = calculate_vector_module(calculate_vector_AtoB3(p,
            lights[SEL_OBJ].campos));
        dist_atten = distance_attenuation(dist);
        accumulate_spot_intensity(
            &I, N, V, Lselobj,
```

```
            F, lights[SEL_OBJ].aperture_cos, lights[SEL_OBJ].I,
            kd, ks, ns, dist_atten);
    }


    //camara seleccionada
    if(lights[SEL_CAM].onoff){
        F = lights[SEL_CAM].camdir;
        dist = calculate_vector_module(calculate_vector_AtoB3(p,
            lights[SEL_CAM].campos));
        dist_atten = distance_attenuation(dist);
        accumulate_spot_intensity(
            &I, N, V, Lselcam,
            F, lights[SEL_CAM].aperture_cos, lights[SEL_CAM].I,
            kd, ks, ns, dist_atten);
    }

    return I;
}
```

En "load_obj.c" tenemos la función que lee los ficheros .obj y además calcula al inicio una única vez (en el sistema local del objeto) los normales de los vértices, las caras, los centros de las caras, etc.

En "matrix_operations.c" están definidas operaciones matemáticas entre puntos, vectores y matrices que utilizo en la mayoría de los demás archivos.

En "seleccionar_material.c" defino la función que utilizo cuando el usuario quiere cargar nuevos objetos. Muestra por pantalla la lista de materiales predefinidos en "lights.h", y sigue preguntando mientras el usuario no introduce el número correspondiente a un material.

Con respecto a los archivos header, contienen las declaraciones de las definiciones en su .c correspondiente que otras unidades de traducción necesitan en el proceso de enlace. Además de ello, están las macros en "camaras.h" para el volumen de proyección, ya sea ortográfica o en perspectiva, y las macros de los materiales de los objetos en "lights.h".

Por último, "obj.h" contiene las declaraciones de los tipos fundamentales utilizados: puntos, vectores, colores, vértices, caras, objetos, luces y lista enlazada de matrices de transformación. He modificado ligeramente las estructuras quitando los campos que no he utilizado. Por ejemplo, he preferido calcular los puntos en coordenas del sistema del observador en variables locales (salvo en el caso de las luces). También he añadido otros: `center` para caras y objetos (para dibujar los vectores de las caras saliendo de dicho punto y para hacer "look at" al centro de los objetos en modo análisis) y `num_transf_analisis_mode` en los objetos (número de transformaciones en modo análisis) para que se pueda saber cuando se hace el "undo" que deshace la transformación "look at" inicial y así volver automáticamente al modo vuelo. Por último, también he definido unos typedefs y enums para mejorar la legibilidad (`bool`, `LightType`, `camara`...). El más interesante es

`real_t`. En todas las partes del código donde usaba `double` lo he sustituido por este typedef, lo que me ha permitido pasar de `double` a `float` cambiando un simple typedef, comprobando que el resultado visual es el mismo pero con 4B en vez de 8B por cada variable de punto flotante utilizado. Esto también ha llevado a la necesidad de definir la macro `REAL_TYPE_SPECIFIER`, ya que para funciones estándar como `printf` y `scanf` no sirve utilizar `%lf` como especificador de `float`s, por lo que esta macro y `real_t` tienen que ser compatibles (y por eso están en un mismo sitio, uno al lado del otro en el código).

# 7 Trabajo futuro

Aquí indico la lista de ciertos cambios o pruebas que no he hecho, principalmente por ahorrar tiempo:

1. Función para **liberar** toda la **memoria reservada** en el heap. En teoría, no es estrictamente necesario, porque el sistema operativo ya se encarga de liberar la memoria de los procesos que acaban. No obstante, suele ser una buena práctica liberar la memoria nosotros mismos, poniendo un `free` por cada `malloc`. Tendríamos que llamar a esta función en el `case` correspondiente a ESC en `teklatua`, antes de llamar a `exit`. Además, seguramente `glut` también tenga otra función para pasarle la función callback al cual llamar cuando el usuario pulsa la cruz que cierra la ventana, y ahí también habría que llamarla (en todos los puntos donde nuestra aplicación termina).

2. Intentar implementar la **proyección manualmente**, en CPU. Creo que el problema en modo "líneas" podría ser debido a la falta de gestión de valores muy pequeños de la cuarta componente "w".

3. Aprender un poco sobre **makefile**s y hacer uno, para mayor comodidad y menos tiempo de espera a la hora de compilar el programa.

4. Probar la **técnica de iluminación de polígonos de Phong**, intersectando los vectores normales en los puntos de corte de "dibujar_triangulo.c" y en la línea horizontal de "dibujar_linea_z.c", y calculando la intensidad en cada punto. Para hacerlo en la CPU puede que sea demasiado (con $XCHANGE = YCHANGE = 0.1$ desde luego que sí).

5. Hacer la implementación de **"ligths.c" más extensible**, para poder añadir cualquier número de luces más cómodamente.

6. Al calcular los vectores normales de los vertices en "load_obj.c" recorro todas las caras por vértice. Podría hacerlo más eficiente guardando previamente una **tabla de los índices de las caras por vértice**, parecido a lo que hacemos al revés (array de índices de vértices por cara) pero solo al cargar los objetos. Aunque habría que ver si eso es más eficiente que la implementación sencilla que tengo ahora.

7. Tratar de extraer lo común entre las funciones de dibujado de poligonos a funciones o macros auxiliares...

8. ... o pensar otra manera (¿tal vez una **máquina de estados**?) en vez de variables booleanas para guardar el estado de la aplicación, para evitar la necesidad de duplicar el número de funciones de dibujado de polígonos o añadir comprobaciones innecesarias.

9. Dar la opción de meter los datos del **material que quiera** el usuario cuando pulse "f", aparte de la opción que le doy ahora de elegir un material predeterminado.

10. Añadir opciones extra como **eliminación** de objetos, cámaras y luces.

Pero aparte de los puntos anteriores, que son más o menos interesantes, el paso clave a futuro sería **aprender OpenGL y lenguajes de shaders** en profundidad para quitarle trabajo innecesario a la CPU y aprovechar la **GPU**.

# References


[1] J. Makazaga eta A. Lasa, "Ordenadore Bidezko Irudigintza", UEU, Concha Jenerala 25, 4. Bilbo, 1998. (519.674 MAK)

[2] "Phong Model Material Properties for Some Common Materials"

`https://people.eecs.ku.edu/~jrmiller/Courses/672/InClass/3DLighting/MaterialProperties.html`