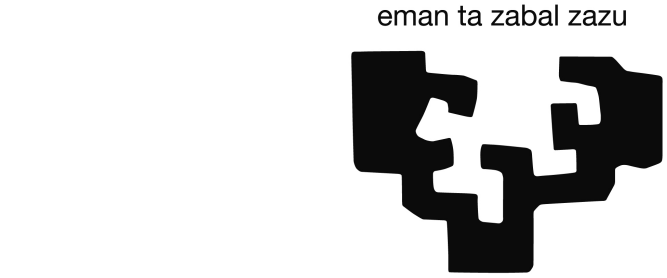
Cámara e iluminación



Unai Lizarralde Imaz Noel Arteche Echeverria
Edu Vallejo Arguinzoniz

24 de diciembre de 2018

Resumen

Esta práctica contempla la utilización de la cámara y la iluminación de los objetos en el espacio. La cámara va a proporcionar la posibilidad de fijar un punto de vista a voluntad y modificable por las transformaciones geométricas que en la anterior práctica se han explorado. La iluminación se encargará de mostrar con el volumen que corresponde a los objetos en la proyección, cuyo efecto ayudará a distinguir distintas caras del polígono, los surcos y las superficies irregulares con mayor precisión de forma mas realista.

Índice

1. Introducción

Este documento sirve de acompañamiento al final de la práctica de la asignatura Gráficos por computador, y viene a documentar el progreso en el desarrollo de las cámaras y los sistemas de iluminación implementados en la práctica.

Cabe destacar, antes de nada, que hay una diferencia fundamental respecto a la última entrega —donde se implementaban transformaciones geométricas sobre los objetos—. Hemos decidido separar de manera clara los componentes de nuestro programa, creando, por un lado, un conjunto de funciones y métodos basados en OpenGL que constituyen un pequeño motor gráfico (bautizado con el nombre de "MKZ Engine"); y, por otro lado, un programa que llama a las funciones facilitadas por la interfaz de este motor para implementar los casos de uso que se exigen en el enunciado de la práctica.

Por lo demás, la principal mejora en cuanto al resultado final ha sido la implementación de cámaras y luces. Hemos incluido la posibilidad de cambiar entre distintos tipos de proyecciones, distintas cámaras, el movimiento de éstas, la incorporación de un Modo Análisis que permite moverse alrededor de un objeto, etc. Asimismo, se ha desarrollado un sistema de iluminación con distintas fuentes de luz y transformaciones sobre ellas.

Este documento contiene información sobre los fundamentos teórico-matemáticos aplicados en el procedimiento de obtención de los resultados deseados, una descripción general de cómo se ha implementado el código para aclara ciertos factores de decisión (sobre todo por parte del MKZ Engine) y, finalmente, información para la correcta ejecución del programa (casos de uso, funcionalidades...).

Este documento se compone de información clave de sus funcionalidades e la implementación de los distintos casos de uso. Para saber acerca de como utilizar cada operación disponible, se ha incluido un manual de usuario aparte que se centra exclusivamente en indicar todo lo relacionado con las opciones a disposición del usuario. Se pretende facilitar al usuario las pautas a seguir de forma individual para explorar todas las posibilidades que ofrece con esta separación.

2. Funcionamiento de la cámara

La cámara es un punto de visualización del entorno situado en el espacio con una matriz de transformación, y un volumen de visión asociado. Para incluir la cámara en la práctica, se ha interpretado la cámara como un objeto sin volumen (no tiene "mesh") en el espacio, y su representación se realiza

mediante una matriz que proporciona la información necesaria para situarla en el espacio, junto con sus ejes. Con ello, se le pueden aplicar transformaciones a la cámara. La cámara, además, siempre estará visualizando en la dirección de su eje -z.

La matriz de transformación esta formado por tres vectores y un punto
El vector del eje x:

$$\overline{x_c} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{pmatrix} \tag{1}$$

El vector del eje y:

$$\overline{y_c} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ 0 \end{pmatrix} \tag{2}$$

El vector del eje z:

$$\overline{z_c} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ 0 \end{pmatrix} \tag{3}$$

Posición p de la cámara en el espacio:

$$p = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \tag{4}$$

Con estos tres vectores se expresa la matriz de transformación de la cámara de la siguiente forma:

$$M_{\mathrm{C}} = \begin{pmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5}$$

La cámara, como es un objeto cualquiera, puede recibir cualquier tipo de transformación en modo vuelo y lo hará en coordenadas locales (la matriz de la transformación se multiplica por la derecha). No obstante, en el modo análisis tiene ciertas restricciones; en concreto, solo tiene habilitado poder trasladarse en el eje z para acercarse o alejarse del objeto y rotar en los ejes.

Todas estas transformaciones se realizan sobre la matriz de transformación de la cámara, nunca sobre el de cambio de sistema de referencia.

Por consiguiente, un objeto puede actuar como si de una cámara se tratara, y visualizar el espacio que lo rodea con la misma estructura aquí arriba planteada. Como en este caso se necesita visualizar desde el propio objeto, la matriz de cambio de referencia del mundo a la cámara será al propio objeto.

2.1. Cambio de sistema de referencia a la cámara

Para mirar a los objetos desde la perspectiva de la cámara (o desde la del objeto) es necesario hacer por un lado una traslación para definir el origen del nuevo sistema de referencia en la cámara y, después, hacer un cambio de base del sistema de referencia del mundo al de la cámara. El resultado es el siguiente:

$$M_{\text{SRC}} = \begin{pmatrix} x_1 & x_2 & x_3 & -p^T \cdot x_c \\ y_1 & y_2 & y_3 & -p^T \cdot y_c \\ z_1 & z_2 & z_3 & -p^T \cdot z_c \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6}$$

Esta matriz se multiplica a la matriz de transformación del objeto por la izquierda para hacer el cambio del sistema de referencia. Esta multiplicación define el MODELVIEW.

$$M_{\text{SRC}} \cdot M_{\text{O}} = \begin{pmatrix} x_1 & x_2 & x_3 & -p^T \cdot x_c \\ y_1 & y_2 & y_3 & -p^T \cdot y_c \\ z_1 & z_2 & z_3 & -p^T \cdot z_c \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} o_{x1} & o_{y1} & o_{z1} & o_{p1} \\ o_{x2} & o_{y2} & o_{z2} & o_{p2} \\ o_{x3} & o_{y3} & o_{z3} & o_{p3} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7}$$

Cuando se trata de un objeto actuando como cámara, se realiza de la misma manera y el MODELVIEW se determinaría mediante la siguiente multiplicación.

$$M_{\text{SRO}} \cdot M_{\text{O}} = \begin{pmatrix} o_{x1} & o_{x2} & o_{x3} & -o_{p1} \cdot o_x \\ o_{y1} & o_{y2} & o_{y3} & -o_{p2} \cdot o_y \\ o_{z1} & o_{z2} & o_{z3} & -o_{p3} \cdot o_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} o_{x1} & o_{y1} & o_{z1} & o_{p1} \\ o_{x2} & o_{y2} & o_{z2} & o_{p2} \\ o_{x3} & o_{y3} & o_{z3} & o_{p3} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

2.2. Modo vuelo

El modo vuelo consiste en modificar en coordenadas locales de la cámara su posición e dirección con las transformaciones geométricas previamente

empleadas en los objetos, pero esta vez sobre la cámara. Entonces, la cámara en este modo es independiente del sistema de referencia del mundo y tiene libertad de moverse por el espacio con total soltura sobre sus propios ejes. Una de las transformaciones, el escalado, sirve para cambiar el volumen de visión y se desarrolla en el apartado 2.4.

Las matrices de transformación siguen siendo las mismas que para los objetos, por lo tanto, suponiendo que se realiza una transformación definida mediante la matriz M_{T}, la nueva matriz asociada a la cámara será la siguiente:

$$M_{\mathrm{C}}{}' = M_{\mathrm{C}} \cdot M_{\mathrm{T}} = \begin{pmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} t_{x1} & t_{y1} & t_{z1} & t_{p1} \\ t_{x2} & t_{y2} & t_{z2} & t_{p2} \\ t_{x3} & t_{y3} & t_{z3} & t_{p3} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9}$$

2.3. Modo análisis

El modo análisis centra la mirada de la cámara seleccionada en el objeto seleccionado. En este modo se pretende poder realizar transformaciones que permitan, siendo el objeto el punto central de la mirada, observarlo desde distintas perspectivas sin perderlo nunca de vista. La excepción es cuando el objeto seleccionado y la cámara seleccionada son lo mismo; en ese caso no se puede utilizar el modo análisis, puesto que sería estar analizando la propia cámara, y además, se deshabilitan todas las transformaciones de todos los objetos, sirve únicamente para visualizar el mundo desde la perspectiva del objeto. Para modificar la cámara y ponerlo mirando al objeto hay una modificación de sus componentes que inicializa la observación en modo análisis, expuesto a continuación.

Rotación encima del objeto. Guardar distancia y y rotar con respecto al eje y de la cámara en global.

Conocer la dirección en la que la cámara necesita fijarse para estar visualizando el objeto directamente. Siendo el punto del objeto O y el punto de la cámara P en coordenadas globales, se obtiene la dirección d normalizada

$$\overline{d} = \frac{O - P}{\|O - P\|} \tag{10}$$

De aquí se pueden obtener, realizando en este orden las operaciones, cada eje normalizado nuevo de la cámara

$$x_c = \frac{\overline{d} \times \overline{y_c}}{\|\overline{d} \times \overline{y_c}\|} \tag{11}$$

$$y_c = \overline{x_c} \times \overline{d} \tag{12}$$

$$z_c = -\overline{d} \tag{13}$$

Este modo está comprometido a observar con fidelidad y precisión el objeto; por tanto, las únicas transformaciones permitidas son la traslación en el eje z de la cámara y la rotación en coordenadas globales. La razón para no permitir la traslación en el resto de ejes se debe a que la distancia de la cámara al objeto se modifica en un eje distinto a z, por ello, si se desea realizar una exploración de la superficie del objeto se recurre a la rotación. El resto de transformaciones carecen de propósito para este modo, por lo tanto, no están habilitados.

En cuanto a la rotación, la cámara se posiciona encima del objeto con una traslación, se rota con respecto al eje y de la cámara en local y se vuelve a reposicionar en el punto correspondiente mediante una traslación.

2.4. Cambio del volumen de visión de la cámara

La cámara tiene un volumen de visión que restringe la cantidad de espacio que considera a la hora de visualizar los objetos. En definitiva, todo aquello que no este dentro del volumen de visión no podrá percibirlo la cámara, mientras que todo aquello que sí lo haga será por que esta dentro de ese volumen de visión. Este volumen es variable y, a pesar de sus valores de inicialización, se puede modificar mediante una transformación local: El escalado de la cámara.

Desde el punto de vista geométrico, al aumentar el tamaño multiplicando a la matriz (no la del cambio de sistema de referencia sino la de transformación) de la cámara por la matriz de escalado resulta que se agranda la cámara y como consecuencia los objetos se vuelven menos significativos en cuanto al espacio que cubren en el volumen de visión; de la misma forma, si se reduce el tamaño de la cámara también lo hará el volumen de visión en consecuencia. Este efecto está estrechamente ligado a la formación del MODELVIEW, donde se multiplica la matriz de cambio de sistema de referencia por la matriz de transformación del objeto, en ese orden.

Al ser el escalado local, sucede que a la hora de calcular la matriz de de cambio de sistema de referencia es la inversa de está multiplicación por la traslación al origen de la cámara. Siendo a, b y c coeficientes de escalado (a multiplica al eje x, b multiplica al eje y y c multiplica al eje z), tanto para

aumentar como para disminuir ocurre que

$$M_{\mathrm{SCR}} = (M_{\mathrm{C}} \cdot M_{\mathrm{E}})^{-1} \cdot M_{\mathrm{T}} = \left(\begin{pmatrix} x_1 & y_1 & z_1 & p_1 \\ x_2 & y_2 & z_2 & p_2 \\ x_3 & y_3 & z_3 & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right)^{-1} \cdot \begin{pmatrix} 0 & 0 & 0 & -p_1 \\ 0 & 0 & 0 & -p_2 \\ 0 & 0 & 0 & -p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (14)$$

es equivalente por propiedad de inversa de multiplicación de matrices a

$$M_{\mathrm{SCR}} = {M_{\mathrm{E}}}^{-1} \cdot {M_{\mathrm{C}}}^{-1} \cdot M_{\mathrm{T}} \quad (15)$$

Como la matriz inversa de un escalado produce el efecto opuesto, cuando se incremente el tamaño de la cámara la matriz de transformación del escalado al invertirse causará que los objetos reduzcan su tamaño y, de forma similar, cuando se reduce el tamaño de la cámara, la matriz de transformación del escalado aumentará el tamaño de los objetos. Como resultado, a pesar de que el volumen de visión sigue siendo el mismo en unidades absolutas, para los objetos habrá un mayor volumen de visión en unidades relativas (aumento) o menor (disminuir).

Por otro lado, en una aproximación mas algebraica, hay que tener en cuenta que en el pipeline la matriz de proyección precede a la del "model-view". Entonces, ocurre que la matriz inversa del escalado multiplicará por la derecha a la de proyección cuyo efecto no es ni mas ni menos que ampliar o encoger los parámetro (n,f,r,l,t,b). Precisamente, sucede que en la diagonal de la matriz de proyección se tiene en el denominador $(r-l)$, $(t-b)$ y $(f-n)$.

Cuando se incrementa en el escalado el tamaño de la cámara, en la inversa la matriz multiplicara por el coeficiente c a la diferencia, lo que aumentará dicho volumen de visión. Por el contrario, al reducir su tamaño, en la inversa el coeficiente c dividirá al denominador, y reducirá el volumen de visión en consecuencia. Siendo la resta entre los límites $(a_i - b_i)$ para $i = 1, 2, 3$

Incrementar el tamaño de la cámara:

$$(a_i - b_i)' = c(a_i - b_i) \quad (16)$$

Disminuir el tamaño de la cámara:

$$(a_i - b_i)' = \frac{(a_i - b_i)}{c} \quad (17)$$

2.5. Proyecciones

Un ordenador muestra su contenido en un panel 2D, y, por ello, necesita proyectar un objeto renderizado en 3D a uno en 2D para visualizarlo a través

del monitor. Primero, transforma el objeto de las coordenadas del observador a coordenadas de recorte ("clip coordinates"). Posteriormente, estas coordenadas se transforman a Volumen de Vista Normalizado ("normalized device coordinates", NDC acortado) dividiendo por la cuarta componente w todos los componentes de cada vector. El NDC consiste en un cubo que se extiende en el interior de cada coordenada menor que la unidad.

En caso de la perspectiva hay que tener en cuenta de que los vértices del objeto no se deben tener en consideración; para solucionarlo, la técnica del "frustum culling" descarta si se va incluir el vértice o no. Si alguna coordenada de recorte es menor a la coordenada $-w_c$ o mayor que w_c, no se dibuja el vértice. Este proceso se lleva a cabo antes de hacer la transformación a NDC.

Un vértice se dibujará si y solo si

$$-w_c \ < \ x_c, y_c, z_c \ < \ w_c \tag{18}$$

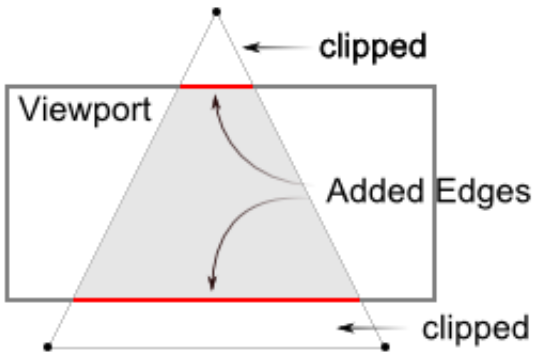


Figura 1: Límites del volumen de visión mediante "frustum culling"

Para la cámara se distinguen dos proyecciones: la proyección ortográfica y la proyección en perspectiva.

2.5.1. Perspectiva

Esta proyección mapea un punto 3D en una pirámide truncada (coordenadas del observador) a un cubo NDC. El rango de la coordenda x de $[l, r]$ a $[-1, 1]$, la coordenada y de $[b, t]$ a $[-1, 1]$ y la coordenada z de $[-n, -f]$ a [-1,1].

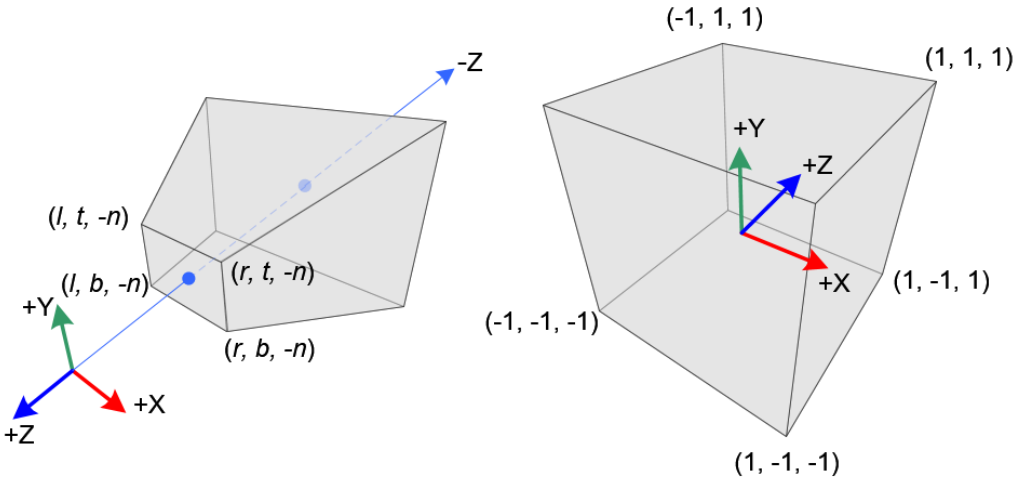


Figura 2: Perspectiva y volumen normalizado de vista

Hay que tener en cuenta que mientras que las coordendas del observador se definen a través del sistema de coordenadas de la mano derecha, el NDC emplea el sistema de coordenadas de la mano izquierda; es decir, la cámara está mirando en el eje $-z$, pero en el eje z en el NDC.

Un punto en el espacio se proyecta al plano "near" (plano de proyección). Se ilustra como se obtiene un punto interior a la pirámide proyectado en dicho plano a continuación.

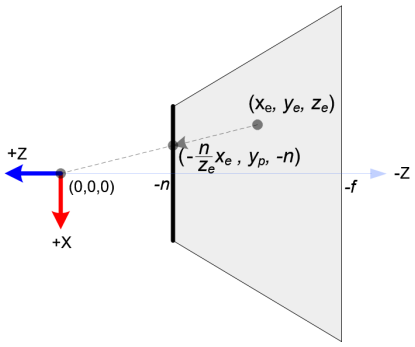


Figura 3: Vista por encima de la pirámide

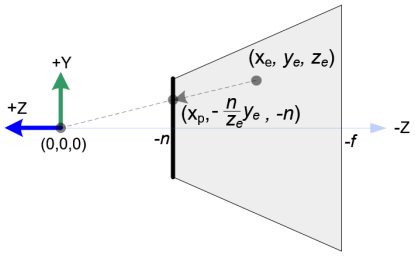


Figura 4: Vista lateral de la pirámide

Por la regla del ratio de triángulos similares se pueden obtener los puntos de x_p y y_p. Tras esto se debe realizar la transformación a NDC para construir la matriz de proyección. El resultado final es la siguiente matriz de proyección:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{19}$$

En el caso de que $r = -l$ y $t = -b$ la matriz de proyección se reduce a lo siguiente:

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{20}$$

Un último apunte acerca de los problemas que puede acarrear la amplitud del intervalo $[-n, -f]$. Cuánta mayor sea la longitud del intervalo, el buffer de profundidad sufre un problema de precisión (z-fighting), cuando el punto se encuentra cerca del plano $-f$. La forma de evitarlo consiste en minimizar la longitud del intervalo en la medida de lo posible.

2.5.2. Ortográfica

En esta proyección se mapean linealmente todos los puntos del espacio de visualización al cubo NDC. Se necesita escalar un cubo rectangular a un cubo y colocarlo en el origen. En esta proyección el componente w queda desestimado.

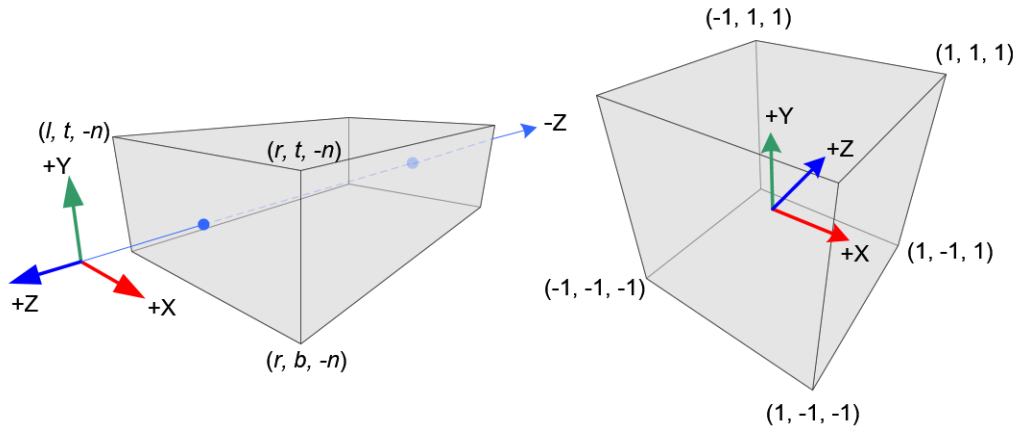


Figura 5: Ortográfica y volumen normalizado de vista

Para este proceso se resuelven las relaciones lineales y se obtiene que la matriz de proyección será:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{21}$$

Se puede simplificar en caso de que el volumen de visualización sea simétrico, esto es, $r = -l$ y $t = -b$.

$$\begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{22}$$

3. Funcionamiento de la iluminación

La iluminación se desarrolla entorno a como los objetos reflejan la luz incidente y los efectos visuales que genera la luz, así como sus propiedades. A la hora de comprender como se compone la estructura de funcionamiento, cabe resaltar que hay dos puntos centrales en este ámbito, las características propias de la luz y el reflejo que genera en cada caso la superficie del objeto iluminada. La iluminación reflejará, por consiguiente, dependiendo de que clase y que cualidades tiene la luz, así como el material que compone a los objetos.

Como añadido, estas fuentes de iluminación pueden provenir del mismísimo objeto seleccionado, y por tanto, tanto la posición como la dirección del objeto se aplicarán al cálculo de la luz sobre el resto de objetos. Una vez más, se puede comprender que una fuente de luz es un objeto mas en el espacio que tiene un rasgo distintivo: emite luz e ilumina la escena.

Para lograr conocer a que distancia se encuentran los objetos a la hora de dibujar se utiliza un z-buffer que da una medida de la distancia y permite dibujar en el orden de aparición de cercanía los objetos. Además, los polígonos son opacos para poder representar la incidencia de la luz en los mismos.

Una fuente de luz se compone de los siguientes parámetros para cada efecto de iluminación: Intensidad de cada color de la fuente (RGB), posición, atenuación, ángulo y la intensidad de distribución de la luz. El material de

un objeto debe indicar el reflejo (RGB) para cada efecto de iluminación que va a recibir.

La luz puede ser de tres tipos diferentes: direccional, puntual o focal. En el primer caso, todos los objetos reciben los rayos de luz desde la misma dirección, por tanto, es un comportamiento similar al del sol. Los rayos emitidos son paralelos entre sí y llegan hasta el infinito. Esta luz no recibe atenuación y únicamente se ve afectado por la rotación, pues se sitúa en el infinito.

En segundo lugar, una luz puntual es un punto en el espacio que emite en todas las dirección rayos de luz que se van atenuando. Las superficies generan un efecto de disminución gradual de la intensidad recibida cuanto mas lejos este de la fuente de luz el objeto. Guarda una similitud directa con el funcionamiento de una bombilla.

En tercera posición, una luz focal emite luz en forma de cono cuyo ángulo determina la amplitud del foco. Es una fuente luz directa y concentrada que ilumina todo aquello que se encuentra dentro del cono y que aplica la dirección de emisión y la atenuación con la superficie que impacta. La anchura del angulo también afecta en lo concentrado que va a se la iluminación que vaya a proveer la fuente de luz a los objetos que se encuentren dentro del área del cono.

3.1. Vectores normales del objeto

Cada objeto esta compuesto de vértices que, a su vez, generan caras del polígono al que pertenecen. Para conocer el comportamiento de la luz de forma aislada para cada cara o vértice, se necesita conocer previamente el vector normal. Con este vector, además de la dirección en la que está mirando, se puede decidir entre si la luz golpea esa cara o vértice y, si lo hace, el ángulo entre el vector normal y el vector dirección de la luz.

Existen dos tipos de formas de iluminar los objetos. Por un lado, se pueden considerar los vectores normales de cada cara del polígono para conocer el efecto que produce la iluminación sobre el objeto, conocido como iluminación “flat”. Por otro lado, en busca de una mayor precisión en el efecto de iluminación, se puede evaluar la actuación de la luz en cada vértice, requiriendo por ello el vector normal de cada vértice. Esta última alternativa se denomina iluminación “smooth”.

Los vectores normales de cada cara del polígono y cada vértice se calculan inmediatamente cuando se cargan los objetos. La metodología consiste en hacer uso del producto vectorial para obtener los vectores normales de cada polígono y, después, calcular los de cada vértice como la suma de los vectores normales de cada cara del polígono en la que se encuentran.

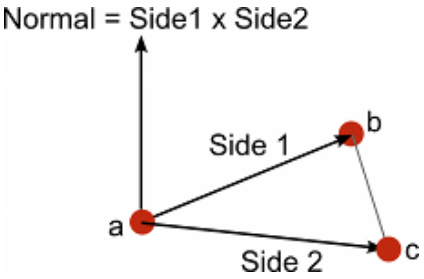


Figura 6: Normal de una cara del polígono dados tres de sus vértices

Sean a, b y c tres vértices que tiene un polígono, se definen los siguientes dos vectores $\vec{v}$ y $\vec{w}$ de la siguiente manera:

$$\vec{v} = b - a = \begin{pmatrix} b_x \\ b_y \\ b_z \\ 1 \end{pmatrix} - \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} b_x - a_x \\ b_y - a_y \\ b_z - a_z \\ 0 \end{pmatrix} \tag{23}$$

$$\vec{w} = c - a = \begin{pmatrix} c_x \\ c_y \\ c_z \\ 1 \end{pmatrix} - \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} c_x - a_x \\ c_y - a_y \\ c_z - a_z \\ 0 \end{pmatrix} \tag{24}$$

El vector normal $\vec{n}$ se obtiene del producto vectorial de ambos y se normaliza

$$\vec{n} = \frac{\vec{v} \times \vec{w}}{\|\vec{v} \times \vec{w}\|} \tag{25}$$

Finalmente, para determinar el vector normal de cada vértice se suman los vectores normales de cada polígono en los que aparece. Siendo los k polígonos p_i con los vectores normales n_i, el vector normal del vértice normalizado $\vec{n_v}$ se calcula de la siguiente manera:

$$\vec{n_v} = \frac{\sum_{i=0}^{k} n_i}{\|\sum_{i=0}^{k} n_i\|} \tag{26}$$

3.2. Inestabilidad en los vectores normales al escalar la cámara

Previamente, en la sección 2.1, se ha presentado una forma de cambiar el volumen de visión que consistía en un escalado en coordenadas locales de la cámara. La raíz del problema no es otro que la inconsistencia que genera esta transformación de la cámara sobre los vectores normales del objeto, puesto que se distorsionan y dejan de ser perpendiculares y, en consecuencia, no son vectores normales. En las fuentes de iluminación genera un problema de posicionamiento e dirección de la luz como secuela de esta práctica, que habría que escalarlo para poder reposicionar e dirigir la luz de forma adecuada. Por todo ello, este método queda descartado.

El mismo problema ocurre cuando se escala un objeto, no obstante, en este caso, se recalculan los vectores normales del objeto otra vez si el escalado es no uniforme (se escala en un solo eje o no por la misma constante en ambos ejes).

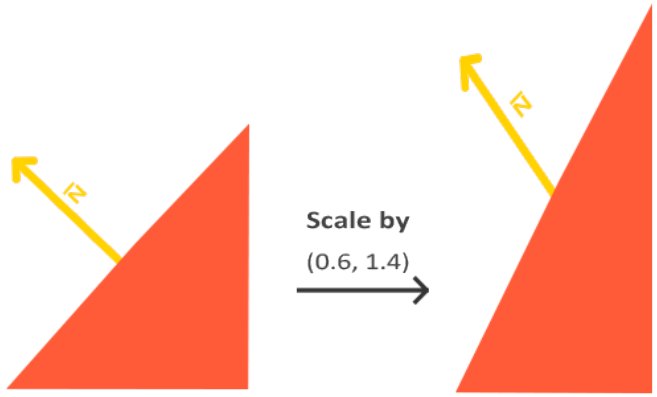


Figura 7: Resultado de un escalado no uniforme en un objeto

Para dar solución a este problema, se van a modificar las constantes de la matriz de proyección, siendo estos (l, r) en el eje x, (t, b) en el eje y y $(-n, -f)$ en el eje z. Para el eje x y el eje y, se ha decidido que para mantener la simetría ambos se alejarán o se encogerán de forma simultánea; esto es, si aumenta el límite superior del intervalo, decrece en la misma medida el intervalo inferior del intervalo. El eje z, por su parte, solo podrá aumentar o disminuir su componente $-f$; la profundidad se podrá seguir creciendo hasta el infinito, aunque un crecimiento desbordante puede degenerar el resultado cerca del plano $-f$ ("z-fighting") que se ha mencionado anteriormente, y podrá disminuir hasta el plano $-n$, en dicha situación la matriz de pro-

yección serían un corte del plano $-n$ a los vértices de los objetos que corta (intersección del plano sobre el objeto).

3.3. Luz ambiental

La luz ambiental afecta a todo el espacio por igual. Para definir una luz ambiental se necesitan su posición e la intensidad I_a de cada color (RGB) y el reflejo ambiental K_a del material. Para cada objeto la luz ambiental se obtiene de la siguiente manera:

$$I_{amb} = K_a I_a \tag{27}$$

3.4. Luz difusa

Una fuente de luz al emitir sus rayos de luz el ángulo de incidencia juega un papel fundamental a la hora de determinar la cantidad de luz que va a recibir el objeto. Cuando una de estas luces impacta en el objeto lo hace con una dirección $\overline{L}$ que representa el vector normalizado obtenido de la resta entre el punto de impacto y el origen de emisión de la luz.

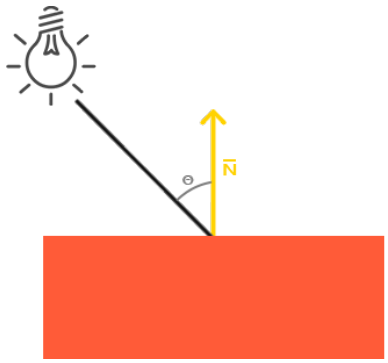


Figura 8: Representación de la luz difusa

Siendo $\overline{N}$ el vector normal normalizado del punto a representar y K_d el reflejo del material para una luz difusa, la luz difusa se fórmula así:

$$I_{diff} = K_a I_a + K_d I_d(\overline{NL}) \tag{28}$$

Es importante que tanto el vector normal como el vector diferencia estén normalizados para dar como resultado el coseno del ángulo que forma únicamente.

3.5. Luz especular

La luz especular pretende generar el reflejo dependiendo de tanto la dirección reflejada de la fuente de luz y la dirección de la perspectiva para causar unos efectos de brillo puntual. La brillantez n ("shinyness") se define mediante un exponente que acentúa y incrementa la intensidad del punto brillante cuanto mas crezca.

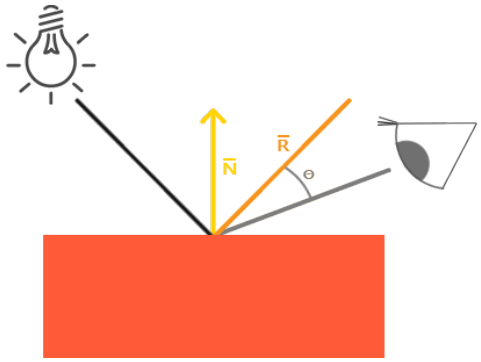


Figura 9: Representación de la luz especular

Para lograr el vector reflejado $\overline{R}$ necesario para calcular este efecto de iluminación, se utilizarán vectores intermediarios que dispondrán el escenario que permitirá calcular dicho vector reflejo. La posición de la fuente de luz se puede proyectar en el vector normal del polígono o del vértice, dependiendo del formato de iluminación activo. Dicho vector se considera el vector $\overline{P}$ de proyección de la posición de la fuente de luz en el vector normal. Esto supone abrir la posibilidad de calcular un vector normal de longitud delimitada o alargada al punto proyectado mediante el producto escalar del vector de dirección de la luz opuesto $-\overline{L}$ con el vector normal del vértice. Una característica de suma importancia es, ni mas ni menos, que el vector normal necesita estar normalizado, mientras que el vector opuesto de la dirección de la luz no debe estarlo. Esto asegura que la longitud del vector resultante alcanzará exactamente al punto proyectado.

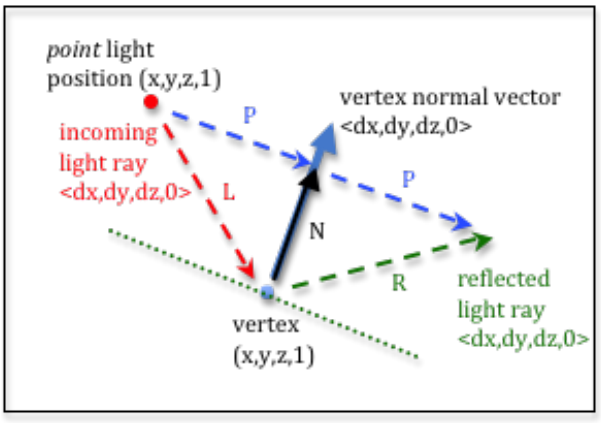


Figura 10: Vector de reflejo de la luz incidente

El cálculo de este vector se puede expresar de la siguiente forma, siendo $\overline{N}_p$ el vector proyectado y $\overline{P}$ el vector perpendicular al vector normal del vértice:

$$\begin{aligned} R &= \overline{N}_p + \overline{P} \\ &= \overline{N}(\overline{N} \cdot -\overline{L}) + (\overline{L} + \overline{N}_p) \\ &= \overline{N}(\overline{N} \cdot -\overline{L}) + (\overline{L} + \overline{N}(\overline{N} \cdot -\overline{L})) \\ &= 2\overline{N}(\overline{N} \cdot -\overline{L}) + \overline{L} \end{aligned} \tag{29}$$

Al final para poder calcular la luz especular de un objeto, teniendo en cuenta que todos los vectores están normalizados, se consigue de la siguiente ecuación para un observador en la dirección $\overline{V}$:

$$I_{esp} = K_s I_s (\overline{R} \cdot \overline{V})^n \tag{30}$$

3.6. Atenuación de la luz

Los rayos de luz recorren una distancia antes de incidir sobre la superficie de un objeto. Esta lejanía puede efectuar un pérdida de la intensidad de la fuente de luz si esta tiene asociada una atenuación. La atenuación se define a través de tres parámetros: atenuación constante a_0, atenuación lineal a_1 y atenuación cuadrática a_2. El coeficiente de atenuación $f(I)$ se calcula de la siguiente manera siendo la distancia d al objeto:

$$f(I) = min\left(1, \frac{1}{a_0 d + a_1 d + a_2 d^2}\right) \tag{31}$$

3.7. Modelo general de la iluminación

El modelo general consiste en combinar todas las luces previamente expuestas (ambiental, difusa y especular). De esta manera, cada objeto será iluminado tomando en cuenta cada una de las luces y ofrecerá un efecto mas realista e vistoso. Como se necesitan tener en cuenta todas las iluminaciones que recibe cada objeto, a la hora de calcular la reflexión final, sera sumar todos ellos. Se expresa de la siguiente manera para k componentes de luz afectando a un objeto:

$$I_{total} = K_a I_a + \sum_{i=0}^{k} f(I_i)(K_d I_{d,i}(\overline{NL}_i) + K_s I_{s,i}(\overline{R}_i \cdot \overline{V})^{n_i}) \tag{32}$$

3.8. Ray-Tracing

El trazado de rayos es una técnica de renderizado que consiste en generar la imagen haciendo uso del recorrido que cada rayo de luz sigue para cada pixel del plano de proyección y mostrar el comportamiento de la luz en las intersecciones con los objetos del espacio. El resultado crea una imagen realista y llena de detalle que simplemente provee de un acercamiento a la realidad sin parangón a costa, desafortunadamente, de un coste computacional superior a los métodos basados en "scanline rendering". Puede generar diversos efectos ópticos, entre los que destacan, reflexión, refracción y dispersión.

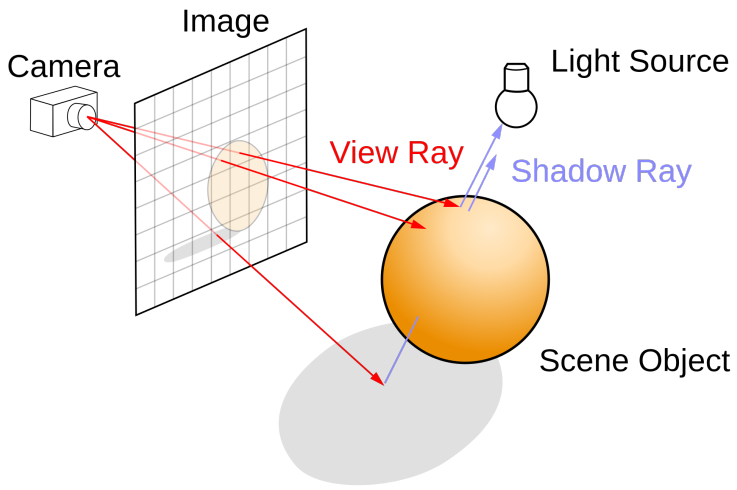


Figura 11: Rayos de luz trazados para cada pixel del plano de proyección

Existen cuatro tipos de rayos para calcular el ray tracing. En primera instancia, el rayo de visión que atraviesa cada pixel del panel de visualización

e intersecta con los objetos del espacio. Cuando lo hace genera un rayo de sombra, que no es otro que la dirección del punto de impacto a la fuente de luz. El rayo del reflejo (r) en la superficie que impacta, cuyo rebote influenciará los objetos con los que intersecte en su camino a su vez. Y, por último el rayo refractado, cuya dirección es la refracción (t) sobre la superficie del objeto. Mediante el cálculo siguiente se puede lograr la intensidad de la luz para cada pixel con k fuentes de luz aplicando el método del ray tracing recursivo.

$$I_{RT} = I_a K_a + + \sum_{i=0}^{k} f(I_i)(K_d I_{d,i}(\overline{N} \cdot \overline{L}_i) + K_s I_{s,i}(\overline{R}_i \cdot \overline{V})^{n_i}) + K_s I_r + K_t I_t \tag{33}$$

4. Metodología e estructura del código

Dada la creciente complejidad de la aplicación, mas casos de uso y integración de características gráficas cada vez mas complicadas, se ha recurrido a técnicas de desarrollo de software escalables.

Por un lado, para una gestión de versiones flexible, se ha adoptado el uso de **git** como herramienta de *versioning* y **github** como repositorio en la nube, el proyecto es publico y se puede visitar en la página `https://github.com/sophyelord/ComputerGraphics`. Esta decisión ha sido clave pues ha permitido gestionar versiones paralelas donde se implementan aspectos diferentes de la aplicación de forma independiente, lo que habilita el trabajo en equipo escalable.

Por otro lado se ha replanteado el diseño de la aplicación entera. Uno de los mayores problemas que se ha encontrado al iterar sobre la aplicación (mejorar las pasadas entregas) ha resultado ser el alto nivel de integración de la cual la aplicación constaba. Las dependencias de partes del código con el resto de la aplicación dificultaba los cambios hasta tal extremo que a veces el código resultaba intratable. Por ello se ha recurrido a un diseño por interfaces en la cual se definen diferentes niveles de abstracción y cada nivel (en teoría) solo depende del nivel de abstracción anterior (un nivel mas concreto) y soporta el siguiente nivel. Como se trata de una aplicación gráfica, una interfaz evidente es la de motor gráfico.

La aplicación que se especifica en la practica consta de casos de uso bastante específicos, por ello que se divida en niveles la implementación del caso de uso tal y como se ha especificado (encima de la interfaz del motor) y la complejidad técnica que este caso de uso supone (en el motor). De este modo el motor solo se encarga de los problemas centrales al generar gráficos por

computador y la aplicación que se programa encima del motor se encarga de los detalles específicos que se piden en la práctica.

Por último se ha usado *eclipse* como IDE para aliviar la carga de trabajo en los apartados mas repetitivos y las herramientas *make* y *gdb* para compilar y depurar la aplicación respectivamente.

5. Implementación

El trabajo presente consta de la implementación de dos aplicaciones: un motor gráfico que trata las dificultades de la generación de gráficos por computador y una aplicación que cumple con las especificaciones de la practica.

5.1. MKZ engine

El motor gráfico al cual denominamos MKZ engine, define una interfaz la cual permite alterar el estado del gestor de ventanas y de la escena que se visualiza, define estructuras de datos que representan diferentes actores de la escena (cámaras, luces y objetos con redes de polígonos) y exporta funciones herramienta para alterar las estructuras de datos.

El motor gráfico mismo esta diseñado también por niveles, lo cual permite, por ejemplo, cambiar de API gráfica re-implementando tan solamente 2 ficheros, o implementar técnicas de simulación gráfica mas avanzadas (*e.g. ray-tracing*) modificando el nivel que trata el dibujado.

El motor gráfico en su totalidad se compone de una colección de *headers* que contienen la interfaz con la que comunicarse con el motor y una librería que implementa las funciones del *header*.

En la entrega, por supuesto, se incluye el código fuente del motor junto con un Makefile que permite la compilación fácil de este.

He aquí una breve explicación de las interfaces y el funcionamiento del motor.

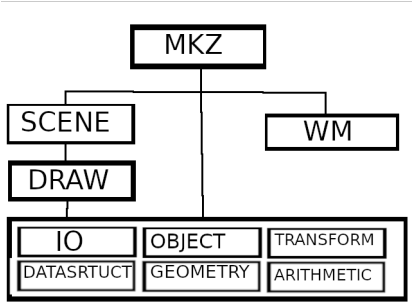


Figura 12: Diagrama del funcionamiento de la interfaz y del motor

5.1.1. WM

WN (Window Manager) es el modulo que se encarga de gestionar la creación de ventanas y de configurar las funciones de remitente (callback). En su implementación actual utiliza la librería glut para desarrollar sus acciones.

5.1.2. SCENE

Este modulo gestiona el estado de la escena, tiene funciones que permiten añadir y eliminar objetos y luces de la escena y establecer una cámara. La interfaz también permite definir los valores que son globales de la escena (independientemente de la cámara) y cambiar las variables (color de fondo, tipo de proyección, dibujado de polígonos, etc...). El modulo de escena también implementa la función de remitente display ya que gestiona la información de la escena. SCENE hace uso del modulo DRAW para cuando se llama a la función de display.

5.1.3. DRAW

Draw es un modulo que (salvo excepciones) carece de estado y implementa funciones de dibujado atómicas. Las funciones permiten establecer luces, dibujar objetos, cambiar de matriz de cámara, limpiar pantalla, establecer modo de dibujo etc. Draw depende de la API gráfica, que es este caso es OpenGL.

5.1.4. Herramientas

El resto de módulos complementan a los 3 módulos principales. Son funciones herramienta y definiciones que son útiles para implementar métodos

mas complicados. Algunas definiciones y herramientas se exportan en la interfaz puesto que se considera que pueden ser útiles para manipular las estructuras de datos y algunas definiciones son necesarias para utilizar las interfaces principales.

Es interesante destacar que las funciones y definiciones exportadas en la interfaz son un subconjunto reducido de las que se definen en el motor dado que muchas definiciones (e.g. estructuras de datos) son solo para uso interno.

5.2. Práctica

La aplicación que implementa la especificación de la práctica esta programada basándose en el motor gráfico y consta de 4 ficheros.

5.2.1. main

Contiene la función de entrada a la aplicación y su finalidad es inicializar la aplicación y entrar en el bucle principal. La inicialización consiste en inizializar el motor gráfico y establecer la función de remitente del teclado.

5.2.2. io

Como su nombre indica, el fichero io se encarga de la interacción con el usuario mediante input y output en el stdin y stdout. io contiene la función del teclado que mapea una funcionalidad a cada tecla definida en la especificación de la práctica. Estas funcionalidades se acceden a través de la interfaz que exporta el fichero KG.

5.2.3. objects

En este módulo se definen los objetos y estructuras de datos que se manipulan en el módulo principal y operaciones básicas sobre estos objetos. Entre otras cosas se define un *wrapper* para los objetos del motor gráfico que habilita guardar matrices de transformación pasadas (para la operación deshacer) y enlazar objetos de modo que la transformación al objeto padre afecta a todos los objetos hijos, útil para cámara del objeto y foco asociado al objeto. También se define una lista enlazada doble y una pila de matrices.

5.2.4. KG

Este es el módulo principal de la aplicación y se encarga de implementar los casos de usos tal y como se han especificado. El *header* del módulo exporta

métodos que se corresponden con cada interacción realizable, estos métodos se llaman desde el fichero de io cuando sucede un evento de teclado concreto.

KG encapsula el estado interno de la aplicación (modo, alcance y objetivo de transformación) y los objetos y luces cargados. La mayoría de operaciones hacen uso extensivo de las interfaces del motor o de los objetos lo cual resulta en un código de alto nivel muy legible.

6. Implementación de las funcionalidades y características

Para comprender específicamente como se han resuelto a nivel de programación los procedimientos establecidos para lograr obtener activas las funcionalidades de tanto la cámara como de la luz, se va a presentar los algoritmos y las inicializaciones e parámetros escogidos relevantes realizados, para explicar como funciona el programa sobre la estructura vertida de forma específica para cada problema presentado.

6.1. Pipeline OpenGL

El pipeline es la secuencia de tareas que realiza para renderizar la imagen final OpenGL. Los apartados que lo componen son la matriz de proyección y la matriz del modelview, que determinan como va a ser la imagen resultante dadas las características de ambas.

En primer lugar se comprueba cual es la proyección que se va establecer (en la variable global projection mode) y, en base a eso, se asigna una matriz de proyección u otra. Tras esto, se especifican las propiedades de la matriz de proyección para el caso que corresponda; es decir, si es una proyección ortográfica o una proyección en perspectiva la definición de la matriz de proyección es distinta.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (projection_mode == MKZ_PROJECTIONMODE_PARALLEL){

  glOrtho(c_left,c_right,c_bottom,c_top,c_near,c_far);
}
else
{
  glFrustum(c_left,c_right,c_bottom,c_top,c_near,c_far);
}
```

La inicialización de las variables que definen el volumen de visión se puede observar mas adelante.

```
projection_mode = MKZ_PROJECTIONMODE_PARALLEL;
```

```
  c_left     = -1;
  c_right    = 1;
  c_bottom   = -1;
  c_top      = 1;
  c_near     = 1;
  c_far      = 1000;
```

En segundo lugar a la hora de dibujar los objetos es necesario construir el modelview para cada objeto. No obstante en OpenGL se almacena siempre la matriz identidad y el cálculo de cada vértice multiplicado por el modelview se hace de forma manual a la hora de dibujar cada objeto. Para cada vértice se multiplica su matriz de transformación por la izquierda y después la matriz de cambio de referencia. El resultado será el vértice transformado que se almacenará en un nuevo buffer de vértices que se utilizará posteriormente para dibujar el objeto.

```
  glLoadIdentity();
  ...
  float transformed_vertex[mesh->num_vertices][3];

  float temp[4];
  temp[3] = 1;

  MKZ_vector3 v1;
  MKZ_vector3 v2;
  MKZ_vector3 v3;

  /* Transformacion del objeto */
  for (v = 0 ; v < mesh->num_vertices ; v++){

    temp[0] = mo->mesh->vertex_table[v].coord.x;
    temp[1] = mo->mesh->vertex_table[v].coord.y;
    temp[2] = mo->mesh->vertex_table[v].coord.z;

    MKZ_ARITHMETIC_transform(mo->obj.transform,temp);
    MKZ_ARITHMETIC_transform(baseChange_mat,temp);

    transformed_vertex[v][0] = temp[0];
    transformed_vertex[v][1] = temp[1];
    transformed_vertex[v][2] = temp[2];

  }
```

Concretamente, la transformación de cada vértice se realiza multiplicando cada matriz secuencialmente, de esta forma.

```
void MKZ_ARITHMETIC_transform(float * mat, float * v3){
  float vx = 0;
  float vy = 0;
  float vz = 0;

  vx = mat[0]*v3[0]   + mat[4]*v3[1]  + mat[8]*v3[2]  + mat[12];
  vy = mat[1]*v3[0]   + mat[5]*v3[1]  + mat[9]*v3[2]  + mat[13];
  vz = mat[2]*v3[0]   + mat[6]*v3[1]  + mat[10]*v3[2] + mat[14];

  v3[0] = vx;
  v3[1] = vy;
  v3[2] = vz;
```

```
}
```

La matriz (mo->obj.transform) de transformación de cada objeto se compondrá de todas las modificaciones generadas por el usuario y se utiliza directamente. No obstante, esa matriz (baseChange mat) de cambio de sistema de referencia se calcula modificando debidamente la matriz de transformación asociada a la cámara mediante la siguiente función.

```
void MKZ_TRANSFORM_to_cameraMatrix(float* transform_mat , float* camera_mat)
{
  MKZ_ARITHMETIC_identityMatrix(camera_mat);

  int i = 0;
  for (i = 0; i < 9 ; i++){
    camera_mat[((i%3)*4) + i/3] = transform_mat[ i%3 + 4*(i/3)];
  }

  for (i = 0 ; i < 3 ; i++ ){
    camera_mat[12 + i] = -MKZ_ARITHMETIC_dotProduct(transform_mat+i*4,
    transform_mat+12);
  }
}
```

6.2. Operaciones de deshacer/rehacer

A lo largo de la implementación se han incluido funciones e decisiones convenientes y reveladoras que se van a explicar próximamente.

Para comenzar, una de las explicaciones que quedo pendiente fue como realizábamos la opción de rehacer y deshacer las transformaciones sobre un objeto. Cada objeto tiene dos matrices, una que se utiliza para llevar todas las transformaciones hasta el momento y otra para rehacer las transformaciones deshechas. Si tras deshacer alguna transformación se realiza alguna transformación, no se pueden recuperar y rehacer los cambios desechados.

```
void KG_undo_transformation(){

  ...

  if (obj->undoStack->matStack != 0){

    MKZ_object * mkz_obj = get_mkz_object(obj);

    float * n_mat = matStack_pop(&obj->undoStack);
    matStack_push(&obj->redoStack,n_mat);

    MKZ_ARITHMETIC_copy_matrix(obj->undoStack->mat,mkz_obj->transform);
    KG_update_children(obj);
  }
}
```

```
void KG_redo_transformation(){

  ...
```

```
  if (obj->redoStack != 0){

      float * n_mat = matStack_pop(&obj->redoStack);
      matStack_push(&obj->undoStack,n_mat);

      MKZ_ARITHMETIC_copy_matrix(obj->undoStack->mat,mkz_obj->transform);
      KG_update_children(obj);
  }
}
```

Para actualizar el hijo de la "linkedlist" de los objetos de forma consecutiva, se hace uso de la siguiente función.

```
void KG_update_children(object * obj){

  linkedList * ch = obj->children;
  MKZ_object * mkzo = get_mkz_object(obj);
  float * tramat = mkzo->transform;

  while (ch != 0){

    object * ch_obj = (object *) ch->content;
    MKZ_object * ch_mkz = get_mkz_object(ch_obj);

    MKZ_ARITHMETIC_matMul(tramat,ch_obj->undoStack->mat,ch_mkz->transform);
    KG_update_children(ch_obj);
    ch = ch->ll_after;
  }
}
```

6.3. Estructura de datos

La estructura de datos que empleamos tiene como objetivo asociar a cada objeto todo lo necesario para trabajar sobre el escenario que se plantea. El orden en el que se va a presentar la estructura de datos será en forma descendente, de mayor a menor magnitud.

Los objetos se almacenan en una "linkedlist", y indican el objeto previo y el siguiente de la misma.

```
struct linkedList{

  void * content;
  struct linkedList * ll_before;
  struct linkedList * ll_after;

};
```

Cada objeto se compone del componente objeto que se va almacenar y su tipo en la linkedlist, las matrices para rehacer y deshacer y sus descendientes en la "LinkedList".

```
typedef struct{

  void * object;
  matStack * undoStack;
  matStack * redoStack;
```

```
  int objectType;
  linkedList * children;

}object;

struct matStack{

  float * mat;
  struct matStack * matStack;
};
```

Se definen tres componentes objeto distintos: Objeto “mesh”, cámara e fuente de luz. Guardan en común el hecho de poseer una matriz de transformación de la instancia en la que se encuentra el componente, si se encuentra activo y el identificado del componente.

```
typedef struct{

  float* transform;
  int active;
  int id;

}MKZ_object;

typedef struct{

  MKZ_object obj;

  MKZ_mesh * mesh;
  MKZ_material * material;


}MKZ_meshedObject;

typedef struct {

  MKZ_object obj;

  int light_type;

  float intensityAmbient;
  float intensityDifuse;
  float intensitySpecular;

  MKZ_color3 color;

  float atenuationConstant;
  float atenuationLinear;
  float atenuationQuadratic;

  float spotExponent;

}MKZ_lightObject;

typedef struct{

  MKZ_object obj;

  MKZ_color3 skybox;

  int projection_mode;
```

```
  double v_x, v_y, v_near, v_far;

  int polygon_mode;
  int culling_enabled;

  int lighting_enable;
  int lighting_mode;


}MKZ_camera;
```

El componente mesh tiene por un lado la información sobre el objeto en su totalidad y como afecta el material que lo constituye en la iluminación.

```
typedef struct {

  int num_vertices;                /* number of vertices in the object*/
  MKZ_vertex *vertex_table;              /* table of vertices */
  int num_faces;                   /* number of faces in the object */
  MKZ_face *face_table;                  /* table of faces */
  MKZ_point3 min;                        /* coordinates' lower bounds */
  MKZ_point3 max;

}MKZ_mesh;

typedef struct{

  float * floatMap; //array size = n*n

  int n;

}MKZ_map;

typedef struct{

  MKZ_map * ambientMapR;
  MKZ_map * ambientMapG;
  MKZ_map * ambientMapB;

  MKZ_map * difuseMapR;
  MKZ_map * difuseMapG;
  MKZ_map * difuseMapB;

  MKZ_map * specularMapR;
  MKZ_map * specularMapG;
  MKZ_map * specularMapB;

  float shininess;

}MKZ_material;
```

EL componente cámara incluye el modo de proyección para elegir el modo de visualización de los objetos deseado. Incluye también la distancia entre cada intervalo que delimita el volumen de visualización (tanto el eje x como el y son simétricos), mientras que en el eje z difiere. La eliminación de las caras trasera se puede activar o desactivar y se puede elegir los modos del polígono "dotted", "wireframe" y "filled". Por último, se puede activar o no

la iluminación para la cámara y decidir si las normales van a ser las caras de los polígonos (flat) o los vértices (smooth).

La iluminación tiene el tipo de luz (direccional, puntual o foco), las intensidades correspondientes a tipo de iluminación (ambiente, difuso y especular), la atenuación, la cantidad de cada color que emite y la dispersión sobre la superficie (exponente).

El material de cada objeto incluye cuanta luz refleja cada tipo de iluminación presente en la escena, así como su luminosidad. La forma de hacer referencia a este es a través de un mapeado simple de una sola posición.

La estructura de datos restante es conocida y no ofrece ninguna novedad a lo ya conocido o proporcionado desde un inicio.

6.4. Cámara

En la cámara se han implementado diversas funciones para poder realizar todos los casos de uso exigidos. Una de las formas de introducir una nueva cámara que se utiliza consiste en duplicar la cámara actual; es decir crea una de características similares que se puede empezar a modificar sin afectar a la que previamente existía y se ha tomado como referencia en su creación.

El volumen de visión ha sido una de ellas y de las que se ha terminado por optar modificar los parámetros de los límites del volumen de visión directamente, dejando a un lado el escalado local. Cuando se va a realizar un escalado uniforme se multiplica por un factor cada límite del intervalo del volumen de visión que se atribuye a la cámara seleccionada en ese momento.

```
void KG_uniform_scale(int sense){

  ...

  switch(t_target){

    case KG_TRANSFORM_TARGET_OBJECT:
      traend = (object*) selectedObject->content;
      break;

    case KG_TRANSFORM_TARGET_CAMERA:
      traend = (object*) selectedCamera->content;
      MKZ_camera * cam = (MKZ_camera *) traend->object;

      cam->v_far *=s_factor;
      cam->v_x *= s_factor;
      cam->v_y *= s_factor;
      return;

    case KG_TRANSFORM_TARGET_LIGHT:
      traend = lList[selectedLight];
      break;
  }

  if (traend == 0)
```

```
    return;

  MKZ_object * obj = get_mkz_object(traend);

  if (t_scope == KG_TRANSFORM_SCOPE_GLOBAL){
    MKZ_TRANSFORM_scaleUniform_global(obj->transform,s_factor);
  }
  else{
    MKZ_TRANSFORM_scaleUniform_local(obj->transform,s_factor);
  }

  KG_update_children(traend);
  KG_save_object_matrix(traend);

}
```

Otra de las necesidades de la cámara ha sido fijar la vista de la cámara en el objeto seleccionado sin perder la posición en la que se encuentra. Este reposicionamiento se logra con la matriz de transformación de la cámara y el punto del objeto y es clave para mirar al objeto en modo análisis.

```
void MKZ_TRANSFORM_look_at(float * modMat, MKZ_point3 * p3){

    float direct_x = p3->x - modMat[12];
    float direct_y = p3->y - modMat[13];
    float direct_z = p3->z - modMat[14];

    MKZ_vector3 direction;
    direction.x = direct_x;
    direction.y = direct_y;
    direction.z = direct_z;

    MKZ_vector3 vup;
    vup.x = modMat[4];
    vup.y = modMat[5];
    vup.z = modMat[6];

    MKZ_vector3 newX;

    MKZ_ARITHMETIC_normalize_vector(&direction);
    MKZ_ARITHMETIC_normalize_vector(&vup);
    MKZ_ARITHMETIC_corssProduct_vector( &direction, &vup, &newX);
    MKZ_ARITHMETIC_corssProduct_vector( &newX , &direction, &vup);


    float  mod = MKZ_ARITHMETIC_eulidean_norm(modMat);
    //mod = 1;
    modMat[0] = newX.x*mod;
    modMat[1] = newX.y*mod;
    modMat[2] = newX.z*mod;

    mod = MKZ_ARITHMETIC_eulidean_norm(modMat+4);
    //mod = 1;
    modMat[4]= vup.x*mod;
    modMat[5]= vup.y*mod;
    modMat[6]= vup.z*mod;

    mod = MKZ_ARITHMETIC_eulidean_norm(modMat+8);
    //mod = 1;
    modMat[8] = -direction.x*mod;
    modMat[9] = -direction.y*mod;
```

```
    modMat[10] = -direction.z*mod;
}
```

El tipo de proyección se puede cambiar modificando en el campo de la cámara correspondiente su valor. Una función dedicada a esta tarea se encarga de ello.

```
void KG_switch_camera_projection(){

  if (selectedCamera == 0)
    return;

  MKZ_camera * cam = MKZ_SCENE_get_camera();

  if (cam->projection_mode == MKZ_PROJECTIONMODE_PARALLEL){
    cam->projection_mode = MKZ_PROJECTIONMODE_PERSPECTIVE;
  }
  else{
    cam->projection_mode = MKZ_PROJECTIONMODE_PARALLEL;
  }
}
```

6.5. Iluminación

Las fuentes de luz suponen un aporte de luz en nuestra representación del espacio. Para lograr este efecto ha sido necesario calcular el vector normal de cada cara del polígono y vértice. El algoritmo empleado primero se encarga de calcular el vector normal de cada cara de un polígono utilizando tres de sus vértices y, una vez terminado el proceso para todas las caras, suma a cada vector el vector normal de las caras a las que pertenece y se normaliza. Cada vez que se vayan a dibujar, si la iluminación esta activa, se calcula como se muestra a continuación.

```
  for (v = 0 ; v < mesh->num_vertices ; v++){
      vertex_normals[v][0] = 0;
      vertex_normals[v][1] = 0;
      vertex_normals[v][2] = 0;
    }

    for (f = 0 ; f < mesh->num_faces ; f++){

      MKZ_face face =  mesh->face_table[f];

      v1.x = transformed_vertex[face.vertex_table[1]][0]
             - transformed_vertex[face.vertex_table[0]][0];
      v1.y = transformed_vertex[face.vertex_table[1]][1]
             - transformed_vertex[face.vertex_table[0]][1];
      v1.z = transformed_vertex[face.vertex_table[1]][2]
             - transformed_vertex[face.vertex_table[0]][2];

      v2.x = transformed_vertex[face.vertex_table[2]][0]
             - transformed_vertex[face.vertex_table[0]][0];
      v2.y = transformed_vertex[face.vertex_table[2]][1]
             - transformed_vertex[face.vertex_table[0]][1];
      v2.z = transformed_vertex[face.vertex_table[2]][2]
```

```
            - transformed_vertex[face.vertex_table[0]][2];

      MKZ_ARITHMETIC_corssProduct_vector(&v1,&v2,&v3);

      for (v = 0 ; v < face.num_vertices ; v++){
        v_index = face.vertex_table[v];
        vertex_normals[v_index][0] += v3.x;
        vertex_normals[v_index][1] += v3.y;
        vertex_normals[v_index][2] += v3.z;
      }

      MKZ_ARITHMETIC_normalize_vector(&v3);
      face_normals[f][0] = v3.x;
      face_normals[f][1] = v3.y;
      face_normals[f][2] = v3.z;
    }

    for (v = 0 ; v < mesh->num_vertices ; v++){
      v3.x = vertex_normals[v][0];
      v3.y = vertex_normals[v][1];
      v3.z = vertex_normals[v][2];

      MKZ_ARITHMETIC_normalize_vector(&v3);

      vertex_normals[v][0] = v3.x;
      vertex_normals[v][1] = v3.y;
      vertex_normals[v][2] = v3.z;

    }
```

Las inicializaciones de nuestras fuentes de iluminación son de importancia para comprender el efecto de iluminación generado al iniciar la aplicación, por tanto, se enseña cuales han sido los parámetros elegidos en la configuración de cada fuente de luz en un comienzo. Nótese que las primeras tres hacen referencia a un sol que rota, una bombilla que se puede mover y un foco asociado al objeto seleccionado en todo momento. Las fuentes de luz se representan median cubos en la escena para conocer su localización en todo momento. Lo primero de todo es definir el material de la superficie de los objetos junto con la matriz de proyección y los valores iniciales para la activación de la iluminación.

```
void MKZ_DRAW_init(){

  identity_mat = MKZ_ARITHMETIC_create_matrix();
  MKZ_ARITHMETIC_identityMatrix(identity_mat);
  baseChange_mat = identity_mat;

  projection_mode = MKZ_PROJECTIONMODE_PARALLEL;

  c_left     = -1;
  c_right    = 1;
  c_bottom   = -1;
  c_top      = 1;
  c_near     = 1;
  c_far      = 1000;

  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
```

```
  glClearColor(0, 0, 0, 1);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glEnable(GL_CULL_FACE);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);


  next_light = 0;
  lighting_enabled = 0;
  lighting_mode = MKZ_LIGHTING_FLAT;

  /*default material */
  MKZ_material * mat = MKZ_GEOMETRY_create_material();
  MKZ_map * map = MKZ_GEOMETRY_create_map(1);
  map->floatMap[0] = 0.2;
  mat->ambientMapR = map;
  mat->ambientMapG = map;
  mat->ambientMapB = map;

  map = MKZ_GEOMETRY_create_map(1);
  map->floatMap[0] = 0.8;
  mat->difuseMapR = map;
  mat->difuseMapG = map;
  mat->difuseMapB = map;

  map = MKZ_GEOMETRY_create_map(1);
  map->floatMap[0] = 0;
  mat->specularMapR = map;
  mat->specularMapG = map;
  mat->specularMapB = map;

  mat->shininess = 100;

  defaultMaterial = mat;
}
```

Le siguen las propiedades de cada fuente de luz que se listan aquí abajo. Además para mostrar las condiciones iniciales de la aplicación se va a enseñar toda la función, extensa, pero que ilustra con detalle las características de la iluminación y la cámara concebidos para la práctica de forma inicial.

Por un lado la inicialización de las variables globales al inicio del mismo permite poder ir creando el escenario de forma progresiva hasta haber completado cada variable global con el contenido apropiado. La transformación inicial es la traslación, las coordenadas son globales y se realizan sobre el objeto todas las transformaciones. El cubo cargado es para representar cada fuente de luz de forma física en el escenario. La cámara se inicializa en perspectiva y con los valores constantes definidos. Para cada fuente de luz se definen sus intensidades para cada tipo de iluminación y la cantidad de cada color que va a emitir en el formato RGB. Finalmente, se posicionan en el espacio con cierto cuidado para mostrar los efectos de iluminación con diversidad.

```
void __KG_init(){

  int i = 0;
```

```
for (i = 0 ; i < 8 ; i++){
  lList[i] = 0;
}

for (i = 0 ; i < 100 ; i++){
  meshList[i] = 0;
}

cameraList = 0;
objList = 0;
selectedLight = 0;
selectedCamera = 0;
selectedObject = 0;

t_type = KG_TRANSFORM_TYPE_TRANSLATE;
t_scope = KG_TRANSFORM_SCOPE_GLOBAL;
t_target = KG_TRANSFORM_TARGET_OBJECT;
special_function = 0;

meshList[0] = MKZ_GEOMETRY_load_mesh("resources/cubo.obj");

/** init scene**/
MKZ_camera * cam  = MKZ_SCENE_get_default_camera();
cam->polygon_mode = MKZ_POLYGONMODE_FILLED;
cam->skybox.r = KG_COL_BACK_R;
cam->skybox.g = KG_COL_BACK_G;
cam->skybox.b = KG_COL_BACK_B;
cam->lighting_enable = 1;
cam->lighting_mode = MKZ_LIGHTING_FLAT;
cam->culling_enabled = 1;

//Set of parameters that are global, i.e. camera independent
MKZ_SCENE_set_global_mask(MKZ_GLOBAL_BG_COLOR | MKZ_GLOBAL_POLYGON |
    MKZ_GLOBAL_LIGHTING_ENABLE | MKZ_GLOBAL_LIGHTING_MODE
    | MKZ_GLOBAL_CULLING);

/** Init first cam*/
cam = MKZ_OBJECT_create_camera();

cam->projection_mode = MKZ_PROJECTIONMODE_PERSPECTIVE;

cam->v_x = KG_VIEW_WIDTH;
cam->v_y = KG_VIEW_HEIGHT;
cam->v_near = KG_VIEW_NEAR;
cam->v_far = KG_VIEW_FAR;

MKZ_vector3 * v3 = MKZ_GEOMETRY_create_vector3();


v3->x = 0;
v3->y = 0;
v3->z = 10;

MKZ_TRANSFORM_translate_global(cam->obj.transform,v3);
MKZ_SCENE_set_camera(cam);

object * obj = create_object_camera(cam);
KG_save_object_matrix(obj);
linkedlist_add(&cameraList,obj);
selectedCamera = cameraList;

/** First light, sun yellow*/
```

```
MKZ_lightObject * lo = MKZ_OBJECT_create_lightObject();
lo->light_type = MKZ_LIGHT_TYPE_DIRECTIONAL;
lo->intensityAmbient = 0.1;
lo->intensityDifuse= 0.3;
lo->intensitySpecular = 3;
lo->color.r = 225.0/255.0;
lo->color.g = 239.0/255.0;
lo->color.b = 23.0/255.0;

MKZ_TRANSFORM_rotateX_global(lo->obj.transform,1);
MKZ_TRANSFORM_rotateY_global(lo->obj.transform,1);
MKZ_SCENE_add_light(lo);

obj = create_object_light(lo);
KG_save_object_matrix(obj);
lList[0] = obj;

/** Second light, red lightbulb **/
lo = MKZ_OBJECT_create_lightObject();
lo->light_type = MKZ_LIGHT_TYPE_POINT;
lo->obj.transform[12] = 1;
lo->obj.transform[13] = 2;
lo->obj.transform[14] = 3;
lo->intensityAmbient = 1;
lo->intensityDifuse = 10;
lo->intensitySpecular = 10;
lo->color.r = 1;
lo->color.g = 0;
lo->color.b = 0;

MKZ_meshedObject * mo = MKZ_OBJECT_create_meshedObject();
mo->mesh = meshList[0];
MKZ_TRANSFORM_scaleUniform_local(mo->obj.transform,0.05);
object * mo_object = create_object_meshed(mo);
KG_save_object_matrix(mo_object);

MKZ_SCENE_add_light(lo);
MKZ_SCENE_add_mesh(mo);

obj = create_object_light(lo);
KG_save_object_matrix(obj);
lList[1] = obj;

linkedlist_add(&obj->children,mo_object);
KG_update_children(obj);


/** luz adicional, verde **/

lo = MKZ_OBJECT_create_lightObject();
lo->light_type = MKZ_LIGHT_TYPE_POINT;
lo->obj.transform[12] = -2;
lo->obj.transform[13] = 2;
lo->obj.transform[14] = 1;
lo->intensityAmbient = 1;
lo->intensityDifuse = 10;
lo->intensitySpecular = 10;
lo->color.r = 0;
lo->color.g = 1;
lo->color.b = 0;

mo = MKZ_OBJECT_create_meshedObject();
```

```
  mo->mesh = meshList[0];
  MKZ_TRANSFORM_scaleUniform_local(mo->obj.transform,0.05);
  mo_object = create_object_meshed(mo);
  KG_save_object_matrix(mo_object);

  MKZ_SCENE_add_light(lo);
  MKZ_SCENE_add_mesh(mo);

  obj = create_object_light(lo);
  KG_save_object_matrix(obj);
  lList[3] = obj;

  linkedlist_add(&obj->children,mo_object);
  KG_update_children(obj);

  /** luz adicional, azul **/

  lo = MKZ_OBJECT_create_lightObject();
  lo->light_type = MKZ_LIGHT_TYPE_POINT;
  lo->obj.transform[12] = -1;
  lo->obj.transform[13] = -1;
  lo->obj.transform[14] = 3;
  lo->intensityAmbient = 1;
  lo->intensityDifuse = 10;
  lo->intensitySpecular = 10;
  lo->color.r = 0;
  lo->color.g = 0;
  lo->color.b = 1;

  mo = MKZ_OBJECT_create_meshedObject();
  mo->mesh = meshList[0];
  MKZ_TRANSFORM_scaleUniform_local(mo->obj.transform,0.05);
  mo_object = create_object_meshed(mo);
  KG_save_object_matrix(mo_object);

  MKZ_SCENE_add_light(lo);
  MKZ_SCENE_add_mesh(mo);

  obj = create_object_light(lo);
  KG_save_object_matrix(obj);
  lList[4] = obj;

  linkedlist_add(&obj->children,mo_object);
  KG_update_children(obj);


  for (i = 0 ; i < 100 ; i++){
    mapList[i] = MKZ_GEOMETRY_create_map(1);
    mapList[i]->floatMap[0] = ((float)rand())/((float)RAND_MAX);
  }


  MKZ_GEOMETRY_free_vector3(v3);
}
```

El foco asociado al objeto no figura entre los inicializados de forma estándar. Esto se debe a que inicialmente podría decidirse que no hubiera ningún objeto al que asociarlo, por ello, su creación y asignación ocurre en el momento de cargar el objeto. Al mismo tiempo es en ese momento en el que se decide de forma aleatoria también el material que lo compondrá.

En un escenario ideal se podría mapear la textura del objeto y recrearlo de forma fiel; sin embargo, no se ha desarrollado hasta ese punto la práctica. La cámara que se incluye como uno de los descendientes de cada objeto se genera también a la hora de cargarlo, para poder acompañarlo por el escenario y utilizarlo cuando resulte necesario.

```
int KG_load_object(char * filename){


  MKZ_mesh * mesh = MKZ_GEOMETRY_load_mesh(filename);

  if (mesh == 0){
    return -1;
  }
  else{

    MKZ_meshedObject * mo = MKZ_OBJECT_create_meshedObject();
    mo->mesh = mesh;


    MKZ_material * mat = MKZ_GEOMETRY_create_material();

    unsigned int r = rand() %100;
    mat->ambientMapR = mapList[r];
    mat->difuseMapR = mapList[r];
    mat->specularMapR = mapList[r];
    r = rand() % 100;
    mat->ambientMapG = mapList[r];
    mat->difuseMapG = mapList[r];
    mat->specularMapG = mapList[r];
    r = rand() % 100;
    mat->ambientMapB = mapList[r];
    mat->difuseMapB = mapList[r];
    mat->specularMapB = mapList[r];

    mat->shininess = 127.0*((float)rand())/((float)RAND_MAX);

    mo->material = mat;

    MKZ_SCENE_add_mesh(mo);

    object * obj = create_object_meshed(mo);
    linkedlist_add(&objList,obj);

    KG_save_object_matrix(obj);

    MKZ_camera * cam = MKZ_OBJECT_create_camera();

    cam->obj.transform[13] = 2;
    MKZ_TRANSFORM_rotateY_global(cam->obj.transform,3.14159265359f);

    cam->v_x = KG_VIEW_WIDTH;
    cam->v_y = KG_VIEW_HEIGHT;
    cam->v_far = KG_VIEW_FAR;
    cam->v_near = KG_VIEW_NEAR;
    cam->projection_mode = MKZ_PROJECTIONMODE_PERSPECTIVE;

    object * obj_cam = create_object_camera(cam);
    linkedlist_add(&obj->children,obj_cam);
```

```
    KG_save_object_matrix(obj_cam);

    MKZ_lightObject * light = MKZ_OBJECT_create_lightObject();
    light->light_type = MKZ_LIGHT_TYPE_FOCAL;
    light->intensityAmbient = 1;
    light->intensityDifuse = 5;
    light->intensitySpecular = 5;
    light->spotExponent = 3;

    light->obj.transform[12]=2;
    light->obj.transform[13]=2;
    light->obj.transform[14]=2;

    MKZ_TRANSFORM_rotateX_local(light->obj.transform,1);
    MKZ_TRANSFORM_rotateY_local(light->obj.transform,2);


    object * obj_light = create_object_light(light);
    linkedlist_add(&obj->children,obj_light);
    KG_save_object_matrix(obj_light);


    MKZ_meshedObject * torch = MKZ_OBJECT_create_meshedObject();
    torch->mesh = meshList[0];
    torch->obj.active = 0;
    MKZ_TRANSFORM_scaleUniform_local(torch->obj.transform,0.01);
    object * torch_obj = create_object_meshed(torch);
    KG_save_object_matrix(torch_obj);
    linkedlist_add(&obj_light->children,torch_obj);

    KG_update_children(obj);
    MKZ_SCENE_add_mesh(torch);


    if (selectedObject == 0){
      selectedObject = objList;

      lList[2] = obj_light;
      MKZ_SCENE_add_light(light);
      torch->obj.active = 1;
    }
    return 0;
  }

}
```

Mediante las llamadas a OpenGL se habilitan las luces y se definen sus propiedades definidas en la inicialización. Este proceso se lleva acabo cuando se añaden las luces creadas a la propia escena. Para cada fuente de luz que llame a la función se informa a OpenGL de su intensidad ambiental, difusa y especular. En caso de ser un foco además también se tiene en cuenta el ángulo de apertura, el difuminado y la dispersión.

```
void MKZ_DRAW_add_light(MKZ_lightObject * lo){

  //printf("Light ptr: %d\n",lo);
  glLoadMatrixf(baseChange_mat);


  GLenum light_ind;
```

```
float f[4];
switch(next_light){

  case 0:
    light_ind = GL_LIGHT0;
    break;
  case 1:
    light_ind = GL_LIGHT1;
    break;
  case 2:
    light_ind = GL_LIGHT2;
    break;
  case 3:
    light_ind = GL_LIGHT3;
    break;
  case 4:
    light_ind = GL_LIGHT4;
    break;
  case 5:
    light_ind = GL_LIGHT5;
    break;
  case 6:
    light_ind = GL_LIGHT6;
    break;
  case 7:
    light_ind = GL_LIGHT7;
    break;

}

glEnable(light_ind);

if (lo->light_type != MKZ_LIGHT_TYPE_DIRECTIONAL)
  glLightfv(light_ind, GL_POSITION, lo->obj.transform+12);
else{


  f[0] = -lo->obj.transform[8];
  f[1] = -lo->obj.transform[9];
  f[2] = -lo->obj.transform[10];
  f[3] = 0;


  glLightfv(light_ind, GL_POSITION, f);

}

f[0] = lo->color.r * lo->intensityAmbient;
f[1] = lo->color.g * lo->intensityAmbient;
f[2] = lo->color.b * lo->intensityAmbient;
f[3] = 1;

glLightfv(light_ind, GL_AMBIENT, f);

f[0] = lo->color.r * lo->intensityDifuse;
f[1] = lo->color.g * lo->intensityDifuse;
f[2] = lo->color.b * lo->intensityDifuse;
f[3] = 1;


glLightfv(light_ind, GL_DIFFUSE, f);
```

```
  f[0] = lo->color.r * lo->intensitySpecular;
  f[1] = lo->color.g * lo->intensitySpecular;
  f[2] = lo->color.b * lo->intensitySpecular;
  f[3] = 1;

  glLightfv(light_ind, GL_SPECULAR, f);


  glLightf(light_ind, GL_CONSTANT_ATTENUATION, lo->atenuationConstant);
  glLightf(light_ind, GL_LINEAR_ATTENUATION,  lo->atenuationLinear);
  glLightf(light_ind, GL_QUADRATIC_ATTENUATION, lo->atenuationQuadratic);

  if (lo->light_type == MKZ_LIGHT_TYPE_FOCAL){

    f[0] = -lo->obj.transform[8];
    f[1] = -lo->obj.transform[9];
    f[2] = -lo->obj.transform[10];
    f[3] = 0;

    glLightfv(light_ind, GL_SPOT_DIRECTION, f);

    float det = MKZ_ARITHMETIC_determinant(lo->obj.transform);
    //printf("det: %f\n",det);

    float a = 1.0/10.0f;

    det = (75.0/(1.0+exp(-a*(log(det)-2)))) +15;
    //Funcion sigmoide parametrizada

    //printf("det: %f\n",det);

    if (det > 90)
      det = 90;

    if (det < 15)
      det = 15;

    glLightf(light_ind, GL_SPOT_CUTOFF, det);

    glLightf(light_ind, GL_SPOT_EXPONENT, lo->spotExponent);
  }

  next_light++;
}
```

Para dibujar la escena con todo lo anteriormente planteado, consistirá en decidir que parámetros van a depender de la cámara particular que será el observador y cuales por defecto mediante las consecutivas máscaras. Después, el dibujado primero se encarga de dibujar y situar con llamadas a OpenGL en la función las fuentes de luz en el escenario y, justo después, se dibuja cada objeto como corresponde.

```
void MKZ_SCENE_draw(){

    MKZ_camera * aux_camera = 0;
  aux_camera = ((global_mask & MKZ_GLOBAL_BG_COLOR) != 0) ?
     default_camera : camera;
    MKZ_DRAW_set_background_color(&aux_camera->skybox);

    aux_camera = ((global_mask & MKZ_GLOBAL_PROJECTION) != 0) ?
```

```
    default_camera : camera;
    MKZ_DRAW_set_projectionMode(aux_camera->projection_mode);

    aux_camera = ((global_mask & MKZ_GLOBAL_POLYGON) != 0) ?
    default_camera : camera;
    MKZ_DRAW_set_poligonMode(aux_camera->polygon_mode);

    aux_camera  = ((global_mask & MKZ_GLOBAL_CULLING) != 0) ?
    default_camera : camera;
    MKZ_DRAW_set_culling(aux_camera->culling_enabled);

    aux_camera = ((global_mask & MKZ_GLOBAL_TRANSFORM) != 0)
    default_camera : camera;
    MKZ_DRAW_set_cameraMat(aux_camera->obj.transform);

    aux_camera = ((global_mask & MKZ_GLOBAL_LIGHTING_ENABLE) != 0) ?
    default_camera : camera;
    MKZ_DRAW_set_lighting(aux_camera->lighting_enable);

    aux_camera = ((global_mask & MKZ_GLOBAL_LIGHTING_MODE) != 0) ?
    default_camera : camera;
    MKZ_DRAW_set_lighting_mode(aux_camera->lighting_mode);

    aux_camera = ((global_mask & MKZ_GLOBAL_RENDER_VOLUME) != 0) ?
    default_camera : camera;
    MKZ_DRAW_set_renderVolume(-aux_camera->v_x/2,aux_camera->v_x/2,
        -aux_camera->v_y/2,aux_camera->v_y/2,
        aux_camera->v_near,aux_camera->v_far);

    MKZ_DRAW_clear();
    MKZ_DRAW_start();

    MKZ_DRAW_clear_lights();

    MKZ_linkedList * aux = lightList;

    while (aux != 0){
          //printf("aux: %d\n",aux);
          MKZ_lightObject * lo = (MKZ_lightObject *)aux->content;
          if (lo->obj.active){
            MKZ_DRAW_add_light(lo);
          }
          aux = aux->ll;

    }

    aux = objectList;

    while (aux != 0){
      //printf("aux: %d\n",aux);
      MKZ_meshedObject * lo = (MKZ_meshedObject *)aux->content;
      if (lo->obj.active){
        MKZ_DRAW_object(lo);
      }
      aux = aux->ll;
    }

    MKZ_DRAW_end();
}
```

7. Casos de uso

A continuación, para mostrar el funcionamiento de nuestra aplicación, se van a realizar distintos casos de uso representativos de tanto la iluminación como de la cámara. En estos casos de uso se va a poner aprueba el funcionamiento de la aplicación en distintos panoramas para determinar si el comportamiento es consecuente con lo expuesto hasta ahora.

7.1. Primer caso de uso

Se va a exponer como las modificaciones a la cámara en el modo vuelo y la utilización del modo análisis pueden utilizarse para proporcionar puntos de vista diversos.

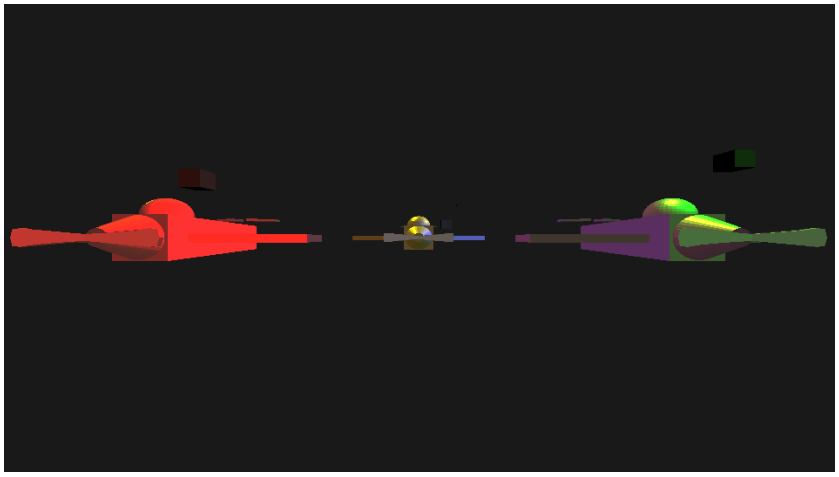


Figura 13: Trasladamos a los lados los dos objetos cargados

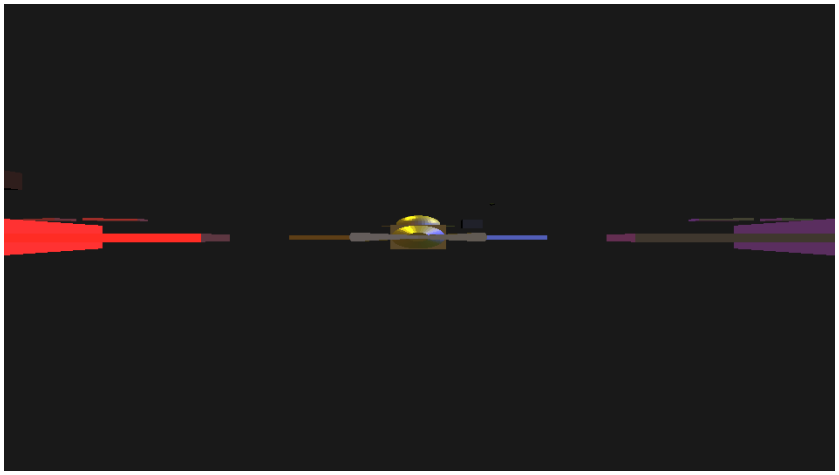


Figura 14: Disminuir el volumen de visión hasta poder ver la mitad de la de cada objeto (eje x)

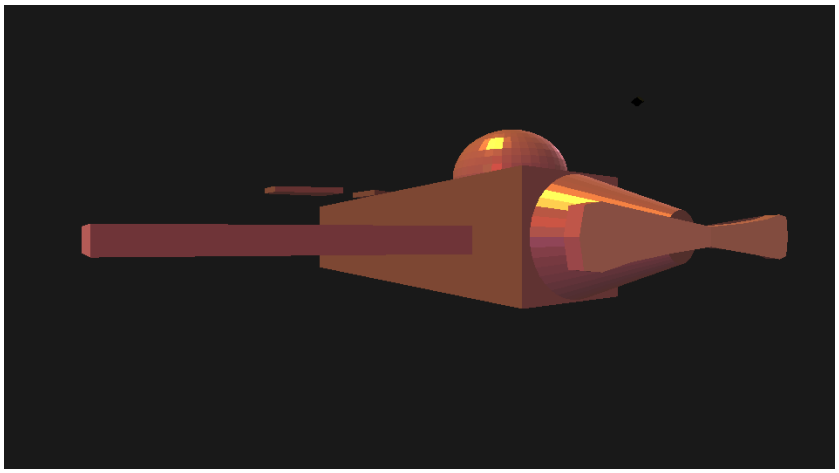


Figura 15: Activar el modo análisis en el objeto de la derecha y acercar hasta perder de vista al objeto izquierdo

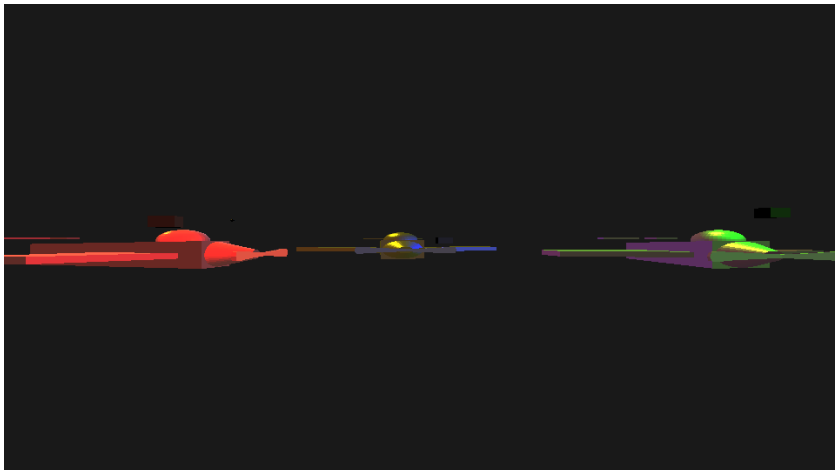


Figura 16: Cambiar de objeto seleccionado, activar modo análisis (para mirar al objeto) e desactivarlo, rotar el objeto hasta que mire al otro objeto

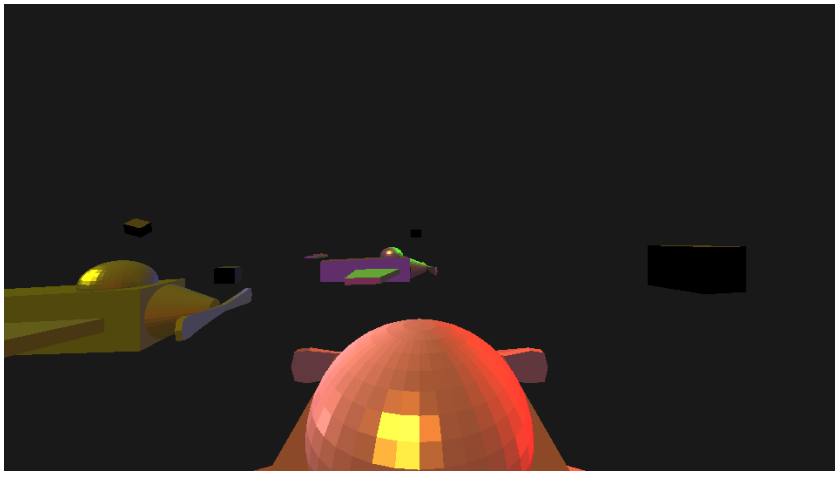


Figura 17: Tomar el objeto seleccionado como la cámara, y visualizar el otro objeto

7.2. Segundo caso de uso

Este caso de uso va a plantear formas de visualizar los objetos mediante varias cámaras tanto en proyección ortográfica como en perspectiva. Para ello, se crean tres cámaras, que en nuestra aplicación supone duplicar la cámara actualmente seleccionada, y se coloca una con vista frontal, una con vista del perfil y otra con la vista superior del objeto.

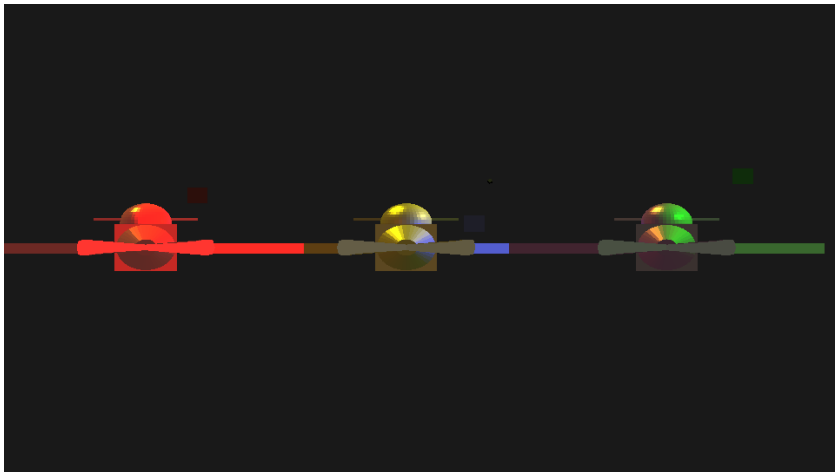


Figura 18: Vista frontal ortográfica

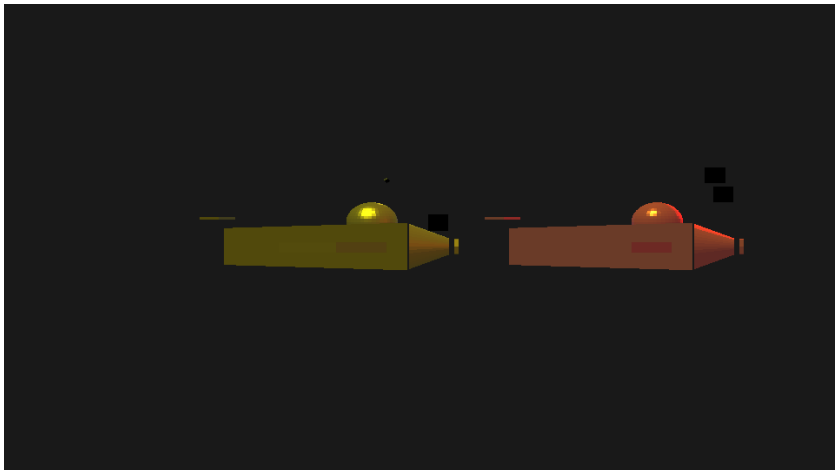


Figura 19: Vista perfil ortográfica

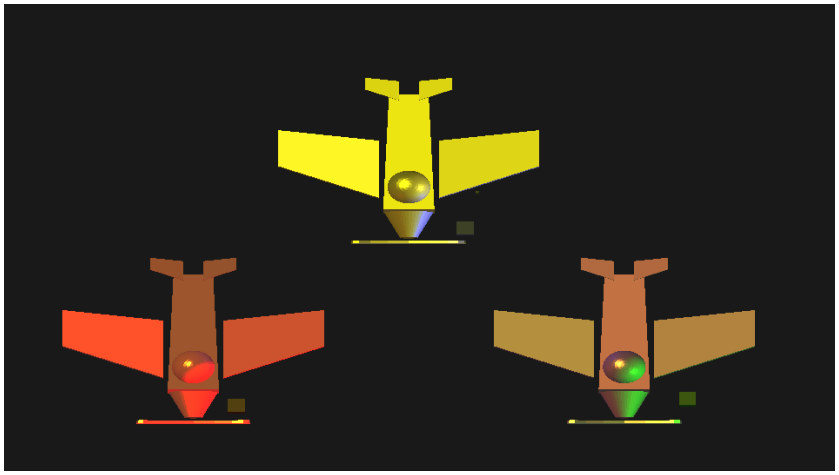


Figura 20: Vista superior ortográfica

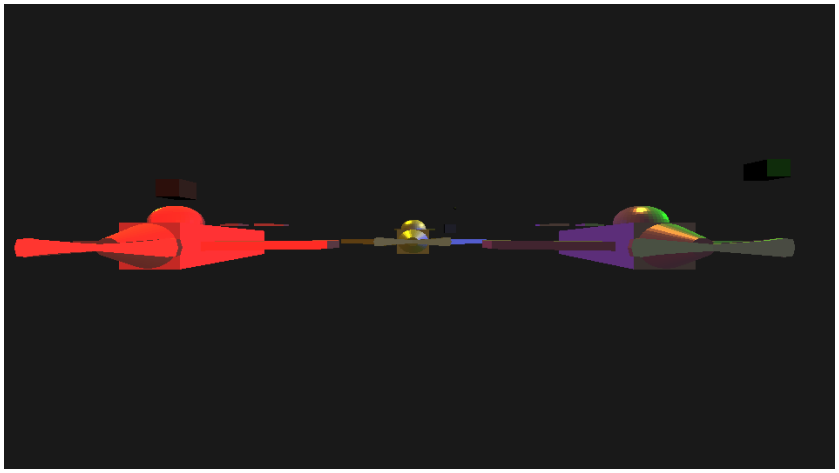


Figura 21: Vista frontal perspectiva

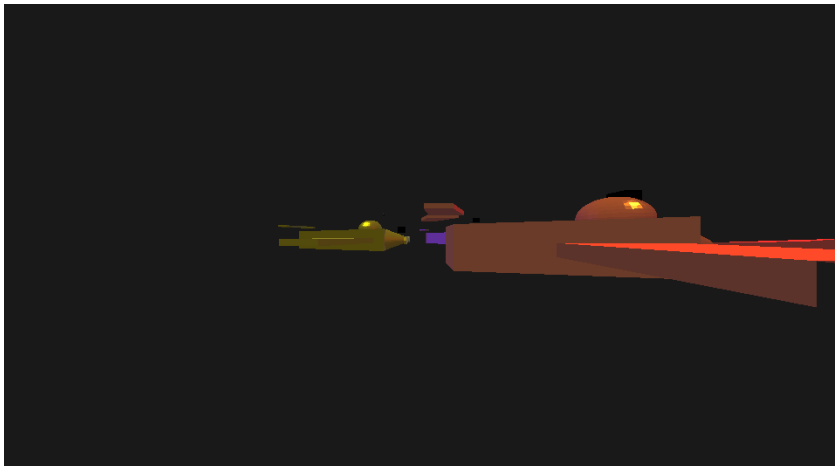


Figura 22: Vista perfil perspectiva

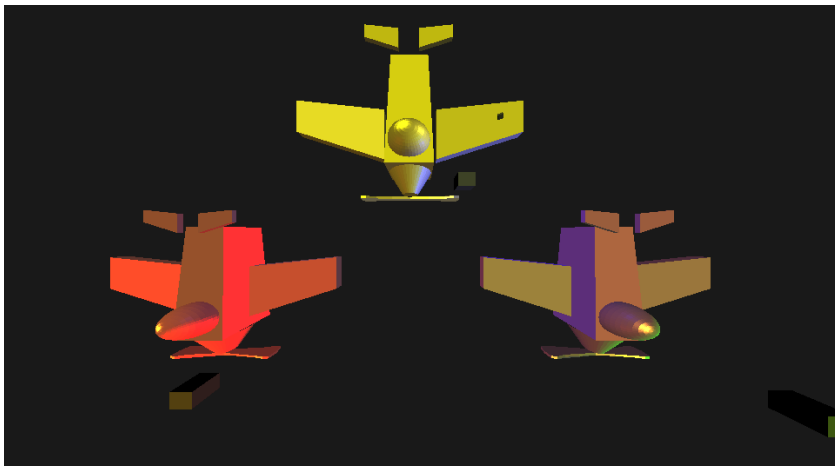


Figura 23: Vista superior perspectiva

7.3. Tercer caso de uso

Para mostrar las fases que trascurren en la iluminación se van a cargar tres objetos y colocar entre ellos una fuente de luz. La luz ambiental proporcionada por el sol también tendrá efecto sobre como se ilumina la superficie.

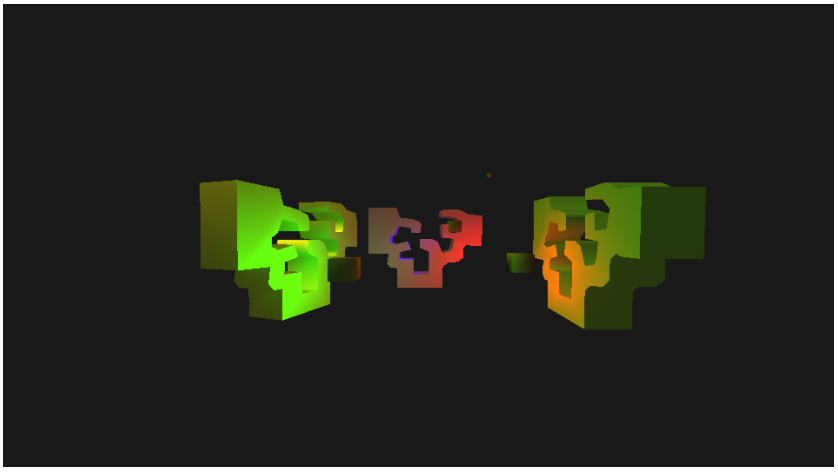


Figura 24: Cargar tres objetos y posicionarlos a una distancia equidistante entre ellos. Colocar (traslación) las fuentes de luz entre dichos objetos. Esas fuentes de luz son puntuales y la luz direccional del sol está activada

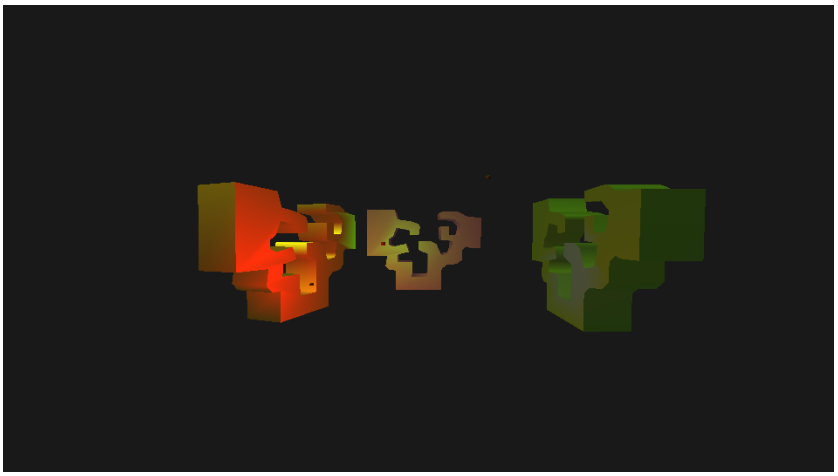


Figura 25: Cambiar las tres fuentes de luz de puntuales a focales, cuya dirección es el vector que indica para un objeto en que dirección está el objeto a su derecha

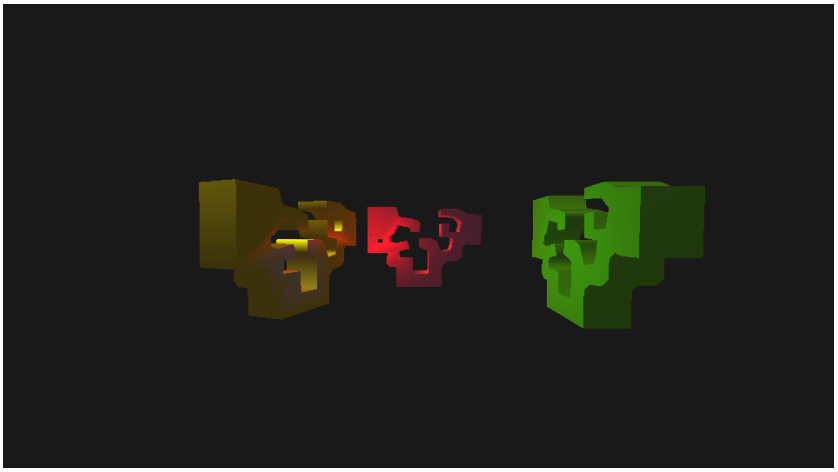


Figura 26: Cambiar la dirección de los focos (rotación) para que iluminen el objeto mas lejano, el objeto que no tiene a los lados sino el adyacente

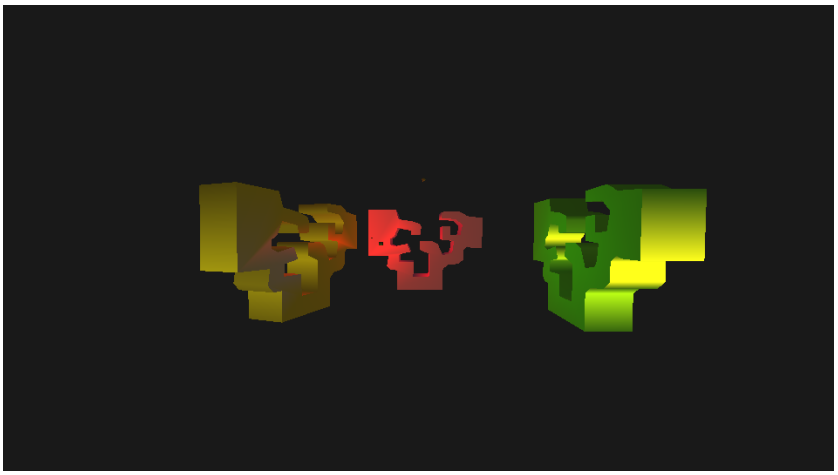


Figura 27: Rotar el sol (luz direccional) hasta colocarlo en la posición opuesta, la mas lejana posible mediante la rotación con respecto a la inicial

7.4. Cuarto caso de uso

Para poner a prueba el foco asociado en todo momento al objeto seleccionado, se va a mostrar diferencia entre tener un foco y no hacerlo y como afecta al resultado final. Para mostrar esta diferencia es necesario apagar todas las luces, excepto el foco, que estén actuando.

Figura 28: Cargar dos objetos, apagar todas las luces excepto el foco al seleccionado

Figura 29: Cambiar de objeto seleccionado

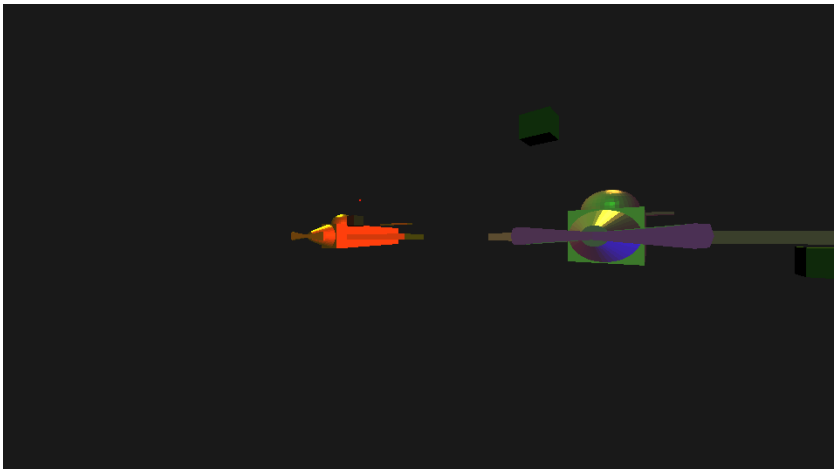


Figura 30: Encender el resto de luces y apagar los focos

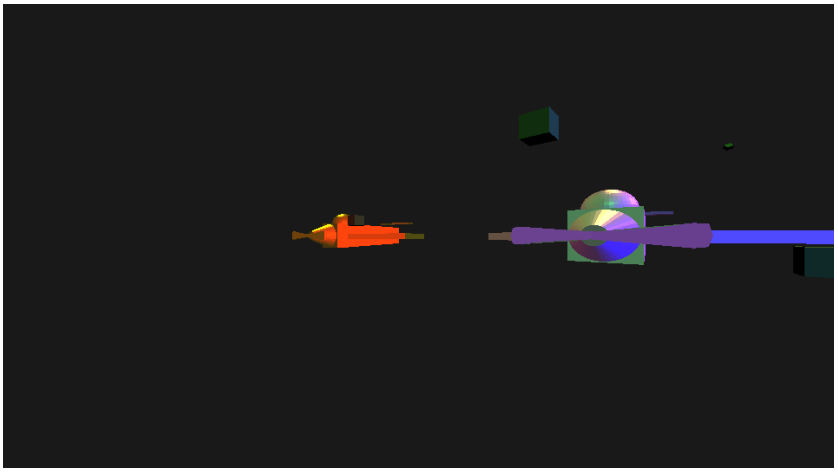


Figura 31: Todas las luces están encendidas, y el foco está en el primer objeto

8. Conclusiones finales

Las conclusiones, dada la documentación y el resultado programado, son bastante evidentes y, en general, muy positivas. El proyecto nos ha permitido pasar de la teoría a la práctica, ver los problemas que se generan al pasar de un modelo teórico de un motor gráfico a una versión real, con sus problemas numéricos, sus fallos de implementación y sus errores de ejecución.

En general, creemos que el resultado ha sido satisfactorio: se han implementado todos los casos de uso especificados en los sucesivos enunciados,

además de funcionalidades adicionales que han servido para profundizar todavía más en los contenidos vistos en clase. Además, gracias a la separación de la interfaz de funciones gráficas por un lado y los casos de uso por otro, el resultado final es el de una aplicación más fácilmente escalable, que podría servir de base para futuras mejoras y ampliaciones del proyecto.

Tanto la asignatura como el proyecto nos han dado una visión de conjunto del terreno de los gráficos por computador, un área en el que los modelos matemáticos son de vital importancia para diseñar e implementar soluciones. Es, de hecho, esta unión entre geometría vectorial e informática la que vuelve tan potentes los resultados obtenidos.

En cuanto a planes de futuro, podríamos citar infinidad de cosas que nos gustaría incorporar al proyecto. Lo más inmediato es, seguramente, ray-tracing, que se añadiría a los sistemas de iluminación existentes como función adicional, así como la incorporación de shaders. Gracias a la separación entre motor y programa, creemos que estas incorporaciones serían razonables de llevar a cabo y resultarían en una mejora sustancial para la aplicación.