Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer
You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Introducción 1

La humanidad siempre ha considerado excitante fantasear con la existencia de mundos virtuales y con la posibilidad de que nuestra existencia no sea real. Lewis Carroll, en "A través del espejo" plantea en 1871 la posibilidad de existencia de realidades alternativas, cuando Alicia accede a un mundo virtual cruzando un espejo (figura 1.1).

Mucho antes, en 1635, Calderón de la Barca escribió la obra de teatro "La vida es sueño" en la que plantea la dificultad para saber si lo que percibimos es real o no[1].

Recientemente, el desarrollo de la Informática Gráfica ha abierto las puertas para que la generación de entornos virtuales sea una realidad, planteando un sin fin de cuestiones éticas al posible uso de las aplicaciones potenciales, mucho antes de que estas puedan ser realidad.

Concretamente, en la década de los 90, el desarrollo de nuevas tecnologías parecía augurar un cambio vertiginoso. Como ejemplo la mítica película "Tron" postulaba la existencia de un mundo virtual en el interior de los ordenadores al que las personas podría viajar. Algo más cerca de la realidad, la película Nivel 13 plantea un futuro en el que los sistemas de entretenimiento sean sistemas de Realidad Virtual en los que el usuario pueda crear mundos virtuales para vivir situaciones imposibles en el mundo real (figura 1.2), llegando a plantearse la posibilidad de que los avatares de ese mundo virtual lleguen a tener consciencia, y que puedan materializarse en el mundo real. Sobre esta misma idea se ha especulado con frecuencia, como ejemplos podemos citar la serie de dibujos animados Código Lyoko, que seguramente será más conocida por los más jóvenes (figura 1.3), o el problema de los tres cuerpos.

No obstante, y a pesar de que la tecnología ha avanzado mucho, no se puede decir que la realidad haya superado a la ficción. La realidad esta muy lejos de ofrecer tan siquiera el grado de inmersión que se plantea en la película «Nivel 13».

En esta asignatura se aborda el problema de la creación y utilización de entornos virtuales. Es decir, el proceso de crear un escenario, su visualización y la interacción con el mismo. Es decir:

- Que abstracciones, modelos y estructuras de datos usar para representar escenas en un sistema informático.
- Que métodos y técnicas utilizar para interaccionar con el escenario.
- Como generar imágenes del mismo.

En la asignatura abordaremos estos tres problemas.

1.1. Informática Gráfica

La Informática Gráfica es una disciplina que se centra en la creación, manipulación y visualización de imágenes y entornos virtuales mediante el uso de tecnología informática. Este campo abarca una amplia gama

Figura 1.1: Lewis Carroll. Alicia en el país de las maravillas: A través del espejo. Madrid: Ediciones Cátedra, 1992.

1: Pedro Calderón de la Barca. La vida es sueño. (Primera edición 1635). Vicens Vives, 2014.

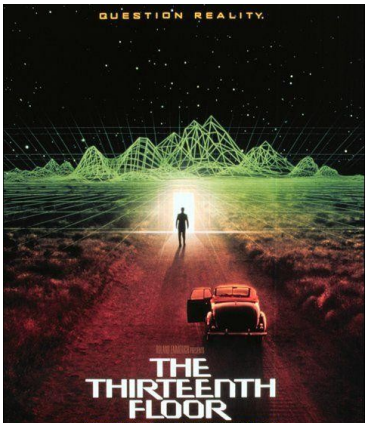

Figura 1.2: Cartel de la película Nivel 13.

Figura 1.3: Cartel de la serie Código Lyoko.

de áreas, desde el diseño de videojuegos y películas de animación hasta la visualización de datos científicos y médicos. A modo de ejemplo citaremos algunos:

Cine. Se utiliza para la creación de efectos visuales en películas.
Videojuegos. Es la herramienta esencial en el desarrollo de videojuegos.
Animación por ordenador. Al igual que en videojuegos, escenarios y personajes se representan como modelos gráficos.
Medicina. Los resultados de pruebas de diagnóstico por imagen (TAC, Resonancia, etc.) se pueden respresentar,visualizary explorar interactivamente como modelos 3D. Estos modelos se pueden usar también para la simulación de procedimientos quirúrgicos.
Arquitectura. Se utiliza para la creación de modelos y visualizaciones de edificios, espacios urbanos.
Diseño industrial. Se utiliza para la creación de modelos de los elementos a diseñar. Estos modelos se suelen usar para realizar simulaciones acortando el proceso de prueba y fabricación.

Figura 1.4: Visualización del modelo 3D de la cabeza de uno de los leones del Patio de los Leones en la Alhambra.

En las aplicaciones gráficas podemos encontrar tres componentes fundamentales: la visualización, el modelado y la interacción.

El modelado se ocupa de la representación de objetos. Incluye los métodos de representación, las estructuras de datos y las operaciones que se realizan con las representaciones. Hay diferentes métodos de modelado, en función de la dimensión del objeto representado y de las operaciones que sea necesario realizar con él.

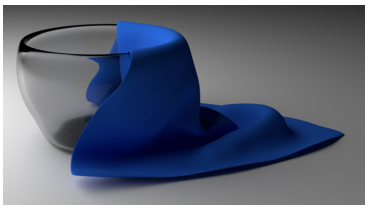

Figura 1.5: Fotograma de la simulación de la caída de una tela realizada con Blender.

La visualización se ocupa de la generación de imágenes de los modelo. Dependiendo de la naturaleza del modelos la visualización puede implicar un amplio rango de acabados, desde la creación de gráficos 2D a la renderización de imágenes fotorealistas en sistemas de Realidad Virtual. La visualización juega un papel crucial en la comunicación de información compleja de manera efectiva.

La interacción es el tercer pilar de la Informática Gráfica y se ocupa de las técnicas que permiten que los usuarios interactúan con los modelos y cómo estos responden a sus acciones. Desde la navegación por entornos en 3D hasta la manipulación de objetos virtuales, la interacción juega un papel crucial en la creación de experiencias inmersivas.

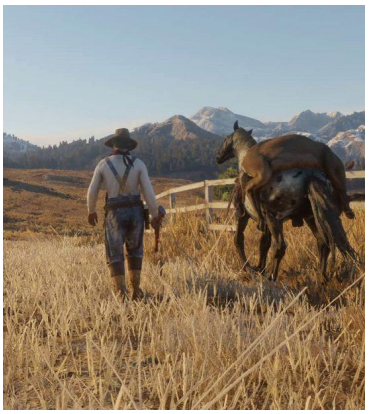

Figura 1.6: Fotograma del juego Red Dead Redemption.

Aunque la mayor parte de las aplicaciones tienen estos tres componentes, encontrándose en la zona rayada del esquema de la Figura 1.7, existen aplicaciones que no tienen alguno de estos componentes. Por ejemplo que calcule propiedades de un modelo previamente creado puede no necesitar ni visualización ni interacción.

1.2. OpenGL

Las prácticas de la asignatura se centrarán en la programación con OpenGL y glut, pero veremos otras herramientas para realizar operaciones específicas, concretamente: Unity 3D, Blender, X3Dom y MeshLab.

OpenGL es una API abierta y estándar que oculta el hardware y hace que la aplicación sea portable (Figura 1.8).

Permite manejar:

- Elementos geométricos
- Propiedades visuales
- Transformaciones
- Especificación de fuentes de luz Hardware
- Especificación de cámara

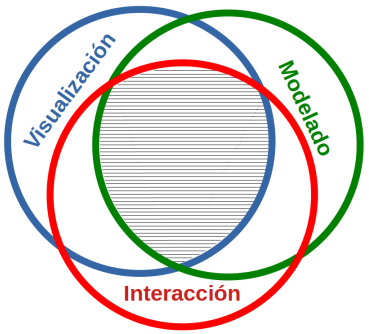

Figura 1.7: Componentes de un sistema gráfico.

Existen muchas implementaciones de OpenGL. La estructura de procesamiento en una implementación típica es un cauce. La aplicación envía ordenes a OpenGL haciendo llamadas a la librería. Las ordenes pueden modificar el estado de OpenGL (p.e. asignar el color de dibujo) o indicar que se dibuje un elemento (p.e. un triángulo).

Cuando OpenGL recibe una orden de dibujo transforma el elemento geométrico, le calcula la iluminación y lo rasteriza, utilizando los parámetros almacenados en el estado (Figura 1.9). Estas operaciones no se realizan necesariamente en este orden, con frecuencia el calculo de iluminación se realiza después de la rasterización.

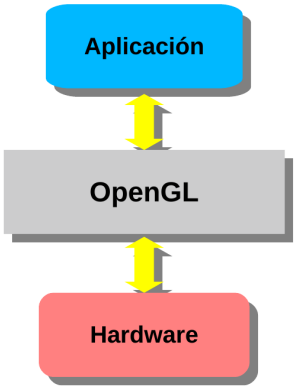

Figura 1.8: OpenGL es una API para la comunicación con el hardware gráfico

OpenGL puede estar implementado por software o hardware. En el primer caso las funciones de OpenGL se ejecutan en CPU, en el segundo caso la mayor parte de las operaciones de OpenGL se ejecutan en la GPU (Unidad Gráfica de Procesamiento). La ejecución en GPU acelera enormemente el dibujo ya que las GPU son procesadores paralelos (tipo SIMD). En su estructura mas simple la GPU tiene dos niveles de procesadores: procesadores de vértices y de fragmentos trabajando en un cauce.

Las primeras GPUs implementaban un cauce de fijo, con funcionalidad fija. Las GPUs modernas (desde 2002) son programables, permitiendo programar tanto los procesadores de vértices como los de fragmentos (a estos programas se les llama shaders). OpenGL permite programar shaders usando OpenGL Shading Language.

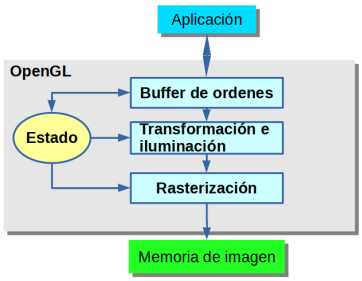

Figura 1.9: Esquema de funcionamiento de OpenGL

A partir de la versión 3 hay dos modos de funcionamiento: el modo de compatibilidad y el **core profile**. El **core profile** es mas reciente y más potente, pero también mas complejo ([KSS16]). En el modo clásico OpenGL incluye una programación por defecto de la GPU, por lo que se puede programar sin necesidad de crear los shaders ([Shr+07]). Comenzaremos trabajando con la versión clásica de OpenGL, conocida también como **compatibility mode**.

OpenGL permite dibujar objetos en 2D y en 3D. En este curso trabajaremos, salvo que se indique lo contrario en tres dimensiones. El sistema de coordenadas que utiliza OpenGL tiene el eje Y en vertical y el eje Z hacia el observador (ver Figura 1.10).

1.2.1. Creación de la ventana de dibujo con glut

Para poder utilizar OpenGL el programa debe crear al menos una ventana de dibujo. Esta operación (que pueden depender bastante del sistema operativo y de sistema de gestión de ventanas) no la realiza OpenGL. Para crear y gestionar las ventanas se debe usar una librería específica como QT, Glut o FLTK.

La alternativa mas simple es usar glut, que es un toolkit sencillo diseñado específicamente para OpenGL [Kil96]. Glut incluye funciones para

interaccionar con el sistema de gestión de ventanas y algunas funciones auxiliares de OpenGL (como la creación de elementos geométricos).

Glut es portable e independiente de la plataforma. Permite crear y gestionar ventanas gráficas y procesar eventos de entrada. Además incluye funciones para dibujar objetos poliédricos simples.

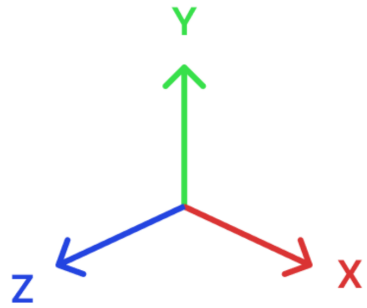

Figura 1.10: Sistema de coordenadas OpenGL

1.2.2. Ejemplo de programa

El siguiente es un ejemplo simple que crea un cubo que gira sobre un plano. El código de este programa (*cubo.c*) se puede descargar de Prado. La figura 1.11 muestra una captura de la aplicación.

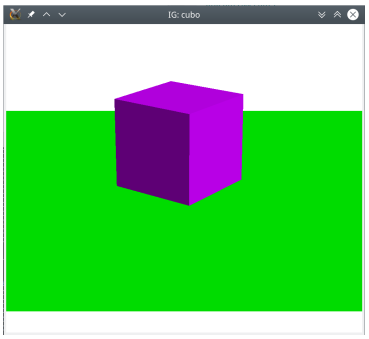

Figura 1.11: Captura de la ejecución del programa cubo

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <GL/glut.h>

float roty=30.0;

void plano( float t ){//Construye un cuadrado horizontal
  glBegin( GL_QUADS );
    glNormal3f( 0.0, 1.0, 0.0 );
    glVertex3f(  t, 0,  t );
    glVertex3f(  t, 0, -t );
    glVertex3f( -t, 0, -t );
    glVertex3f( -t, 0,  t );
  glEnd();
}

void Dibuja( ){
  float pos[4] = {5.0, 5.0, 10.0, 0.0 };
  float morado[4]={0.8,0,1,1}, verde[4]={0,1,0,1};
  glClearColor(1,1,1,1); // Fondo blanco
  glClear( GL_COLOR_BUFFER_BIT  | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  glTranslatef(-0.5,-0.5,-100);
  glLightfv( GL_LIGHT0, GL_POSITION, pos );
  glRotatef( 20, 1.0, 0.0, 0.0 );
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, verde );
  plano(30);
  glRotatef( roty, 0.0, 1.0, 0.0 );
  glTranslatef(0,5,0);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, morado );
  glutSolidCube(10);
  glutSwapBuffers();
}

void Ventana(GLsizei ancho,GLsizei alto) {
  float D=ancho; if(D<alto) D=alto;
  glViewport(0,0,ancho,alto); //fija el area de dibujo
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(-ancho/D,ancho/D,-alto/D,alto/D,5,250);
  glMatrixMode(GL_MODELVIEW);
}

```

```
45 void idle(){
46   roty +=0.15;
47   glutPostRedisplay();
48 }
49
50 int main( int argc, char *argv[] )
51 {
52   glutInit( &argc, argv );
53   glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
54   glutCreateWindow("IG: cubo");
55   glutDisplayFunc( Dibuja );
56   glutReshapeFunc( Ventana );
57   glutIdleFunc(idle);
58   glEnable( GL_LIGHTING );
59   glEnable( GL_LIGHT0 );
60   glEnable(GL_DEPTH_TEST);
61   glutMainLoop();
62 return 0;
63 }
```

La línea 4 incluye la cabecera de la librería glut, no es necesario incluir OpenGL ya que glut.h la incluye.

La línea 6 declara e inicializa la variable que se va a usar para indicar la rotación que se va a aplicar al cubo, expresado en grados.

La función plano (entre las líneas 8 y 10) crea un cuadrado de tamaño t x t, centrado en el origen de coordenadas en el plano y=0. Para crear el cuadrado se le indica a OpenGL que se le van a pasar cuadriláteros (QUADS). La información de las primitivas se pasa entre una llamada a glBegin y una llamada a glEnd. En este bloque se especifica la normal (vector perpendicular de módulo). En este caso, como el plano es horizontal la normal es el vector (0,1,0). A continuación se indican los vértices del cuadrilatero, que se deben dar en sentido antihorario cuando se ve el cuadrilatero por la cara delantera (Figura 1.12).

La función **Dibuja** (líneas 18 a 35) dibuja la escena. Esta función es llamada por glut cada vez que es necesario redibujar la escena. La variable **pos** indica la posición de la fuente de luz (fijate que está dada como un vector de cuatro componentes, mas adelante explicaremos el significado de la cuarta coordenada). En este caso, la luz está colocada en la coordenada (5,5,10) de la escena. Las variables morado y verde definen dos colores. Están expresadas también como vectores de cuatro componentes, que indican los valores de rojo, verde, azul y opacidad. Las componentes se dan normalizadas entre 0 y 1, indicando el 1 el valor máximo. [2]

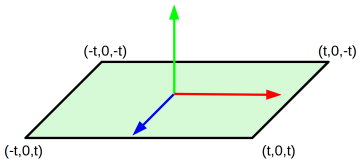

Figura 1.12: Esquema del cuadrilatero dibujado por la función **plano**.

2: Para que se puedan dibujar objetos transparentes no basta con usar valores de opacidad menores que 1.

Cada vez que se llama a **Dibuja** se visualiza un nuevo frame. Para evitar que el dibujo se realice mezclado con la imagen del frame anterior es necesario borrar el framebuffer (la memoria de imagen). En la línea 21 **glClearColor** indica el color con el que queremos que se borre (en este caso blanco), y en la linea 22 **glClear** borra el framebuffer. Observa que tiene como parámetro una operación or de dos constantes, cada una indica un valor a borrar. En este caso se está borrando la información de color y la de profundidad. Esta última se usa para evitar que elementos que están mas lejos de la cámara tapen a objetos que están mas cerca.

Para colocar los objetos en la escena se les aplican transformaciones geométricas, en este ejemplo se usan traslaciones (líneas 24 y 30) y rotaciones

(líneas 26 y 29). Cuando se indica una transformación geométrica se guarda en el estado de OpenGL y se aplica a todos los elementos que se dibujen a partir de ese momento. Para evitar que las transformaciones de un frame afecten a los frame sucesivos inicializamos la transformación con la identidad antes de empezar a dibujar (línea 23). La traslación 24 empuja toda la escena 100 unidades hacía la parte negativa de z y -0,5 en X e Y. [3].

3: La cámara está colocada en el origen de coordenadas mirando a la parte negativa de Z, de este modo la escena se coloca en el campo visual de la cámara

La línea 25 indica la posición en la que está la luz (cada luz tiene un identificador de GL_LIGHTn, en este caso estamos colocando la luz 0).

En la 26 se indica una rotación de 20 grados respecto al eje X (el eje se indica dando un vector, en este caso el (1,0,0), que es el eje X).

La línea 27 indica que el color de la cara delantera de lo que se dibuje a continuación debe ser verde. Este color se aplicará solo al plano, ya que en la línea 31 se cambia el color a morado, antes de dibujar el cubo. El cubo se dibuja con una función de glut **glutSolidCube** (línea 32), que dibuja un cubo del tamaño indicado centrado en el origen. Antes de dibujar el cubo también se aplica una rotación de **roty** grados respecto al eje Y, que servirá para hacer que el cubo rote y una traslación de 5 unidades en Y, para colocarlo sobre el plano. La última línea de **Dibuja** llama a la función **glutSwapBuffers**, que hace que se muestre el frame que se acaba de dibujar.

La siguiente función **Ventana** (líneas 36 a 43) indica en que parte de la ventana debe dibujarse y como se proyecta el modelo en la pantalla. En la línea 38, **glViewport** especifica el área de la ventana que se va a usar para dibujar, en este caso es toda la ventana. La función **glFrustrum** indica la transformación de perspectiva que se va a usar, esto es equivalente a indicar los parámetros de la cámara en una fotografía. Veremos estas funciones mas adelante.

La función **idle**, en las líneas 45 a 48 se ejecuta como función de fondo (siempre que no hay otra función que procesar). Aumenta el ángulo de rotación (línea 46) e indica que es necesario generar un nuevo frame llamando a **glutPostRedisplay** (línea 47).

El programa principal (líneas 50 a 63) crea la ventana de dibujo usando las funciones de glut, declara las funciones que deben responder a los distintos evento y establece algunos parámetros de OpenGL. Para crear la ventana de dibujo inicializa glut (línea 52); establece los parámetros de la ventana con **glutInitDisplayMode** y crea la ventana llamando a **glutCreateWindow**. Esta filosofía de separación de asignación de parámetros de la función que realiza la acción la encontramos con frecuencia en funciones de OpenGL y glut, también hemos visto ya el paso de parámetros haciendo un or. En este caso los parámetros que se pasan para la creación de la ventana son:

GLUT_RGBA Indica que se use formato **RGB**α para los colores.

GLUT_DOUBLE Usa dos copias de la memoria de imagen. Se dibuja en la que está oculta y se intercambian con la llamada a **glutSwapBuffers**. Esto evita el parpadeo (**flickering**).

GLUT_DEPTH Usa un buffer de profundidad para almacenar la distancia a la cámara a la que está el objeto que es visible en cada pixel. Es necesario para hacer eliminación de partes ocultas usando Z-Buffer.

Las tres instrucciones siguientes (**glutDisplayFunc**, **glutReshapeFunc** y **glutIdleFunc**) indican las funciones de nuestro programa que se deben ejecutar cuando: sea necesario redibujar la escena, se cambie el tamaño de la ventana o cuando no haya ninguna acción que procesar. Los argumentos que reciben son las funciones **Dibuja**, **Ventana** e **idle**.

Las siguientes instrucciones son llamadas a la función **glEnable** que activa opciones de OpenGL. Concretamente:

GL_LIGHTING Activa el cálculo de iluminación.
GL_LIGHT0 Enciende la luz cero (**LIGHT0**).
GL_DEPTH_TEST Activa la eliminación de partes ocultas usando el algoritmo Z-Buffer.

Para compilar el programa se puede usar la siguiente orden:

```
1 gcc cubo.c -lglut -lGLU -lGL -o cubo
```

1.2.3. Transformaciones geométricas

Las transformaciones geométricas son funciones que modifican las coordenadas de los puntos. Existen varias transformaciones geométricas simples que son muy útiles en Informática Gráfica para la definición de escenas y animaciones:

- **Traslación:** desplazar todos los puntos del espacio de igual forma, es decir, en la misma dirección y la misma distancia.
- **Escalado:** estrechar o alargar las figuras en una o varias direcciones.
- **Rotación:** rotar los puntos un ángulo en torno a un eje.

La transformación de **traslación** $T_{\mathbf{d}}$ en $\mathbb{E}_3$ es una transformación geométrica que desplaza cualquier punto $\mathbf{p} = (x, y, z)$ sumándole un vector[4] $\mathbf{d} = (d_x, d_y, d_z)$, es decir:

$$T_{\mathbf{d}}(\mathbf{p}) \equiv \mathbf{p} + \mathbf{d} = (x, y, z) + (d_x, d_y, d_z) = (x + d_x, y + d_y, z + d_z)$$

La Figura 1.13 muestra el ejemplo de la aplicación de traslaciones a un objeto simple. El vector de traslación está representado en rojo.

Las transformaciones de **escalado** multiplican cada componente por un factor de escala. Una transformación de **escalado** $E_{\mathbf{s}}$ con factor de escala $\mathbf{s} = (s_x, s_y, s_z)$ transforma el punto $\mathbf{p} = (x, y, z)$ según:

$$E_{\mathbf{s}}(\mathbf{p}) \equiv (s_x x, s_y y, s_z z)$$

La Figura 1.14 muestra el ejemplo de la aplicación de un escalado al objeto de la Figura 1.13.

4: Usamos negrita para denotar vectores

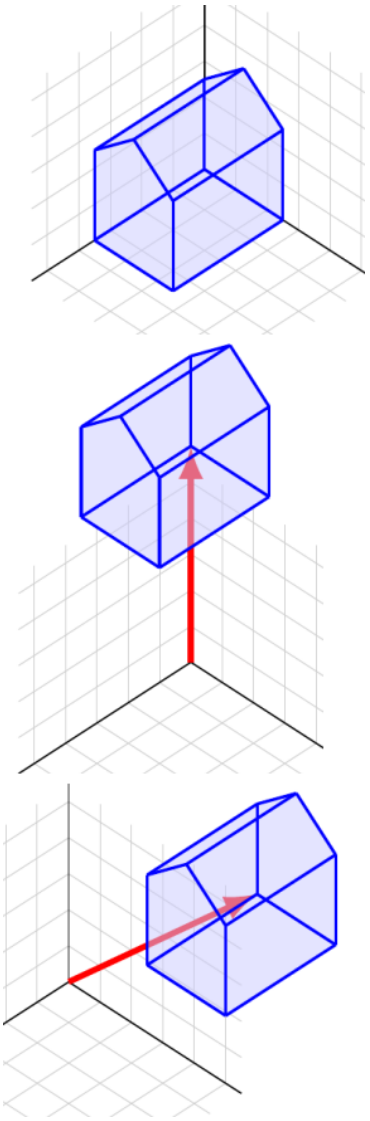

Figura 1.13: Ejemplos de traslaciones. El objeto de la figura superior se ha trasladado con $\mathbf{d} = (0, 6, 0)$ en la central y con $\mathbf{d} = (6, 6, 0)$ en la inferior.

1.3. Eliminación de partes ocultas: ZBuffer

Cuando se visualiza un modelo 3D es normal que se proyecten distintas superficies en la misma zona de pantalla. Para que la visualización sea entendible y se corresponda con la imagen real es necesario que solo se dibujen los elementos que están mas cerca de la cámara (que deben tapar a los que quedan detras) [5]. A este proceso se le denomina **eliminación**

de partes ocultas. Las figuras 1.15 y 1.16 corresponden a la misma escena, visualizada con y sin eliminación de partes ocultas.

La eliminación de partes ocultas suele realizarse con el algoritmo **Z-Buffer**, que realiza el proceso a nivel de pixel. Para ello se utiliza un **buffer de profundidad**, un array 2D de las mismas dimensiones del **frame buffer** en el que para cada pixel se almacena la distancia a la cámara a la que se encuentra el objeto dibujado en el pixel. Esto permite realizar el proceso de un modo muy simple: Para cada pixel que se va a dibujar se calcula su profundidad respecto a la cámara y solo se dibuja si no se ha dibujado previamente un objeto mas próximo. Antes de empezar a dibujar el buffer de profundidad se inicializa a la distancia máxima. El algoritmo es el siguiente:

```
color canvas[resx,resy] // Memoria de imagen
int zbuffer[resx,resy]=Zfar // buffer de profundidad
for each primitiva
  Rasterizar(primitiva) // se descompone en un conjunto de
     fragmentos
     for each fragment
       if zbuffer[x,y] > fragment.depth
         zbuffer[x,y] = fragment.depth
         canvas[x,y] = fragment.color
```

En este proceso la función Rasterizar descompone la primitiva en un conjunto de pixeles (fragmentos). Cada uno de ellos tiene asociado el color y la profundidad (depth). Las primitivas son los elementos geométricos que se dibujan y los fragmentos se corresponden con pixeles.

La figura 1.17 muestra un esquema del proceso, la escena está en la parte central de la figura, la imagen generada a la derecha y el buffer de profundidad a la izquierda.

Las profundidades se transforman a un intervalo:

$$z' = \frac{Far + Near}{Far - Near} - \frac{2}{Z}\frac{Far\ Near}{Far - Near}$$

Siendo **Z** la profundidad del fragmento, **Far** la distancia máxima y **Near** la mínima. Esta expresión genera profundidades normalizadas entre -1 y 1, que ee discretizan, para almacenarlas como enteros sin signo.

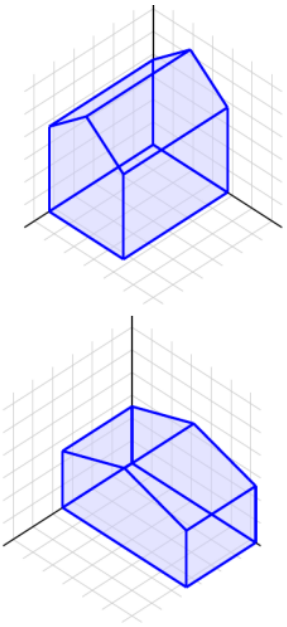

Figura 1.14: Ejemplos de escalados. El objeto de la figura superior se ha escalado con s=(1.5,1.5,1.5) en la inferior con s=(2.5,1,1) en la inferior.

5: Salvo que las superficies sean transparentes

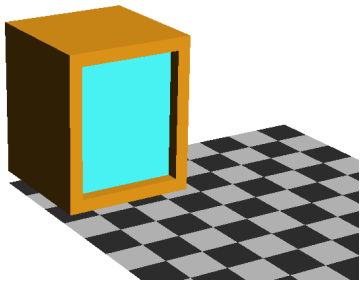

Figura 1.15: Escena visualizada con eliminación de partes ocultas

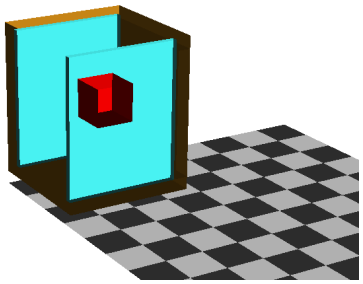

Figura 1.16: Escena visualizada sin eliminación de partes ocultas

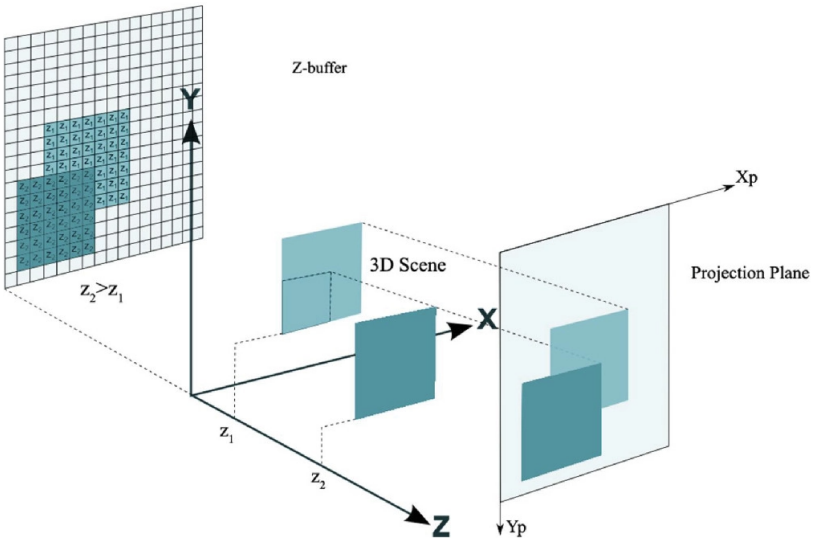

Figura 1.17: Esquema del funcionamiento del Z-Buffer
https://docs.hektorprofe.net/graficos-3d/24-profundidad-con-z-buffer/

1.4. Ejercicios

1. Quita el parámetro **GLUT_DOUBLE** de la inicialización de la venta en el programna **cubo** y comprueba y explica el resultado.
2. Comprueba y explica el resultado de quita el parámetro **GLUT_-DEPTH**.
3. Comprueba y explica el resultado de comentar la activación de la luz cero.
4. Cual será el resultado de cambiar el valor del incremento de **roty** en la línea 46.
5. ¿Que pasá si el valor del incremento de **roty** es negativo?
6. Modifica el programa para que hacer que el plano se vea marrón.
7. Añade una esfera azul a la derecha del cubo.
8. Piensa ejemplos de aplicaciones que utilice Gráficos pero no tengan los tres componentes: visualización, interacción o modelado.
9. La figura 1.18 muestra dos paralelepípedos, ¿Cual puede ser la razón de que se vea rallada la parte central?
10. ¿Que parámetros se podrían cambiar para reducir este efecto?
11. Los dos últimos parámetros de la función**glFrustrum** son las distancias a los planos de recortado delantero y trasero (los elementos que estén mas cerca que el plano delantero o mas lejos que el trasero no se dibuja). ¿Por que se introducen estos parámetros?

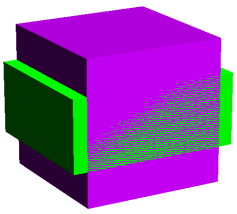

Figura 1.18: Imagen con dos paralelepípedos

Bibliografía

[Bot+10] Mario Botsch et al. *Polygon mesh processing*. CRC press, 2010.

[Cig+08] Paolo Cignoni et al. «MeshLab: an Open-Source Mesh Processing Tool». En: *Eurographics Italian Chapter Conference*. Ed. por Vittorio Scarano, Rosario De Chiara y Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[Kil96] Mark J Kilgard. *The OpenGL utility toolkit (GLUT) programming interface API version 3*. [accedido 26-Abril-2024]. 1996. URL: https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf.

[KSS16] John Kessenich, Graham Sellers y Dave Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. [accedido 26-Abril-2024]. Addison-Wesley Professional, 2016. URL: https://www.cs.utexas.edu/users/fussell/courses/cs354/handouts/Addison.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf.

[Shr+07] Dave Shreiner et al. *OpenGL (R) programming guide: The official guide to learning OpenGL (R), version 2.1*. [accedido 26-Abril-2024]. Addison-Wesley Professional, 2007. URL: https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/The%20Official%20Guide%20to%20Learning%20OpenGL%2C%20Version%202.1.pdf.