Dpto. de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingenierías Informática y Telecomunicación

Prácticas de Informática Gráfica

Autores:
Pedro Cano
Antonio López
Domingo Martín
Francisco J. Melero

Curso 2017/18

La Informática Gráfica

La gran ventaja de los gráficos por ordenador, la posibilidad de crear mundos virtuales sin ningún tipo de límite, excepto los propios de las capacidades humanas, es a su vez su gran inconveniente, ya que es necesario crear toda una serie de modelos o representaciones de todas las cosas que se pretenden obtener que sean tratables por el ordenador.

Así, es necesario crear modelos de los objetos, de la cámara, de la interacción de la luz (virtual) con los objetos, del movimiento, etc. A pesar de la dificultad y complejidad, los resultados obtenidos suelen compensar el esfuerzo.

Ese es el objetivo de estas prácticas: convertir la generación de gráficos mediante ordenador en una tarea satisfactoria, en el sentido de que sea algo que se hace "con ganas".

Con todo, hemos intentado que la dificultad vaya apareciendo de una forma gradual y natural. Siguiendo una estructura incremental, en la cual cada práctica se basará en la realizada anteriormente, planteamos partir desde la primera práctica, que servirá para tomar un contacto inicial, y terminar generando un sistema de partículas con animación y detección de colisiones.

Esperamos que las prácticas propuestas alcancen los objetivos y que sirvan para enseñar los conceptos básicos de la Informática Gráfica, y si puede ser entreteniendo, mejor.

Índice general

Práctica 1

Introducción. Modelado y visualización de objetos 3D sencillos

1.1. Objetivos

Con esta práctica se quiere que el alumno aprenda:

- A crear estructuras de datos que permitan representar objetos 3D sencillos
- A utilizar las primitivas de dibujo de OpenGL para dibujar los objetos

1.2. Desarrollo

Para el desarrollo de esta práctica se entrega el esqueleto de una aplicación gráfica basada en eventos, mediante GLUT y Qt 5, y con la parte gráfica realizada por OpenGL. Para facilitar su uso, la aplicación permite abrir una ventana, mostrar unos ejes y mover una cámara básica.

El alumno deberá crear y visualizar un tetraedro y un cubo. Para ello, creará las estructuras de datos que permitan representarlos mediante sus vértices y caras. Usando dicha información y las primitivas de dibujo de OpenGL los visualizará con los siguientes modos:

- Puntos
- Alambre
- Sólido
- Ajedrez

Para poder visualizar en modo alambre (también para el modo sólido y ajedrez) lo que se hace es mandar a OpenGL como primitiva los triángulos, `GL_TRIANGLES`, y cambiar la forma en la que se visualiza el mismo mediante la instrucción `glPolygonMode`, permitiendo el dibujar los vértices, las aristas o la parte sólida.

Para el modo ajedrez basta con dibujar en modo sólido pero cambiando alternativamente el color de relleno.

Se usarán las siguientes teclas para activar los distintos modos y objetos:

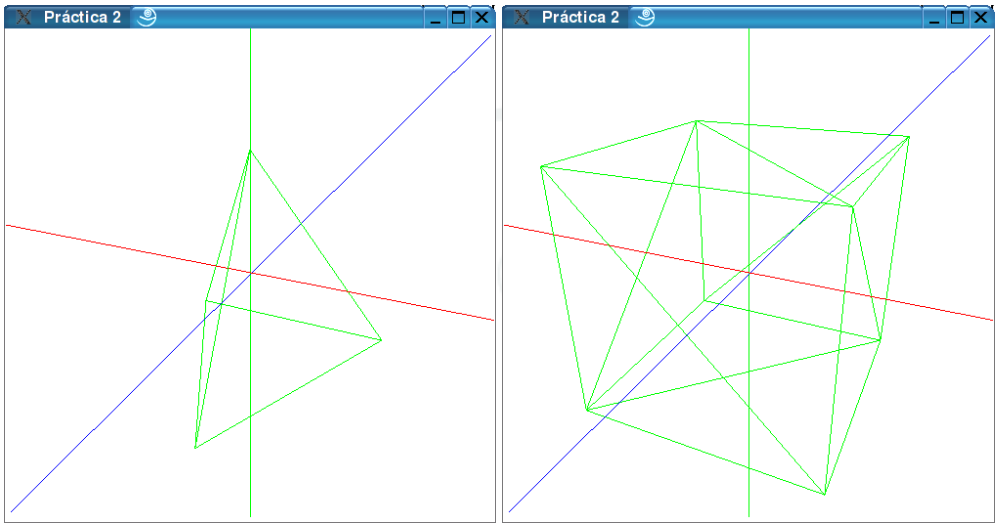
- Tecla p: Visualizar en modo puntos
- Tecla l: Visualizar en modo líneas/aristas
- Tecla f: Visualizar en modo relleno
- Tecla c: Visualizar en modo ajedrez
- Tecla 1: Activar tetraedo
- Tecla 2: Activar cubo

1.3. Evaluación

La evaluación de la práctica, sobre 10 puntos, se hará del modo siguiente:

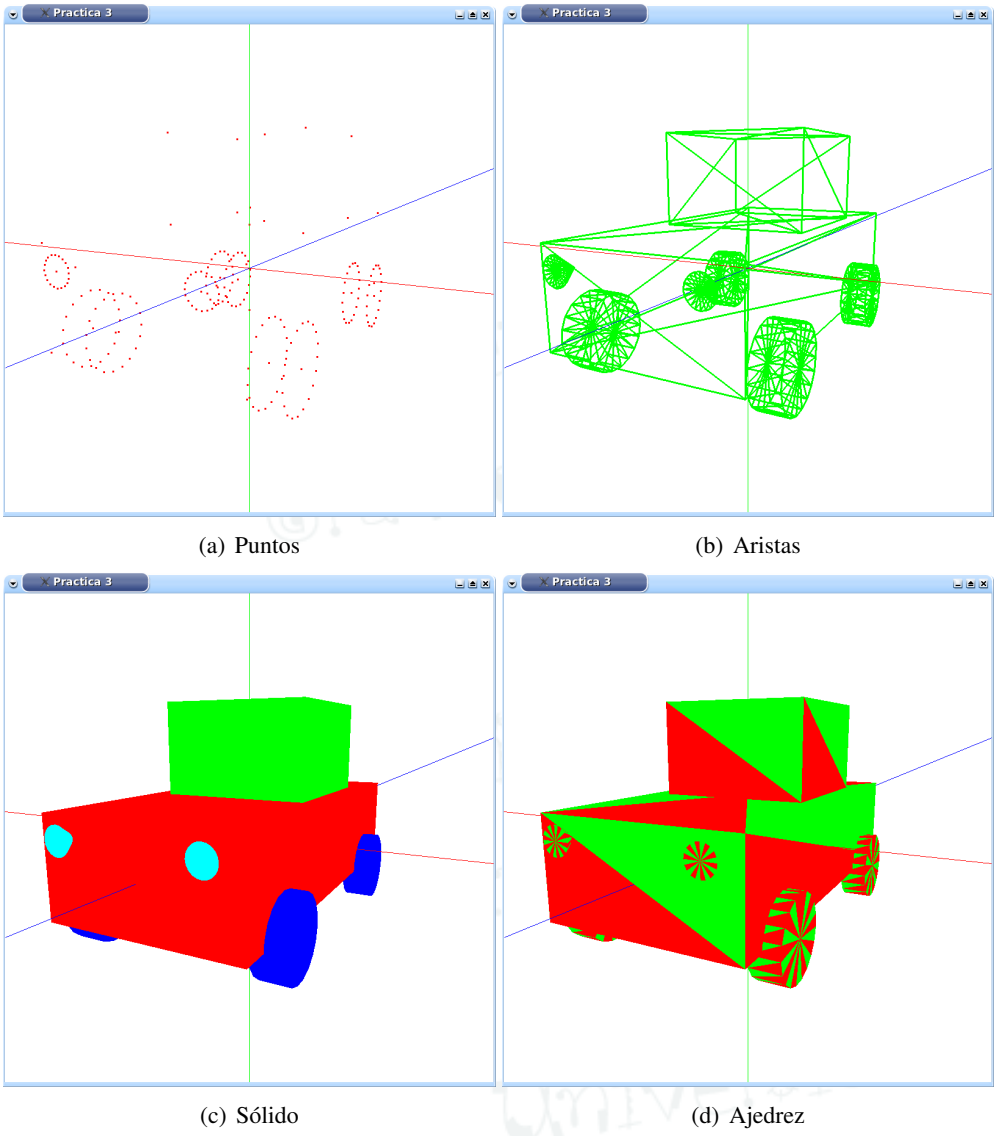
- Creación de las estructuras de datos para modelar un tetraedro y un cubo mediante vértices. Visualización en modo puntos (6 pt.)

- Creación del código que permite visualizar en los modos alambre, sólido y ajedrez. (4pt)



(a) Tetraedro (b) Cubo

Figura 1.1: Tetraedro y cubo visualizados en modo alambre.



(a) Puntos (b) Aristas

(c) Sólido (d) Ajedrez

Figura 1.2: Coche mostrado con los distintos modos de visualización.

1.4. Extensiones

1.5. Duración

La práctica se desarrollará en 1 sesión

1.6. Bibliografía

- Mark Segal y Kurt Akeley;*The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- M. E. Mortenson; *Geometric Modeling*; John Wiley & Sons,1985

Práctica 2

Modelos PLY y Poligonales

2.1. Objetivos

Aprender a:

- A leer modelos guardados en ficheros externos en formato PLY (Polygon File Format) y su visualización.
- Modelar objetos sólidos poligonales mediante técnicas sencillas. En este caso se usará la técnica de modelado por revolución de un perfil alrededor de un eje de rotación.

2.2. Desarrollo

PLY es un formato para almacenar modelos gráficos mediante listas de vértices, caras poligonales y diversas propiedades (colores, normales, etc.) que fue desarrollado por Greg Turk en la universidad de Stanford durante los años 90. Para más información consultar:
http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html

Para la realización de la práctica, en primer lugar, se visualizarán modelos de objetos guardados en formato PLY usando los modos de visualización implementados en la primera práctica. Para ello, se entregará el código de un lector básico de ficheros PLY para objetos únicamente compuestos por vértices y caras triangulares, que devuelve un vector de coordenadas de los vértices y un vector de los índices de vértices que forman cada cara. Se creará una estructura de datos que guarde los vectores anteriores (se recomienda usar STL y el fichero auxiliar vertex.h).

En segundo lugar, se ha de crear un código que a partir del conjunto de puntos que representen un perfil respecto a un plano principal ($X = 0$ ó $Y = 0$ ó $Z = 0$), de un parámetro que indique el número de lados longitudinales del objeto a generar por revolución y de un eje de rotación, calcule el conjunto de vértices y el conjunto el caras que representan el sólido obtenido.

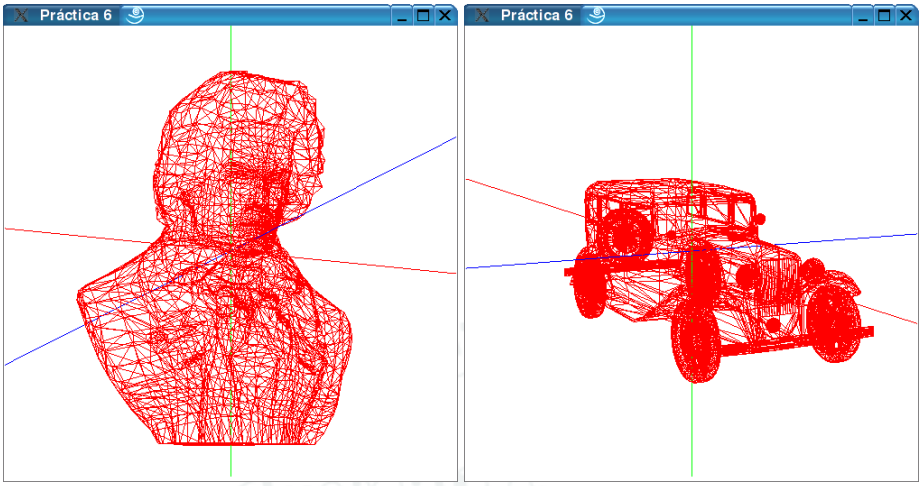
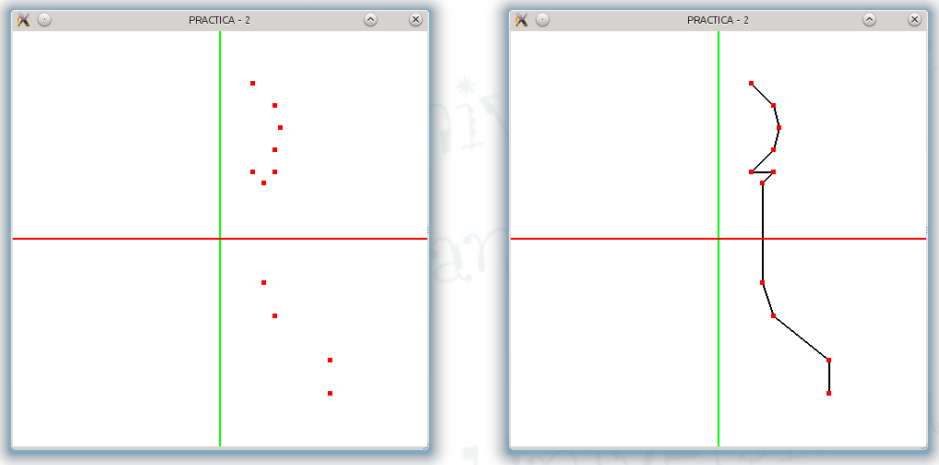


Figura 2.1: Objetos PLY.



(a) Puntos (b) Polilínea

Figura 2.2: Perfil inicial.

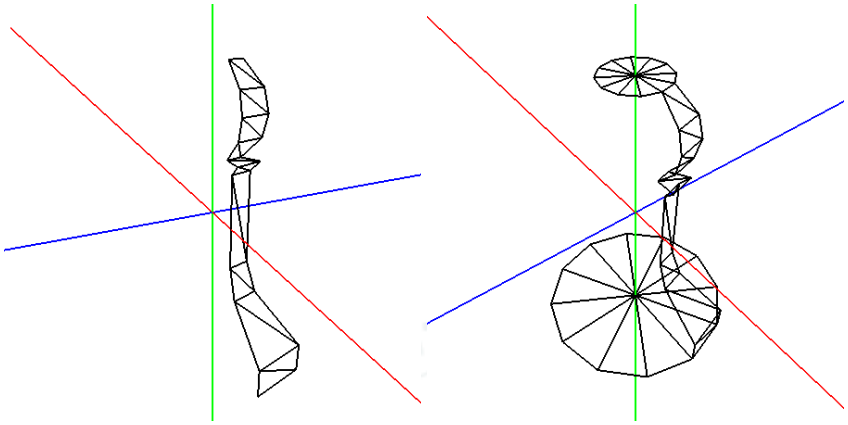
Los pasos para crear un sólido por revolución serían:

- Sea, por ejemplo, un perfil inicial Q_1 en el plano $Z = 0$ definido como:

$$Q_1\left(p_1(x_1,y_1,0),...,p_M(x_M,y_M,0)\right),$$

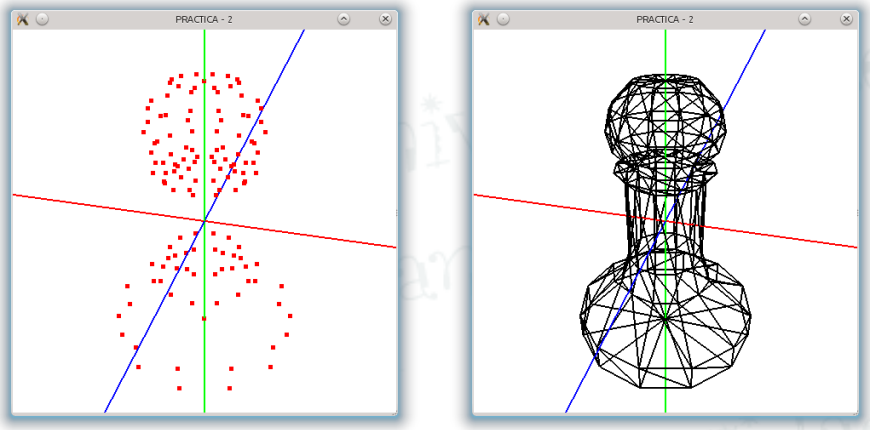
  siendo $p_i(x_i,y_i,0)$ con $i = 1,...M$ los puntos que definen el perfil (ver figura 2.2).

- Se toma como eje de rotación el eje Y y si N es número lados longitudinales, se obtienen los puntos o vértices del sólido poligonal a construir multiplicando Q_1 por N sucesivas transformaciones de rotación con respecto al eje Y, a las que notamos por



(a) (b)

Figura 2.3: Caras del sólido a construir: (a) logitudinales (solo un lado es mostrado) y (b) incluyendo las tapas superior e inferior.



(a) Puntos (b) Aristas

(c) Sólido (d) Ajedrez

Figura 2.4: Sólido generado por revolución con distintos modos de visualización.

$R_Y(\alpha_j)$ siendo α_j los valores de N ángulos de rotación equiespaciados. Se obtiene un conjunto de $N \times M$ vértices agrupados en N perfiles Q_j, siendo:

$$Q_j = Q_1 R_Y(\alpha_j), \quad \text{con } j = 1, \ldots, N$$

- Se guardan $N \times M$ los vértices obtenidos en un vector de vértices según la estructura de datos creada en la práctica anterior.
- Las caras longitudinales del sólido (triángulos) se crean a partir de los vértices de dos perfiles consecutivos Q_j y Q_{j+1}. Tomando dos puntos adyacentes en cada uno de los dos perfiles Q_j y Q_{j+1} y estando dichos puntos a la misma altura, se pueden crear dos triángulos. En la figura 2.3(a) se muestran los triángulos así obtenidos solamente para un lado longitudinal para una mejor visualización. Los vértices de los triángulos tienen que estar ordenados en el sentido contrario a las agujas del reloj.
- A continuación creamos las tapas del sólido tanto inferior como superior (ver figura 2.3(b)). Para ello se han de añadir dos puntos al vector de vértices que se obtienen por la proyección sobre el eje de rotación del primer y último punto del perfil inicial. Estos dos vértices serán compartidos por todas las caras de las tapas superior e inferior.
- Todas las caras, tanto las longitudinales como las tapas superior e inferior, se almacenan en la estructura de datos creada para las caras en la práctica anterior.

El modelo poligonal finalmente obtenido también se podrá visualizar usando cualquiera de los distintos modos de visualización implementados para la primera práctica (ver figura 2.4).

Dos consideraciones sobre la implementación del código para el objeto por revolución: primera, se puede hacer un tratamiento diferenciado cuando uno o ambos puntos extremos del perfil inicial están situados sobre el eje de rotación y segunda, el perfil inicial se puede leer de un fichero PLY cuyo contenido sólo ha de tener las coordenadas de los puntos de éste (no es difícil crear manualmente un perfil con un fichero PLY, véase el siguiente ejemplo, donde hay 11 vértices y una sola cara que no se utilizará)

```
ply
format ascii 1.0
element vertex 11
property float32 x
property float32 y
property float32 z
element face 1
property list uchar uint vertex_indices
end_header
1.0 -1.4 0.0
1.0 -1.1 0.0
0.5 -0.7 0.0
0.4 -0.4 0.0
0.4 0.5 0.0
```

```
0.5 0.6 0.0
0.3 0.6 0.0
0.5 0.8 0.0
0.55 1.0 0.0
0.5 1.2 0.0
0.3 1.4 0.0
3 0 1 2
```

Se usarán las siguientes teclas para activar los distintos modos y objetos:

- Tecla p: Visualizar en modo puntos
- Tecla l: Visualizar en modo líneas/aristas
- Tecla f: Visualizar en modo relleno
- Tecla c: Visualizar en modo ajedrez
- Tecla 1: Activar tetraedo
- Tecla 2: Activar cubo
- Tecla 3: Activar objeto PLY cargado
- Tecla 4: Activar objeto por revolución

2.3. Evaluación

La evaluación de la práctica, sobre 10 puntos, se hará del modo siguiente:

- Lectura y visualizaciónde ficheros PLY. y en modo puntos (3 pts.).
- Creación del código para el modelado de objetos por revolución (7 pts.).

2.4. Extensiones

Se propone como extensión modelar sólidos por barrido a partir de un contorno cerrado.

2.5. Duración

La práctica se desarrollará en 3 sesiones

2.6. Bibliografía

- Mark Segal y Kurt Akeley; *The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.
- J. Vince; *Mathematics for Computer Graphics*; Springer 2006.

Práctica 3

Modelos jerárquicos

3.1. Objetivos

Con esta práctica el alumno aprenderá a:

- Diseñar modelos jerárquicos de objetos articulados.
- Controlar los parámetros de animación de los grados de libertad de modelos jerárquicos usando OpenGL.
- Gestionar y usar la pila de transformaciones de OpenGL.

3.2. Desarrollo

Para realizar un modelo jerárquico es importante seguir un proceso sistemático, tal y como se ha estudiado en teoría, poniendo especial interés en la definición correcta de los grados de libertad que presente el modelo.

Para modificar los parámetros asociados a los grados de libertad del modelo utilizaremos el teclado. Para ello tendremos que escribir código para modificar los parámetros como respuesta a la pulsación de teclas.

Las acciones a realizar en esta práctica son:

1. Diseñar un modelo jerárquico con al menos 3 grados de libertad distintos (al menos deben aparecer giros y desplazamientos). Puedes tomar como ejemplo el diseño de una grúa semejante a las del ejemplo (ver figura 3.1). En el ejemplo, estas gruas tienen al menos tres grados de libertad: ángulo de giro de la torre, giro del brazo y altura del gancho.

2. Diseñar el grafo del modelo jerárquico del objeto diseñado, determinando el tamaño de las piezas y las transformaciones geométricas a aplicar (tendrás que entregar el grafo del modelo en papel, o en formato electrónico: pdf, jpeg, etc, cuando entregues la práctica).

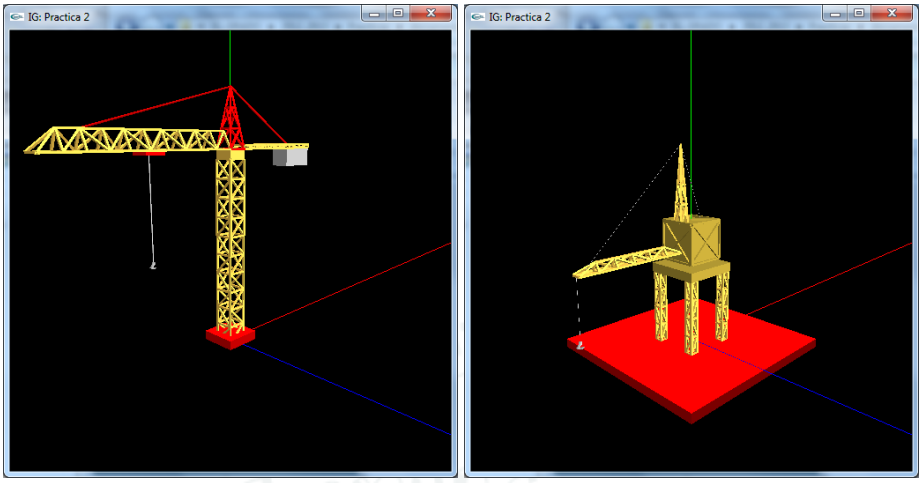


Figura 3.1: Ejemplos del resultado de la práctica 3.

3. Crear las estructuras de datos necesarias para almacenar el modelo jerárquico. El modelo debe contener la información necesaria para definir los parámetros asociados a la construcción del modelo (medidas de elementos, posicionamiento, etc.) y los parámetros que se vayan a modificar en tiempo de ejecución (grados de libertad, etc.). Hay que tener en cuenta que un modelo jerárquico esta formado por otros objetos, normalmente más sencillos, entre los que existen relaciones de dependencia hijo-padre, por lo que se deberá poder almacenar en la estructura de datos los distintos componentes que lo forman , asi como las transformaciones que le afectan a cada uno con su tipo de transformación y sus parámetros.

   Si el modelo no almacena explícitamente el grafo jerárquico, sino que dicha jerarquía se contruye en base objetos ya creados y a la gestión de la pila de transformaciones de OpenGL, se deberá crear el código que permite representar el modelo jerárquico en base a los parámetros de construcción y grados de libertad que se definan en el modelo.

4. Inicializar el modelo jerárquico diseñado para almacenar en la estructura de datos los componentes y parámetros necesarios para su construcción.

5. Crear el codigo necesario para visualizar el modelo jerárquico utilizando los valores de los parámetros del modelo almacenado en la estructura de datos. El alumno podra utilizar las funciones de visualización implementadas en las anteriores practicas que permiten visualizar los modelos con las distintas técnicas implementadas.

6. Incorporar código para cambiar los parámetros modificables del modelo y para controlar su movimiento. Añadir la ejecución de dicho código en las funciones de control de pulsación de teclas para modificar el modelo de forma interactiva. Hay que tener presente los límites de cada movimiento.

7. Ejecutar el programa y comprobar que los movimientos son correctos.

3.2.1. Reutilización de elementos

En esta práctica para construir los modelos jerárquicos se deben utilizar otros elementos más sencillos que al combinarse mediante instanciación utilizando las transformaciones geométricas necesarias, nos permitirán construir modelos mucho más complejos. Se puede partir de cualquier primitiva que nos ofrezcan las propias librerías de OpenGL, o reutilizar los elementos implementados en las prácticas anteriores.

3.2.2. Resultados entregables

El alumno entregará un programa que represente y dibuje un modelo jerárquico con al menos tres grados de libertad, cuyos parámetros se podrán modificar por teclado. Para esta práctica se deberá incorporar el control con las siguientes teclas para activar/desactivar cada acción posible de las realizadas en las 3 primeras prácticas:

- Tecla p: Visualizar en modo puntos
- Tecla l: Visualizar en modo líneas/aristas
- Tecla f: Visualizar en modo relleno
- Tecla c: Visualizar en modo ajedrez
- Tecla 1: Activar tetraedo
- Tecla 2: Activar cubo
- Tecla 3: Activar objeto PLY cargado
- Tecla 4: Activar objeto por revolución
- Tecla 5: Activar objeto jerárquico
- Tecla Z/z: modifica primer grado de libertad del modelo jerárquico (aumenta/disminuye)
- Tecla X/x: modifica segundo grado de libertad del modelo jerárquico (aumenta/disminuye)
- Tecla C/c: modifica tercer grado de libertad del modelo jerárquico (aumenta/disminuye)

3.3. Evaluación

La evaluación de la práctica, sobre 10 puntos, se hará del modo siguiente:

- Diseño del modelo jerárquico y del grafo correspondiente que muestre las relaciones entre los elementos (2 puntos)

- Creación de las estructuras necesarias para almacenar el modelado jerárquico (4 puntos)
- Creación del código para poder visualizar de forma correcta el modelo (1 punto)
- Control interactivo del cambio de los valores que definen los grados de libertad del modelo (3 puntos)

3.4. Extensiones

Como extensión optativa se propone la incorporación de animación automática de los componentes del modelo. Para ello:

- Añade al modelo lo que estimes necesario para almacenar una velocidad de movimiento para cada uno de los parámetros.
- Añade opciones en el control de teclas para fijar la velocidad de cada parámetro a un valor positivo, negativo o cero.
- Ahora puedes animar el modelo haciendo que el valor del parámetro que modifica cada grado de libertad se modifique segun su velocidad.

Para animar el modelo utiliza una función de fondo que se ejecute de forma indefinida en el ciclo de ejecución de la aplicación OpenGL, en la que puedes realizar la actualización de cada parámetro en función de su velocidad. La función de fondo estará creada en el fichero principal y se activa desde el programa principal con:

```
glutIdleFunc( idle );
```

siendo ”void idle()” el nombre de la función que se llama para modificar los parámetros. Esta función de animación debe cambiar los parámetros del modelo en función de la velocidad y para que se redibuje, llamar a:

```
glutPostRedisplay();
```

Cambia finalmente el procedimiento de entrada para que desde el teclado se pueda cambiar también las velocidades de modificación de los grados de libertad:

- Tecla B/b: incrementa/decrementa la velocidad de modificación del primer grado de libertad del modelo jerárquico
- Tecla N/n: incrementa/decrementa la velocidad de modificación del segundo grado de libertad del modelo jerárquico
- Tecla M/m: incrementa/decrementa la velocidad de modificación del tercer grado de libertad del modelo jerárquico

3.5. Duración

La práctica se desarrollará en 3 sesiones.

3.6. Bibliografía

- Mark Segal y Kurt Akeley; *The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.

3.7. Algunos ejemplos de modelos jerárquicos

En las figuras 3.2 y 3.3 podéis ver algunos ejemplos de modelos jerárquicos que se pueden construir para la práctica (simplificando todo lo que se quiera los distintos elementos que los componen).

Estudiar con detalle cada uno y seleccionar el que os interese, o diseñar otro que tenga al menos 3 grados de libertad similares a los de las gruas que tenéis en el ejemplo.

Figura 3.2: Ejemplos de posibles modelos jerárquicos.

Figura 3.3: Ejemplos de posibles modelos jerárquicos.

Práctica 4

Iluminación y texturas

4.1. Objetivos

Los objetivos de esta práctica son conseguir un mayor realismo mediante el uso de la iluminación y las texturas. Para ello necesitamos hacer lo siguiente:

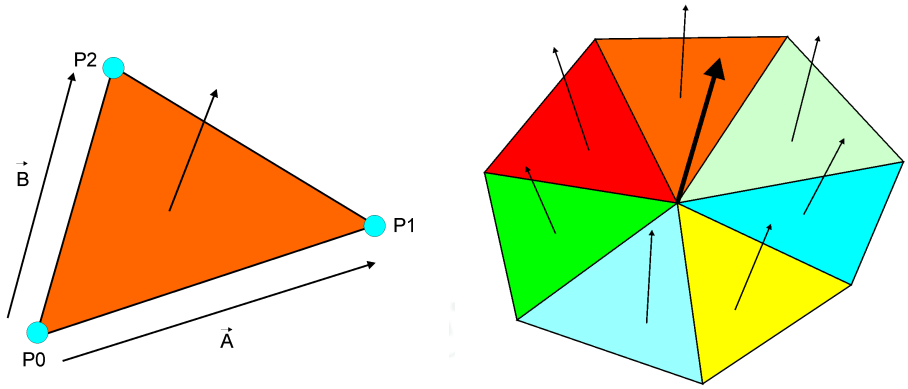
- Iluminación
  - Aprender a generar las normales de las caras y de los vértices para un modelo poligonal hecho con triángulos.
  - Añadir y usar fuentes de luz
  - Añadir y usar materiales
  - Modificar y usar un modelo de reflexión.
  - Iluminar un objeto
- Texturas
  - Calcular las coordenadas de textura para un objeto sencillo
  - Cargar y usar visualizar, con y sin iluminación, una textura

4.2. Desarrollo

El objetivo de esta práctica es conseguir un mayor realismo en la visualización de las escenas y para ello vamos a integrar el proceso de iluminación y el uso de texturas.

Para poder aplicar la iluminación es necesario disponer de al menos un objeto, una fuente de luz, y un modelo que nos indique cómo se refleja la luz en dicho objeto. Para calcular la reflexión no sólo es importante la posición de la luz (otros parámetros como el color son menos importantes) sino que también debemos saber la orientación de la superficie. Para ello es necesario calcular las normales.

Aunque con la iluminación se obtiene una substancial mejora en el realismo, para mejorarlo recurrimos a las texturas, que consiste, de una manera muy general, en pegarle una



(a) Normal de un triángulo (b) Normal de un vértice

Figura 4.1: Cálculo de las normales

foto a un objeto o parte del mismo. Para ello tendremos que ver cómo se relacionan imagen y objeto mediante las coordenadas de textura.

4.2.1. Cálculo de normales.

Lo primero que vamos a hacer es el cálculo de las normales. En una superficie de la que disponemos de su descripción exacta, el problema consistiría en calcular la normal en un punto de la superficie. Nuestros modelos de vértices y triángulos en general[1] son aproximaciones de modelos curvos y continuos por lo que obtenemos también, en general, son aproximaciones.

Dado que tenemos triángulos y vértices, podremos calcular las normales de dichos elementos. Empezamos por el cálculo de las normales de los triángulos.

El cálculo de la normal de un triángulo, es muy sencillo. La normal del plano que incluye al triángulo se obtiene de la siguiente manera. Dados los puntos P_0, P_1 y P_2, podemos calcular los vectores $\vec{A} = P_1 - P_0$ y $\vec{B} = P_2 - P_0$. Si aplicamos el producto vectorial $\vec{A} \times \vec{B}$ obtenemos el vector normal, $\vec{N}$, cuyo sentido viene dado por la regla de la mano derecha (en general, se entiende que la normal apunta hacia el lado exterior del polígono) (ver figura 4.1(a)).

La normal de la cara se usa no solo para cálculos de iluminación sino que también sirve para determinar la orientación de la cara, si la misma mira hacia adentro del objeto o hacia afuera. Es importante que mire hacia afuera para que la cara pueda ser visualizada, en el caso de que se indique, mediante la instrucción glPolygonMode, que sólo se muestren las caras orientadas hacia adelante, mediante el valor GL_FRONT. Una solución consiste en visualizar las caras que miran hacia adelante y hacia atrás, mediante el valor GL_FRONT_AND_BACK, pero salvo que se quiera hacer porque se planee el introducirse en el interior del objeto, dicha opción evitará que se pueda mejorar el rendimiento, pues si

[1] Si el objeto es poligonal, como por ejemplo un cubo, la representación es exacta

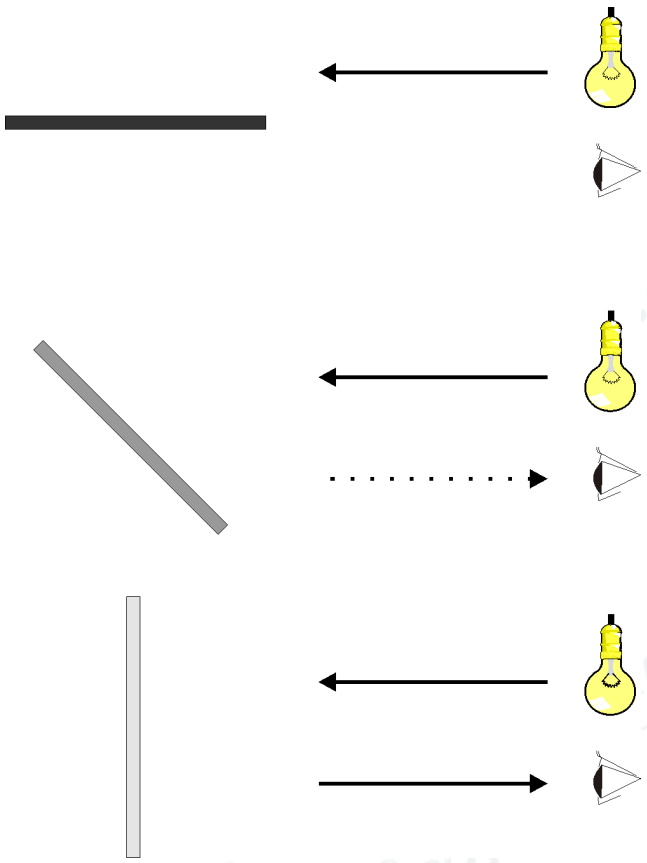


Figura 4.2: Reflexión difusa: cómo afecta la orientación

así se indica, OpenGL puede eliminar de los cálculos todas aquellas caras que no son visibles en relación al observador. Para determinar si una cara es visible o no desde la posición del observador, basta con hacer el producto escalar entre la normal y el vector que se forma entre la posición del observador y una posición de la cara.

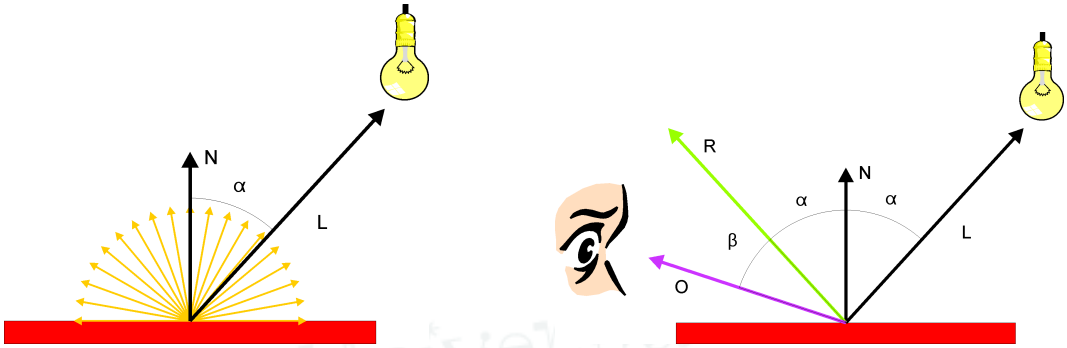
Para el cálculo de la normal de un vértice en un modelos de triángulos se parte de las normales de los triángulos que confluyen en dicho vértice y se calcula el valor medio de las normales (ver figura 4.1(b)). Esto es:

$$\vec{N} = \frac{\sum_{i=1}^{n} \vec{N}_i}{n}$$

Es muy importante que una vez que hayamos calculado las normales las normalicemos.

4.2.2. Iluminación

Una vez que disponemos de las normales vamos a ver cómo se incluyen los distinto elementos para producir la iluminación. Para alcanzar un mayor realismo mediante la



(a) Reflexión difusa (b) Reflexión especular

Figura 4.3: Tipos de reflexión

iluminación vamos a implementar el suavizado plano, FLAT_SHADING, el suavizado de Gouraud, GOURAUD_SHADING.

Para poder simular la iluminación nos hace falta un modelo de reflexión. El que vamos a usar es un modelo sencillo que tiene tres componentes: ambiental, difusa y especular.

Empecemos por la componente más sencilla de entender, y en muchos casos la que mayor importancia tiene, la componente difusa. La idea básica para entender su comportamiento consiste en entender que la orientación del objeto respecto a la fuente de luz modifica la cantidad de luz reflejada: cuanto más perpendicular esté a los rayos de luz más reflejará. La figura 4.2 muestra la idea.

Dado que las superficies con las que estamos trabajando son triángulos, esto implica que cuanto mas perpendicular sea el plano en el que se encuentra inscrito el triángulo mayor será la cantidad de luz reflejada. ¿De qué manera queda definido el plano? ¡Mediante la normal! Si usamos la normal para definir la orientación del triángulo, la reflexión será 0 cuando sea perpendicular al vector que indica la dirección de la luz, y será máxima cuando los dos vectores sean paralelos. O sea cuando los vectores son perpendiculares el valor debe ser 0 y cuando son paralelos debe ser máximo. Podemos ver que el producto escalar tiene esa propiedad. O si tenemos en cuenta el ángulo que se forma entre ambos vectores, cuando el ángulo es 0 el valor debe ser máximo y cuando el ángulo es 90 grados, el valor debe máximo. La función coseno cumple con la propiedad, si entendemos que el valor que devuelve servirá de modulador. Así, $\cos(0) = 1$ y $\cos(90) = 0$. Para calcular el coseno vamos a usar el producto escalar: $\vec{N} \cdot \vec{L} = |\vec{N}| * |\vec{L}| * \cos(\alpha)$. Si tenemos los vectores normalizados, entonces $|\vec{N}| = 1$ y $|\vec{L}| = 1$ y por tanto, la ecuación queda así: $\vec{N} \cdot \vec{L} = \cos(\alpha)$. La figura 4.3(a) muestra el efecto de la reflexión difusa y la disposición de los vectores. Es importante hacer notar que para esta componente la posición del observador no afecta el resultado. Esto es, cuando llega un rayo de luz el objeto refleja en todas las direcciones. La mayor parte de los objetos que vemos suelen tener una componente mayoritariamente difusa.

También es importante tener en cuenta que el producto escalar puede dar valores negativos, en cuyo caso implica que la parte delantera de la cara está orientada hacia atrás. Este método también se puede usar para evitar cálculos de caras que no son visibles, es el

llamado *culling* que puede ser habilitado en OpenGL.

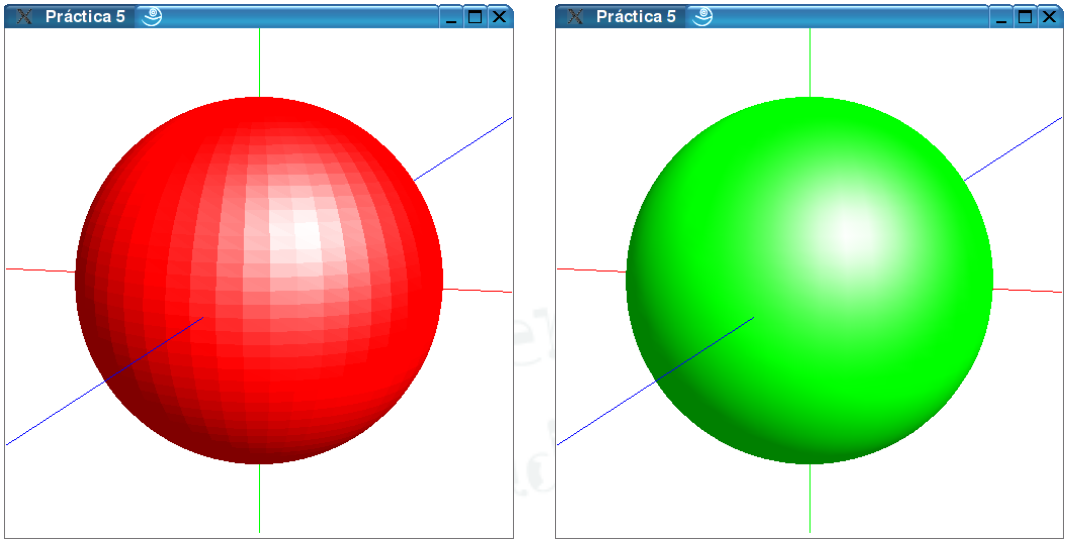
Existen objetos que reflejan la luz de otra manera, que vamos a llamara reflexión especular. Un ejemplo son los espejos, casi un reflector especular ideal. En este tipo de objetos, un rayo de entrada es reflejado en un rayo de salida con el mismo ángulo de entrada que en la salida. La idea se muestra en la figura 4.3(b). En este caso sí que es importante la posición del observador pues dependiendo de la misma es posible que vea el rayo reflejado o no. En un reflecto especular ideal, el ángulo entre el rayo reflejado y la dirección del observador tendría que ser 0. Lo que ocurre normalmente es que los reflectores especulares no son ideales y se produce una cierta dispersión alrededor del rayo reflejado. Para modelar este comportamiento de nuevo recurrimos al coseno, pero esta vez, para el ángulo β que se forma entre $\vec{R}$ y $\vec{0}$: cuando β es 0 se obtiene la máxima reflexión y cuando es 90^o se obtiene 0. Dado que podemos encontrar objetos más o menos especulares, es necesario añadir un modificador de la función coseno: elevar el coseno a una potencia, con valores entre 0 e infinito. En las implementaciones, se usan valores finitos. Por tanto, la reflexión especular nos quedaría de la siguiente manera $\vec{R} \cdot \vec{O} = \cos^n(\beta)$, estando los vectores $\vec{R}$ y $\vec{O}$ normalizados y siendo n el exponente.

La última componente que nos falta es la ambiental. Con esta componente se pretende modelar una especie de flujo de luz constante que viene de todas direcciones. Este flujo no is imaginado sino que tiene un fundamento físico. Imaginemos una habitación pequeña sin ninguna iluminación salvo la luz que entra por un pequeño agujero. Un observador que esté en el interior podrá ver las distintas partes de la misma. El motivo es que la luz se empieza a reflejar en las paredes una y otra vez haciendo visible lo que en principio no estaba iluminado directamente. Por tanto, es una componente que forma a partir de las numerosas reflexiones de los rayos de luz en los distintos objetos. Es fácil ver que conforme los rayos se reflejan irán adquiriendo la tonalidad del material del objeto que produce la reflexión. En un modelo de iluminación global, todos estos reflejos se tienen en cuenta. En el modelo sencillo de iluminación que estamos explicando, un modelo local, esta componente se simplifica mediante el uso de un valor constante de baja intensidad. Además, evita el efecto de que las caras que no están iluminadas directamente se vean de color negro, lo cual resulta poco natural.

Recapitulemos. Nuestro modelo de reflexión de las luz es el siguiente: $I_r = A + D + E$. Esto es, la intensidad de luz reflejada por un objeto es la suma de las componentes ambiental, difusa y especular. Hemos visto que el valor de las reflexiones difusa y especular depende de la orientación del objeto con respecto a la luz en el caso de la reflexión difusa, y del observador con respecto al rayo reflejado en la reflexión especular. Ahora tenemos que incluir la intensidad de la fuente de luz y el material: si no hay luz no podemos iluminar nada, el material del objeto modula la reflexión, por ejemplo un color oscuro refleja menos luz que un color claro.

Si la intensidad de la fuente de luz es I_l, si la capacidad del material para reflejar la componente difusa es K_d, la capacidad del material para reflejar la componente especular es K_e, y la reflexión ambiental se modela con K_a, la fórmula nos queda: $I_r = I_l * K_a + I_l * K_d * cos(\alpha) + I_l * K_e * \cos^n(\beta)$

Dado que tanto la luz como los materiales se definen como colores RGB, la formula anterior se debe extender a las tres componentes.



(a) Sombreado plano (b) Sombreado de Gouraud

Figura 4.4: Tipos de sombreado/suavizado

Un detalle importante a tener en cuenta en la ecuación que hemos explicado es la posibilidad de obtener valores de intensidad reflejada mayor que 1, cuando en OpenGL, la máxima intensidad es 1. Veamos un ejemplo. Si suponemos que el valor de la luz es $I_l = 1$, luz blanca, la componente ambiental es $K_a = 0{,}25$, la componente difusa es $K_d = 0{,}8$, y la componente especular es $K_d = 0{,}5$, un gris medio, podemos ver que como mínimo la intensidad reflejada será $I_r = 1 * 0{,}25 \rightarrow 0{,}25$, pero el valor máximo se dará cuando α y β sean 0. En tal caso el valor será:$I_r = 1 * 0{,}25 + 1 * 0{,}8 * 1 + 1 * 0{,}5 * 1 \rightarrow 1{,}55$. Este valor tiene que ser recortado a 1. Lo que veremos es que muchas partes del objeto se ven con una intensidad máxima. Por tanto, es importante modular correctamente los valores de los materiales, ya que será bastante normal usar fuentes de luz de color blanco $I_l(1,1,1)$.

Por último veamos las dos posibilidades que ofrece OpenGL para aplicar la ilumina-cion: sombreado plano (GL_FLAT) y sombreado de Gouraud (GL_SMOOTH). La diferencia está en que en el sombreado plano sólo se asigna una normal a los tres vértices de cada triángulo, la normal del triángulo y por tanto se obtiene un color uniforme para toda la cara (Figura 4.4(a)). En el sombreado de Gouraud se asigana a cada vértice su normal correspondiente. Por tanto tendremos tres colores en los vértices y para las posiciones intermedias OpenGL aplica una interpolación produciendo el efecto de que parece que la superficie es curva (Figura 4.4(b)).

4.2.3. Texturas

En este ejemplo vamos a hacer uso de una de las funcionalidades que permite obtener mayor realismo: las texturas.

Imaginemos que queremos producir una imagen 2D realista: un buen pintor conseguirá

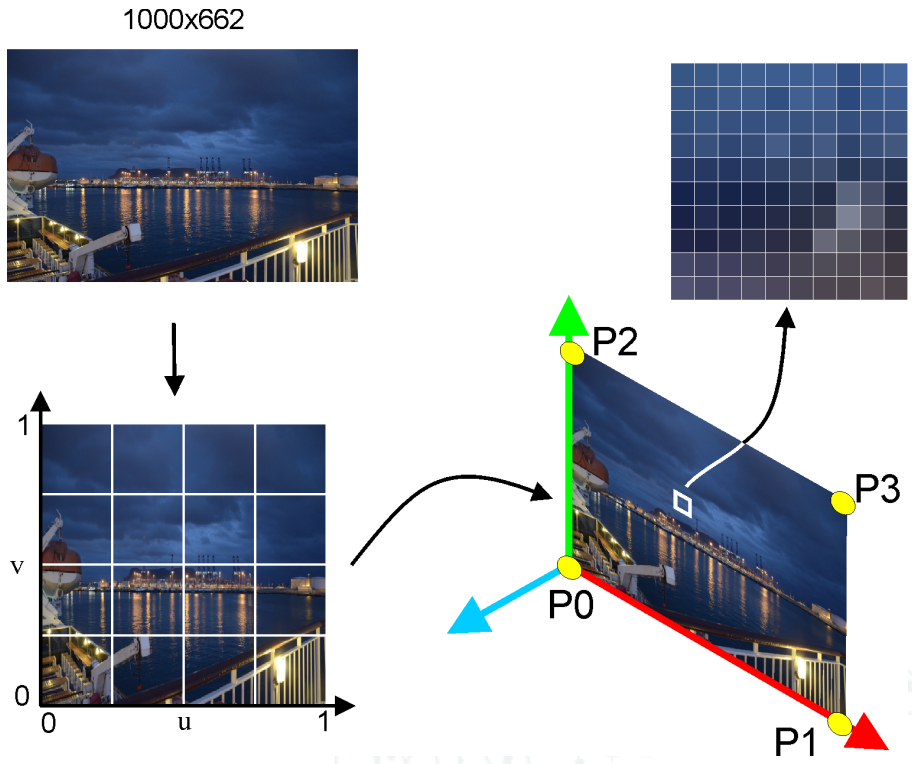


Figura 4.5: Pasos en la aplicación de una textura

un buen resultado con mucho esfuerzo, pero nada será mejor que una fotografía. Esta idea se puede extender a la visualización de modelos realistas: sólo con vértices y colores es muy difícil que consigamos un alto grado de realismo, y si se consigue sólo es posible con un gran número de vértices.

Pongamos un ejemplo. Imaginemos que queremos modelar un tablón de madera. La forma del mismo es muy fácil de modelar con un paralelepípedo, un cubo estirado. Con 8 vértices y 8 colores, poco se puede conseguir para simular la madera. La solución consistiría en incrementar el número de vértices hasta que consiguiéramos el nivel de detalle deseado.

La solución que se propone con las texturas es la siguiente: hacemos una foto de cada lado del tablón y cada foto es pegada a un lado del modelo. La geometría en sencilla pero la visualización es realista. No vamos a entrar en cómo se puede pega la imagen en el modelo, pero ayuda el pensar que la fotografía se ha imprimido en una tela que es deformable, de tal manera que la ajustamos al objeto.

Los pasos para aplicar una textura se muestran en la figura 4.5. El primer paso consiste en pasar una imagen matricial a un sistema de coordenadas normalizado, con coordenadas u y v, cumpliendo que $0 \leq u \leq 1$ y $0 \leq v \leq 1$. Esta normalización permite independizar el tamaño real de la imagen de su aplicación al modelo.

El siguiente paso consiste en asignarle a cada punto del modelo las coodenadas de textura correspondientes. En el ejemplo de la imagen, para representar un rectángulo necesitamos

2 triángulos. Si tenemos 4 puntos, P_0, P_1, P_2 P_3, el triángulo T_0 podría estar compuesto por los puntos $(P_2, P_0, P1)$ y el triángulo T_1 por los puntos $(P_1, P_3, P2)$. Dado que queremos mostrar toda la textura, eso implica la siguiente asignación de coordenadas de textura:

- $(0,0) \rightarrow P_0$
- $(1,0) \rightarrow P_1$
- $(0,1) \rightarrow P_2$
- $(1,1) \rightarrow P_3$

Si solo quisiéramos mostrar la parte central, las coordenadas podrían ser las siguientes:

- $(0{,}25, 0{,}25) \rightarrow P_0$
- $(0{,}75, 0{,}25) \rightarrow P_1$
- $(0{,}25, 0{,}75) \rightarrow P_2$
- $(0{,}75, 0{,}75) \rightarrow P_3$

Es importante observar con estos dos ejemplos que la diferencia en lo que se muestra se ha conseguido simplemente cambiado las coordenada de textura, no las coordenadas de los puntos.

Una vez que se ha hecho la correspondencia, OpenGL se encarga del trabajo duro, realizando la correspondencia entre lo píxeles de la imagen de entrada y los píxeles de la imagen de salida, resolviendo los distintos problemas de escala que hay, realizando la interpolación, ajustando la perspectiva, si la hay, etc.

Un detalle a tener en cuenta es que los formatos de imágenes suelen usar un sistema de coordenadas izquierdo, con el origen en la esquina superior izquierda mientras que OpenGL usa un sistema de coordenadas derecho con el origen en la esquina inferior izquierda. Por ello aplicamos una operación de reflejo horizontal.

4.2.4. Implementación

El alumno entregará un programa que permitar aplicar la ilumación a los objetos que se hayan creado en las anteriores prácticas. Para la visualización de la textura se usará el tablero, al cual también se le podrá aplicar la iluminación. Las tareas serán:

- Iluminación
  - Calcular las normales al crear los objetos.
  - Definir al menos tres materiales (glMaterial).
  - Definir al menos dos luces, una en el infinito (glLight). Se deben poder mover.
- Texturas

- Crear un objeto que sea un cuadrado con mxm divisiones (un tablero)
- Cargar una imagen mediante el programa auxiliar.
- Asignar la imagen al tablero.

Las teclas que controlan la aplicación son las siguientes:

- Tecla p: Visualizar en modo puntos
- Tecla l: Visualizar en modo líneas/aristas
- Tecla f: Visualizar en modo relleno
- Tecla c: Visualizar en modo ajedrez
- Tecla 1: Activar tetraedo
- Tecla 2: Activar cubo
- Tecla 3: Activar objeto PLY cargado
- Tecla 4: Activar objeto por revolución
- Tecla 5: Activar objeto jerárquico
- Tecla 6: Activar tablero
- Tecla Z/z: modifica primer grado de libertad del modelo jerárquico (aumenta/disminuye)
- Tecla X/x: modifica segundo grado de libertad del modelo jerárquico (aumenta/disminuye)
- Tecla C/c: modifica tercer grado de libertad del modelo jerárquico (aumenta/disminuye)
- Tecla i: Activar/desactivar la iluminación
- Tecla t: Activar/desactivar la textura
- Tecla s: Alternar entre sombreado plano (GL_FLAT) y sombreado de Gouraud (GL_SMOOTH).
- Tecla F1: Modificar la posición/orientación de la fuente de luz 1 item Tecla F2: Modificar la posición/orientación de la fuente de luz 2
- Tecla F5: Activar material 1
- Tecla F6: Activar material 2
- Tecla F7: Activar material 3

4.3. Evaluación

La evaluación de la práctica, sobre 10 puntos, se hará del modo siguiente:

- Creación y funcionamiento correcto de las normales (1 punto)
- Creación y funcionamiento correcto de las luces (2 punto)
- Creación y funcionamiento correcto de los materiales luces (2 punto)
- Implementación correcta de los modos de sombreado (1 punto)
- Creación del tablero (1 punto)
- Funcionamiento correcto de la textura en diferentes posiciones (3 puntos)

4.4. Extensiones

Aplicar la textura a los objetos de revolución.

4.5. Duración

La práctica se desarrollará en 3 sesiones.

4.6. Bibliografía

- Mark Segal y Kurt Akeley; *The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.

Práctica 5

Interacción

5.1. Objetivos

El objetivo de esta práctica es:

- Aprender a desarrollar aplicaciones gráficas interactivas, gestionando los eventos de entrada de ratón
- Aprender a realizar operaciones de selección de objetos en la escena.
- Afianzar los conocimientos de los parámetros de la cámara para su correcta ubicación y orientación en la escena.

5.2. Desarrollo

Partiendo de las prácticas anteriores, se añadirá la siguiente funcionalidad:

1. Colocar varias cámaras en la escena
2. Mover la cámara usando el ratón y el teclado en modo *primera persona*.
3. Seleccionar objetos de la escena usando el ratón de forma que la cámara pase a funcionar en modo *examinar*.

Adicionalmente, como funcionalidad extra, se podrá implementar:

1. Una cámara en *tercera persona* que siga un objeto en movimiento (aleatorio o guiado por teclado) en la escena.
2. la función de zoom para una cámara con una proyección ortogonal

5.2.1. Colocar varias cámaras en la escena

Se añadirá al código un vector de cámaras (objetos de una clase `Camara` si se está programando en C++), y se incluirá en la escena un mecanismo de control para saber qué cámara de las existentes está activa. Al menos se habrán de colocar tres cámaras, ofreciendo las vistas clásicas de frente, alzado y perfil de la escena completa. Al menos una de ellas deberá tener proyección ortogonal y al menos otra proyección en perspectiva. Se activarán con las teclas F1, F2 y F3.

5.2.2. Mover la cámara usando el ratón y el teclado en modo *primera persona*

Con las teclas A,S,D y W se moverá la cámara activa por la escena (W: avanzar, S: retroceder, A: desplazamiento a la izquierda, D: desplazamiento a la derecha, R: reiniciar posición), conservando la dirección en la que se está mirando. Para ello será necesario modificar únicamente el punto VRP o *eye*.

Por otro lado, girar la cámara a derecha o izquierda, arriba o abajo (modificar el VPN o el *lookAt* se realizará siguiendo los movimientos del ratón con el botón derecho pulsado.

Para controlar la cámara con el ratón es necesario hacer que los cambios de posición del ratón afecten a la posición de la cámara, y en *glut* eso se hace indicando las funciones que queremos que procesen los eventos de ratón (en el programa principal antes de la llamada a *glutMainLoop*();):

```
glutMouseFunc( clickRaton );
glutMotionFunc( ratonMovido );
```

y declarar estas funciones en el código.

La función *clickRaton* será llamada cuando se actúe sobre algún botón del ratón. La función *ratonMovido* cuando se mueva el ratón manteniendo pulsado algún botón.

El cambio de orientación de la cámara activa se gestionará en cada llamada a *ratonMovido*, que solo recibe la posición del cursor, por tanto debemos comprobar el estado de los botones del ratón cada vez que se llama a *clickRaton*, determinando que se puede comenzar a mover la cámara sólo cuando se ha pulsado el botón derecho. La información de los botones se recibe de *glut* cuando se llama al callback:

```
void clickRaton( int boton, int estado, int x, int y );
```

por tanto bastará con analizar los valores de *boton* y *estado*, y almacenar información que nos permita saber si el botón derecho está pulsado y la posición en la que se encontraba el cursor cuando se pulsó

```
if ( boton == GLUT_RIGHT_BUTTON )
{   if ( estado == GLUT_DOWN ){
// Se pulsa el botón, por lo que se entra en el estado "moviendo cámara"
    }
    else{
// Se levanta el botón, por lo que se sale del estado "moviendo cámara"
    }
```

```
}
```

En la función *ratonMovido* comprobaremos si el botón derecho está pulsado, en cuyo caso actualizaremos la posición de la cámara a partir del desplazamiento del cursor

```
void ratonMovido( int x, int y )
{   ......
    ......
    if ( estadoRaton==MOVIENDO_CAMARA_FIRSTPERSON)
    {
        escena.camaras[camaraActiva].girar(x-xant,y-yant);
                            xant=x;yant;=;
   }
   glutPostRedisplay();
```

En el método `Camara::girar`, cada cámara recalcula el valor de sus parámetros (VPN o `lookAt`) en función del incremento de x e y recibido.

Si hasta ahora en la práctica la transformación de visualización se hacía, por ejemplo, en

```
void change_observer()
{
    // posicion del observador
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0,0,-Observer_distance);
    glRotatef(Observer_angle_x,1,0,0);
    glRotatef(Observer_angle_y,0,1,0);
}
```

ahora habrá que hacer

```
void change_observer()
{
    // posicion del observador
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    escena.camaras[camaraActiva].setObservador();
}
```

de forma que en cada posicionamiento de la cámara se invoque la cámara con los parámetros adecuados.

5.2.3. Seleccionar objetos de la escena usando el ratón de forma que la cámara pase a funcionar en modo *examinar*

Se incluirán en la escena los objetos generados en las prácticas 2, 3 y 4, y cualquier otro objeto que se desee. El usuario, al hacer clic con el botón izquierdo del ratón sobre

uno de los objetos de la escena, centrará el foco de atención sobre el centro de dicho objeto (calculado como el centro de su caja envolvente). A partir de ese momento, la cámara entrará en modo EXAMINAR, y el movimiento del ratón con el botón derecho pulsado permitirá observar el objeto seleccionado desde cualquier ángulo.

Movimiento de la cámara en modo EXAMINAR

En el método `Camara::girar`, cuando se está en modo examinar, la cámara recalcula el valor de sus parámetros (VRP o `eye`) en función del incremento de x e y recibido, de forma que orbite en torno al punto `lookAt`, que es el centro del objeto seleccionado.

En el modo EXAMINAR, las teclas A,S,D y W no tienen efecto sobre la cámara, salvo que se implemente la funcionalidad extra de zoom, que se realizará con las teclas W y S.

Selección

Para seleccionar se debe crear una función de selección (*pick*). Hay dos formas de hacerlo: usando el modo GL_SELECT de OpenGL, o usando una codificación de colores en el buffer trasero.

Selección modo GL_SELECT

Cuando se pulse el botón izquierdo desde la función *clickRaton* se debe llamar a una función que gestione el `pick`:

```
int pick( int x, int y)
```

siendo, x, y la posición del cursor que se va a usar para realizar la selección, y devuelve el ID del objeto que se ha seleccionado (-1 si no se ha seleccionado nada).

Para poder seleccionar los distintos componentes de la escena se deben añadir identificadores (*names*) al dibujarlos.

El procedimiento de selección (*pick*), debe realizar los siguientes pasos:

```
// 1. Declarar buffer de selección
      glSelectBuffer(...)
// 2. Obtener los parámetros del viewport
      glGetIntegerv (GL_VIEWPORT, viewport)
// 3. Pasar OpenGL a modo selección
      glRenderMode (GL_SELECT)
// 4. Fijar la transformación de proyección para la seleccion
      glMatrixMode (GL_PROJECTION);
      glLoadIdentity ();
      gluPickMatrix (x,(viewport[3] - y),5.0, 5.0, viewport);
      MatrizProyeccion(); // SIN REALIZAR LoadIdentity !
// 5. Dibujar la escena con Nombres
      dibujarConNombres();
// 6. Pasar OpenGL a modo render
```

```
        hits = glRenderMode (GL_RENDER);
// 7. Restablecer la transformación de proyección (sin gluPickMatrix)
// 8. Analizar el contenido del buffer de selección
// 9. Devolver el resultado
```

Interpretación del buffer de selección

En el buffer devuelve, para cada primitiva intersecada (la variable `hits` devuelta al hacer `glRenderMode(GL_RENDER)` nos dice el número de primitivas intersecadas con el clic del ratón):

- el número de identificadores o nombres asociados a la primitiva
- el intervalo de profundidades de la primitiva: zmin y zmax.
- los nombres asociados a la primitiva.

Se deberá crear un método en la escena o una función que gestione el buffer de nombres, y devuelva el identificador del objeto más cercano al observador.

Nombres

Para distinguir un objeto de otro, OpenGL utiliza una pila de enteros como identificadores. Dos primitivas se considera que corresponden a objetos distintos cuando el contenido de la pila de nombres es distinto. Para ello proporciona las siguientes funciones:

```
glLoadName(i) // Sustituye el nombre activo por i
glPushName(i) // Apila el nombre i
glPopName() // Desapila un nombre
glInitNames() // Vacia la pila de nombres
```

El método o función `dibujarConNombres` difiere del dibujar normal en que hace uso de la pila de nombres, por lo que es el que se invoca en el modo GL_SELECT. Antes de ponerte a codificar, debes decidir cómo colocar los identificadores y añadirlos a los objetos, teniendo en cuenta lo que necesitas seleccionar.

Procura gestionar bien la pila de nombres, asegurándote de que está vacia al comienzo del ciclo de dibujo y que los `glPushName` y `glPopName` están balanceados

Selección por codificación de colores

Hay un mecanismo más simple aún para determinar qué primitiva ha sido seleccionada. Se trata de usar un código de color para cada objeto seleccionable. Se trata de crear una función de dibujado distinta para cuando queremos seleccionar, y cuando el usuario hace clic, se pinta la escena “para seleccionar” en el buffer trasero y se lee el color del pixel donde el usuario ha hecho clic. Si no se hace un intercambio de buffers, el usuario jamás verá esa escena “rara”, y el programa seguirá su proceso natural.

En resumen, los pasos a seguir son:

- Llamar a la función o método dibujaSeleccion()
- Leer el pixel (x,y) dado por la función gestora del evento de ratón
- Averiguar a qué objeto hemos asignado el color de dicho pixel
- **No intercambiar buffers**

Un código de ejemplo, que dibuja cuatro patos cada uno de un color sería:

```
void Escena::dibujaSeleccion() {

glDisable(GL_DITHER); // deshabilita el degradado
for(int i = 0; i < 2; i++)
   for(int j = 0; j < 2; j++) {
         glPushMatrix();
         switch (i*2+j) { // Un color para cada pato
                  case 0: glColor3ub(255,0,0);break;
                  case 1: glColor3ub(0,255,0);break;
                  case 2: glColor3ub(0,0,255);break;
                  case 3: glColor3ub(250,0,250);break;
         }
         glTranslatef(i*3.0,0,-j * 3.0);
         pato.dibuja();
         glPopMatrix();
   }
glEnable(GL_DITHER);
}
```

Para comprobar el color del pixel, usaremos la función glReadPixels:

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format, GLenum type, GLvoid *pixels);
```

donde

- x,y : la esquina inferior izquierda del cuadrado a leer (en nuestro caso el x,y, del pick)
- width,height: ancho y alto del área a leer (1,1 en nuestro caso)
- format: Tipo de dato a leer (coincide con el format del buffer, GL_RGB o GL_RGBA).
- type: tipo de dato almacenado en cada pixel, según hayamos definido el glColor (p.ej. GL_UNSIGNED_BYTE de 0 a 255, o GL_FLOAT de 0.0 a 1.0)
- pixels: El array donde guardaremos los pixels que leamos. Es el resultado de la función.

En el caso del procesamiento del pick, una vez dibujado el buffer, llamaríamos a un método o función que, en función del color del pixel nos miraría en la tabla de asignación de colores a objetos qué objeto estaríamos seleccionando.

Para que esto funcione, hay varias cosas a tener en cuenta:

- Los colores se han de definir con glColor3ub, es decir, como enteros de 0 a 255
- Es posible que el monitor no esté en modo trueColor y no devuelva exactamente el valor que pusimos (hay que tenerlo en cuenta si no funciona bien).
- Hay que desactivar el GL_DITHER, GL_LIGHTING, GL_TEXTURE

5.3. Evaluación

La evaluación de la práctica, sobre 10 puntos, se hará del modo siguiente:

- Posicionar varias cámaras (2 punto)
- Gestión con ratón y teclado de cámara en primera persona (3 puntos)
- Selección de objeto (2 puntos)
- Gestión con ratón de cámara en modo examinar (3 puntos)

5.4. Extensiones

Se podrá realizar una animación de un objeto (p.ej. un animal que se mueva por el suelo) y se le pondrá una cámara que le realice un seguimiento desde atrás, lo que viene a ser una cámara en *tercera persona*. El movimiento del objeto podrá ser automático o controlado por teclado. En este caso, tanto la posición `eye` como `lookAt` vienen determinadas, de forma directa o indirecta, por el objeto que se persigue.

Como debe haber al menos una cámara en modo ortogonal, se podrá implementar la función zoom para dicha cámara, pues sabemos que la imagen resultante es independiente de la distancia del observador al plano de proyección. Para ello habrá que modificar los parámetros del frustum.

5.5. Duración

La práctica se realizará durante 3 sesiones.

5.6. Bibliografía

- Mark Segal y Kurt Akeley;*The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000

- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- M. E. Mortenson; *Geometric Modeling*; John Wiley & Sons,1985
- http://www.lighthouse3d.com/opengl/picking/index.php