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

Prácticas de Informática Gráfica

Autores:
Pedro Cano
Antonio López
Domingo Martín
Javier Melero
Juan Carlos Torres
Carlos Ureña

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 estudiante aprenda:

- A utilizar las primitivas de dibujo de OpenGL para dibujar objetos
- A distinguir entre lo que es crear un modelo y a lo que es visualizarlo.
- A crear y utilizar clases que permitan representar objetos 3D sencillos.

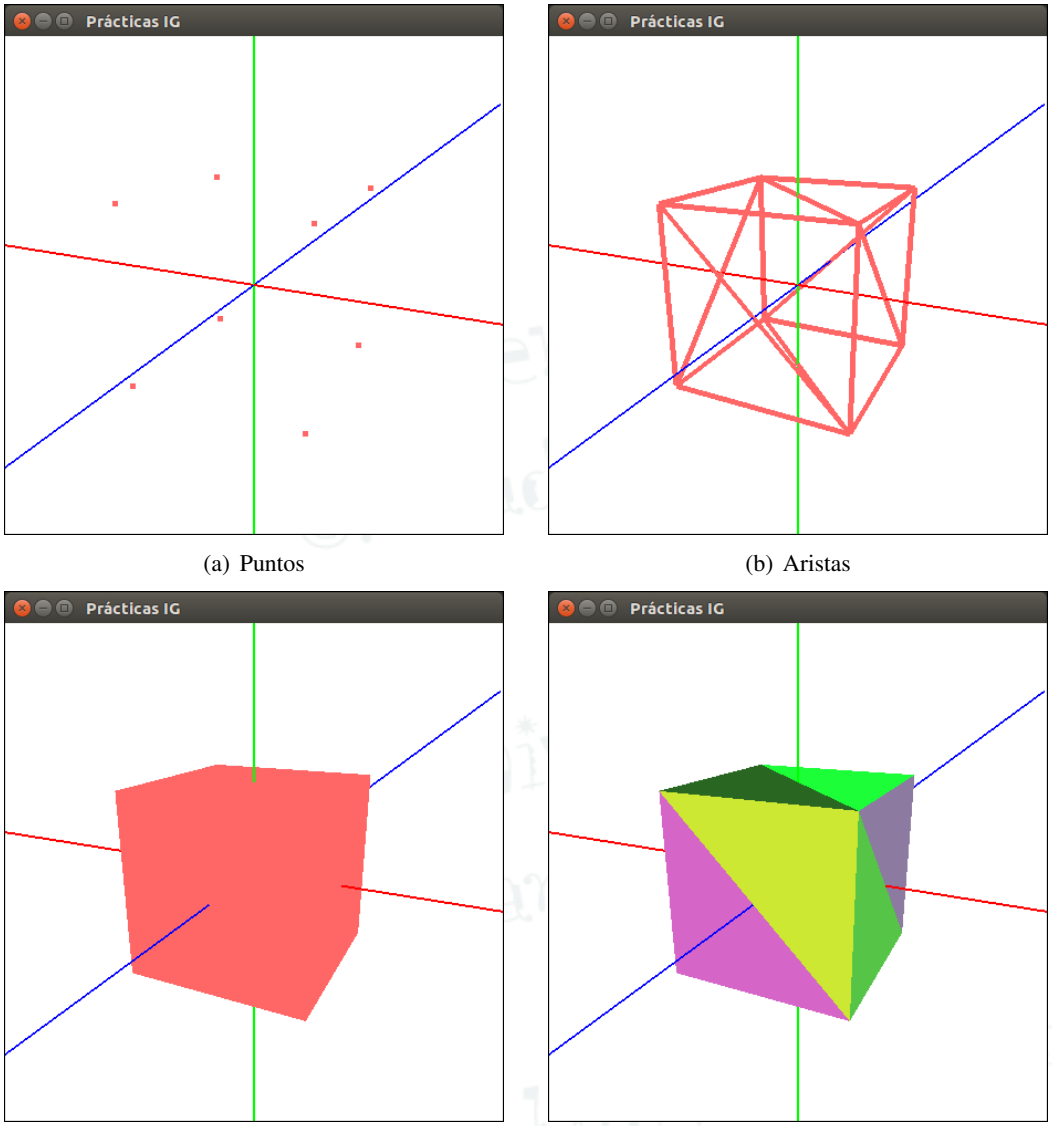
1.2. Desarrollo

Para el desarrollo de esta práctica se entrega el esqueleto de una aplicación gráfica basada en eventos, mediante glut, y con la parte gráfica realizada con OpenGL. La aplicación contiene el código de inicialización de OpenGL, la captura eventos y se ha implementado una estructura de clases que permite representar objetos 3D, incluyendo unos ejes cartesianos y una pirámide y dos modos de dibujado, punto y alambre. También está implementada una cámara que rota con respecto al centro de coordenadas con las teclas de cursor y con las teclas página adelante y página atrás para acercarse y alejarse respectivamente.

El alumno deberá estudiar el código entregado. Hecho esto, añadirá funciones que permitan dibujar en modo sólido con un único color para todas las caras y en modo sólido con un color diferente para cada cara. Además, siguiendo la jerarquía ya definida, deberá crear una clase para un cubo o bien para otro tipo de objeto que sea sencillo de definir manualmente.

Por tanto, al finalizar la práctica se dispondrá de los siguientes modos de dibujado:

- Punto
- Alambre
- Sólido
- Sólido Colores



(a) Puntos (b) Aristas

(c) Sólido (d) Sólido colores

Figura 1.1: Cubo mostrado con los distintos modos de visualización.

Para poder visualizar en modo sólido se usa como primitiva de dibujo los triángulos, `GL_TRIANGLES`, y se cambia la forma en que se visualizan los mismos mediante la instrucción `glPolygonMode`, para rellenar su interior.

Para el modo sólido con un color diferente para cada cara, se añade a la clase triángulo un vector de colores y en el constructor de cada figura se asignan éstos.

1.3. Evaluación

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

- Creación de la clase cubo (4 puntos).
- Creación del código que permita dibujar en modo relleno con un sólo color (2 puntos).
- Creación del código que permita dibujar en modo relleno con un color diferente en cada cara (4 puntos).

1.4. Duración

La práctica se desarrollará en una sesión.

1.5. 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. Theory and Practice, 2 Edition*; Addison-Wesley,1996

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 [1].
- Modelar objetos sólidos poligonales mediante algoritmos sencillos como modelado por revolución de un perfil alrededor de un eje de rotación o por extrusión de un polígono.

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 números flotantes con las coordenadas de los vértices y un vector de enteros con los índices de los vértices que forman cada cara. Se creará una clase para estos modelos PLY.

En segundo lugar, dado un código que permite generar objetos por revolució, habrá que modificarlo para incluir las tapas de los objetos y casos particulares (cilindro, cono y esfera). Observación: hay que evitar tener vértices repetidos que dan lugar a múltiples

[1] Existen numerosos formatos para modelos gráficos como STL (Stereolithography), OBJ (Object), DAE (Digital Asset Exchange), IGES (Initial Graphics Exchange Specification), etc.

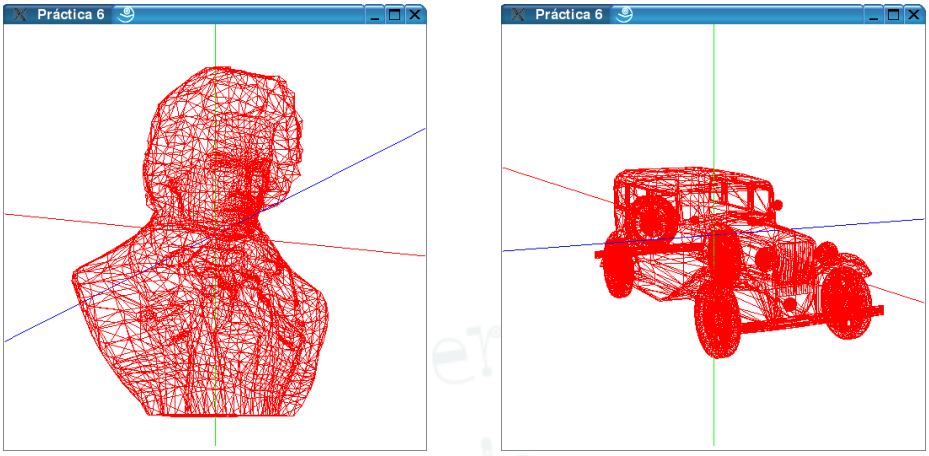
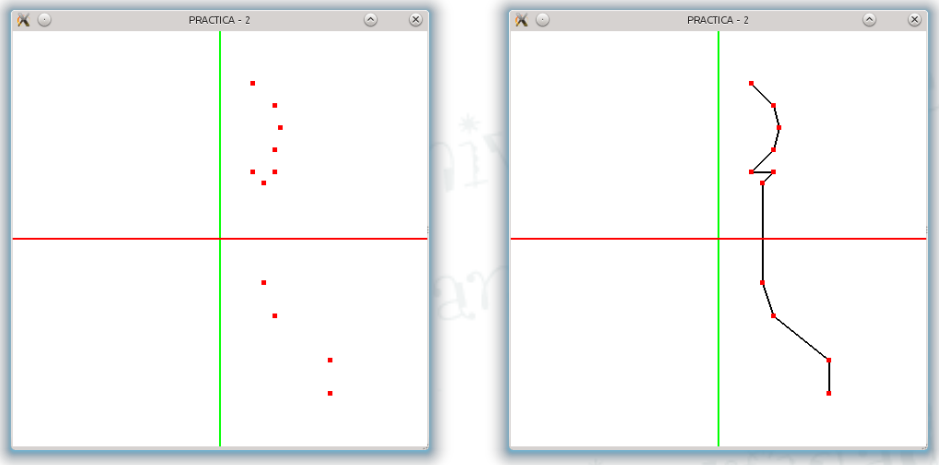


Figura 2.1: Objetos PLY.



(a) Puntos (b) Polilínea

Figura 2.2: Ejemplo de perfil inicial.

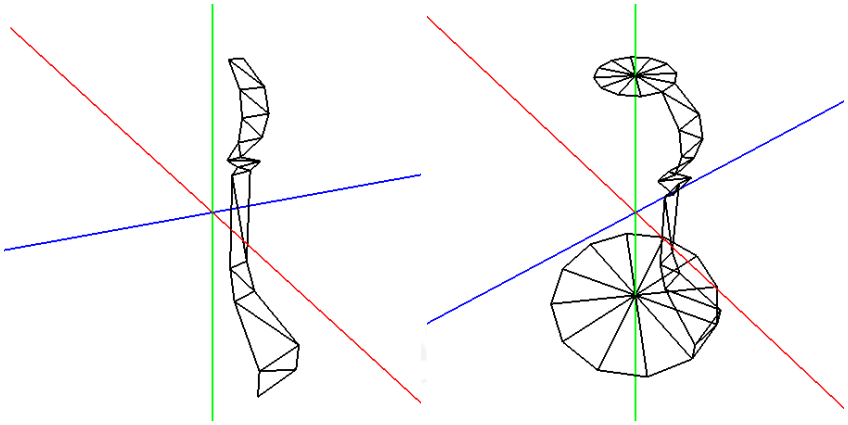
triángulos solapados.

Los pasos para crear un objeto o modelo por revolución serían:

- Sea un perfil inicial Q_1 en el plano $Z = 0$ (para perfiles en los planos $Y = 0$ o $Y = 0$ los pasos son semejantes) definido como:

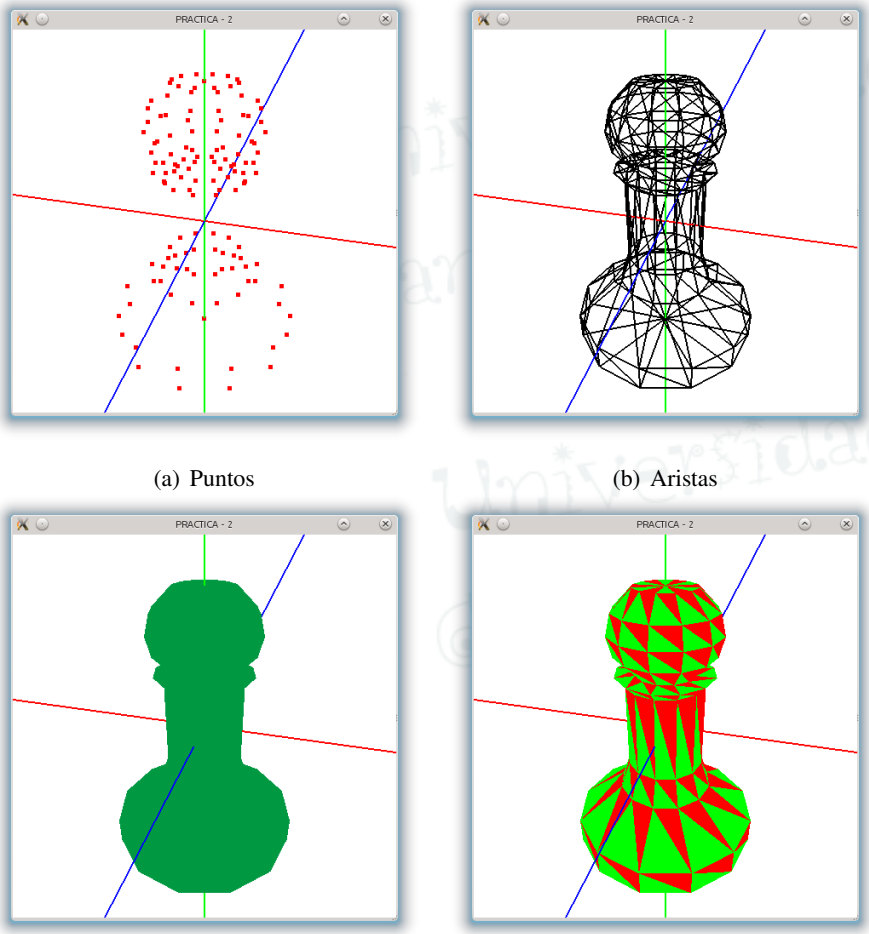
$$Q_1 \left(p_1(x_1, y_1, 0), ..., p_M(x_M, y_M, 0) \right),$$

  siendo $p_i(x_i, y_i, 0)$ con $i = 1, ...M$ los puntos que definen el perfil (ver figura 2.2, donde se muestra un ejemplo de perfil y se han trazado líneas rectas entre los puntos para que se distinga mejor).



(a) (b)

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



(a) Puntos (b) Aristas

(c) Sólido (d) Sólido con dos colores

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

- Se toma como eje de rotación el eje Y y si N es el 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 $R_Y(\alpha_j)$ 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 objeto (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 se pueden crear las tapas del modelo, tanto la inferior como la 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 respectivamente.

- 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 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 añdiendo dos tapas, una o ninguna 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
```

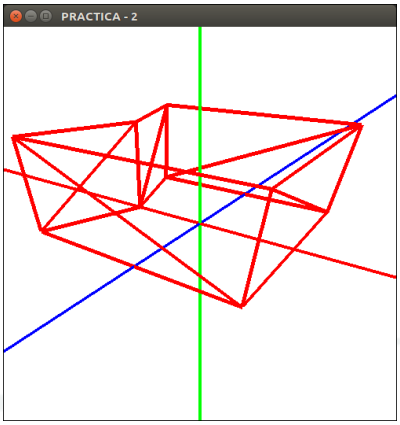


Figura 2.5: Extrusión básica

```
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
```

En el código proporcionado se ha incluido una clase para generar modelos por extrusión básica a partir de un polígono definido en el plano $Y = 0$ que se desplaza sumando a sus vértices un vector de translación (ver figura 2.5).

Se han usado las siguientes teclas:

- Tecla 1: visualizar en modo puntos
- Tecla 2: visualizar en modo aristas
- Tecla 3: visualizar en modo sólido
- Tecla 4: visualizar en modo sólido colores
- Tecla P: seleccionar pirámide
- Tecla C: seleccionar cubo
- Tecla O: seleccionar modelo PLY
- Tecla R: seleccionar modelo por revolución
- Tecla X: seleccionar modelo por extrusión

Añadir teclas para cono, cilindro y esfera.

2.3. Evaluación

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

- Leído un archivo PLY, introducir sus elementos dentro de las estructuras de datos ya definidas en la práctica anterior (1.5 puntos).
- Creación de código para las tapas de los objetos por revolución (3 puntos).
- Creación de clases para obtener un cono, un cilindro y una esfera (4 puntos).
- Generar un objeto por revolución a partir de un perfil definido en un fichero PLY (1.5 puntos).

2.4. Duración

La práctica se desarrollará en 2 sesiones.

2.5. 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 estudiante aprenderá a:

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

3.2. Desarrollo

Para realizar un modelo jerárquico artículado 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 presenten las articulaciones del modelo.

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

Las acciones a realizar en esta práctica son:

1. Diseñar el grafo de un modelo jerárquico articulado, determinando el tamaño de las piezas y las transformaciones geométricas a aplicar (se tendrá que entregar el grafo del modelo en papel o en formato electrónico: pdf, jpeg, etc.). Ha de tener al menos cuatro articulaciones. En el código proporcionado se ha realizado una excavadora como ejemplo, con articulaciones en la pala, brazos y cabina (ver figura 3.1).

2. Crear las clases necesarias para implementar el modelo jerárquico. Cada clase permitirá definir las piezas del objeto, incluyendo las transformaciones para modelarlas (como escalados para las medidas de las piezas, y rotaciones y traslaciones para su posicionamiento) así como las transformaciones y variables necesarias para su articulación, en su caso, que se podrán modificar en tiempo de ejecució. La relación jeráquica, en el ejemplo entregado, se implementa mediante la inclusión de las instancias de los objetos hijos en los objetos padres (existen otras posibilidades). Dos consejos para realizar la práctica:

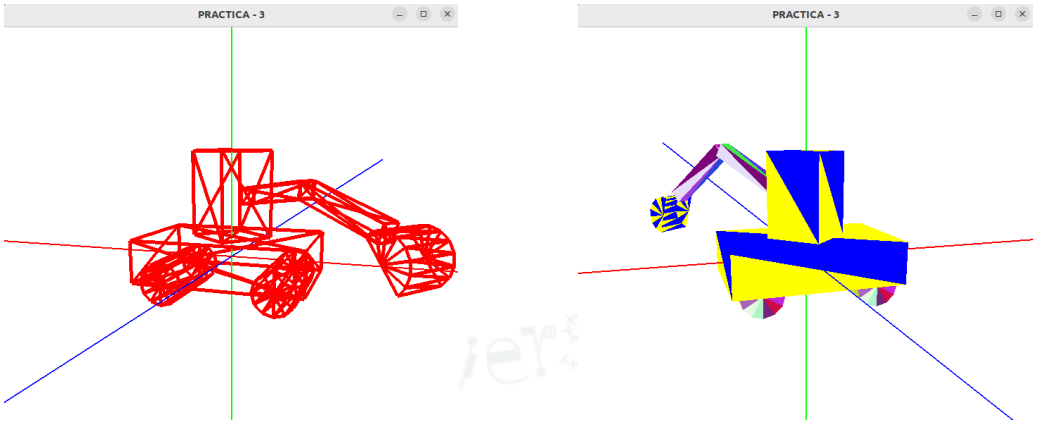


Figura 3.1: Ejemplo de modelo de excavadora realizado para la práctica 3.

a) Crear primero la versión estática del modelo, esto es, aquella que lo muestra en un estado inicial pero con las dimensiones y el posicionamiento correctos.

b) Modificar el modelo para incluir las transformaciones que permiten su movimiento. Esto implica que se pod/'an modificar los parámetros que las controlan cuando se pulsen las teclas asociadas.

Para construir los modelos jerárquicos se han de utilizar elementos más sencillos que al combinarse mediante instanciación utilizando las transformaciones geométricas pertinentes nos permitirán construir modelos más complejos. Se pueden usar los elementos creados en las prácticas anteriores y nuevos si se consideran necesarios para la actual.

3.2.1. Animación

Una vez se han implementado las transformaciones que aplican el movimiento, se puede modificar pulsando las teclas correspondientes. Pero también se puede conseguir que el movimiento sea automático, para ello hay que hacer que los parámetros cambien con el tiempo sin nuestra intervención. También se podría ajustar la velocidad a la que se producen los cambios.

Para automatizar el cambio de los parámetros, vamos a hacer uso del evento *idle* de GLUT (para Qt hay que usar un QTimer con un tiempo 0). Este evento, si se activa, hace que se llame a la función que se haya indicado cuando en gestor de eventos está desocupado, *idle*. En general, este suele ser el estado en el que se encuentra el gestor pues los eventos, si están bien implementados, se deben resolver rápidamente.

Este evento se activa desde el programa principal con:

```
glutIdleFunc(function_idle );
```

siendo $function_idle()$ el nombre de la función que se ha de encargar de actualizar los valores de los parámetros ue controlan las transformaciones asociadas al movimiento. Por ejemplo, el ángulo para un rotación.

Para una animación, básicamente, haremos cambios de estado, $P' = f(P)$. Esto es, calculamos un nuevo valor para el parámetro P usando el valor previo.

Por ejemplo, podemos calcular el siguiente valor del ángulo de una rotación como: $\alpha' = \alpha + \delta$, siendo δ el incremento que se aplica en cada paso. Según el valor de δ los cambios de estado serán más o menos suaves.

Hay recordar que una vez se han actualizado las variables hay que indicar que se redibuje la escena llamando a $glutPostRedisplay()$.

3.2.2. Resultados entregables

El alumno entregará un programa que represente y dibuje un modelo jerárquico con al menos cuatro articulaciones, cuyos parámetros se podrán modificar por teclado. Se pueden incorporar las siguientes teclas a las ya usadas en las prácticas anteriores:

- Tecla a: Activar objeto jerárquico
- Tecla s: Activar/desactivar la animación automática
- Teclas de función F1 ...F12: modificar parámetros articulaciones.

3.3. Evaluación

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

- Diseño del grafo correspondiente al modelo jerárquico que muestre las relaciones entre los elementos (1.5 puntos).
- Creación de las clases necesarias para modelar y visualizar un modelo jerárquico con al menos cuatro articulaciones (7 puntos). A considerar en la evaluación aspectos como detalle del modelo, creación de piezas específicas y originalidad.
- Control interactivo de los valores que definen los movimientos en las articulaciones incluyendo restricciones en los mismos (0.5 puntos).
- Animación automatica (1 punto). Se recomienda que sea coherente con el objeto jerárquico diseñado.

3.4. Duración

La práctica se desarrollará en 3 sesiones.

3.5. 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. Theory and Practice, 2 Edition*; Addison-Wesley,1996.
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.

3.6. Prácticas de estudiantes

En la figura 3.2 podéis ver modelos jerárquicos creados por estudiantes de cursos pasados.

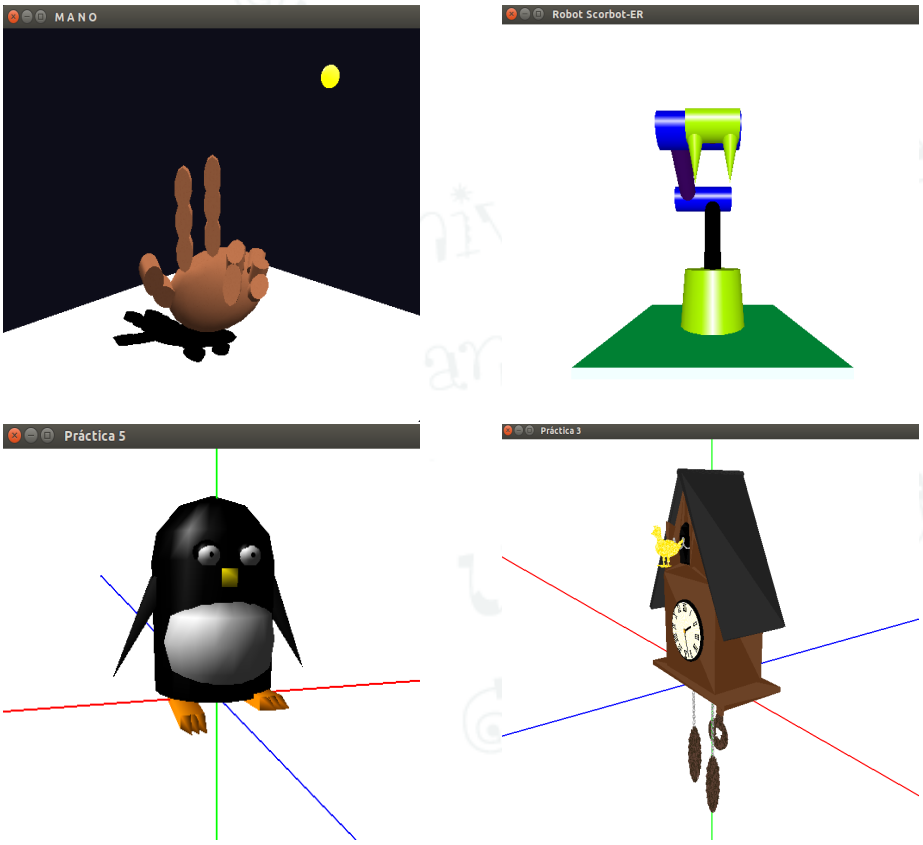


Figura 3.2: Ejemplos de modelos jerárquicos.

Práctica 4

Iluminación y texturas

4.1. Objetivos

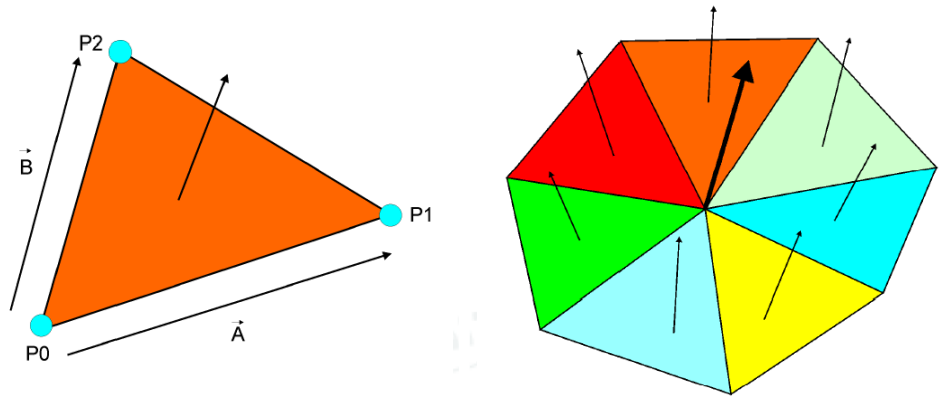
Los objetivos de esta práctica son conseguir un mayor realismo mediante el uso de iluminación local y texturas. Para ello necesitamos hacer lo siguiente:

- Iluminación
  - Para los modelos poligonales (mallas de triángulos) añadir las normales a las caras y a los vértices y definir materiales.
  - Definir fuentes de luz.
  - Crear dos nuevos métodos de visualización (iluminación plano y con suavizado de Gouraud).
- Texturas
  - Determinar las coordenadas de textura para un objeto sencillo
  - Cargar una textura y visualizarla sin iluminación.

4.2. Desarrollo

El objetivo de esta práctica es conseguir un mayor realismo en la visualización de objetos poligonales y para ello vamos a integrar iluminación local, suavizado y uso de texturas.

La iluminación local comprende tres componentes: ambiental, difusa (ley de Lambert) y especular o brillo (ley de Phong). Es necesario incorporar a la estructura de datos usada para las mallas de triángulos las normales a las caras en la llamada iluminación plana y las normales a los vértices si queremos iluminación con suavizado. Además, se añade información sobre coeficientes ambientales, difusos y especulares que simulan el material con el que estarían formados los objetos (metal, plástico, madera, tela, etc.) y se definen fuentes de luz. Todo lo anterior, determinará el coloreado de los objetos.



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

Figura 4.1: Cálculo de normales.

Para mejorar el realismo recurrimos a las texturas que consisten, de una manera muy general, en pegar una imagen a un objeto o parte del mismo. A tal efecto, tendremos que ver cómo se relacionan imagen y objeto mediante las coordenadas de textura.

4.2.1. Cálculo de normales.

A partir de la tabla de coordenadas de vértices y la tabla de caras de una malla de triángulos, se deben calcular las dos tablas de normales (de caras y normales de vértices) que quedarán almacenadas en la estructura o instancia de clase que representa la malla, junto con el resto de tablas (coordenadas de vértices, índices de caras, etc.).

En primer lugar, se calcula tabla de normales de las caras triángulares. Para ello se debe recorrer la tabla caras de un objeto para acceder a los tres vértices que las conforman. Sean, por ejemplo, tres vértices (puntos) P_0, P_1 y P_3; a partir de sus coordenadas se calculan los vectores $\vec{A} = P_1 - P_0$ y $\vec{B} = P_2 - P_0$ para obtener un vector perpendicular a la cara (normal) con su producto vectorial $\vec{M}_c = \vec{A} \times \vec{B}$ (ver figura 4.1(a)). El vector $\vec{M}_c$ se ha de normalizar, obteniendo la normal normalizada a una cara $\vec{N}_c = \vec{M}_c / \|\vec{M}_c\|$.

Las normales a los vértices se obtienen mediante una aproximación usando las normales a las caras, aunque hay objetos en los que se pueden calcular de forma exacta como el cubo y la esfera. Para cada vértice, el vector $\vec{N}_v$, es un vector normalizado que se aproxima mediante la suma de los vectores normales normalizados de todas las caras que tienen en común a dicho vértice dividida por la norma de la mencionada suma, es decir (ver figura 4.1b)):

$$\vec{N}_v = \frac{\sum_{i=0}^{k-1} \vec{N}_{c,i}}{\| \sum_{i=0}^{k-1} \vec{N}_{c,i} \|}$$

donde $\vec{N}_{c,i}$ es el vector perpendicular normalizado a la i-ésima cara adyacente al vértice (suponemos que hay k de ellas).

En general, se entiende que los vectores normales normalizados $\vec{N}_c$ y $\vec{N}_v$ apuntan hacia el lado exterior de las caras, aunque hay situaciones que interesa que estos vectores apunten

al lado interior, como cuando se sitúa una luz puntual dentro de un objeto y queremos visualizar su parte interior.

4.2.2. Iluminación.

Para poder simular la iluminación se utiliza un modelo de iluminación local que tiene tres componentes:

$$I_r = A + D + S, \tag{4.1}$$

siendo A la componente ambiental, D la difusa y S especular. Se calcula la intensidad reflejada I_r (color que se percibiría de un objeto) bien en cada cara de la malla de triángulos (iluminación plana) o bien en los tres v/'ertices de cara triángulo para interpolar en los píxeles interiores. Pasamos a detallar la expresión anterior.

$$I_{r,\lambda} = A_\lambda + D_\lambda + S_\lambda = I_{a,\lambda}K_{a,\lambda} + I_{l,\lambda}K_{d,\lambda}\vec{N}\,\vec{L} + I_{l,\lambda}K_s(\vec{R}\,\vec{V})^n \tag{4.2}$$

- La ecuación anterior ha de calcularse tres veces, una para cada color del modelo RGB, por ello se ha indicado que depende de la longitud de onda λ tanto la luz como los materiales que se indiquen para los objetos.

- $I_{l,\lambda}$ es la intensidad de la fuente de luz. Las características materiales de los objetos se expresan con los coeficientes ambiental $K_{a,\lambda}$, difuso $K_{d,\lambda}$ y especular K_s. Lo normal es hacer que el coeficiente ambiental sea igual al difuso ya que éste determina el color intrínseco de los objetos, $K_{a,\lambda} = K_{d,\lambda}$. El coeficiente especular no depende de la longitud de onda, sin embargo en OpenGL se pueden dar valores diferentes a cada parámetro RGB.

- La componente ambiental $I_{a,\lambda}K_{a,\lambda}$ aproxima toda la iluminación indirecta, la luz procedente de múltiples reflexiones en los objetos de la escena que produce una iluminación de fondo. Se asigna un valor constante para la componente ambiental a cada punto de una cara y además evita el efecto de que las caras que no estén iluminadas directamente por una fuente de luz se vean de color negro. La intensidad ambiental $I_{a,\lambda}$ ha de tomar valores pequeños.

- La componente difusa modela la reflexión de la luz en objetos con superficies rugosas que dan lugar a que la luz entrante se refleje en múltiples direcciones de salida. Se obtiene como $I_{l,\lambda}K_{d,\lambda}\vec{N}\,\vec{L} = I_{l,\lambda}cos(\alpha)$, dependiendo del coseno del ángulo α que forma un vector normal en el punto de la superficie de un objeto, *vecN*, y un vector hacia la luz que sale del mencionado punto, *vecL*. El $cos(\alpha)$ se puede calcular como el producto escalar $\vec{N}\,\vec{L}$ si estos dos vectores están normalizados.

  La orientación del objeto respecto a la fuente de luz modifica la cantidad de luz reflejada difusamente: cuanto más perpendicular esté a los rayos de luz más reflejará. La reflexión será 0 cuando $\vec{N}$ sea perpendicular a $\vec{L}$, o bien cuando $\alpha = 90°$ vector y será máxima cuando los dos vectores sean paralelos, o sea cuando $\alpha = 0°$. Es importante tener en cuenta que el producto escalar puede dar valores negativos, lo que implica que la parte delantera de la cara está orientada en dirección contraria a la fuente de luz y por ello se le asignará como color sólo lo obtenido con la componente ambiental.

La componente difusa no está afectada por la posición del observador y a señalar que la mayor parte de los objetos que vemos son difusos lo que les proporciona un aspecto mate.

- La componente especular modela el brillo, siendo éste el reflejo de una fuente de luz en objetos con superficies pulidas donde cada rayo de entrada se refleja en una sola dirección de salida. Se obtiene como $I_{l,\lambda}K_s(\vec{R}\,\vec{V})^n = I_{l,\lambda}K_s(cose(\beta))^n$, dependiendo del coseno del ángulo β que forma un vector que representa la reflexión de la luz en un punto de la superficie *vecR*, y un vector hacia el observador (posición de la cámara) desde el mencionado punto, *vecV*. El $cos(\beta)$ se puede calcular como el producto escalar $\vec{R}\,\vec{V}$ si estos dos vectores están normalizados. El exponente n, llamado coeficiente de Phong, sirve para modelar el tamaño del brillo, a mayor n se expande menos.

  En objetos con componente especular el rayo de entrada en un punto de una superficie y el reflejado tienen el mismo ángulo con respecto a la normal al punto. En este caso sí que es importante la posición del observador pues dependiendo de la misma es posible que éste vea el rayo reflejado o no. Cuando β es 0 se obtiene la máxima reflexión especular o brillo (es como si al observador "viese directamente.ª través de la reflexión a la fuente de luz) y cuando $|\beta| \geq 90$ el brillo es cero.

  El color del brillo es el de la fuente de luz ya que el brillo es su reflejo, por ello el coeficiente especular K_s es un escalar y no un vector RGB. Objetos especulares típicos son los metales pulidos y cuando un objeto básicamente difuso tiene algo de brillo con reflejos extensos (n pequeño) se le suele denominar satinado.

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 $I_{l,\lambda} = (1,1,1)$, luz blanca (los tres componentes RGB iguales a 1), el coeficiente ambiental es $K_{a,\lambda} = (0.25, 0.25, 0.25)$ el difuso es $K_{d,\lambda} = (0.8, 0.8, 0.8)$ y el especular es $K_s = 0.5$, un gris claro. Podemos ver que como mínimo la intensidad reflejada será:

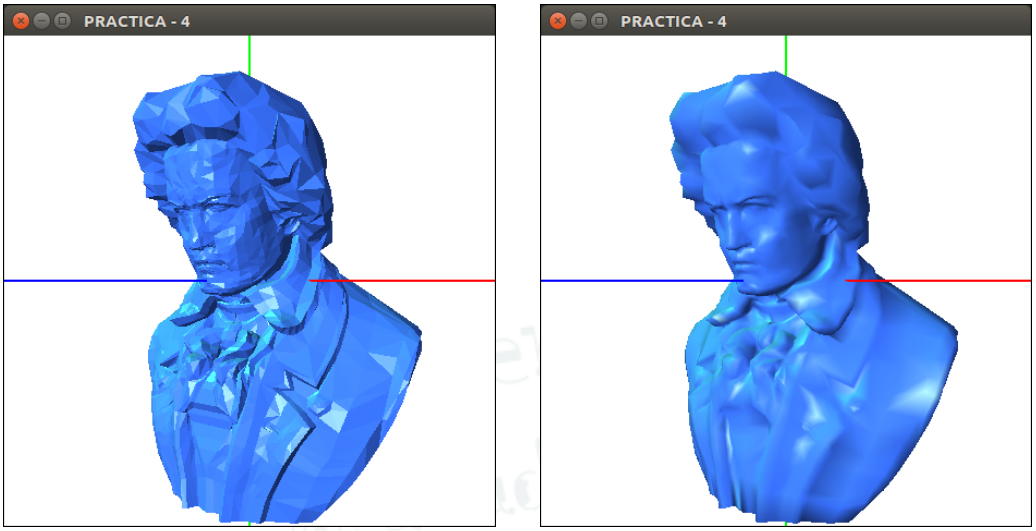
$$I_{r,\lambda} = (1,1,1) \cdot (0.25, 0.25, 0.25) = (0.25, 0.25, 0.25). \tag{4.3}$$

El valor máximo se dará cuando α y β sean 0. En tal caso, se tiene:

$$\begin{aligned} I_{r,\lambda} \;=\; & (1,1,1) \cdot (0.25, 0.25, 0.25) + \\ & (1,1,1) \cdot (0.8, 0.8, 0.8) \times \cos(0) + (1,1,1) \times 0.5 \times \cos(0) = \\ & (1.55, 1.55, 1.55), \end{aligned} \tag{4.4}$$

estos valores $(1.55, 1.55, 1.55)$ se recortan a 1 y veremos que muchas partes del objeto tendrán una intensidad máxima (imagen saturada). Por tanto, es importante modular correctamente los valores de los materiales, ya que será bastante frecuente usar fuentes de luz de color blanco.

OpenGL ofrece dos opciones para aplicar la iluminación: plana (*GL_FLAT*) y con suavizado de Gouraud (*GL_SMOOTH*). La diferencia es que para la iluminación plana se usa



(a) Plana (b) Con suavizado de Gouraud

Figura 4.2: Tipos de iluminación en OpenGL.

la normal a cada triángulo y por tanto se obtiene un color uniforme para toda la cara (figura 4.2(a)); mientras que en el suavizado de Gouraud se usan las normales a los vértices de la cara y por tanto, tendremos tres colores (uno en cada vértice) y para las posiciones intermedias OpenGL aplica una interpolación produciendo un efecto que hace parecer a la superficie de un objeto como curva (figura 4.2(b)).

Para poder iluminar la escena lógicamente hace falta que definamos al menos una fuente de luz. En OpenGL Se define con la función *glLight* que permite controlar aspectos como la posición, el color, si la luz es de tipo focal, etc. Para definir la posición y el color se emplea de la siguiente forma: $glLightXY(LUZ, PARAMETRO, VALOR)$. El argumento *LUZ* es el nombre de la fuente de luz que queremos cambiar, disponiendo en OpenGL de ocho que se denominan con las constantes GL_LIGHT0 hasta GL_LIGHT7S. Como vamos a cambiar la posición usamos como *PARAMETRO* el valor *GL_POSITION* en cuyo caso en *VALOR* ponemos un vector con las coordenadas de la posición de la luz (luz puntual) o su dirección (luz direccional o situada en el infinito). En *VALOR* hay que mandar 4 coordenadas y la última llamada w se interpreta como luz puntual si $w = 1$ y luz direccional si $w = 0$.

Un detalle muy importante a tener en cuenta cuando se indica la posición de la fuente de luz es que se pueden aplicar transformaciones que se hayan definida en la matriz *GL_MODELVIEW*. Esto permite conseguir que las luces puedan cambiar de posición mediante traslaciones y rotaciones.

Para la luz GL_LIGHT0 el color está definido por defecto como blanco y para el resto de luces se ha de fijar. Por tanto, si añadimos una segunda luz habrá que indicar su posición y también su color cambiando el argumento *PARAMETRO* por *GL_DIFFUSE*, *GL_SPECULAR*, etc.

Recordar que hay que habilitar la iluminación y las luces que se vayan a usar para que se pueda visualizar correctamente la escena.

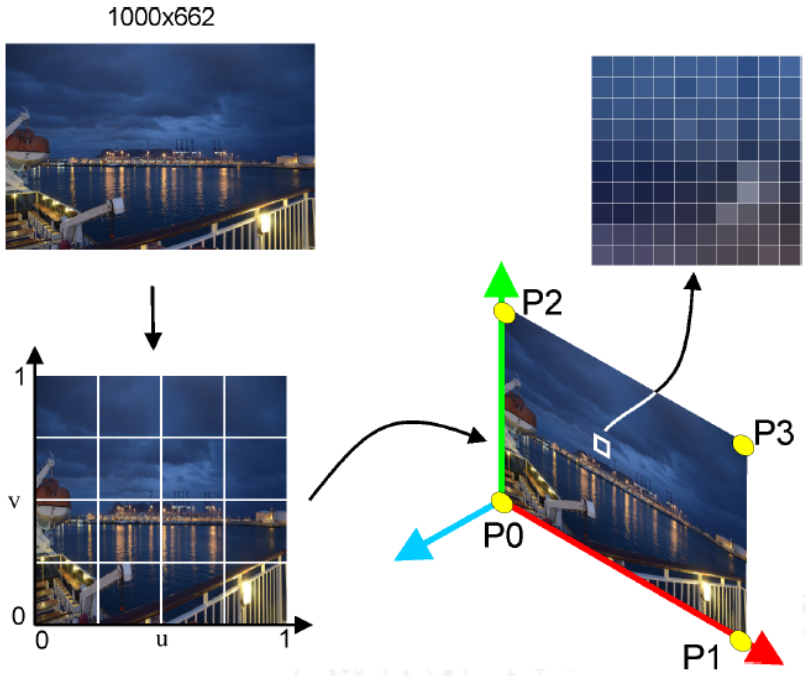


Figura 4.3: Pasos en la aplicación de una textura.

Los materiales de los objetos se definen mediante la función *glMaterial*. El formato es el siguiente: $glMaterialXY(LADO, PARAMETRO, VALOR)$. El argumento *LADO* es parte de la cara sobre el que queremos hacer la modificación que puede ser el lado delantero (anverso), el trasero (reverso) o los dos. Es semejante a *glPolygonMode*. *PARAMETRO* puede ser $GL\_DIFFUSE$, $GL\_SPECULAR$, $GL\_AMBIENT$ y $GL\_SHININESS$, que tienen una relación directa con el material (madera, plástico, etc.) y color que se quieren simular en un objeto. El argumento *VALOR* será un vector con tres componentes que se corresponde con intensidades RGB, excepto para $GL\_SHININESS$ que será un escalar que indica el tamaño del brillo.

4.2.3. Texturas

Las texturas se utilizan para aumentar el realismo en la síntesis de la imagen de una escena sin incrementar la complejidad de los objetos (aumentar el número de caras para tener más detalles).

Sea un ejemplo. Imaginemos que queremos modelar un tablón de madera. La forma del mismo es muy fácil de realizar con un paralelepípedo, un cubo escalado. Con 8 vértices y 8 colores, poco se puede conseguir para simular el veteado de la madera. Una solución consistiría en incrementar el número de vértices (por tanto, colores y caras) hasta que consiguiéramos el nivel de detalle deseado.

La solución que se propone con las texturas es la siguiente: hacemos una fotografía de cada lado de un tablón de madera real y se pega en el lado correspondiente del modelo. La

geometría del modelo sigue siendo sencilla pero la visualización es realista. No vamos a entrar en detalle a cómo se pega una imagen en las caras de un objeto, aunque se verán dos etapas o pasos básicos.

El primer paso (ver figura 4.3) consiste en utilizar para una imagen matricial definida en píxeles 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 de la imagen de su aplicación al modelo. Además se cambia el origen de la imagen matricial que es la esquina superior izquierda por la esquina inferior izquierda en este sistema normalizado.

El segundo paso consiste en asignarle a cada vértice del modelo las coordenadas de textura correspondientes. En el ejemplo de la figura 4.3, para representar un rectángulo necesitamos 2 triángulos. Si tenemos 4 puntos, P_0, P_1, P_2 y P_3, un triángulo T_0 podría estar compuesto por los puntos (P_2,P_0,P_1) y un triángulo T_1 por los puntos (P_1,P_3,P_2). Dado que queremos mostrar toda la textura en estas dos caras, se tiene 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 serían 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 muestre se consigue simplemente cambiado las coordenadas de textura, no las coordenadas de los puntos.

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

4.2.4. Implementación

Se han de realizar las siguientes tareas:

- Crear el código que permita calcular las normales a los triángulos y a los vértices. Se han de añadir dos tablas a la clase triángulos para su almacenamiento.

- Crear el código que permita dibujar los objetos con iluminación plana. Se usa la tabla de normales a triángulos.
- Crear el código que permita dibujar los objetos con iluminación con suavizado de Gouraud. Se usa la tabla de normales a vértices.
- Colocar dos fuentes de luz: una direccional blanca y otra puntual de color diferente al blanco. Esta segunda luz ha de rotar alrededor de uno de los ejes principales.
- Para el objeto jeráquico asignar al menos tres materiales con colores y acabados diferentes (mate, satinado y brillante) [1].
- Para la esfera calcular de forma exacta sus normales a los vértices.
- Pegar una textura a un objeto sencillo tal como una pirámide o un cubo (es posible que sea necesario definir nuevas clases pirámide o cubo) y crear una nueva función de dibujado con textura.

Se añaden nuevas teclas para el control de eventos de teclado con respecto a las prácticas anteriores, que queda de la siguiente manera (se pueden usar otras teclas):

- Tecla 1: visualizar en modo puntos.
- Tecla 2: visualizar en modo aristas.
- Tecla 3: visualizar en modo sólido.
- Tecla 4: visualizar en modo sólido colores.
- Tecla 5: visualizar en modo iluminación plano.
- Tecla 6: visualizar en modo iluminación con suavizado de Gouraud.
- Tecla 7: visualizar un objeto sencillo con textura.

- Tecla P: seleccionar pirámide.
- Tecla C: seleccionar cubo.
- Tecla O: seleccionar modelo PLY.
- Tecla R: seleccionar modelo por revolución.
- Tecla X: seleccionar modelo por extrusión.
- Tecla L: seleccionar cilindro.
- Tecla N: seleccionar cono.
- Tecla E: seleccionar esfera.

[1] En http://www.it.hiof.no/ borres/j3d/explain/light/p-materials.html se pueden ver ejemplos de materiales.

- Tecla A: seleccionar modelo jerárquico.
- Tecla S: Activar/desactivar la animación automática.
- Teclas de función F1 ...F12: modificar parámetros articulaciones.
- Tecla I: Activar/desactivar la segunda luz.
- Tecla O y P: rotación positiva o negativa de la segunda luz.

4.3. Evaluación

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

- Cálculo de las tablas de normales de caras y vértices (1 puntos) para un objeto.
- Cálculo de la tabla de normales de vértices para la esfera (0,5 puntos)
- Implementación de los modos de dibujado iluminación plana e iluminación con suavizado de Gouraud (2 puntos)
- Definición correcta de materiales para el objeto jerárquico (2 puntos)
- Definición de fuentes de luz y modificación de la posición de una de ellas (2 puntos)
- Visualización de textura en un objeto sencillo (2.5 puntos)

4.4. Duración

Esta tercera práctica se desarrollará en 2 sesiones.

4.5. 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. Theory and Practice, 2 Edition*; Addison-Wesley,1996.
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.

Práctica 5

Interacción

5.1. Objetivos

Con esta práctica se quiere que el estudiante aprenda:

- a utilizar los eventos del ratón para mover una cámara.
- a crear una cámara con proyección ortogonal y realizar un zoom con la misma
- a realizar la operación de selección, pick, para un objeto.

5.2. Desarrollo

Se tendrá que añadir código para que los dos grados de libertad en el movimiento de un ratón se conviertan en los dos giros de la cámara que hay por defecto. El movimiento de la rueda se usará para el acercamiento y alejamiento de la cámara.

Mediante el teclado se podrá cambiar entre una cámara con proyección en perspectiva y otra ortogonal. Se guardarán los valores correspondientes para que el cambio no produzca discontinuidades.

Se ha a implementar un pick tal que seleccione partes o piezas que configuren la jerarquía del objeto desarrollado en la práctica tercera. La parte seleccionada deberá cambiar de color.

5.2.1. Mover la cámara en perspectiva usando el ratón

Para controlar la cámara con el ratón es necesario hacer que los cambios de posición (movimiento en un plano) del mismo afecten a la orientación de la cámara (rotaciones), y en *glut* se utilizarán funciones 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 y la función *ratonMovido* cuando se mueva éste manteniendo pulsado algún botón.

El cambio de orientación de la cámara se gestionará en cada llamada a *ratonMovido*, que recibe la posición del cursor, debiendo comprobar el estado de los botones del ratón con *clickRaton*, ya que sólo se comenzará a mover la cámara cuando se tenga 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 );
```

los valores de *boton* y *estado* permiten saber que botón está pulsado, siendo x e y la posición en la que se encontraba el cursor cuando se pulsó. Se guardará el botón pulsado, el estado y la posición.

```
if ( boton == GLUT_RIGHT_BUTTON )
{  if ( estado == GLUT_DOWN )
   {
   // se pulsa el botón derecho, entrado en el estado "moviendo cámara"
     estadoRaton=1;
     xc=x;
     yc=y;
   }
   else estadoRaton=0;
   // se levanta el botón derecho, saliendo del estado "moviendo cámara"
}
if ( boton == GLUT_LEFT_BUTTON )
{  if ( estado == GLUT_DOWN )
   {
   // se pulsa el botón izquierdo, entrado en el estado "pick"
      estadoRaton = 2;
      xc=x;
      yc=y;
      pick(xc, yc);
   }
}
// estadoRaton=0 cuando se deja de pulsar el botón derecho
// estadoRaton=1 cuando se pulsa el botón derecho
// estadoRaton=2 cuando se pulsa el botón izquierdo
// xc, yc almacenan la posición del ratón
```

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 mientras se ha mantenido pulsado el botón derecho.

```
void ratonMovido( int x, int y )
{
 if(estadoRaton==1)
    {
      Observer_angle_y=Observer_angle_y-(x-xc);
      Observer_angle_x=Observer_angle_x+(y-yc);
      xc=x;
      yc=y;
```

```
      glutPostRedisplay();
    }
}
```

La transformación de visualización se hace en la función *change$_o$bserver*, donde se verá reflejada la actualización de los dos ángulos de giro de la cámara.

```
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);
}
```

5.2.2. Cámara ortogonal

Mediante el teclado se ha de poder cambiar la cámara en perspectiva que venimos usando en las prácticas a una cámara con proyección ortogonal que puede ser una vista en alzado, perfil o planta. Al regresar a la cámara en perspectiva la vista en ella ha ser igual a la que se tuviera antes de realizar el cambio.

Sobre la vista ortogonal se ha de realizar un zoom mediante la rueda del ratón, por ejemplo girando hacia abajo se disminuye la imagen generada y girando hacia arriba se aumenta. Se crea con la función

```
glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
         GLdouble near, GLdouble far)
```

donde

- left, right, bottom, top delimitan el tamaño del plano de proyección
- near y far son los planos de recorte delantero y trasero que conforman un volumen de vista para la proyección ortográfica

El zoom se implementa escalando los valores de left, right, bottom y top.

5.2.3. Selección por codificación de colores

Hay un mecanismo simple para determinar qué primitiva ha sido seleccionada en lugar de utilizar la selección proporcionada por OpenGL. Se trata de usar un código de color para cada objeto seleccionable. Se trata de crear una función de dibujado distinta para el buffer trasero y el delantero cuando queremos seleccionar. Cuando se pinta en el buffer delantero y el usuario hace clic, se pinta la escena “para seleccionar” en el buffer trasero y se lee el color del píxel de este buffer correspondiente con el punto 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 una función o método *seleccion*() que crea en el buffer trasero una imagen con colores distintos para cada elemento a seleccionar.
- Leer el píxel (x,y) dado por la función gestora del evento de ratón.
- Averiguar a qué objeto hemos asignado el color de dicho píxel.
- **No intercambiar buffers**. Se usa glFlush().

Un código de ejemplo, que dibuja cuatro patos con *pato.dibuja*() y cada uno de un color distinto sería:

```
void escena::seleccion() {
    glDrawBuffer(GL_BACK);
        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();
           }
    glDrawBuffer(GL_FRONT);
        }
```

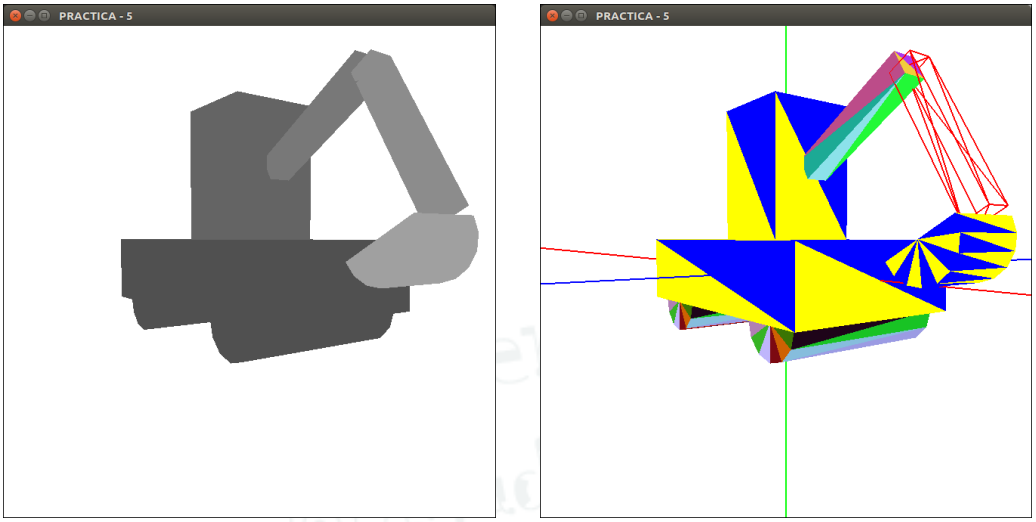
Con la función *glDrawBuffer*(...) se conmutan los buffers (*GL_BACK* o *GL_FRONT*). Para leer el color del pixel en el buffer trasero usaremos la función glReadPixels:

```
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 punto x,y, obtenido del pick)
- width,height: ancho y alto del área a leer (1,1 en nuestro caso)
- format: tipo de dato a leer (bien 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 píxeles que leamos. Es el resultado de la función.

En el caso del procesamiento del pick del objeto jerárquico de la tercera práctica, una vez dibujado el buffer trasero, se llamará a un método o función que según el color del píxel



(a) Imagen en el buffer trasero (*GL_BACK*) (b) Imagen en el buffer delantero (*GL_FRONT*)

Figura 5.1: Selección de una pieza de un objeto jerárquico. El color leído en el buffer trasero se relaciona con un brazo de la excavadora que se visualiza de modo diferente en el buffer delantero para mostrar que ha sido seleccionado.

leído pinte de un modo diferente la pieza correspondiente en el buffer delantero como forma de realimentación (ver figura 5.1).

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 tener desactivados GL_LIGHTING y GL_TEXTURE.

5.3. Evaluación

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

- Mover la cámara con el ratón (1.5 puntos)
- Implementar una cámara ortogonal y el zoom de la misma (2 puntos)
- Implementar la selección por color de las piezas del objeto jerárquico de la práctica tercera (6.5 puntos)

5.4. Duración

La práctica se realizará durante dos sesiones.

5.5. 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. Theory and Practice, 2 Edition*; Addison-Wesley,1996.
- http://www.lighthouse3d.com/tutorials/opengl-selection-tutorial/