PREGUNTAS-IG-de-examen-o-posible...

flowerpower22

Informática Gráfica

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

1. **Explique los diferentes métodos que se pueden usar para realizar la selección o pick.**

- Identificación por color: a cada objeto identificable se le asigna un identificador (número natural) que se convierte a color. Al dibujar ese objeto se usa dicho color asociado almacenados en el framebuffer. Al seleccionar un píxel del objeto con coordenadas x e y, se inspecciona en el framebuffer dicha posición y se identifica el color. Para pasar del identificador al color se usan máscaras de bits para obtener cada parte.
- Intersección rayo escena: el usuario indica con el ratón el objeto más cercano en la posición del cursor. Se obtiene la posición x e y del cursor y se convierten a coordenadas de vista. Se traza un vector desde el CP y el punto indicado y se obtiene la ecuación de dicha recta con la que se calcula la intersección con los objetos intersectables. Si se ha producido una intersección se añade a la lista guardando el identificador del objeto y su profundidad. Por último, se ordena dicha lista por profundidad y devolvemos el identificador del objeto más cercano (el primero de la lista, el de menor profundidad)
- Subvolumen de visión: se marca una posición con el ratón obteniendo la posición, alrededor de la cual se crea una ventana de unos pocos píxeles y se identifican dichos píxeles. Se dibuja cada objeto con su correspondiente identificador. Si al convertir los objetos en píxeles coincide con algunos de la ventana se ha producido una selección. Se guarda el identificador del objeto y la profundidad y se hace una ordenación por profundidad para quedarnos con el identificador del objeto más cercano.

2. **Explica el funcionamiento del Z-buffer.**

Método de eliminación de caras o partes ocultas. Devuelve para cada píxel el color del objeto más cercano al observador. Puede emplearse cualquier objeto, no solo por el que está definido por caras planas. El algoritmo necesita una memoria donde va almacenando los valores de profundidad para cada píxel en una matriz (llamada z-buffer). La matriz se inicializa inicialmente a infinito. Se itera sobre cada polígono de la esfera sobre todos los píxeles que la componen, de los que se calculará la distancia en z hacia la cámara y se actualiza la matriz según el siguiente criterio:

- Si z < z-buffer (el objeto está más cerca de la cámara) -> se reemplaza el valor de ese pixel en el z-buffer por el valor del píxel del polígono actual.
- Si z > z-buffer (el objeto está más lejos que otro objeto ya visto) -> no se modifica la matriz.

3. **Explique los pasos que se siguen en OpenGL para utilizar una imagen como textura.**

1. Partimos de una imágen de textura y una geometría. El objetivo es encontrar una función que dada una coordenada de textura en dos dimensiones, devuelva un punto en tres dimensiones del modelo.
2. Se asignan las coordenadas de textura de forma explícita o implícita, automatizado con una función.
3. Dentro de la forma implícita, hay varias formas de hacerlo:
   - Coordenadas paramétricas
   - Coordenadas cilíndricas
   - Coordenadas esféricas.
4. Por último, se habilita la textura y se aplica la iluminación para darle realismo a la imagen.

4. **¿Qué matriz de OpenGL almacena la transformación de vista?**

GL_MODELVIEW es la que guarda la matriz de transformaciones en OpenGL.

5. **Indique cuáles son los parámetros para definir la cámara y cómo se usan para obtener la transformación de la vista.**

Los parámetros que definen la transformación de la vista son:

- VRP: posición donde está la cámara. Es un punto dado en el SCM (sistema de coordenadas de mundo).
- VPN: hacia donde mira la cámara. Es un vector dado en el SCM.
- VUP: indica la orientación hacia arriba. Es un vector dado en el SCM.

La transformación de la vista es un cambio del sistema de referencia que se obtiene con cálculo de matrices que representan una traslación y una rotación usando los ángulos de Euler.

6. **Indique los pasos que hay que realizar en OpenGL y los elementos que intervienen y por tanto han de estar definidos, para conseguir que una escena se vea iluminada.**

Definir los vectores normales de cada cara: hay que definir un vector normal (perpendicular a la superficie apuntando hacia fuera de la parte visible) por cada vértice de nuestra representación.
Situar las luces: para iluminar una escena será necesario situar las luces. Dos tipos de iluminación:

- Luz ambiental: ilumina toda la escena por igual (no proviene de una dirección específica).
- Luz difusa: viene de una dirección específica, y depende de su ángulo de incidencia para iluminar una superficie en mayor o menor medida.

Definiendo materiales: OpenGL permite controlar la forma en que la luz se refleja sobre nuestros objetos, que es lo que se conoce como definición de materiales.

7. **Las lentes de las cámaras con zoom permiten cambiar la zona visible, desde ángulos más grandes (gran angular) hasta más pequeños (tele). ¿Cómo se podría conseguir el mismo efecto con los parámetros de una cámara perspectiva? Explíquelo y ponga ejemplos de valores.**

Este efecto se puede conseguir con los parámetros bottom y top, left y right, near y far de la cámara, que definen el volumen de visualización. Así, se puede elegir si se quieren ver las cosas lejanas, cercanas, etc. El ángulo en particular se cambia con los parámetros left, right, top y bottom. Todo lo que queda fuera del frustum definido es ignorado y no se dibuja.

8. **¿Qué es una partícula en Informática Gráfica? Describe su ciclo de vida. ¿Puedes poner algún ejemplo?**

Las partículas son elementos lógicos a los que se debe otorgar propiedades gráficas para que sean visibles. Tiene un ciclo de vida con estas fases:

- La partícula se crea en un sitio.
- La partícula cambia o no a lo largo del tiempo, moviéndose o variando el color, dando el efecto de animación cuando se juntan muchas partículas.
- La partícula desaparece cuando su tiempo de vida, que puede ser aleatorio o fijo, termina.

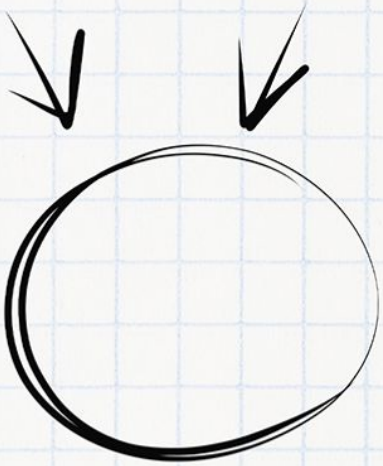
Ejemplo de uso: crear un fuego, donde las partículas van cambiando de color de rojo a amarillo y se crean más en el centro que a los lados.
Otro ejemplo: fuegos artificiales o efecto del agua cuando alguien se tira a la piscina.

Imagínate aprobando el examen

Necesitas tiempo y concentración

Planes	PLAN TURBO	PLAN PRO	PLAN PRO+
Descargas sin publi al mes	10	40	80
Elimina el video entre descargas	✓	✓	✓
Descarga carpetas	✗	✓	✓
Descarga archivos grandes	✗	✓	✓
Visualiza apuntes online sin publi	✗	✓	✓
Elimina toda la publi web	✗	✗	✓
Precios Anual	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo, ¿Qué nota vas a sacar?



WUOLAH

Informática Gráfica

Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

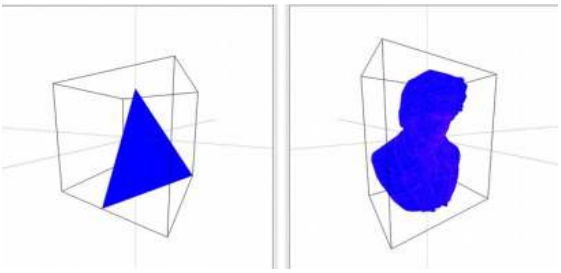
1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

9. **Explique el modelo de iluminación que se usa en OpenGL.**

El modelo de iluminación de OpenGL tiene tres componentes:
- Componente difusa: modela la reflexión de objetos que no son brillantes (objetos mates y difusos), por ejemplo una pared de yeso. La reflexión depende del ángulo entre el vector a la fuente de luz y la normal del objeto. No depende de la dirección desde la que miramos.
- Componente especular: modela la reflexión de objetos que son brillantes, por ejemplo espejo. La reflexión depende de la posición y la orientación de la luz y de la dirección en la que miramos (el reflejo del brillo cambia si cambiamos la cámara).
- Componente ambiental: es constante. Simula la iluminación de fondo y evita que superficies y objetos que no estén directamente iluminados se vean negros. No depende de la posición del observador, ni de la normal de las superficies.

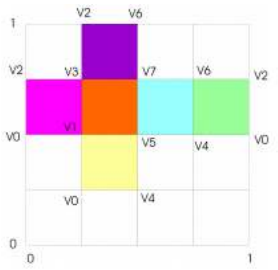
10. **Dado un objecto 3D que se define con definido por sus vértices, vector<_vertex3f> Vertices, y triángulos, vector<_vertex3ui> Triangles, implemente mediante pseudocódigo o C++ el programa que calcularía la caja frontera (la caja frontera es el menor paralelepípedo que incluye a la figura; basta con salvar el vértice con los coordenadas menores y el vértice con las mayores coordenadas).**

```
Pos_min = _vertex3f(1e8,1e8,1e8);
Pos_max = _vertex3f(-1e8,-1e8,-1e8);
for (unsigned int i=0; i<Vertices.size(); i++){
   if (Vertices[i].x < Pos_min.x)
      Pos_min.x = Vertices[i].x;
   if (Vertices[i].y < Pos_min.y)
      Pos_min.y = Vertices[i].y;
   if (Vertices[i].z < Pos_min.z)
      Pos_min.z = Vertices[i].z;
   if (Vertices[i].x > Pos_max.x)
      Pos_max.x = Vertices[i].x;
   if (Vertices[i].y > Pos_max.y)
      Pos_max.y = Vertices[i].y;
   if (Vertices[i].z > Pos_max.z)
      Pos_max.z = Vertices[i].z;
}
```



11. **Tenemos un cubo definido por sus 8 vértices y 12 triángulos y queremos aplicarle la textura de esta manera. Indicar si se puede hacer o no. Si no se puede, exponer cómo se resolvería. Indicar los valores de las coordenadas de textura para cada vértice.**

Nombramos los vértices como (imagen). Como se puede ver, el problema es que un mismo vértice (y sus coordenadas de textura) aparecen repetidos. Eso implica que la posición del vértice es la misma, pero para cada cara tiene unas coordenadas de textura.Por tanto, la solución pasa por repetir los vértices y tener una descripción para los triángulos que usan esos vértices nuevos. Para cada uno de estos nuevos vértices debe haber un elemento que guarde las coordenadas de textura correspondientes. Por ejemplo, tendríamos: V0a → (0,0.5), V0b → (0.25,0.25) y V0c → (1,0.5). De la misma forma se pondrían las coordenadas para cada vértice del cubo.



12. ¿Qué es la transformación de la vista?

La transformación de una vista es una transformación que permite cambiar de sistema de coordenadas, desde el Sistema de Coordenadas de Mundo al Sistema de Coordenadas Vista (SCV). Esta transformación permite simular el posicionamiento de la cámara en cualquier posición y orientación aplicando transformaciones geométricas (traslaciones, rotaciones, etc.).

13. Cree un ejemplo de transformación de vista incluyendo las llamadas de OpenGL.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0,0,-10);
glRotatef(37,1,0,0);
glRotatef(45,0,1,10);
```

14. Cree un ejemplo de almacenar la transformación de la vista

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslatef(0.5, 0.5, 0);
glRotated(90, 0, 0, 1);
glScalef(0.5, 1, 2);
objeto.draw(modo);
glPopMatrix();
```

15. Enumere y explique las propiedades de la transformación de perspectiva.

Acortamiento perspectivo: los objetos más lejanos producen una proyección más pequeña.
Puntos de fuga: cualquier par de líneas paralelas convergen en un punto llamado punto de fuga.
Inversión de vista: los puntos que están detrás del centro proyección se proyectan invertidos.
Distorsión topológica: cualquier elemento geométrico que tenga una parte delante y otra detrás del centro proyección produce dos proyecciones semi infinitas.

16. Queremos realizar acercarnos a un objeto para ver sus detalles. Explicar cómo se podría hacer en una proyección de perspectiva.

La solución más sencilla es acercarse al objeto. El problema es que si no se cambia el plano delantero habrá un momento en el que se alcanza el objeto y lo recortará.
Si colocamos la cámara en una posición donde los planos de corte no recorten el objeto, se puede hacer un zoom simplemente cambiando el tamaño de la ventana de proyección.

17. Dado un cubo definido por sus vértices, vector<_vertex3f> Vertices, y sus triángulos, vector<_vertex3ui> Triangles, indique lo siguiente si queremos mostrar el cubo texturado:

a. La estructura de datos para guardar las coordenadas de textura.

Sería un vector para vértices en dos dimensiones. Su tamaño sería igual al número de vértices:

```
Vector<_vertex2d> Vertices_texcoordinates;
Vertices_texcoordinates.resize(vertices.size());
```

b. La función que dibujaría el objeto texturado.

```
glBegin(GL_TRIANGLES);
for (unsigned int i=0; i<Triangles.size();i++){
   glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._0]);
   glVertex3fv((GLfloat *) &Vertices[Triangles[i]._0]);
   glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._1]);
   glVertex3fv((GLfloat *) &Vertices[Triangles[i]._1]);
   glTexCoord2fv((GLfloat *) &Vertices_texcoordinates[Triangles[i]._2]);
   glVertex3fv((GLfloat *) &Vertices[Triangles[i]._2]);
}
glEnd();
```

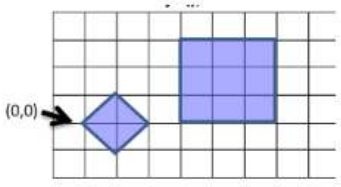
c. Un ejemplo de coordenadas de textura para cada vértice, si la textura se aplica a todas las caras sin repetirla (cada cuadrado NO muestra la textura completa).

Sin tener en cuenta el problema de los puntos repetidos, bastaría con desplegar el cubo sobre la textura y asignar los valores de las coordenadas de textura correspondientes. Por ejemplo:

```
Vertices_texcoordinates[0]=_vertex2f(0,0.5);
Vertices_texcoordinates[1]=_vertex2f(0.25,0.5);
Vertices_texcoordinates[2]=_vertex2f(0,0.75);
Vertices_texcoordinates[3]=_vertex2f(0.25,0.75);
Vertices_texcoordinates[4]=_vertex2f(0.75,0.5);
Vertices_texcoordinates[5]=_vertex2f(0.75,0.5);
Vertices_texcoordinates[6]=_vertex2f(0.5,0.75);
Vertices_texcoordinates[7]=_vertex2f(0.5,0.5);
```

18. Dada la figura inferior indique qué transformaciones son necesarias para obtener la figura de la derecha. La figura se dibuja con la función cuadrado.dibujar().

```
glPushMatrix();
glTranslatef(3, 0, 0);
glScalef(3/2, 3/2, 1);
glRotatef(45, 0, 0, 1);
cuadrado.dibujar();
glPopMatrix();
```



19. ¿Cómo se calculan las normales a un vértice?

El vector normal de un vértice es la suma de las normales de los triángulos adyacentes a dicho vértice. Para ello, calculamos las normales de todos los triángulos como el producto vectorial de dos de sus aristas, teniendo en cuenta que las normales tienen dirección hacia el exterior del objeto.

Las normalizamos con la siguiente fórmula: $\frac{n_C}{||N_C||}$

A continuación, sumamos para cada cara, el valor de su normal a la normal de cada uno de sus vértices. Una vez acabado, normalizamos las normales de los vértices: $\frac{n_V}{||N_V||}$

20. Describa las transformaciones de vista que se pueden aplicar a una cámara.

- glFrustum (left, right, bottom, top, near, far); describe una matriz de perspectiva que produce una proyección en perspectiva.

- glOrtho (left, right, bottom, top, near, far); describe una matriz de perspectiva que produce una proyección paralela.
En ambas, la matriz actual se multiplica por esta matriz y el resultado reemplaza a la matriz actual.

21. Describe brevemente para que sirven, en un programa OpenGL/glut, cada una de estas cuatro funciones:

- glutDisplayFunc(f): se llamará cada vez que se dibuje la ventana.
- glutReshapeFunc(f): control del cambio de tamaño de la ventana de visualización.
- glutKeyboardFunc(f): control de eventos con el teclado.
- glutSpecialFunc(f): control de eventos con el teclado para cuando se ha pulsado una tecla especial.

22. Explique el sistema de colores que se usa en OpenGL.

El sistema de color empleado en OpenGL es el sistema RGB (rojo, verde y azul, los colores primarios aditivos). A cada uno de estos colores se le asigna un valor, generalmente 0 y 1 (OpenGL). El valor 1 es la mayor cantidad posible de ese color, y 0 es ninguna cantidad de ese color. Podemos mezclar estos tres colores para obtener una gama completa de colores.

23. Sobre las proyecciones. Indicar si es verdadero V o falso F.

- La proyección de perspectiva acorta los objetos más lejanos (V)
- Dos líneas paralelas en el modelo sólo fugan si no son paralelas al plano de proyección (F)
- El vector Z del sistema cartesiano del observador (punto de mira punto del observador) y el vector de inclinación pueden tener cualquier orientación (V)
- El vector de inclinación siempre coincide con el eje Y del observador (F)
- La ventana debe estar centrada para que se pueda realizar la proyección (F)
- En algunos casos es obligatorio poner el plano delantero detrás del plano trasero (F)
- Si un objeto tiene todos sus vértices detrás del centro de proyección, no se podrá proyectar correctamente (V)
- En una proyección paralela el zoom se puede implementar moviendo los planos de corte (V)

24. ¿Cómo hacer zoom en una escena en perspectiva cambiando los parámetros?

Dos formas:
- Moviendo el centro de proyección (CP) y cambiando el radio de visión. Si hacemos el radio más grande, los objetos se verán más pequeños, y viceversa.

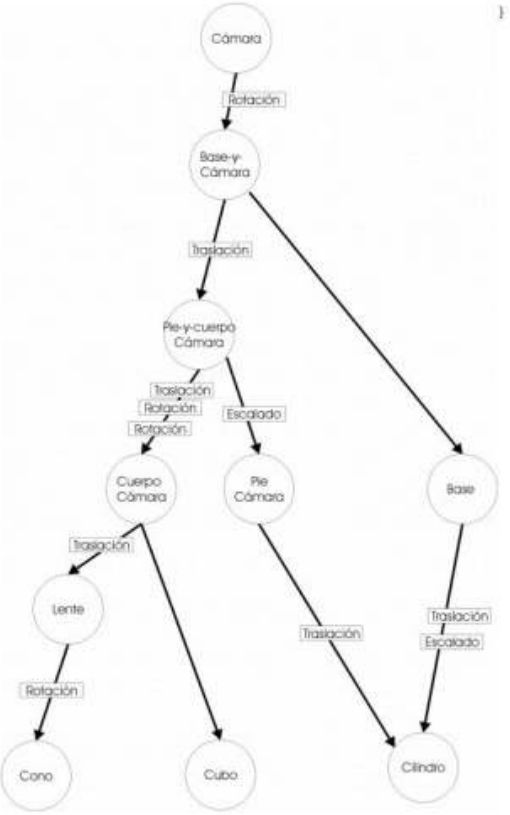
  Problema: si no se cambia el plano delantero habrá un momento en el que se alcance el objeto y se recorte.

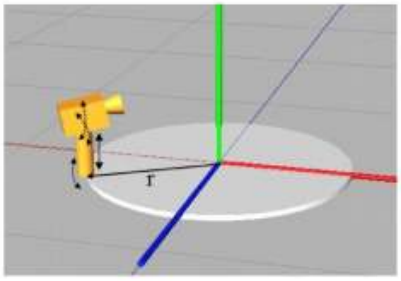
  Ventaja: solución más sencilla.

- Cambiando el tamaño de la imagen, por ejemplo la vista del otro con el factor, pues ese factor también se puede poner en la vista en perspectiva. Dicho factor hace el plano más grande o pequeño. Con la vista en perspectiva podemos poner los planos más lejos (los que van desde el CP al punto de corte) o multiplicando el plano por un factor y hacerlo más grande.

  Ventaja: para conseguirlo, se coloca la cámara en una posición donde los planos de corte no recorten el objeto. Si se fija el tamaño tendríamos que cambiar el ángulo de visión sin tocar el tamaño de la imagen, la distancia que hay desde el centro de proyección al plano.

25. Generar el grafo de escena incluyendo las transformaciones, tal que, partiendo de un cubo, un cilindro y un cono unidad, se pueda realizar el modelo de una cámara de TV situada en el borde de una plataforma de radio r. La plataforma (cilindro) rota con respecto al eje Y, la cámara rota toda ella con respecto a su base (cilindro), y permite subir y bajar, y el cuerpo de la cámara (cubo) y el objetivo (cono) rotan arriba abajo e izquierda derecha. Las medidas de las partes del modelo se dejan a discreción. Implementar el modelo en C++ y OpenGL. (nota: se han dibujado los ejes cartesianos como cilindros y no forman parte del modelo)





26. Diferencia entre fragment (fragment shader) y un pixel (vertex)

La diferencia semántica entre fragmento y pixel en OpenGL es que un pixel es un cuadrado que contiene un color y ocupa una posición (x, y) determinada, mientras que un fragmento es el mismo cuadrado, con un color, pero está en la etapa de ser procesado. Por ejemplo, puede que algunos fragmentos se eliminen, por lo que no se producirá el píxel de ese.

- initializeGL: para inicializar OpenGL.
- resizeGL: cuando la ventana cambia de tamaño y cuando carga la ventana, para conocer el tamaño inicial de la ventana.
- paintGL: dibuja cada vez que lo necesite, por lo que nos permite dibujar usando los shaders.

27. Sistemas de coordenadas

- Sistema de coordenadas del modelo: define el objeto en aquellas unidades que le sean más propias, siendo un sistema de coordenadas 3D que utiliza flotantes. Se suelen encontrar los objetos en el centro de coordenadas.
- Sistema de coordenadas del mundo: todos los objetos del escenario (incluso fuentes de luz y cámara) se definen con este sistema de coordenadas. Define las unidades más apropiadas para la escena, siendo un sistema de coordenadas 3D que utiliza flotantes.
- Sistema de coordenadas de la cámara: aquí hay que transformar de 3D a 2D. Para ello los objetos tienen que definirse con respecto a este sistema de coordenadas.
- Sistema de coordenadas de dispositivo normalizado: el volumen de visión se transforma a este sistema de coordenadas, que facilita ciertas operaciones.
- Sistema de coordenadas de dispositivo: cuando se realiza la proyección, convierte esos objetos en píxeles, que se dibujan a través de este sistema de coordenadas, dependiendo de las dimensiones del viewport. Es un sistema de coordenadas 2D de enteros.

28. Proyecciones

La idea es mostrar objetos que se definen en 3D en dispositivos que solo permite mostrarlo de forma bidimensional (2D). La proyección que se encarga de transformar objetos en 3D a 2D. Hay 2 tipos:

- Proyección perspectiva: muestra de forma más natural los objetos, como el funcionamiento del ojo humano. Se caracteriza por el acortamiento perspectivo (los objetos que se ven a lo lejos se ven más pequeños que los objetos que están más cerca).
  glFrustum: necesita los valores de la ventana, la distancia con respecto al origen del plano delantero y la distancias con respecto al origen del plano trasero.
- Proyección paralela: por muy lejos que esté el objeto siempre se ven del mismo tamaño, ya que el centro de proyección converge en el infinito.
  glOrtho
  En realidad la proyección paralela no existe, pero se usa para el dibujo técnico y otro tipo de reproducciones en los que no queremos que se cambien las medidas con respecto a la distancia.

Los planos de corte recortan el objeto que está por delante del panel delantero y/o detrás del panel trasero, evitando uno de los principales problemas, la distorsión topológica, y cargar un objeto complejo en unos pocos píxeles cuando el objeto está demasiado lejos.

29. Transformaciones

Se representan mediante matrices 4x4 que representan las transformaciones a realizar.
La transformación es la multiplicación de la matriz por el vector que representa la posición de cada vértice del objeto.
Traslación: mover un objeto de una posición a otra. Siendo p0 el punto original y T el valor a trasladar, la posición donde se moverá el punto sería: p0′ = p0 + T. Con lo cual:

$$x' = x * Tx$$
$$y' = y * Ty$$
$$z' = z * Tz$$

Escalado: cambiar el tamaño del objeto. Siendo p0 el punto original y S el factor de escala, la nueva posición del punto es: p0′ = p0 * S. Con lo cual:

$$x' = x * Sx$$
$$y' = y * Sy$$
$$z' = z * Sz$$

El escalado se realiza con respecto al origen de coordenadas.
Si S > 1: el objeto se agranda.
Si 0 < S < 1: el objeto encoge.
Si S = 0: el objeto se destruye poniendo todos los puntos del objeto en el (0,0,0).
Si S < 0: el objeto se invierte.
Si S = S': forman un escalado homogéneo.
Si S != S': forman un escalado heterogéneo.
Rotación: cambiar la orientación del objeto. Como estamos en 3D podemos rotar con respecto al eje x, y y z. De tal forma para rotar se aplica los siguientes cálculos:

Eje X

$$x' = x$$
$$y' = y * cos(\alpha) - z * sin(\alpha)$$
$$z' = y * sin(\alpha) + z * cos(\alpha)$$

Eje Y

$$x' = x * cos(\alpha) + z * sin(\alpha)$$
$$y' = y$$
$$z' = -x * sin(\alpha) + z * cos(\alpha)$$

Eje Z

$$x' = x * cos(\alpha) - y * sin(\alpha)$$
$$y' = x * sin(\alpha) + y * cos(\alpha)$$
$$z' = z$$

Sobre el eje al que queremos rotar no cambia. En caso de que queramos rotar con respecto a otro pivote que no sean (x, y o z), tendrá que convertirlo en el origen.
La matriz 4x4 contendrá todas las transformaciones y se aplicará a cada punto. Para realizar los cambios se hace en un orden específico con respecto al centro de coordenadas, ya que si lo hacemos en otro orden el resultado no es el mismo.

1º Escalado o rotación
2º Traslación

Al usar una matriz 4x4 estamos pasando de 3D a 4D.

¿Cómo pasamos un punto en coordenadas 3D a 4D? Usamos la siguiente fórmula, siendo w != 0:

$$x' = x * w$$
$$y' = y * w$$
$$z' = z * w$$

Con lo cual, las coordenadas son (x*w, y*w, z*w, w).
Y a la inversa (4D a 3D) →(x/w, y/w, z/w, 1)
En vertex shader aparecen 3 matrices que representan las transformaciones principales:

- Transformación de modelado: pasar de coordenadas de modelado a coordenadas de mundo.
- Transformación de la cámara: pasar de coordenadas de mundo a coordenadas de cámara.
- Transformación de proyección: pasar de coordenadas de cámara a coordenadas de dispositivo normalizado.

Esta separación es muy importante, ya que nos permite transformar un punto a cualquier sistema de coordenadas definidas por las transformaciones.

30. ¿Qué es la selección?

Una selección es cuando, dado el conjunto de elementos de la escena, queremos poder seleccionar individualmente cada uno de ellos para aplicarle algún tipo de proceso (borrarlo, moverlo, etc).

31. Componentes de una cámara

- Posición de la cámara: transforma el sistema de coordenadas del mundo de los vértices del objeto por el sistema de coordenadas de vista. Para ese cambio hay que hacer transformaciones geométricas.
- Dirección de la cámara
- Orientación de la cámara: vertical, apaisada u otro ángulo.
- Zoom: el cierre o apertura de la lente.

32. Triángulos

En OpenGL casi todo se construye con triángulos. Esto se ha definido así, puesto que es un polígono sencillo y por sus propiedades para su visualización. Cualquier objeto que queramos representar y que tenga superficie, habrá que hacerlo a través de triángulos.
Cada triángulo se crea mediante 3 vértices y habrá que indicarle en OpenGL que esos 3 vértices son un triángulo y no como vértices independientes o como una recta.

33. Interpolación de colores

La interpolación es necesaria porque se usa para la iluminación y para implementar otros procedimientos que necesiten valores intermedios, por los extremos. En el dibujado de objetos no hay diferencia entre distintos objetos, ya que todos están compuestos por triángulos, sólo varía el número de triángulos. Tenemos 3 funciones: model (aplica las transformaciones al modelo), view y projection.

34. glPolygonMode(MODO_CARA, MODO_VISUALIZACIÓN);

MODO_CARA: qué lados de la cara afecta. Tenemos GL_FRONT (cara de alante), GL_BACK (cara de atrás) y GL_FRONT_AND_BACK (ambas caras delantera y trasera).
MODO_VISUALIZACIÓN: temenos GL_POINT (sólo vértices), GL_LINE (sólo las líneas de los triángulos que forman cada 3 vértices) y GL_FILL (rellena el hueco que une los 3 vértices).

35. Revolución de objetos

Barrido por revolución: dada una curva, podemos girar el objeto alrededor de un eje (x,y,z) un número de veces y a partir de dichos perfiles obtener la superficie.
Los puntos de perfil generatriz giran con respecto a un eje y obteniendo una serie de perfiles a partir de los cuales obtener las caras. Para realizar el giro o las revoluciones se usan las funciones seno y coseno, ya que con eso podemos calcular las posiciones en una circunferencia.
Una vez creado el perfil del objeto, se obtienen los triángulos que representan la superficie. Para hacerlo hay que olvidarse la forma del objeto y crear una forma más sencilla como si aplastásemos el objeto obteniendo un plano del objeto, manteniendo la posición relativa de los vértices. Para cerrar el objeto hay que repetir los puntos del principio al final, para unir los puntos y formar el objeto con los triángulos.

36. Modelado de objetos

Para el modelado de objetos necesitamos objetos, observador y luces.
En un modelo hay que tener una representación de los objetos tratados computacionalmente. Un objeto real ocupa espacio, color, material y otros atributos.
Físicamente están formados por partículas que se agrupan en diferentes niveles, el nivel más básico son los *quarks* que componen los átomos.

37. ¿Cuál es la representación tratable computacionalmente de un objeto?

Al ser un computador, el punto se consideraría una partícula sin dimensión, que debe estar referenciado a un sistema de coordenadas. Tipos de modelos:
- Modelado de puntos: crea un objeto tridimensional con otra de 0D. Su principal característica es que el punto se posiciona en un espacio tridimensional (x,y,z) en coordenadas cartesianas o polares. Dado un conjunto de vértices podemos obtener un objeto, pero el problema es cómo asociar los puntos a un objeto. La solución es utilizar los más representativos.
- Modelado de alambres: el modelado de puntos no nos permite ver con claridad qué objeto es, excepto si hay una gran densidad de puntos. La solución es mostrar la relaciones de los puntos, de tal forma que visualiza las aristas, mostrando la relación de los puntos.
- Modelado de superficie: con el modelo de alambres se ve algo más claro el objeto, pero puede producir una solución ambigua y poco realista, entonces la solución es mostrar la superficie (modelo de fronteras).

38. Optimización de modelos

Los procedimientos para dibujar los puntos, aristas y caras no son muy eficientes, ya que generan una llamada por primitiva y no reutilizan la información. Se soluciona usando primitivas especializadas:
- Tira de cuadriláteros
- Tira de triángulos
- Abanico de triángulos

39. Modelado jerárquico sin movimiento

Si se cambia la transformación de modelado y se vuelve a dibujar el cubo sin borrar la escena, se obtendría un segundo cubo al compilar y ejecutar el programa, de tal forma que solo necesitamos crear un solo objeto cubo y podremos pintar tantas veces queramos ese objeto.
Si añado una tercera transformación y se vuelve a dibujar el cubo, se añade un tercer cubo al ejecutar el programa.
La jerarquización es básicamente establecer dependencias.

40. Iluminación

Para hacer que el objeto sea aún más real tenemos 3 aproximaciones al realismo (de menos a más):
- Sombreado plano: se hace con los cálculos de sombreado en el *vertex shader*.
- Sombreado de Gouraud: se hace con los cálculos de sombreado en el *vertex shader*.
- Sombreado de Phong: se hacen con los cálculos de sombreado en el *fragment shader*, ya que no se puede hacer sobre un vector (*vertex shader*).

La iluminación necesita un modelo de reflexión. En OpenGL usamos 3 componentes sencillas: la ambiental, la difusa y la especular.
La diferencia entre Gouraud y Phong es que Gouraud es una versión simplificada de Phong. En Gouraud se obtienen los valores de reflexión en los vértices, se calculan los colores correspondientes y los mismos se interpolan para obtener el resto de posiciones (fragmentos/pixeles), realizando la interpolación con el *fragment shader*. Pero con Phong se quiere obtener un resultado más exacto haciendo que se realicen los cálculos de reflexión en todos los puntos del objeto.

41. Texturas

La textura es como un envoltorio, es como si le pegásemos una foto en un objeto y se adaptase a la forma del objeto. Pasos para aplicar la textura:

- Pasamos la imagen que está en forma matricial, en un sistema de coordenadas normalizado, con las coordenadas u y v, con rangos entre 0 y 1 ambos incluidos ($0 \leq u \leq 1{,}0 \leq v \leq 1$). Esto permite independizar el tamaño real de la imagen de su aplicación al modelo.
- Después le asignamos a cada punto del modelo las coordenadas de textura correspondientes.

42. Ejemplo brazo mecánico (modelo jerárquico con movimiento)

El brazo mecánico consta de una pinza, la mano, brazo1, brazo2 y la base.

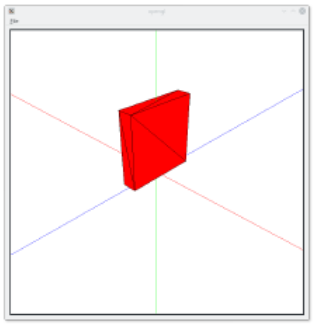


Figura 13.1: Pinza

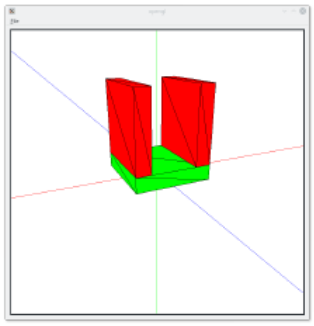


Figura 13.2: Mano

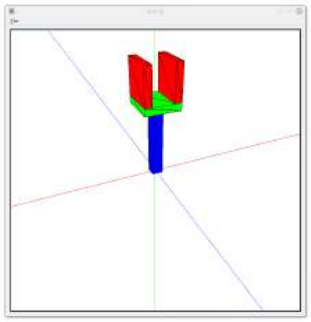


Figura 13.3: Brazo1

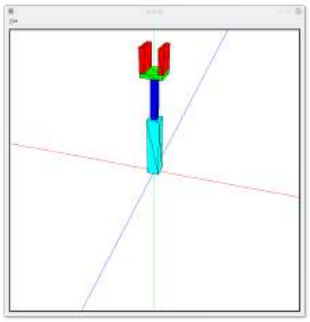


Figura 13.4: Brazo2

Jerarquía de dependencia: pinza mano brazo1 brazo2 base
Para hacer los movimientos, en el caso de la mano, lo que vamos a hacer es que las pinzas se acerquen o se alejen aplicando una transformación sobre el eje X.

```
//pinza 1
glPushMatrix();
  glTranslatef(-traslacion,0.5,2.5);
  glTranslatef(0.4,0.2,0);
  pinza.draw();
glPopMatrix();
//pinza 2
glPushMatrix();
  glTranslatef(traslacion,0.5,2.5);
  glTranslatef(0.4,0.2,0);
  pinza.draw();
glPopMatrix();
```

Vamos a añadir otro movimiento al brazo1, haciendo que rote:

```
//rotación del brazo1
glPushMatrix();
  glTranslatef(0,2,0);
  glRotatef(angulo,0,1,0);
  brazo1.draw();
glPopMatrix();
```

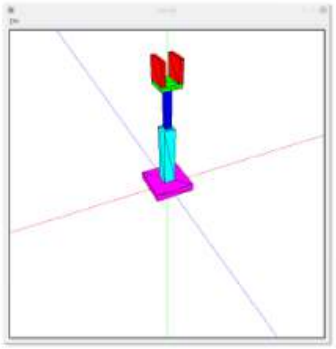


Figura 13.5: Base

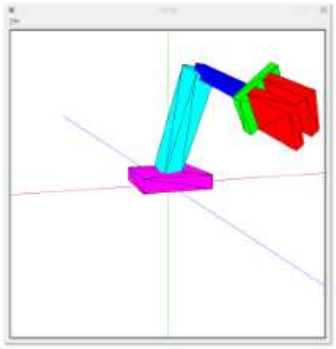


Figura 13.6: Ejemplo completo con movimiento

43. Describe la estructura de aristas aladas, y explica (apoyando la explicación con pseudocódigo si lo ves conveniente) el proceso que se debe seguir para construir una estructura de aristas aladas a partir de un archivo PLY.

Las estructuras de aristas aladas son una forma de representar un modelo 3D en términos de sus características geométricas básicas, como vértices, caras y aristas. Estas estructuras se utilizan comúnmente en la computación gráfica para procesamiento de modelos 3D y su visualización.
Para construir una estructura de aristas aladas a partir de un archivo PLY (formato de almacenamiento de modelos 3D), es necesario seguir los siguientes pasos:
1. Leer el archivo PLY: utilizar una biblioteca de lectura de archivos PLY para leer el archivo y almacenar los datos en variables.
2. Procesar los vértices: recorrer cada vértice del modelo y almacenarlo en una estructura de datos, como un vector.
3. Procesar las caras: recorrer cada cara del modelo y almacenar los índices de los vértices que forman la cara en una estructura de datos, como un vector o una matriz.
4. Procesar las aristas: recorrer cada arista del modelo y almacenar los índices de los vértices que forman la arista en una estructura de datos, como un vector o una matriz.
5. Construir la estructura de aristas aladas: utilizar las estructuras de datos de vértices, caras y aristas para construir la estructura de aristas aladas.

44. Dibujar un cuadrado en 2D color negro.

```
glColor3f(0.0,0.0,0.0);
   glLineWidth(3);   // para que dibuje (solo) las líneas (bordes)
   glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
   glBegin(GL_POLYGON);
      glVertex3f(-0.2,0.65,0.0);
      glVertex3f(0.2,0.65,0.0);
      glVertex3f(0.2,0.85,0.0);
      glVertex3f(-0.2,0.85,0.0);
   glEnd();
```

45. Dibujar puntos, líneas y sólido 3D.

```
void _puntos3D::draw_puntos(float r, float g, float b, int grosor){
	int i;
	glPointSize(grosor);
	glColor3f(r,g,b);
	glBegin(GL_POINTS);
	for (i=0;i<vertices.size();i++){
		glVertex3fv((GLfloat *) &vertices[i]);
	}
	// glPointSize(1);
	glEnd();
}

void _triangulos3D::draw_aristas(float r, float g, float b, int grosor){
	int i;
	glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
	glLineWidth(grosor);
	glColor3f(r,g,b);
	glBegin(GL_TRIANGLES);
	for (i=0;i<caras.size();i++){
		glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
		glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
		glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
	}
	glEnd();
}

void _triangulos3D::draw_solido(float r, float g, float b){
	int i;
	glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
	glColor3f(r,g,b);
	glBegin(GL_TRIANGLES);
	for (i=0;i<caras.size();i++){
		glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
		glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
		glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
	}
	glEnd();
}
```

46. Dibujar con un color diferente cada cara.

```
void _triangulos3D::draw_solido_colores_caras( ){
      int i;
      glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
      glBegin(GL_TRIANGLES);
      for (i=0;i<caras.size();i++){
            glColor3f(colores_caras[i].r,colores_caras[i].g,colores_caras[i].b);
            glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
            glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
            glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
      }
      glEnd();
}
```

47. Clase cubo.

```
//vertices
vertices.resize(8);
vertices[0].x=-tam; vertices[0].y=0; vertices[0].z=tam;
vertices[1].x=tam; vertices[1].y=0; vertices[1].z=tam;
vertices[2].x=tam; vertices[2].y=0; vertices[2].z=-tam;
vertices[3].x=-tam; vertices[3].y=0; vertices[3].z=-tam;
vertices[4].x=-tam; vertices[4].y=tam*2; vertices[4].z=tam;
vertices[5].x=tam; vertices[5].y=tam*2; vertices[5].z=tam;
vertices[6].x=tam; vertices[6].y=tam*2; vertices[6].z=-tam;
vertices[7].x=-tam; vertices[7].y=tam*2; vertices[7].z=-tam;

// triangulos
caras.resize(12);

// base
caras[0]._0=0; caras[0]._1=1; caras[0]._2=2;
caras[1]._0=0; caras[1]._1=2; caras[1]._2=3;
// techo
caras[2]._0=4; caras[2]._1=5; caras[2]._2=6;
caras[3]._0=4; caras[3]._1=6; caras[3]._2=7;
// frente
caras[4]._0=0; caras[4]._1=1; caras[4]._2=4;
caras[5]._0=1; caras[5]._1=5; caras[5]._2=4;
// atras
caras[6]._0=3; caras[6]._1=2; caras[6]._2=6;
caras[7]._0=3; caras[7]._1=6; caras[7]._2=7;
// izquierda
caras[8]._0=0; caras[8]._1=4; caras[8]._2=3;
caras[9]._0=3; caras[9]._1=4; caras[9]._2=7;
// derecha
caras[10]._0=1; caras[10]._1=2; caras[10]._2=6;
caras[11]._0=1; caras[11]._1=6; caras[11]._2=5;
```

48. Clase pirámide.

```
// vertices
vertices.resize(5);
vertices[0].x=-tam;vertices[0].y=0;vertices[0].z=tam;
vertices[1].x=tam;vertices[1].y=0;vertices[1].z=tam;
vertices[2].x=tam;vertices[2].y=0;vertices[2].z=-tam;
vertices[3].x=-tam;vertices[3].y=0;vertices[3].z=-tam;
vertices[4].x=0;vertices[4].y=al;vertices[4].z=0;

// caras
caras.resize(6);
caras[0]._0=0;caras[0]._1=1;caras[0]._2=4;
caras[1]._0=1;caras[1]._1=2;caras[1]._2=4;
caras[2]._0=2;caras[2]._1=3;caras[2]._2=4;
caras[3]._0=3;caras[3]._1=0;caras[3]._2=4;
caras[4]._0=3;caras[4]._1=1;caras[4]._2=0;
caras[5]._0=3;caras[5]._1=2;caras[5]._2=1;
```

49. Dibujar gama azules.

```
void _triangulos3D::colores_gama_azules(float umbral){
	int i;
	float z;
	colores_caras.resize(caras.size());
	for (i = 0; i < caras.size(); i++){
		z=rand()%1000/1000.0;
		if(z > 0.5){
			colores_caras[i].r=z;
			colores_caras[i].g=z;
		}
		else{
		colores_caras[i].r=0.0;
		colores_caras[i].g=0.0;
		}
	colores_caras[i].b=rand()%1000/1000;
	}
}
```

50. Clase cilindro.

```
_cilindro::_cilindro(float radio, float altura, int lados){
	vector<_vertex3f> perfil;
	_vertex3f aux;
	aux.x=radio; aux.y=-altura; aux.z=0.0;
	perfil.push_back(aux);
	aux.x=radio; aux.y=altura; aux.z=0.0;
	perfil.push_back(aux);
	parametros(perfil, lados, 1, 1, 0);
}
```

51. Clase esfera.

```
_esfera::_esfera(float radio, int latitud, int longitud){
	vector<_vertex3f> perfil;
	_vertex3f aux;
	int i;
	for (i=1;i<latitud;i++){
		aux.x=radio*cos(M_PI*i/(latitud*1.0)-M_PI/2.0);
		aux.y=radio*sin(M_PI*i/(latitud*1.0)-M_PI/2.0);
		aux.z=0.0;
		perfil.push_back(aux);
	}
	parametros(perfil,longitud,1,1,1);
}
```

52. Clase cono.

```
_cono::_cono(float radio, float altura, int lados){
	vector<_vertex3f> perfil;
	_vertex3f aux;
	aux.x=radio; aux.y=0.0; aux.z=0.0;
	perfil.push_back(aux);
	aux.x=0.0; aux.y=altura; aux.z=0.0;
	perfil.push_back(aux);
	parametros(perfil, lados, 1, 1, 2);
}
```

53. Objeto por revolución.

```
// perfil
vector<_vertex3f> perfil, poligono;
_vertex3f aux;
aux.x=1.0; aux.y=-1.5; aux.z=0.0;
perfil.push_back(aux);
aux.x=1.4; aux.y=-0.5; aux.z=0.0;
perfil.push_back(aux);
aux.x=1.4; aux.y=0.5; aux.z=0.0;
perfil.push_back(aux);
aux.x=1.0; aux.y=1.5; aux.z=0.0;
perfil.push_back(aux);
rotacion.parametros(perfil,6,1,1);   // con las dos tapas: 1,1
```

54. Dibujar colores random y colores ajedrez.

```
void _triangulos3D::colors_random(){
  int i, n_c;
  n_c=caras.size();
  colores_caras.resize(n_c);
  srand (time(NULL));
  for (i=0;i<n_c;i++){
    colores_caras[i].r=rand()%1000/1000.0;
    colores_caras[i].g=rand()%1000/1000.0;
    colores_caras[i].b=rand()%1000/1000.0;
  }
}

void _triangulos3D::colors_chess(float r1, float g1, float b1, float r2,
float g2, float b2){
  int i, n_c;
  n_c=caras.size();
  colores_caras.resize(n_c);
  for (i=0;i<n_c;i++){
    if (i%2==0){
      colores_caras[i].r=r1;
      colores_caras[i].g=g1;
      colores_caras[i].b=b1;
    }
    else{
      colores_caras[i].r=r2;
      colores_caras[i].g=g2;
      colores_caras[i].b=b2;
    }
  }
}
```

55. Objeto excavadora.

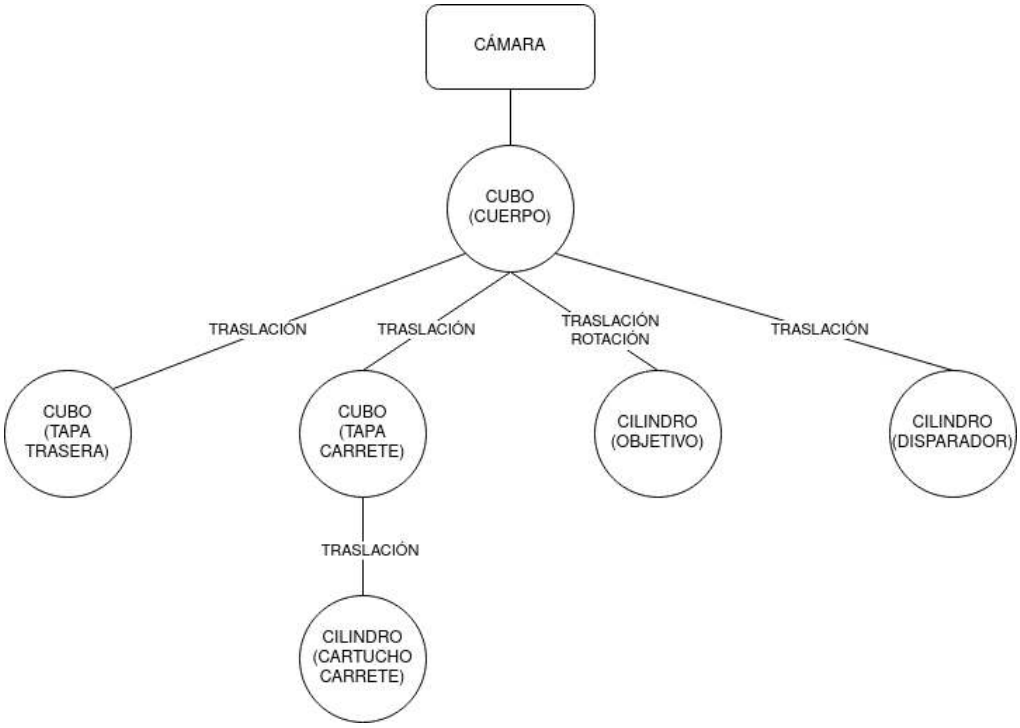
```
//*************************************************************************
// piezas
//*************************************************************************

//*************************************************************************
// pala
//*************************************************************************
_pala::_pala(float radio, float ancho, int num){
  vector<_vertex3f> perfil;
  _vertex3f vertice_aux;
  _vertex3i cara_aux;
  int i, j;
  vertice_aux.x=radio; vertice_aux.y=0; vertice_aux.z=-ancho/2.0;
  perfil.push_back(vertice_aux);
  vertice_aux.z=ancho/2.0;
  perfil.push_back(vertice_aux);
```


```
  // tratamiento de los vértices
  for (j=0;j<=num;j++){
    for (i=0;i<2;i++){
      vertice_aux.x=perfil[i].x*cos(M_PI*j/(1.0*num))-
                    perfil[i].y*sin(M_PI*j/(1.0*num));
      vertice_aux.y=perfil[i].x*sin(M_PI*j/(1.0*num))+
                    perfil[i].y*cos(M_PI*j/(1.0*num));
      vertice_aux.z=perfil[i].z;
      vertices.push_back(vertice_aux);
    }
  }

  // tratamiento de las caras
  for (j=0;j<num;j++){
    cara_aux._0=j*2;
    cara_aux._1=(j+1)*2;
    cara_aux._2=(j+1)*2+1;
    caras.push_back(cara_aux);
    cara_aux._0=j*2;
    cara_aux._1=(j+1)*2+1;
    cara_aux._2=j*2+1;
    caras.push_back(cara_aux);
  }

  // tapa inferior
  vertice_aux.x=0;
  vertice_aux.y=0;
  vertice_aux.z=-ancho/2.0;
  vertices.push_back(vertice_aux);

  for (j=0;j<num;j++){
    cara_aux._0=j*2;
    cara_aux._1=(j+1)*2;
    cara_aux._2=vertices.size()-1;
    caras.push_back(cara_aux);
  }

  // tapa superior
  vertice_aux.x=0;
  vertice_aux.y=0;
  vertice_aux.z=ancho/2.0;
  vertices.push_back(vertice_aux);

  for (j=0;j<num;j++){
    cara_aux._0=j*2+1;
    cara_aux._1=(j+1)*2+1;
    cara_aux._2=vertices.size()-1;
    caras.push_back(cara_aux);
  }
  colors_chess(1.0,1.0,0.0,0.0,0.0,1.0);
}
```

```
//*************************************************************************
// brazo
//*************************************************************************
_brazo::_brazo(){
  ancho=0.6;
  alto=0.1;
  fondo=0.1;
  colors_chess(1.0,1.0,0.0,0.0,0.0,1.0);
};

void _brazo::draw(_modo modo, float r, float g, float b, float grosor){
  glPushMatrix();
    glScalef(ancho, alto, fondo);
    glTranslatef(0.5,0,0);
    cubo.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//*************************************************************************
// cabina
//*************************************************************************
_cabina::_cabina(){
  ancho=0.4;
  alto=0.6;
  fondo=0.4;
  cubo.colors_chess(1.0,1.0,0.0,0.0,0.0,1.0);
};

void _cabina::draw(_modo modo, float r, float g, float b, float grosor){
  glPushMatrix();
    glScalef(ancho, alto, fondo);
    cubo.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//*************************************************************************
// sustentación
//*************************************************************************
_sustentacion::_sustentacion(){
  ancho=1.2;
  alto=0.3;
  fondo=0.8;
  radio=0.15;
  base.colors_chess(1.0,1.0,0.0,0.0,0.0,1.0);
};

void _sustentacion::draw(_modo modo, float r, float g, float b, float
grosor){
  glPushMatrix();
    glTranslatef(2*ancho/6,-alto/2.0,0);
    glRotatef(90,1,0,0);
    glScalef(radio, fondo/2.2, radio);
```

```
    rueda.draw(modo, r, g, b, grosor);
  glPopMatrix();
  glPushMatrix();
    glTranslatef(-2*ancho/6,-alto/2.0,0);
    glRotatef(90,1,0,0);
    glScalef(radio, fondo/2.2, radio);
    rueda.draw(modo, r, g, b, grosor);
  glPopMatrix();
  glPushMatrix();
    glScalef(ancho, alto, fondo);
    base.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//*************************************************************************
// excavadora (montaje del objeto final)
//*************************************************************************
_excavadora::_excavadora(){
  giro_cabina = 0.0;
  giro_primer_brazo = 0.0;
  giro_primer_brazo_max = 0;
  giro_primer_brazo_min = -90;
  giro_segundo_brazo = 0.0;
  giro_segundo_brazo_max = 30;
  giro_segundo_brazo_min = 0;
  giro_pala = 0.0;
  giro_pala_max = 50.0;
  giro_pala_min = -90.0;
  tamanio_pala=0.15;
};

void _excavadora::draw(_modo modo, float r, float g, float b, float
grosor){
  glPushMatrix();
    sustentacion.draw(modo, r, g, b, grosor);
    glTranslatef(0,(cabina.alto+sustentacion.alto)/2.0,0);
    glRotatef(giro_cabina,0,1,0);
    cabina.draw(modo, r, g, b, grosor);
    glTranslatef(cabina.ancho/2.0,0,0);
    glRotatef(giro_segundo_brazo,0,0,1);
    brazo.draw(modo, r, g, b, grosor);
    glTranslatef(brazo.ancho,0,0);
    glRotatef(giro_primer_brazo,0,0,1);
    brazo.draw(modo, r, g, b, grosor);
    glTranslatef(brazo.ancho,0,0);
    glRotatef(giro_pala,0,0,1);
    glTranslatef(tamanio_pala,0,0);
    glScalef(tamanio_pala, tamanio_pala, tamanio_pala);
    pala.draw(modo, r, g, b, grosor);
  glPopMatrix();
};
```

56. Grafo modelo cámara.



57. Objeto cámara.

```
//************************************************************************
// base de la camara
//************************************************************************
_base::_base(){
  ancho=1.7;
  alto=1.0;
  fondo=0.4;
  cubo_base.colors_chess(0.5,0.2,0.0,1.0,1.0,0.5);
};

void _base::draw(_modo modo, float r, float g, float b, float grosor) {
  glPushMatrix();
    glScalef(ancho, alto, fondo);
    cubo_base.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//************************************************************************
// tapa trasera de la camara
//************************************************************************
_tapa::_tapa() {
  ancho=1.5;
```

```
  alto=0.8;
  fondo=0.1;
  cubo_tapa.colors_chess(1.0,1.0,0.5,0.5,0.2,0.0);
};

void _tapa::draw(_modo modo, float r, float g, float b, float grosor) {
  glPushMatrix();
    glScalef(ancho, alto, fondo);
    glTranslatef(-ancho/3,0,-fondo);
    cubo_tapa.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//*************************************************************************
// cartucho de la camara
//*************************************************************************
_cartucho::_cartucho() {
  alto2=0.1;
  fondo2=0.15;
  radio=0.15;
  cilindro_cartucho.colors_chess(0.5,0.2,0.0,1.0,1.0,0.5);
};

void _cartucho::draw(_modo modo, float r, float g, float b, float grosor)
{
  glPushMatrix();
    glScalef(radio, alto2, fondo2);
    glTranslatef(0,alto2*10,0);
    cilindro_cartucho.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//*************************************************************************
// palanca carrete de la camara
//*************************************************************************
_palanca::_palanca() {
  ancho=0.3;
  alto=0.1;
  fondo=0.1;
  cubo_palanca.colors_chess(0.5,0.2,0.0,1.0,1.0,0.5);
};

void _palanca::draw(_modo modo, float r, float g, float b, float grosor) {
  glPushMatrix();
    glScalef(ancho, alto, fondo);
    glTranslatef(ancho*1.5,alto*2.5,0);
    cubo_palanca.draw(modo, r, g, b, grosor);
  glPopMatrix();
};


//*************************************************************************
```

```
// disparador de la camara
//************************************************************************
_disparador::_disparador() {
  alto=0.1;
  fondo=0.15;
  radio=0.15;
  cilindro_disparador.colors_chess(0.5,0.2,0.0,1.0,1.0,0.5);
};

void _disparador::draw(_modo modo, float r, float g, float b, float
grosor) {
  glPushMatrix();
    glScalef(radio, alto, fondo);
    glTranslatef(0,alto*10,0);
    cilindro_disparador.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//************************************************************************
// objetivo de la camara
//************************************************************************
_objetivo::_objetivo() {
  alto=0.2;
  fondo=0.4;
  radio=0.4;
  cilindro_objetivo.colors_chess(0.5,0.2,0.0,1.0,1.0,0.5);
};

void _objetivo::draw(_modo modo, float r, float g, float b, float grosor)
{
  glPushMatrix();
    glScalef(radio, alto, fondo);
    glTranslatef(0,alto*5,0);
    cilindro_objetivo.draw(modo, r, g, b, grosor);
  glPopMatrix();
};

//************************************************************************
// camara (montaje del objeto final)
//************************************************************************
_camara::_camara() {
  giro_disparador = 0.0;
  giro_objetivo = 0.0;
  giro_cartucho = 0.0;
  giro_palanca = 0.0;
  giro_palanca_max = 135.0;
  giro_palanca_min = 0.0;
  giro_tapa = 0.0;
  giro_tapa_max = -90.0;
  giro_tapa_min = 0.0;
};
void _camara::draw(_modo modo, float r, float g, float b, float grosor){
```


```
  glPushMatrix();
    base.draw(modo, r, g, b, grosor);
    glTranslatef(base.ancho/3.0,
                 (cartucho.alto2*2+base.alto)/2.0-cartucho.alto2,
                  0.0);
    glRotatef(giro_cartucho,0,1,0);
    cartucho.draw(modo, r, g, b, grosor);
    glPushMatrix();
      glTranslatef(0.0,palanca.alto*1.3,0.0);
      glRotatef(giro_palanca,0,0,1);
      palanca.draw(modo, r, g, b, grosor);
    glPopMatrix();
  glPopMatrix();
  glPushMatrix();
    glTranslatef(tapa.ancho/2,0.0,-base.fondo/1.75);
    glRotatef(giro_tapa,0,1,0);
    tapa.draw(modo, r, g, b, grosor);
  glPopMatrix();
  glPushMatrix();
    glTranslatef(-base.ancho/3.0,
             (disparador.alto*2+base.alto)/2.0-disparador.alto,
             0.0);
    glRotatef(giro_disparador,0,1,0);
    disparador.draw(modo, r, g, b, grosor);
  glPopMatrix();
  glPushMatrix();
    glTranslatef(0.0,0.0,base.fondo-objetivo.alto);
    glRotatef(90,1,0,0);
    glRotatef(giro_objetivo,0,1,0);
    objetivo.draw(modo, r, g, b, grosor);
  glPopMatrix();
};
```

58. Animación modelo cámara.

```
// variables para la animación automática
float giro1 = 0, giro2 = 0, giro3 = 0, giro4 = 0, giro5 = 0;
int pulsar = 0;
int paso = 0;

//*************************************************************************
// Función de animación automática
//*************************************************************************
void animacion(){
  switch(paso){
    // abrir la tapa trasera (para introducir carrete)
    case 0: camara.giro_tapa-=giro3;
      if (camara.giro_tapa < -135){
        paso = 1;
      }break;

    // cerrar la tapa trasera
```

```
    case 1: camara.giro_tapa+=giro3;
      if (camara.giro_tapa >= 0){
        paso = 2;
      }break;
    // regular el objetivo
    case 2: camara.giro_objetivo+=giro1;
      if (camara.giro_objetivo > 160){
        paso = 3;
      }break;
    // capturar la imagen con el disparador
    case 3: camara.giro_disparador-=giro2;
      if (camara.giro_disparador < -120){
        paso = 4;
      }break;
    // levantar pestaña del carrete
    case 4: camara.giro_palanca+=giro4;
      if (camara.giro_palanca > 135){
        paso = 5;
      }break;
    // girar carrete para recogerlo
    case 5: camara.giro_cartucho+=giro5;
      if (camara.giro_cartucho > 180){
        paso = 0;
        camara.giro_palanca = 0.0;
        camara.giro_cartucho = 0.0;
        camara.giro_tapa = 0.0;
        camara.giro_objetivo = 0.0;
        camara.giro_disparador = 0.0;
      }break;
  }
  glutPostRedisplay();
}

void normal_key(unsigned char Tecla1,int x,int y){
  switch (toupper(Tecla1)){
    case 'Q':exit(0);
    case '1':modo=POINTS;break;
    case '2':modo=EDGES;break;
    case '3':modo=SOLID;break;
    case '4':modo=SOLID_COLORS;break;
    case 'P':t_objeto=PIRAMIDE;break;
    case 'C':t_objeto=CUBO;break;
    // resto de teclas que utilizamos...
    case 'S':
      if(pulsar == 0){
        giro1 = 1.0;
        giro2 = 1.0;
        giro3 = 1.0;
        giro4 = 1.0;
        giro5 = 1.0;
        pulsar = 1;
      }
```

```
      else{
        giro1 = 0.0;
        giro2 = 0.0;
        giro3 = 0.0;
        giro4 = 1.0;
        giro5 = 1.0;
        pulsar = 0;
      }break;
  }
  glutPostRedisplay();
}

//************************************************************************
// Función de animación manual
//************************************************************************
void special_key(int Tecla1,int x,int y){
  switch (Tecla1){
    case GLUT_KEY_LEFT:Observer_angle_y--;break;
    case GLUT_KEY_RIGHT:Observer_angle_y++;break;
    case GLUT_KEY_UP:Observer_angle_x--;break;
    case GLUT_KEY_DOWN:Observer_angle_x++;break;
    case GLUT_KEY_PAGE_UP:Observer_distance*=1.2;break;
    case GLUT_KEY_PAGE_DOWN:Observer_distance/=1.2;break;

    // MOVIMIENTOS CAMARA
    case GLUT_KEY_F1: camara.giro_palanca+=1;
      if (camara.giro_palanca > camara.giro_palanca_max)
        camara.giro_palanca = camara.giro_palanca_max;break;
    case GLUT_KEY_F2: camara.giro_palanca-=1;
      if (camara.giro_palanca < camara.giro_palanca_min)
        camara.giro_palanca = camara.giro_palanca_min;break;
    case GLUT_KEY_F3:camara.giro_tapa-=1;
      if (camara.giro_tapa < camara.giro_tapa_max)
        camara.giro_tapa = camara.giro_tapa_max;break;
    case GLUT_KEY_F4:camara.giro_tapa+=1;
      if (camara.giro_tapa > camara.giro_tapa_min)
        camara.giro_tapa = camara.giro_tapa_min;break;
    case GLUT_KEY_F5: camara.giro_disparador+=5;break;
    case GLUT_KEY_F6: camara.giro_disparador-=5;break;
    case GLUT_KEY_F7: camara.giro_cartucho+=5;break;
    case GLUT_KEY_F8: camara.giro_cartucho-=5;break;
    case GLUT_KEY_F9: camara.giro_objetivo+=5;break;
    case GLUT_KEY_F10: camara.giro_objetivo-=5;break;
  }
  glutPostRedisplay();
}

En el main llamamos a:
  // animación automática
  glutIdleFunc(animacion);
```

59. Dibujar con texturas.

```
// IDENTIFICADOR DE TEXTURA
GLuint textura_id;
GLuint textura_id2;

// OBJETO A DIBUJAR
GLfloat vertices[] = {-3.0, 0.0, 3.0,  3.0, 0.0, 3.0,   0.0, 5.0, 0.0,
                       3.0, 0.0, 3.0,  3.0, 0.0, -3.0,  0.0, 5.0, 0.0,
                       3.0, 0.0, -3.0, -3.0, 0.0, -3.0, 0.0, 5.0, 0.0,
                      -3.0, 0.0, -3.0, -3.0, 0.0, 3.0,  0.0, 5.0, 0.0};

GLfloat vertices2[] = {2.0, 2.0, 1.0,  2.0, 0.0, 1.0,  4.0, 0.0, 1.0,
                       4.0, 2.0, 1.0,  4.0, 2.0, 1.0,  4.0, 0.0, 1.0,
                       4.0, 0.0, -1.0, 4.0, 2.0, -1.0, 4.0, 2.0, -1.0,
                       4.0, 0.0, -1.0, 2.0, 0.0, -1.0, 2.0, 2.0, -1.0,
                       2.0, 2.0, -1.0, 2.0, 0.0, -1.0, 2.0, 0.0, 1.0,
                       2.0, 2.0, 1.0,  2.0, 2.0, -1.0, 2.0, 2.0, 1.0,
                       4.0, 2.0, 1.0,  4.0, 2.0, -1.0, 2.0, 0.0, -1.0,
                       2.0, 0.0, 1.0,  4.0, 0.0, 1.0,  4.0, 0.0, -1.0};

GLfloat texVertices[] = {0.0,1.0,      0.5,1.0,       0.25,0.5,
                         0.5,1.0,      1.0,1.0,       0.75,0.5,
                         0.0,0.5,      0.5,0.5,       0.25,0.0,
                         0.5,0.5,      1.0,0.5,       0.75,0.0};

GLfloat texVertices2[] = {// delante
                          0.0, 0.0,  0.0, 0.5,  0.5, 0.5,  0.5, 0.0,
                          // derecha
                          0.5, 0.0,  0.5, 0.5,  1.0, 0.5,  1.0, 0.0,
                          // detrás
                          1.0, 1.0,  1.0, 0.5,  0.5, 0.5,  0.5, 1.0,
                          // izquierda
                          0.5, 0.0,  0.5, 0.5,  1.0, 0.5,  1.0, 0.0,
                          // superior e inferior
                          0.0, 0.5,  0.0, 1.0,  0.5, 1.0,  0.5, 0.5,
                          0.0, 0.5,  0.0, 1.0,  0.5, 1.0,  0.5, 0.5};

float latitud=0.0,longitud=0.0,radio=24;
int contador = 0;

void Init(int argc, char **argv){
   glutInit(&argc, argv);
   glutInitWindowSize(700,700);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutCreateWindow("Práctica 4 IG");
   glEnable(GL_DEPTH_TEST);
   //glEnable(GL_COLOR_MATERIAL);
   glMatrixMode(GL_PROJECTION);
   gluPerspective( 40.0, 700/700.0f, 0.1, 150);
   prepara_textura("./arbol_navidad.jpg", &textura_id);
   prepara_textura("./caja.jpg", &textura_id2);
}
```

```
...

void OnDraw(void){
...
   dibuja();

   //no delete this line
   glutSwapBuffers();
}
...

int main(int argc, char** argv) {
   Init(argc, argv);
   //Enter the callbacks
   glutDisplayFunc(OnDraw);
   glutSpecialFunc(SpecialKeys);

   glutMainLoop(); // begin the loop

   // LIBERA LA TEXTURA
   libera_textura(&textura_id);
   libera_textura(&textura_id2);
   return 0;
}

// ------------------------------------------------------------------------
void prepara_textura (char *file, GLuint *tex_id )
{
   vector<unsigned char> data;
   CImg<unsigned char> image;
   image.load(file);

   // empaquetamos bien los datos
   for (long y = 0; y < image.height(); y ++)
      for (long x = 0; x < image.width(); x ++){
       unsigned char *r = image.data(x, y, 0, 0);
       unsigned char *g = image.data(x, y, 0, 1);
       unsigned char *b = image.data(x, y, 0, 2);
       data.push_back(*r);
       data.push_back(*g);
       data.push_back(*b);
      }
   glGenTextures(1, tex_id);
   glBindTexture(GL_TEXTURE_2D, *tex_id);
   contador++;
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

   // TRASFIERE LOS DATOS A GPU
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image.width(), image.height(),
             0, GL_RGB, GL_UNSIGNED_BYTE, &data[0]);
}
```

```
void dibuja (void){
   glEnable(GL_TEXTURE_2D);
   glEnableClientState(GL_VERTEX_ARRAY);
   glEnableClientState (GL_TEXTURE_COORD_ARRAY_EXT);

   // árbol navidad
   glBindTexture(GL_TEXTURE_2D, textura_id);

   glVertexPointer(3, GL_FLOAT, 0, vertices);
   glTexCoordPointer(2, GL_FLOAT, 0, texVertices);
   glDrawArrays(GL_TRIANGLES, 0, 12);

   glPushMatrix();
   glTranslatef(0,2.5,0);
   glScalef(0.75,0.75,0.75);
   glDrawArrays(GL_TRIANGLES, 0, 12);
   glPopMatrix();

   glPushMatrix();
   glTranslatef(0,4.5,0);
   glScalef(0.5,0.5,0.5);
   glDrawArrays(GL_TRIANGLES, 0, 12);
   glPopMatrix();

   glPushMatrix();
   glTranslatef(0,6.25,0);
   glScalef(0.25,0.25,0.25);
   glDrawArrays(GL_TRIANGLES, 0, 12);
   glPopMatrix();

   // regalo
   glBindTexture(GL_TEXTURE_2D, textura_id2);

   glVertexPointer(3, GL_FLOAT, 0, vertices2);
   glTexCoordPointer(2, GL_FLOAT, 0, texVertices2);

   glPushMatrix();
   glTranslated(2.0, 0, 0);
   glDrawArrays(GL_QUADS, 0, 24);
   glPopMatrix();

   glDisableClientState(GL_VERTEX_ARRAY);
   glBindTexture(GL_TEXTURE_2D, 0);
   glDisable(GL_TEXTURE_2D);
}

void libera_textura (GLuint *tex_id)
{
   for (int i = 1; i <= contador; i++){
      glDeleteTextures(i, tex_id);
   }
}
```

60. Dibujar en modo sólido con colores diferentes para cada vértice.

```
void _triangulos3D::draw_solido_colores_vertices( ){
  int i;
  glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
  glShadeModel(GL_SMOOTH);
  glBegin(GL_TRIANGLES);
  for (i=0;i<vertice.size();i++){
    glColor3f(colores_vertices[caras[i]._0].r,
              colores_vertices[caras[i]._0].g,
              colores_vertices[caras[i]._0].b);
    glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
    glColor3f(colores_vertices[caras[i]._1].r,
              colores_vertices[caras[i]._1].g,
              colores_vertices[caras[i]._1].b);
    glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
    glColor3f(colores_vertices[caras[i]._2].r,
              colores_vertices[caras[i]._2].g,
              colores_vertices[caras[i]._2].b);
    glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
  }
  glEnd();
  glShadeModel(GL_FLAT);
}
```

61. Dibujar en modo sólido con textura.

```
void _triangulos3D::draw_solido_textura(){
  int i;
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, id_tex);
  glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
  glBegin(GL_TRIANGLES);
  for (i=0;i<caras.size();i++){
    glTexCoord2f(texturas_vertices[caras[i]._0].x,
                 texturas_vertices[caras[i]._0].y);
    glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
    glTexCoord2f(texturas_vertices[caras[i]._1].x,
                 texturas_vertices[caras[i]._1].y);
    glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
    glTexCoord2f(texturas_vertices[caras[i]._2].x,
                 texturas_vertices[caras[i]._2].y);
    glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
  }
  glEnd();
  glDisable(GL_TEXTURE_2D);
}
```

62. Dibujar en modo sólido con iluminación y suavizado.

```
void _triangulos3D::draw_solido_phong_flat(){
  int i;
  glShadeModel(GL_FLAT);
  glEnable(GL_NORMALIZE);
  glEnable(GL_LIGHTING);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, (GLfloat *) &ambiente);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, (GLfloat *) &difuso);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, (GLfloat *) &especular);
  glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, brillo);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glBegin(GL_TRIANGLES);
  for (i=0;i<caras.size();i++){
    glNormal3f(normales_caras[i].x,
               normales_caras[i].y,
               normales_caras[i].z);
    glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
    glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
    glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
  }
  glEnd();
  glDisable(GL_LIGHTING);
}

void _triangulos3D::draw_solido_phong_gouraud(){
  int i;
  glShadeModel(GL_SMOOTH);
  glEnable(GL_NORMALIZE);
  glEnable(GL_LIGHTING);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, (GLfloat *) &ambiente);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, (GLfloat *) &difuso);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, (GLfloat *) &especular);
  glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, brillo);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glBegin(GL_TRIANGLES);
  for (i=0;i<caras.size();i++){
    glNormal3f(normales_vertices[caras[i]._0].x,
               normales_vertices[caras[i]._0].y,
               normales_vertices[caras[i]._0].z);
    glVertex3fv((GLfloat *) &vertices[caras[i]._0]);
    glNormal3f(normales_vertices[caras[i]._1].x,
               normales_vertices[caras[i]._1].y,
               normales_vertices[caras[i]._1].z);
    glVertex3fv((GLfloat *) &vertices[caras[i]._1]);
    glNormal3f(normales_vertices[caras[i]._2].x,
               normales_vertices[caras[i]._2].y,
               normales_vertices[caras[i]._2].z);
    glVertex3fv((GLfloat *) &vertices[caras[i]._2]);
  }
  glEnd();
  glDisable(GL_LIGHTING);
}
```

63. Calcular normales caras.

```
void _triangulos3D::calcular_normales_caras(){
  int i, n_c;
  _vertex3f va, vb;
  float modulo;
  n_c=caras.size();
  normales_caras.resize(n_c);
  for (i=0;i<n_c;i++){
    va=vertices[caras[i]._1]-vertices[caras[i]._0];
    vb=vertices[caras[i]._2]-vertices[caras[i]._1];
    normales_caras[i].x=va.y*vb.z-va.z*vb.y;
    normales_caras[i].y=va.z*vb.x-va.x*vb.z;
    normales_caras[i].z=va.x*vb.y-va.y*vb.x;
    modulo=sqrt(normales_caras[i].x*normales_caras[i].x+
                normales_caras[i].y*normales_caras[i].y+
                normales_caras[i].z*normales_caras[i].z);
    normales_caras[i].x/=modulo;
    normales_caras[i].y/=modulo;
    normales_caras[i].z/=modulo;
  }
  normales_caras_calculadas = 1;
}
```

64. Calcular normales vértices.

```
void _triangulos3D::calcular_normales_vertices(){
  int i, n_c, n_v;
  float modulo;
  if (normales_caras_calculadas == 0) calcular_normales_caras();
  n_v=vertices.size();
  normales_vertices.resize(n_v);
  n_c=caras.size();
  for (i=0;i<n_v;i++)
    normales_vertices[i]=_vertex3f(0.0,0.0,0.0);
  for (i=0;i<n_c;i++){
    normales_vertices[caras[i]._0]+=normales_caras[i];
    normales_vertices[caras[i]._1]+=normales_caras[i];
    normales_vertices[caras[i]._2]+=normales_caras[i];
  }
  for (i=0;i<n_v;i++){
    modulo=sqrt(normales_vertices[i].x*normales_vertices[i].x+
                normales_vertices[i].y*normales_vertices[i].y+
                normales_vertices[i].z*normales_vertices[i].z);
                normales_vertices[i].x/=modulo;
                normales_vertices[i].y/=modulo;
                normales_vertices[i].z/=modulo;
  }
}
```

65. Dibujar en difuso.

```
void _triangulos3D::colors_diffuse_flat (float kr, float kg, float kb,
                                         float lpx, float lpy, float lpz){
  int i, n_c;
  float modulo, escalar;
  _vertex3f l;
  n_c=caras.size();
  colores_caras.resize(n_c);
  for (i=0;i<n_c;i++){
    l.x=lpx-vertices[caras[i]._0].x;
    l.y=lpy-vertices[caras[i]._0].y;
    l.z=lpz-vertices[caras[i]._0].z;
    modulo=sqrt(l.x*l.x+l.y*l.y+l.z*l.z);
    l.x/=modulo;
    l.y/=modulo;
    l.z/=modulo;
    escalar=l.x*normales_caras[i].x+
            l.y*normales_caras[i].y+
            l.z*normales_caras[i].z;
    if (escalar>0.0){
      colores_caras[i].r=kr*escalar;
      colores_caras[i].g=kg*escalar;
      colores_caras[i].b=kb*escalar;
    }
    else{
      colores_caras[i].r=0.0;
      colores_caras[i].g=0.0;
      colores_caras[i].b=0.0;
    }
  }
}

void _triangulos3D::colors_diffuse_gouraud(float kr, float kg, float kb,
                                           float lpx, float lpy, float lpz){
  int i, n_v;
  float modulo, escalar;
  _vertex3f l;
  n_v=vertices.size();
  colores_vertices.resize(n_v);
  for (i=0;i<n_v;i++){
    l.x=lpx-vertices[i].x;
    l.y=lpy-vertices[i].y;
    l.z=lpz-vertices[i].z;
    modulo=sqrt(l.x*l.x+l.y*l.y+l.z*l.z);
    l.x/=modulo;
    l.y/=modulo;
    l.z/=modulo;
    escalar=l.x*normales_vertices[i].x+
            l.y*normales_vertices[i].y+
            l.z*normales_vertices[i].z;
    if (escalar>0.0){
      colores_vertices[i].r=kr*escalar;
```

```
      colores_vertices[i].g=kg*escalar;
      colores_vertices[i].b=kb*escalar;
    }else{
      colores_vertices[i].r=0.0;
      colores_vertices[i].g=0.0;
      colores_vertices[i].b=0.0;
    }
  }
}
```

66. Luces.

```
int alfa = 90;
int inc_alfa = 10;
bool onoff = true;
void luces(){
    GLfloat luz_ambiente[]={0.1,0.1,0.1,1.0},
            luz_difusa[]={1.0,1.0,1.0,1.0},
            luz_especular[]={1.0,1.0,1.0,1.0},
            luz_posicion[]={20.0,20.0,20.0};
    GLfloat luz_ambiente2[]={0.0,0.05,0.0,1.0},
            luz_difusa2[]={0.0,1.0,0.0,1.0},
            luz_especular2[]={0.0,1.0,0.0,1.0},
            luz_posicion2[]={0.0,10.0,20.0,1.0};
    glLightfv (GL_LIGHT1, GL_AMBIENT, luz_ambiente);
    glLightfv (GL_LIGHT1, GL_DIFFUSE, luz_difusa);
    glLightfv (GL_LIGHT1, GL_SPECULAR, luz_especular);
    glLightfv (GL_LIGHT1, GL_POSITION, luz_posicion);
    glLightfv (GL_LIGHT2, GL_AMBIENT, luz_ambiente2);
    glLightfv (GL_LIGHT2, GL_DIFFUSE, luz_difusa2);
    glLightfv (GL_LIGHT2, GL_SPECULAR, luz_especular2);
    glPushMatrix();
      glRotatef(alfa,0,0,1);
      glLightfv (GL_LIGHT2, GL_POSITION, luz_posicion2);
    glPopMatrix();
    glDisable (GL_LIGHT0);
    glEnable (GL_LIGHT1);
    if(onoff){
        glEnable (GL_LIGHT2);
    }else{
        glDisable (GL_LIGHT2);
    }
}

void normal_key(unsigned char Tecla1,int x,int y){
  switch (toupper(Tecla1)){ ...
    case 'I':onoff = !onoff;break;
    case 'Y':alfa += inc_alfa;break;
    case 'U':alfa -= inc_alfa;break;
  }
  glutPostRedisplay();
}
```

67. Movimiento de ratón y de cámara.

```
int estadoRaton, xc, yc;
int cambio = 0;
float izquierda = -5.0,
      derecha   =  5.0,
      abajo     = -5.0,
      arriba    =  5.0,
      factor    =  1.0;

void cambio_camara(){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho (izquierda*factor, derecha*factor,
           abajo*factor, arriba*factor, -100, 100);
  glRotatef(90,1,0,0);  // vista desde arriba
  // glRotatef(-90,1,0,0);  // vista desde abajo
  // glRotatef(90,0,1,0);  // vista de perfil
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

void draw(void){
  glDrawBuffer(GL_FRONT);
  clean_window();
  change_projection();

  if(cambio == 0){
    change_observer();
  }
  else{
    cambio_camara();
  }
  draw_axis();
  draw_objects();
  luces();

  if (t_objeto==CAMARA){
    glDrawBuffer(GL_BACK);
    clean_window();
    change_projection();
    if(cambio == 0){
      change_observer();
    }
    else{
      cambio_camara();
    }
   camara.seleccion();
  }
  glFlush();
}
```


```
void normal_key(unsigned char Tecla1,int x,int y){
  switch (toupper(Tecla1)){
    ...
    case ';':cambio=0;break;
    case ',':cambio=1;break;
  }
  glutPostRedisplay();
}

//************************************************************************
// funciones para la selección por color
//************************************************************************
void procesar_color(unsigned char color[3]){
  int i;
  for (i=0;i<camara.piezas;i++){
    if (color[0]==camara.color_select[i].r &&
        color[1]==camara.color_select[i].g &&
        color[2]==camara.color_select[i].r){
      if (camara.activo[i]==0){
        camara.activo[i]=1;
      }
      else{
      camara.activo[i]=0;
      }
      glutPostRedisplay();
    }
  }
}

void pick_color(int x, int y)   {
  GLint viewport[4];
  unsigned char pixel[3];
  glGetIntegerv(GL_VIEWPORT, viewport);
  glReadBuffer(GL_BACK);
  glReadPixels(x,viewport[3]-y,1,1,GL_RGB,GL_UNSIGNED_BYTE,(GLubyte *)
&pixel[0]);
  printf(" valor x %d, valor y %d, color %d, %d, %d \n",
         x,y,pixel[0],pixel[1],pixel[2]);
  procesar_color(pixel);
}

//************************************************************************
// funciones para manejo de eventos del ratón
//************************************************************************
void clickRaton(int boton, int estado, int x, int y){
  if(boton==GLUT_RIGHT_BUTTON){
    if(estado==GLUT_DOWN){
      estadoRaton=1;
      xc=x;
      yc=y;
    }else estadoRaton=0;
  }
```

```
  if(boton==GLUT_LEFT_BUTTON){
    if(estado==GLUT_DOWN){
      estadoRaton=2;
      xc=x;
      yc=y;
      pick_color(xc, yc);
    }
  }
  if(boton==3){
    Observer_distance/=1.2;
  }
  if(boton==4){
    Observer_distance*=1.2;
  }
}

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();
  }
}

En el main añadimos los siguientes eventos ratón:
glutMouseFunc(clickRaton);
glutMotionFunc(RatonMovido);
```