Preguntas-Resueltas.pdf

Cristobal02

Informática Gráfica

3º Grado en Ingeniería Informática

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

Preguntas teóricas 2.0

(1) ¿Qué es la transformación de vista? ¿Qué parámetros están implicados? ¿Qué matriz de OpenGL la almacena?

Es una transformación que permite cambiar de sistema de coordenadas, desde el sistema de coordenadas mundiales al sistema de coordenadas del observador.

Esta transformación permite simular el posicionamiento de la cámara en cualquier posición y orientación

Los parámetros que definen la transformación de vista son:

VRP: Posición donde está la cámara y origen de sistema de coordenadas del observador. Es un punto dado en el sistema de coordenadas mundiales.

VPN: Punto hacia donde mira la cámara. Es el eje z del sistema de coordenadas del observador y un vector dado en el sistema de coordenadas mundiales.

VUP: Indica la orientación hacia arriba. Es un vector dado en el SCM.

GL_MODELVIEW es la matriz que almacena la transformación de vista.

Cree un ejemplo incluyendo las llamadas de OpenGL

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0,-10);
glRotatef(37.1,0.0);
glRotatef(45,0.1,0);
```

(2) Enumere y explique las propiedades de la transformación de perspectiva.

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

(3) Queremos acercarnos a un objeto para ver sus detalles. Explicar cómo se podría hacer en una proyección de perspectiva. Ventajas e inconvenientes.

La solución más sencilla consiste en acercarse al objeto. El problema está en 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

④ Dado un cubo definido por sus vertices, vector<_vertex3f> vertices y triangulos, vector<_vertex3ui> Triangles, indique lo siguiente si queremos mostrar el cubo texturizado:

1) La estructura de datos para guardar las coordenadas de textura.

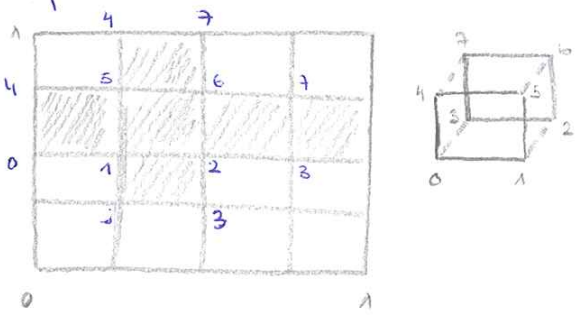
Usaremos un vector de dos coordenadas en punto flotante: vector<_vertex2f> TexCoordinates;

2) La función que dibujaría el objeto texturizado.

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

3) Un ejemplo de coordenadas de textura para cada vértice si la textura se aplica a todas las caras sin repetirla.

Suponemos:




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

⑤ Explique lo más detalladamente posible, las distintas formas de hacer un pick en OpenGL.

1. Usando el buffer de selección (Hasta OpenGL v3): Devuelve los identificadores de los objetos incluidos en un subvolumen de visión centrado en una posición del viewport.
2. Intersección rayo-escena: Se dibuja la escena con la iluminación desactivada usando como colores los identificadores de los objetos en un frame-buffer no visible. Se lee el pixel que corresponde a la posición del cursor y se decodifica el color para obtener el identificador.
3. Codificando el id del objeto como color y leyendo el frame buffer: Se crea un objeto de tipo frame buffer y se hace rasterización con ese objeto como imagen de destino, o se utiliza un buffer trasero donde se visualizan las primitivas y un buffer delantero, que es el que se visualiza en pantalla.

```
void color_pick(){
    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]);
}
```

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Quiero el cash

Consulta condiciones **aquí**

do your thing

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

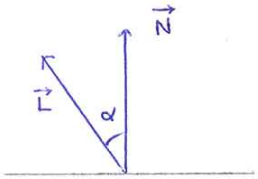
coordenadas 3D de un modelo hasta que tenemos una imagen. Indique el propósito de cada etapa y el resultado obtenido tras cada una de las transformaciones.

1. Transformación del modelo: Situarlo en escena, cambiarlo de tamaño y crear modelos compuestos a partir de otros más simples.
2. Transformación de vista: Poner al observador en la posición deseada *
3. Transformación de perspectiva: Calcular para cada píxel su color, teniendo en cuenta la primitiva que se muestra, el color, material, texturas, luces, etc. *
4. Rasterización: Pasar de 3D a 2D *
5. Transformación del dispositivo: Adaptar la imagen 2D a la zona de dibujado.

⑦ Explique cómo funciona la iluminación en OpenGL

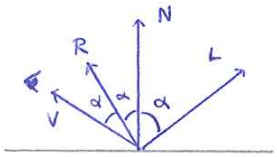
La iluminación en OpenGL tiene 3 componentes: difusa, ambiental y especular.

La componente difusa modela la reflexión de la luz en objetos mates y difusos. Depende del ángulo entre el vector de la fuente de luz y la normal del objeto. No depende de la dirección desde la que miramos:



$$R_D = I_D K_d \cdot \cos\alpha = I_D K_d \cdot (\vec{N}\,\vec{L})$$

La componente especular modela la reflexión de los objetos brillantes. Depende de la posición y orientación de la luz y de la dirección en la que miramos:



$$R_E = I_E \cdot K_E \cdot \cos(\alpha)^n$$

n es la componente que modela el brillo del objeto

La componente ambiental es constante y su función es simular la iluminación de fondo y evitar que superficies y objetos que no estén directamente iluminados no se vean negros. No depende de la posición del observador, ni de la normal de las superficies.

$$R_{AMB} = I_{AMB} \cdot K_{AMB} \cdot K_{MAMB}$$

K_{MAMB} es la constante ambiental del modelo y K_{AMB} es la constante ambiental del material.

$$R = R_D + R_E + R_A$$

(8) ¿Qué es el Z-Buffer? ¿Cómo funciona?

Es una técnica utilizada para saber qué píxeles deben ser pintados.
El algoritmo consiste en medir la distancia de cada píxel a la cámara y comprobar, si la distancia actual es menor que la guardada anteriormente en el buffer este se actualiza con la nueva distancia y se dibuja el objeto en la pantalla. Si la distancia es mayor, esto quiere decir que otro objeto está tapándolo, por lo que no se pintará.
Utilizar el Z-Buffer aumenta considerablemente la complejidad y el uso de recursos.
Para utilizarlo en OpenGL hay que llamar a glEnable(GL_DEPTH_TEST), y para limpiarlo: glClear(GL_DEPTH_TEST | GL_COLOR_BUFFER_BIT);

(9) ¿Cómo se calculan las normales de los vértices? ¿Y de las caras?

Para calcular la normal de una cara, suponiendo que esa cara está formada por tres vértices; P0, P1 y P2, lo que tenemos que hacer es obtener dos vectores, $\vec{A} = P_1 - P_0$ y $\vec{B} = P_2 - P_0$ y hacer el producto vectorial entre ambos; $\vec{N} = \vec{A} \times \vec{B}$, sin embargo, este producto vectorial debe estar normalizado; $\vec{N} = \frac{\vec{N}}{|\vec{N}|}$

Para calcular la normal de un vértice, hacen falta las normales de todas las caras que comparten ese vértice, de modo que se sumen todas esas normales y se divide entre el número de caras que comparten ese vértice; $\vec{N}_v = \frac{\sum_{i=0}^{n} \vec{N}_c}{n}$. Al igual que en el caso de las caras, el vector normal al vértice debe estar normalizado $\Rightarrow \vec{N}_v = \frac{\vec{N}_v}{|\vec{N}_v|}$

```
void CalcularNormalesCaras(){
  vector<_vertex3f> normalesCaras; // Defino el vector donde irán las normales
  for(int i=0; i<Triangles.size(); i++){
    _vertex3f P0 = Vertices[Triangles[i]_0];
    _vertex3f P1 = Vertices[Triangles[i]_1];
    _vertex3f P2 = Vertices[Triangles[i]_2];

    _vertex3f A = P1-P0;
    _vertex3f B = P2-P0;
    _vertex3f P = ProductoVectorial(A,B);
    _vertex3f N;

    if(Modulo(P) != 0){
      N = Normalizar(P); // Normalización del vector
      normalesCaras.pushback(N);
    }
  }
}
```

(10) Explique los tipos de sombreado.

→ Sombreado plano: Se calcula una vez por cada cara que forme el modelo, asigna el mismo color a todos los píxeles de la cara. Es muy eficiente si el objeto es sencillo. Apropiado para objetos poliédricos.

→ Sombreado de vértices (suave): Se calcula una vez por cada vértice y cada color obtenido se utiliza para interpolar los colores de los píxeles de cada polígono o cara. Eficiencia similar al plano pero mucho más realista

→ Sombreado de píxeles: La normal de cada píxel se calcula interpolando las normales de los vértices. Computacionalmente más costosa, pero produce resultados más realistas.

(11) Describa las formas de interacción de la luz como partícula según la superficie de los objetos y el proceso que sigue para realizar un suavizado de Gouraud.

Si la luz interactúa con una superficie pulida opaca, es una reflexión especular. Si la luz interactúa con una superficie rugosa opaca, es una reflexión difusa.

El suavizado de Gouraud se realiza mediante las normales de los vértices para calcular la intensidad de la luz. Luego se interpolan esa intensidad para encontrar los valores en los píxeles en los que se proyecta el polígono en pantalla.

(12) 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.

1. Definir las normales de cada cara: Es necesario definir un vector perpendicular a cada cara de nuestro modelo que apunte hacia afuera.
2. Definir las normales de cada vértice: Para obtener una iluminación más realista es necesario definir un vector perpendicular a cada vértice de cada cara del modelo.
3. Situar las luces: Situamos las luces en escena. OpenGL maneja dos tipos de luces:
   → Luz ambiental: Ilumina toda la escena por igual ya que esta no proviene de una dirección predeterminada
   → Luz difusa: Viene de una dirección específica, y depende de su ángulo de incidencia para iluminar una superficie en mayor o menor medida.
4. Definir materiales: Es la manera en que la luz se refleja sobre nuestros objetos

(13) 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 ~~paralela~~ en perspectiva
→ glOrtho(left, right, bottom, top, near, far): Describe una matriz de perspectiva que produce una proyección paralela.

(14) Explique el sistema de colores que usa OpenGL.

El sistema de color que usa OpenGL es el sistema RGB que tiene tres componentes: Rojo, Verde y azul, de modo que a cada componente OpenGL le asigna un valor entre 0 y 1 siendo 1 la máxima cantidad de color y 0 la ausencia de ese color. Podemos mezclar estos tres colores para obtener una gama completa de colores y generalmente se suelen almacenar en un vector de tres componentes en punto flotante: vector<_vertex3f> colores.

15) Describa brevemente para que sirven en un programa OpenGL/glut cada una de estas cuatro funciones

→ glutDisplayFunc (function): La función "function" se llamará cada vez que se dibuje la ventana

→ glutReshapeFunc (function): La función "function" se llamará cada vez que se redimensione la ventana.

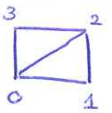
→ glutKeyboardFunc (function): Función de control de eventos a través del teclado

→ glutSpecialFunc (function): Función de control de eventos a través del teclado cuando se pulsa una tecla.

16) Dado un objeto descrito con vértices y caras (vector<_vertex3f> Vertices, vector<_vertex3ui> Triangles), escribir en pseudocódigo el programa que comprueba si el objeto es abierto o cerrado (abierto = tiene aristas con una sola cara)

La clave está en saber como se define una cara con _vertex3ui ya que para dibujarla se necesitan dos componentes del vector por ejemplo, para unir los vértices P_0, P_1, P_2 y P_3 tendríamos:

```
Triangles[0] = _vertex3ui(0,1,2);
Triangles[1] = _vertex3ui(3,0,2);
```



De manera que lo que tenemos que comprobar es si se repite alguna unión. En ese caso, para todos los triángulos, podremos decir que el objeto es cerrado.

```
bool Consistente(){
    bool es_consistente = true;
    Para cada triángulo del objeto && es_consistente = true{
        Comprobamos si ese triángulo tiene alguna arista repetida
        Si la tiene continuamos
        Si no la tiene then es_consistente = false (paramos de buscar)
    }
    return es_consistente;
}
```

17) Sea la siguiente representación de una malla de triángulos, describa cómo se puede determinar si la orientación de las normales de las caras es consistente (están orientadas hacia el exterior del objeto)

```
class Puntos3D{
    vector<_vertex3f> vertices;
};
class _triangulos3D: public _puntos3D{
    _triangulos3D();
    void CalcularNormales();
    vector<_vertex3ui> caras;
    vector<_vertex3f> normales_caras;
};
```

Para determinar la orientación de las normales podemos hacer lo siguiente:

1. Utilizamos la función CalcularNormales() para calcular todas las normales de la cara del objeto.
2. Para cada cara, obtenemos el vector que une el centro de la cara con el punto que indica la posición de la cámara.
3. Calculamos el producto escalar entre la normal de esa cara y el vector obtenido en el paso 2, de manera que si el resultado es positivo, la cara está orientada hacia el exterior, en caso de que sea negativo, esa cara estaría orientada hacia el interior del objeto.
4. Con que una cara esté orientada hacia el interior, podremos decir que el objeto tiene una orientación de las caras no consistente.

(20) Explica cómo se obtiene un volumen de vista en perspectiva, nombrando los parámetros necesarios. Para OpenGL indica las distintas formas de realizar un zoom para una cámara en perspectiva (glFrustum()) y además, cómo orientar este tipo de cámara para tener un plano de proyección picado oblicuo a 45 grados respecto al objeto enfocado.

Un volumen de vista en perspectiva se define como una pirámide truncada donde el vértice superior indica el lugar donde está la cámara y la base representa el plano de proyección.

Los parámetros necesarios son los siguientes:

1. Punto en el plano de proyección (PO) dado en sistema de coordenadas mundiales
2. Punto de mira o punto hacia donde apunta la cámara (PM) dado en sistema de coordenadas mundiales
3. Vector de inclinación de la cámara (VI)
4. Vector normal al plano de proyección (PM-PO) dado en sistema de coordenadas mundiales
5. Centro de proyección (CP) dado en sistema de coordenadas mundiales. Define el origen del sistema de coordenadas del observador.
6. Planos delantero y trasero que representan la distancia sobre el eje Z del sistema de coordenadas del observador
7. Ventana de proyección (Wxmin, Wxmax, Wymin, Wymax) en coordenadas mundiales 2D.

Las distintas formas de hacer zoom para una cámara en perspectiva son:

1. Modificar el plano delantero (parámetro 'near' de glFrustum) cuidando que no se recorte al objeto
2. Modificando el tamaño de la ventana de proyección parámetros ('left', 'right', 'bottom' y 'top') también cuidando que no se recorte al objeto.

Para orientar la cámara, para tener un plano de proyección perspectiva oblicuo a 45 grados, podemos hacer lo siguiente:

```
GLfloat right = Back_plane * tan( rad(45.0)/2.0 );
GLfloat left = -right;
GLfloat bottom = -r;
GLfloat top = right;
glFrustum( left, right, bottom, top, Front_Plane, Back_Plane);
```

(19) La imagen siguiente corresponde a la visualización de un poliedro regular de radio 1, centrado en el origen que aproxima una esfera de color azul medio. Indica:

a) ¿Qué fuentes de luz y qué propiedades de material se han usado para generarla?

Se han usado dos luces blancas direccionales tal que si la luz 1 tiene coordenadas (x,y,z), la luz 2 tiene coordenadas (-x,y,z).

El material tiene una componente difusa de azul medio, y una componente especular de color blanco. Además, podemos ver que no tiene o tiene muy poca componente ambiental de color azul.

b) ¿Qué cambios se deben hacer en el código para que la visualización simule mejor una esfera, indicando cómo se calcularían las normales?

Hay que ~~cambiar~~ poner glShadeModel(GL_SMOOTH);

Y calcular las normales de los vértices, donde cada normal se obtiene calculando el sumatorio de las normales de las caras que comparten dicho vértice y dividiendo entre el número de caras compartidas:

$$\vec{N_v} = \frac{\sum_{i=0}^{n} \vec{N_c}}{n}$$

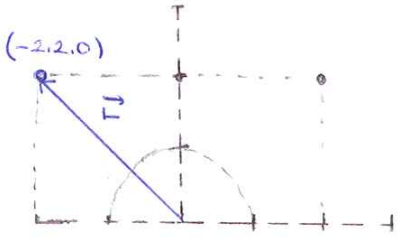
```
for(int i=0; i< Triangles.size(); i++){
  glNormal3f( n_v[Triangles[i]._0].x,
              n_v[Triangles[i]._0].y,
              n_v[Triangles[i]._0].z);
  glVertex3f((GLfloat*) &Vertices[
              Triangles[i]._0]);
  :
}
```

(2D) Supongamos que tenemos una esfera difusa y especular de radio 1 unidad centrada en el origen, una fuente de luz puntual en $(-2,2,0)$

a) Dónde tendremos el máximo valor de iluminación difusa en la esfera?

b) Si el observador está en $(0,2,0)$, ¿dónde estará situado el máximo brillo?

c) Idem para el observador en $(2,2,0)$



a) La expresión de la iluminación difusa viene dada por: $R_d = I_d \cdot K_d \cdot \cos(\varphi)$, donde φ es el ángulo que forman el vector normal a la superficie $(\vec{N})$ y el vector que indica la orientación de la luz $(\vec{L})$. Si queremos el máximo valor, $\cos(\varphi) = 1$, y eso se obtiene cuando el ángulo es 0°.
Esto quiere decir que obtendremos el máximo valor de iluminación difusa cuando el vector normal al objeto tenga las mismas coordenadas que el vector de la luz; $\vec{L} = \vec{N} = (-2,2,0)$

b) En este caso, la expresión de la ilumin queda determinada por:

$$I = R_D + R_E = I_D \cdot K_D \left(|\vec{N}| \cdot |\vec{L}|\right) + I_E \cdot K_E \left(|\vec{L}| \cdot (\vec{V}|^n\right)$$

$R_E = I_E \cdot K_E \cdot \cos(\alpha)^n$, donde α es el ángulo que forman el vector normal al plano $(\vec{N})$ y el vector que indica la posición del espectador $(\vec{V})$

$R_D = I_D \cdot K_D \cdot \cos(\varphi)$, donde φ es el ángulo que forman el vector normal al plano $(\vec{N})$ y el vector que indica en qué posición se encuentra la luz $(\vec{L})$

Por lo tanto el máximo brillo se obtiene cuando $\left(|\vec{L}| \cdot |\vec{V}|\right)^n$ es el valor máximo; para ello, $\vec{L}$ y $\vec{V}$ deben ser paralelos. ~~no tienen por qué ser paralelos~~

c) En este caso es exactamente lo mismo. Algo que hay que destacar es que en la reflexión especular, el ángulo que forman $\vec{R}$ y $\vec{V}$ es el mismo que el que forman $\vec{N}$ y $\vec{L}$, por lo tanto, ~~este~~ podemos decir que tanto para el caso del apartado b como en este, el máximo brillo se obtendrá cuando $\vec{N}$ y $\vec{L}$ sean paralelos.

(2A) La función glFrustum en OpenGL se utiliza para definir una matriz de proyección de perspectiva y ajustar sus parámetros puede simular el efecto de cambiar la zona visible, simular cómo funcionan las lentes de las cámaras con zoom, la matriz de proyección de perspectiva define la forma en que el mundo tridimensional se proyecta en la pantalla bidimensional.

21) c) Los ... de los ... con zoom permiten cambiar la zona visible, desde ángulos más grandes a más pequeños ¿Cómo se podría conseguir el mismo efecto con los parámetros de glFrustum? Explicarlo y poner ejemplos.

Tenemos que modificar los parámetros "left", "right", "bottom", "top", "near" y "far" de glFrustum;

1. Simulamos un ángulo más grande:

glFrustum (-1, 1, -1, 1, 0.1, 100);

→ Usamos valores pequeños para simular la ventana de visualización

→ Aumentamos el valor de "near" para simular que hay más objetos en la escena

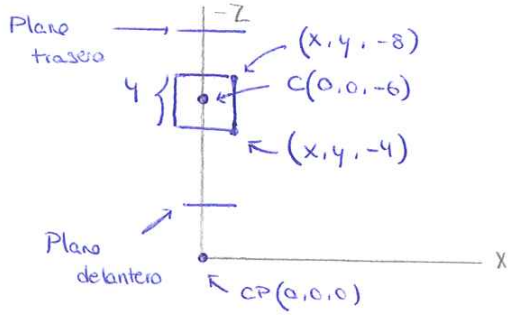
2. Simulamos un ángulo más pequeño:

glFrustum(-2, 2, -2, 2, 5, 100);

→ Usamos valores grandes para simular la ventana de visualización

→ Disminuimos el valor de "near" para simular que hay menos objetos en la escena.

22) Dado un cubo cuyo tamaño de arista es 4 y tiene su centro en el punto (0,0,-6), indicar el valor de los parámetros de glFrustum tal que haga que el plano delantero esté a la mitad de la distancia entre la cara delantera y el centro de proyección, y el tamaño de la ventana sea el doble de la distancia de las proyecciones de los vértices de la cara delantera en el eje X y la razón de aspecto de la ventana 4:3. La cámara está en la posición inicial de OpenGL. El plano trasero está a 4 veces la distancia entre el CP y la cara delantera



glFrustum(-4/2, +4/2, -4/2 · 3/4, 4/2 · 3/4, 4/2, 4·4

↓

glFrustum(-2, 2, -1.5, 1.5, 2, 16);

Distancia cara delantera y CP = 4

~~Distancia cara~~

25) Escriba el código necesario para disponer de funcionalidad "Zoom +" y "Zoom -" tanto en una cámara con proyección perspectiva como en una cámara con proyección ortogonal.

```
// Definimos una variable zoom
GLfloat zoom;

//En el método que usamos para emplear las teclas del teclado
void special_key(){
  case GLUT_KEY_PAGE_UP: zoom *= 0.5;
  case GLUT_KEY_PAGE_DOWN: zoom /= 0.5;
  .
  .
}
glFrustum(-1/zoom, 1/zoom, -1/zoom, 1/zoom, 1, 50); // Dividir simula el acercamiento
glOrtho(-1*zoom, 1*zoom, -1*zoom, -1*zoom, 1, 50); // Multiplicar ajusta el tamaño de la ventana de vista
```

(23) Dibuje el siguiente modelo [illegible] y como influyen sus parámetros en la iluminación de una superficie dada

$$I_\lambda = I_{a\lambda} + I_{L\lambda}\left(K_{d\lambda}(\vec{N}\vec{L}) + K_s(\vec{R}\cdot\vec{V})^n\right)$$

El modelo de iluminación se corresponde con el modelo de Phong, de modo que sus parámetros son:

1. $I_{a\lambda}$: Hace referencia a la componente de iluminación ambiental, que se encarga de iluminar uniformemente toda la escena, de manera que los elementos que no estén iluminados directamente no se vean negros. Esta iluminación no depende de la posición del observador
2. $I_{L\lambda}$: Hace referencia a la iluminación procedente de una fuente de luz puntual, que ilumina al objeto de manera ~~y~~ directa y que por tanto arroja resultados que ~~dependen~~ varían dependiendo de la posición del observador
3. $K_{d\lambda}$: Es la constante de reflexión difusa del material que indica qué porcentaje de la luz incidente es reflejado de manera difusa por una superficie.
4. $\vec{N}\cdot\vec{L}$: Es el producto escalar entre el vector normal a la superficie ($\vec{N}$) y el vector que indica la posición donde se encuentra la luz puntual ($\vec{L}$), junto con K_d, estos parámetros modifican la cantidad de luz difusa que recibe una superficie.
5. K_s: Es la constante de reflexión especular del material que indica qué porcentaje de la luz incidente es reflejado de manera ~~difus~~ especular por una superficie
6. $\vec{R}\cdot\vec{V}$: Es el producto escalar entre el vector que indica la posición del observador ($\vec{V}$) y el vector que indica la posición del rayo reflejado de luz ($\vec{R}$)
7. n: Modela el brillo de la superficie. Junto con K_s y $\vec{R}\cdot\vec{V}$ modifica la cantidad de luz especular que recibe la superficie

(24) Explique lo más detallado que pueda las distintas formas de hacer un pick en OpenGL

1. Identificación por color: A cada objeto se le asigna un identificador (número natural), que se convierte a un color. Cuando se dibuja el objeto se usa el color que tiene asociado. Al mover el cursor y pulsar para realizar la selección se guardan las coordenadas x e y del píxel seleccionado. Se lee el píxel del buffer en la posición x e y y se convierte el color al identificador. Para pasar del identificador al color se usan máscaras de bits para obtener cada parte y para pasar del ~~(identificador)~~ color al identificador se hacen los pasos inversos
2. Lanzando un rayo: La idea es seleccionar el objeto más cercano a la posición del cursor. Para ello obtenemos la posición x e y del cursor en coordenadas del observador y las pasamos a coordenadas ~~del objeto~~ mundiales. Hacemos pasar una línea recta por el centro de proyección y la nueva posición y calculamos la intersección con los objetos. Si hay intersección se añade a la lista, guardando el identificador del objeto y la profundidad. Finalmente ordenamos por profundidad y nos quedamos con el identificador del más cercano.
3. Por ventana: La idea consiste en crear una pequeña ventana alrededor de la posición x e y del cursor cuando se hace click. Una vez identificados los píxeles que conforman la ventana, sólo hay que dibujar el objeto, al que se le asigne un identificador. Si al convertir el objeto a píxeles, coincide con alguno o varios de la ventana, entonces hay selección. Se guarda el identificador del objeto y la profundidad y finalmente hacemos una ordenación por profundidad y seleccionamos el objeto con el identificador más cercano

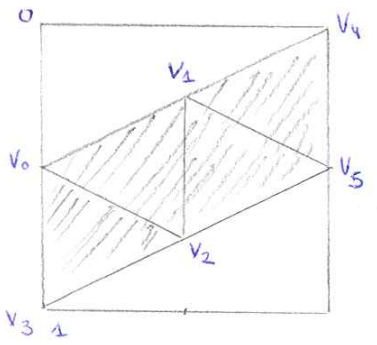
(26) ¿En qué consiste la técnica del ray-tracing?

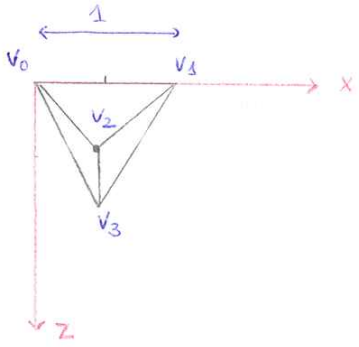
Es una técnica de renderizado en tiempo real que simula la forma en que los rayos de luz interactúan con los objetos en una escena 3D para producir imágenes más realistas y detalladas.

Basado en el ray-casting (trazado de rayos desde el observador hacia los objetos y ver cómo se interrumpen esos rayos) y en la óptica geométrica (luz como energía).

Consigue simular superficies difusas y especulares, sombras arrojadas, transmisión y reflexión

Ejercicio texturas triangulo



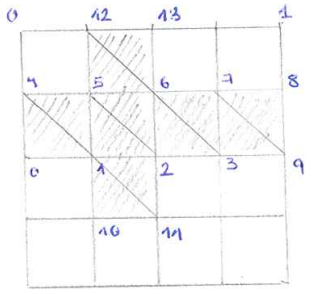


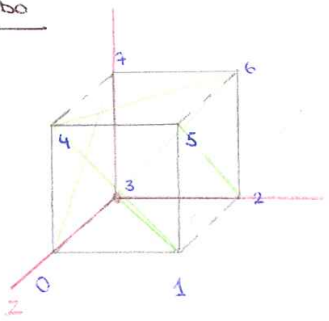
Definimos las estructuras:

```
vector<_vertex3f> Vertices;
vector<_vertex3ui> Triangles;
vector<_vertex2f> Textures;
```

```
Vertices.resize(4); // Definimos los vertices
Vertices[0] = _vertex3f(0,0,0);
Vertices[1] = _vertex3f(1,0,0);
Vertices[2] = _vertex3f(0.5,1,0.5);
Vertices[3] = _vertex3f(0.5,0,1);
Triangles.resize(8); // Definimos las caras
Triangles[0] = _vertex3ui(0,3,2);
Triangles[1] = _vertex3ui(3,0,2); // Cara izquierda
Triangles[2] = _vertex3ui(3,1,2);
Triangles[3] = _vertex3ui(1,2,3); // Cara derecha
Triangles[4] = _vertex3ui(1,0,2);
Triangles[5] = _vertex3ui(0,1,2); // Cara trasera
Triangles[6] = _vertex3ui(0,1,3);
Triangles[7] = _vertex3ui(1,0,1); // Cara inferior
Textures.resize(6); // Definimos las texturas
Textures[0] = _vertex2f(0,0.5);
Textures[1] = _vertex2f(0.5,0.25);
Textures[2] = _vertex2f(0.5,0.75);
Textures[3] = _vertex2f(1,1);
Textures[4] = _vertex2f(1,0);
Textures[5] = _vertex2f(1,0.5);
glBegin(GL_TRIANGLES);
for(int i=0; i<Triangles.size(); i++){
  glTexCoord2f((GLfloat*)&Textures[Triangles[i]._0]);
  glVertex3f((GLfloat*)&Vertices[Triangles[i]._0]);
  glTexCoord2f((GLfloat*)&Textures[Triangles[i]._1]);
  glVertex3f((GLfloat*)&Vertices[Triangles[i]._1]);
  glTexCoord2f((GLfloat*)&Textures[Triangles[i]._2]);
  glVertex3f((GLfloat*)&Vertices[Triangles[i]._2]);
}
glEnd();
```

Ejercicio de texturas cubo





Definimos las estructuras:

```
vector<_vertex3f> vertices;
vector<_vertex3ui> caras;
vector<_vertex2f> textures;
```

* El vértice 3 está centrado en el origen.

```
vertices.resize(8); // Definimos los vértices
vertices[0] = _vertex3f(0,0,1);
vertices[1] = _vertex3f(1,0,1);
vertices[2] = _vertex3f(1,0,0);
vertices[3] = _vertex3f(0,0,0);
vertices[4] = _vertex3f(0,1,1);
vertices[5] = _vertex3f(1,1,1);
vertices[6] = _vertex3f(1,1,0);
vertices[7] = _vertex3f(0,1,0);
caras.resize(12); // Definimos las caras
caras[0] = _vertex3ui(0,1,4);
caras[1] = _vertex3ui(1,4,5); // Cara delantera
caras[2] = _vertex3ui(1,2,5);
caras[3] = _vertex3ui(2,5,6); // Cara izquierda
caras[4] = _vertex3ui(2,3,6);
caras[5] = _vertex3ui(3,6,7); // Cara trasera
caras[6] = _vertex3ui(0,3,4);
caras[7] = _vertex3ui(3,4,7); // Cara derecha
caras[8] = _vertex3ui(4,5,6);
caras[9] = _vertex3ui(4,6,7); // Cara superior
caras[10] = _vertex3ui(0,1,3);
caras[11] = _vertex3ui(1,2,3); // Cara inferior
textures.resize(14); // Definimos las coord. de text.
textures[0] = _vertex2f(0,0.5);
textures[1] = _vertex2f(0.25,0.5);
textures[2] = _vertex2f(0.5,0.5);
textures[3] = _vertex2f(0.75,0.5);
textures[4] = _vertex2f(0,0.25);
textures[5] = _vertex2f(0.25,0.25);
textures[6] = _vertex2f(0.5,0.25);
textures[7] = _vertex2f(0.75,0.25);
textures[8] = _vertex2f(1,0.25);
textures[9] = _vertex2f(1,0.5);
textures[10] = _vertex2f(0.25,0.75);
textures[11] = _vertex2f(0.5,0.75);
textures[12] = _vertex2f(0.25,0);
textures[13] = _vertex2f(0.5,0);
```

```
glBegin(GL_TRIANGLES);
for(int i=0; i<caras.size(); i++){
  glTexCoord2f((GLfloat*)&textures[
    Triangles[i]._0]);
  glVertex2f((GLfloat*)&vertices[
    Triangles[i]._0]);
  glTexCoord2f((GLfloat*)&textures[
    Triangles[i]._1]);
  glVertex2f((GLfloat*)&vertices[
    Triangles[i]._1]);
  glTexCoord2f((GLfloat*)&textures[
    Triangles[i]._2]);
  glVertex2f((GLfloat*)&vertices[
    Triangles[i]._2]);
}
glEnd();
```