

Examen 1

1. Terminales raster, memoria de refresco y color en los terminales raster.

Los terminales raster permiten visualizar en ellos imágenes en movimiento gracias a un haz de electrones que recorre una superficie de fósforos haciendo que estos emitan o no luz de acuerdo con la imagen que se desea proyectar. Al contrario que en los terminales vectoriales, donde el haz de electrones se desplaza aleatoriamente, en los terminales raster se recorre siempre de forma ordenada la matriz de píxeles que conforman la pantalla, desde la parte superior izquierda a la inferior derecha.

Para proyectar imágenes en color, se sitúan los fósforos en grupos de tres, cada uno de ellos para iluminarse en uno de los tres colores primarios (rojo, verde y azul), y un cañón de electrones para cada color. Se añade, además, una parrilla justo delante de la pantalla de fósforo de tal forma que para cada tríada sólo hay un agujero en el que inciden los tres rayos. Los valores de color y brillo de la imagen que se debe dibujar en la pantalla se almacenan en una memoria conocida como *framebuffer*.

La información del color de una imagen se almacena en un determinado número de bits por píxel (denominado profundidad de color). A mayor cantidad de bits, mayor es el número de colores diferentes que podemos almacenar. Con 8 bits, dispondríamos de 256 colores diferentes, mientras que con 16 bits lograríamos 65.536. Con 24 bits de color podemos almacenar 16.777.216 colores, que es aproximadamente lo que el ojo humano puede percibir.

En los primeros terminales, la tasa de refresco era de entre 25 y 30Hz. Se utilizaba una estrategia para reducir el parpadeo sin aumentar el refresco conocida como *interlace* (pares-impares). Esta estrategia consiste en utilizar dos memorias para almacenar la imagen a representar, una de las cuales contiene las líneas pares de la imagen y otra las impares.

2. Tengo una bicicleta por piezas. Todos sus elementos están centrados en el origen y tienen un tamaño unitario. El cuadro debe tener un tamaño de 1'2 veces el unitario, mientras que la rueda o'6. La rueda estará situada a o'7 metros en el eje X respecto al origen del cuadro y gira a o'3 radianes por segundo. Indica qué secuencia de instrucciones se utilizaría para dibujarla. El frame rate es 1.

```
//Pasamos a grados los 0.3 radianes.
const GLdouble VELOCIDAD_GIRO = (0.3*3.14159)/180;

GLdouble angulo_giro = 0.0;

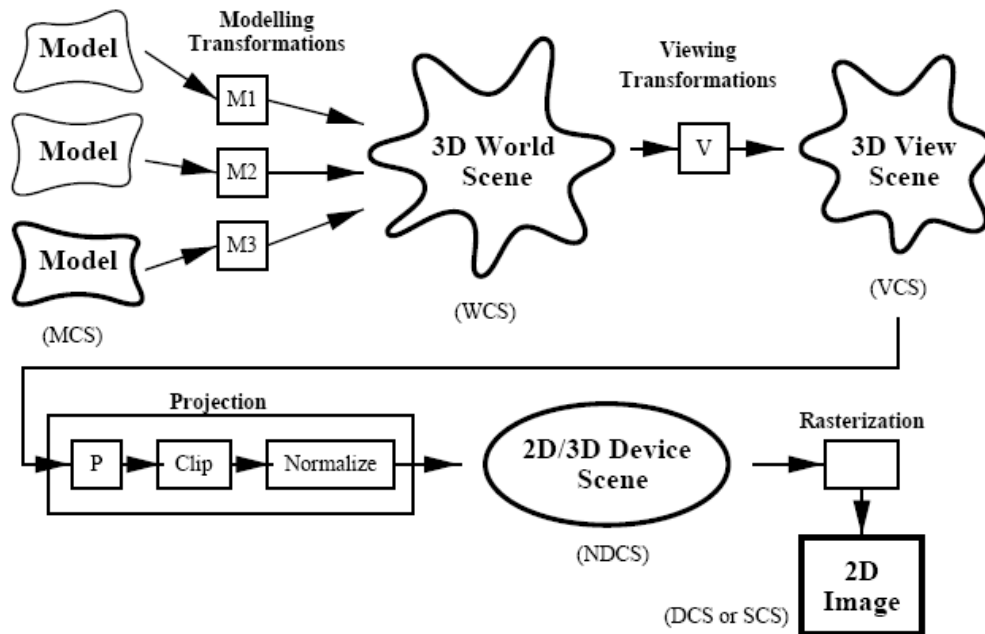
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //Cuadro.
    glPushMatrix();
    glScaled(1.2, 1.2, 1.2);
    glutWireCube(1.0);
    glPopMatrix();
    //Rueda 1.
    glPushMatrix();
    glTranslated(0.7, 0.0, 0.0);
    glScaled(0.6, 0.6, 0.6);
    glRotated(angulo_giro, 0.0, 0.0, -1.0);
    glutWireSphere(1.0, 20, 20);
    glPopMatrix();
    //Rueda 2.
    glTranslated(-0.7, 0.0, 0.0);
    glScaled(0.6, 0.6, 0.6);
    glRotated(angulo_giro, 0.0, 0.0, -1.0);
    glutWireSphere(1.0, 20, 20);
    glutSwapBuffers();
    glFlush();
}

void idle() {
    angulo_giro += VELOCIDAD_GIRO;
    if (angulo_giro >= 360)
        angulo_giro -= 360;
    glutPostRedisplay();
}

bool init() {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearDepth(1.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    return true;
}

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitWindowPosition(POS_X_VENTANA, POS_Y_VENTANA);
    glutInitWindowSize(ANCHO_VENTANA, ALTO_VENTANA);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutCreateWindow("Bicicleta =D");
    glutDisplayFunc(display);
    glutIdleFunc(idle);
    if (!init())
        return 1;
    glutMainLoop();
    return 0;
}
```

3. Haz un esquema de la visión 3D. Explícalo.



En primer lugar, se parte de modelos de diferentes objetos, a los que se les pueden aplicar diferentes transformaciones (de modelado), y que conforman una escena de un mundo en tres dimensiones. Aplicando a este mundo las transformaciones de vista o visualización pertinentes, obtenemos una vista de esa escena en tres dimensiones.

Esta vista en tres dimensiones pasa a continuación por un proceso de proyección en el que se calcula cómo representarla en una pantalla en dos dimensiones. A esta nueva representación en dos dimensiones se le aplica un proceso denominado *clipping* cuya finalidad es ajustar la imagen que se quiere visualizar al espacio disponible en la ventana o pantalla en la que se proyectará.

Por último, a esta proyección en dos dimensiones de nuestra escena en tres dimensiones ya ajustada para ser encuadrada en el espacio disponible para su proyección se le aplica un último proceso conocido como *rasterización*, que convierte toda la información que se tiene de la imagen a un conjunto de píxeles que pueden ser introducidos en el *framebuffer* de un dispositivo para que este los proyecte.

4. ¿Qué es un modelo poliédrico? ¿Qué debe tener en cuenta?

Un modelo poliédrico es aquel que utiliza poliedros (cuerpos geométricos tridimensionales formados por polígonos planos) para representar tridimensionalmente estructuras complejas. Los polígonos más frecuentemente utilizados para formar los poliedros son los triángulos.

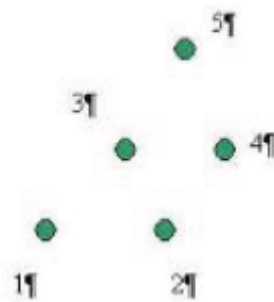
El modelo exige que en un poliedro se cumpla que:

- Un **vértice** debe **pertenecer al menos a una arista**.
- Los **polígonos no deben intersectarse** entre sí, **salvo en las aristas**.
- En una **arista no deben confluir más de dos polígonos**.
- En un **vértice** pueden coincidir **cualquier número de polígonos**.

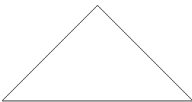
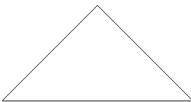
En una buena representación poliédrica, debemos:

- **Evitar la degeneración de los polígonos** (lados con longitud cero).
- **Evitar vértices en T**, ya que puede provocar problemas a la hora de rellenar el color por interpolación entre los vértices, y otras singularidades en ciertos algoritmos.
- **Utilizar primitivas** poligonales que permitan especificar el objeto **sin repetir varias veces los vértices** que pertenezcan a varios polígonos.
- Cuidarnos de la **falta de coplanariedad** (cualidad de estar en un mismo plano) en polígonos de más de tres vértices.

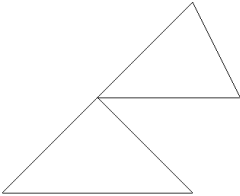
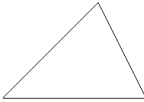
5. ¿Qué obtienes a partir de los siguientes puntos si ejecutas los siguientes comandos con `glPolygonMode(GL_FRONT, GL_FILL)`? Dibújalo.



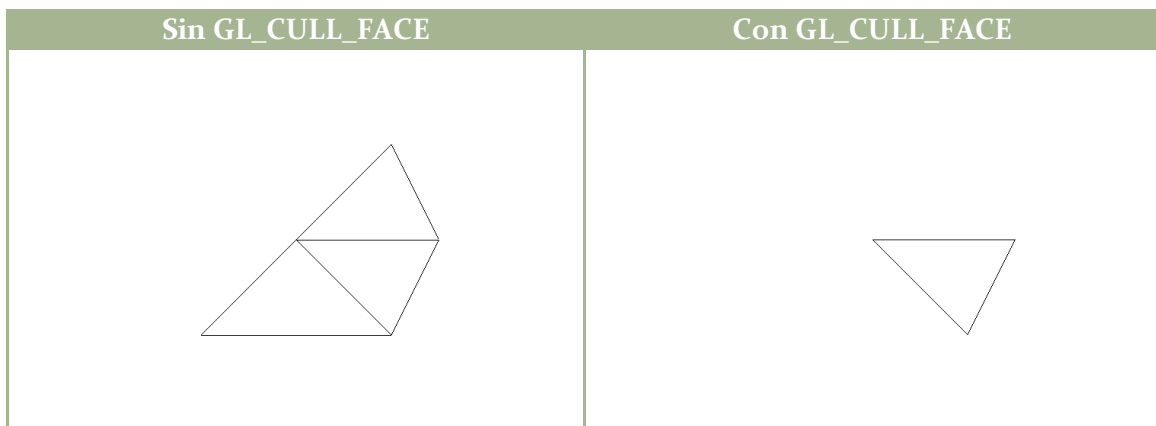
- `glBegin(GL_TRIANGLES);`
 - Secuencia: 1, 2, 3, 4, 5.

Sin GL_CULL_FACE	Con GL_CULL_FACE
	

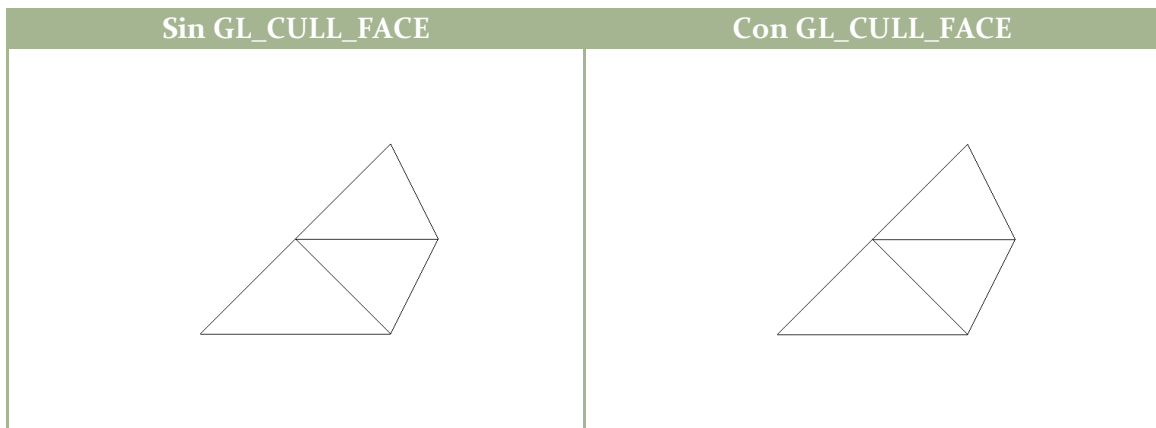
- Secuencia: 3, 2, 1, 3, 4, 5.

Sin GL_CULL_FACE	Con GL_CULL_FACE
	

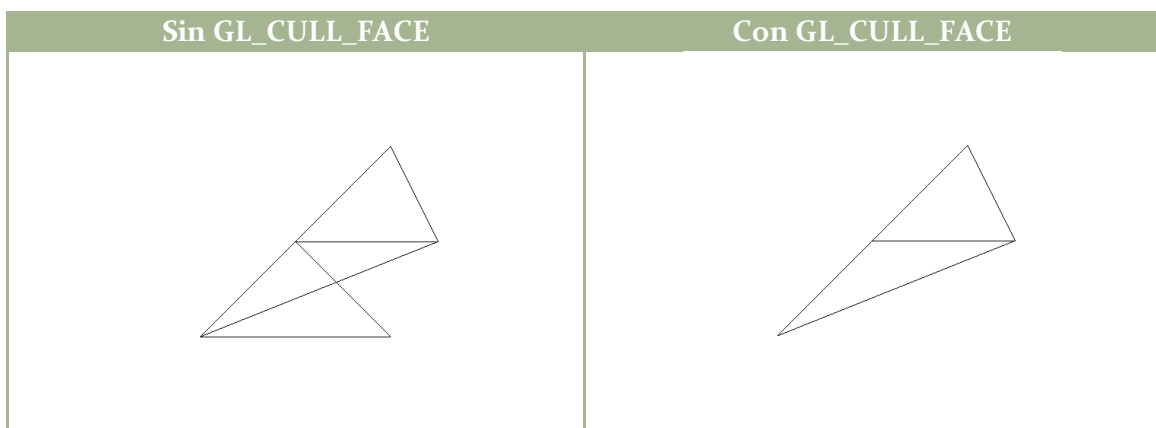
- Secuencia: 2, 1, 3, 4, 3, 2, 3, 5, 4.



- glBegin(GL_TRIANGLE_STRIP);
 - Secuencia: 1, 2, 3, 4, 5.



- Secuencia: 2, 1, 3, 4, 5.



6. Si te hablo de los u y v de una textura, ¿de qué te estoy hablando?

Son los vectores de superficie, los ejes u y v de dicha textura. Existe otro vector, w , que se escapa de la textura sobre la que se sitúa, paralelamente a la normal del plano sobre el que se encuentra la textura. Para calcular el valor de textura en un píxel determinado, es necesario utilizar la relación entre las coordenadas (s, t) de la textura en el plano y los vectores de superficie (u, v) .

7. Dada la cara $P(2, 0, 0)$, $Q(3, 0, 0)$, $R(3, 0, -1)$ y la partícula situada en $S(0, 3, 0)$ supuestamente puntual (radio 0) que se desplaza en la dirección $(-1, -1, 0)$, ¿chocará la partícula con el plano que contiene a la cara? ¿Y con la cara? Justifica tus conclusiones.

Vamos a calcular dos vectores del plano que contiene la cara mediante los puntos que tenemos, por ejemplo:

$$\overrightarrow{PQ} = Q - P = (3, 0, 0) - (2, 0, 0) = \boxed{(1, 0, 0) = \overrightarrow{PQ}}$$

$$\overrightarrow{PR} = R - P = (3, 0, -1) - (2, 0, 0) = \boxed{(1, 0, -1) = \overrightarrow{PR}}$$

Utilizando estos vectores y uno de los puntos, por ejemplo P , podemos obtener la ecuación general del plano resolviendo el siguiente determinante, que se forma poniendo X , Y y Z en la primera fila y restándole a estos el punto P ¹. En las otras dos filas, se ponen los vectores que calculamos antes.

$$\begin{vmatrix} X-2 & Y & Z \\ 1 & 0 & 0 \\ 1 & 0 & -1 \end{vmatrix} = 0$$

$$\begin{aligned} &((X-2) \times 0 \times -1) + (Y \times 0 \times 1) + (Z \times 0 \times 1) \\ &- [(Z \times 0 \times 1) + ((X-2) \times 0 \times 0) + (Y \times 1 \times -1)] = 0 \end{aligned}$$

$$\boxed{Y = 0 \equiv \alpha}$$

Ya tenemos la ecuación de nuestro plano, que llamaremos α . Ahora hallaremos la recta sobre la que se mueve la partícula utilizando el punto en el que está y su vector director.

$$\begin{cases} X = 0 - 1 \times t \\ Y = 3 - 1 \times t \\ Z = 0 + 0 \times t \end{cases} \equiv r$$

Para saber si la partícula choca contra el plano que contiene a la cara, comprobaremos si la recta corta al plano, sustituyendo los valores de X , Y y Z de la recta en la ecuación del plano.

$$3 - 1 \times t = 0$$

¹ $(X, Y, Z) - (2, 0, 0) = (X-2, Y, Z)$

$$\underline{t = 3}$$

Ahora simplemente tenemos que sustituir t en las ecuaciones paramétricas de la recta que tenemos de antes para obtener las coordenadas del punto en el que se cortan.

$$\left. \begin{array}{l} X = 0 - 1 \times 3 \\ Y = 3 - 1 \times 3 \\ Z = 0 + 0 \times 3 \end{array} \right\} \Rightarrow \boxed{\boxed{P(-3, 0, 0)}}$$

Para saber ahora si la partícula choca contra la cara, sólo tenemos que comprobar que el punto que obtuvimos antes está dentro de la cara. Podemos hacerlo incluso con un dibujo simple y ya vemos que el punto está fuera del triángulo, así **no chocan**.

Examen 2

1. Memoria de refresco cuantificación del color y LUT.

Sobre la memoria de refresco y la cuantificación del color se habla en la primera pregunta del examen 1: Terminales raster, memoria de refresco y color en los terminales raster.

Una LUT (**look-up table**) es una tabla con una entrada para cada posible valor de un píxel en una determinada imagen. Su utilización permite optimizar el uso de la memoria, ya que genera una paleta con los colores que se van a utilizar, indexándolos con el mínimo número de bits necesarios. De esta forma, se pueden almacenar los datos de color de la imagen utilizando los índices correspondientes de la LUT, lo cual resultará en un menor uso de memoria.

2. ¿Qué son las coordenadas homogéneas? ¿Para qué sirven? Ventajas e inconvenientes. Pon ejemplos.

Las **coordenadas homogéneas** son un instrumento utilizado para describir un punto (x, y) representándolo con la terna $(x/w, y/w, w)$, donde $w \in \mathbb{R}$. Su uso permite tratar cualquier transformación geométrica como una multiplicación de matrices, facilitando el cálculo de transformaciones.

El uso de coordenadas homogéneas ofrece las siguientes ventajas:

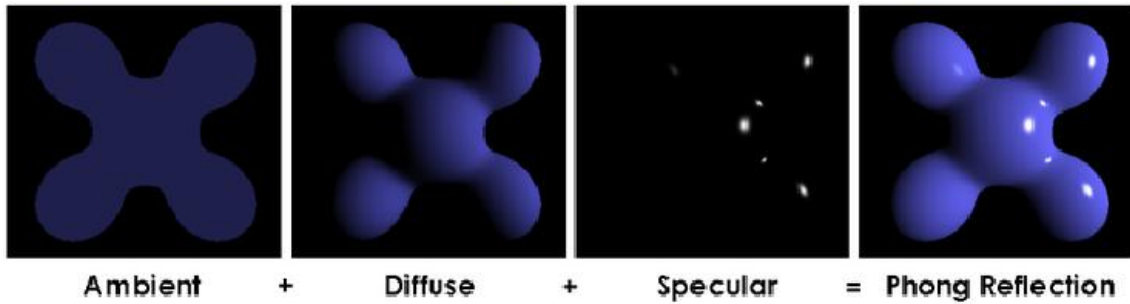
- Facilitan las operaciones de concatenación de transformaciones mediante el producto de matrices.
- Las tarjetas aceleradoras de gráficos implementan en hardware las operaciones matriciales con coordenadas homogéneas, mejorando el rendimiento de estas operaciones. Si las operaciones se realizasen por software requerirían un gran poder de cómputo.

Los ejemplos que los busque Julián, que yo no tengo ganas.

3. Modelo de iluminación simple. Modelo de Phong.

Lo de modelo iluminación simple no me apetece buscarlo, así que, que le den.

El **modelo de iluminación de Phong** es una técnica de iluminación digital en tres dimensiones en el que la luz y sus efectos sobre los objetos se divide y simplifica en tres tipos: **luz ambiente**, **luz difusa** y **luz especular**.



La **luz ambiente** no está asociada a ningún foco de luz ni a ninguna dirección. Provoca cambios uniformes en la iluminación de los objetos, sin generar sombras ni matices en el color.

La **luz difusa** sí está asociada a un foco de luz y proviene de una dirección en particular. Se refleja en todas las direcciones. Su variación se asocia con cambios de intensidad o posición del foco. Genera sombras y matices de color en función del ángulo de incidencia.

La **luz especular** también está asociada a un foco de luz proveniente de una dirección concreta, pero se refleja únicamente en un ángulo determinado. Su variación está también asociada a los cambios de intensidad y posición del foco, al igual que la luz difusa; pero, además, también es dependiente de la posición del observador. Con ella se obtienen efectos de brillo sobre las superficies.

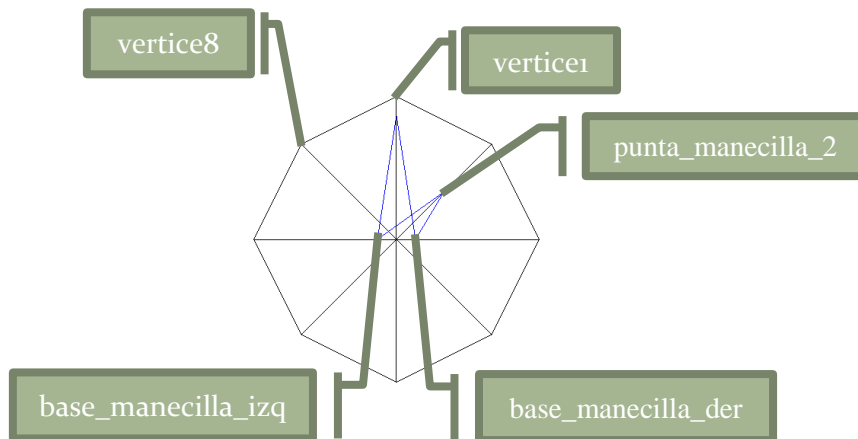
4. Tengo una escena con un tren que va por una vía y una estación. En la estación está el reloj 1 con sus manecillas y en el tren el reloj 2 con sus manecillas también. Haz un esquema de dibujo del tren, la estación y los dos relojes. Los modelos originales tienen un tamaño unitario. El tren es 10 veces la unidad y la estación 100, mientras que el reloj 8. El tren se mueve a 100Km/h.

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslated(pos_estacion_x, pos_estacion_y, pos_estacion_z);
    glPushMatrix();
    //Estación.
    glPushMatrix();
        glScaled(100.0, 100.0, 100.0);
        dibujar_estacion();
    glPopMatrix();
    //Reloj.
    glTranslated(pos_reloj_est_x, pos_reloj_est_y, pos_reloj_est_z);
    glScaled(8.0, 8.0, 8.0);
    dibujar_reloj();
    glPushMatrix();
        glRotated(angulo_manecilla_1, 0.0, 0.0, 1.0);
        dibujar_manecilla_1();
    glPopMatrix();
    glRotated(angulo_manecilla_2, 0.0, 0.0, 1.0);
    dibujar_manecilla_2();
    glPopMatrix();
    glTranslated(pos_tren_x, pos_tren_y, pos_tren_z);
    //Tren.
    glPushMatrix();
        glScaled(10.0, 10.0, 10.0);
        dibujar_tren();
    glPopMatrix();
    //Reloj.
    glTranslated(pos_reloj_tren_x, pos_reloj_tren_y, pos_reloj_tren_z);
    glScaled(8.0, 8.0, 8.0);
    dibujar_reloj();
    glPushMatrix();
        glRotated(angulo_manecilla_1, 0.0, 0.0, 1.0);
        dibujar_manecilla_1();
    glPopMatrix();
    glRotated(angulo_manecilla_2, 0.0, 0.0, 1.0);
    dibujar_manecilla_2();
}

void idle() {
    pos_tren_x += DESPL_TREN_X;
    pos_tren_y += DESPL_TREN_Y;
    pos_tren_z += DESPL_TREN_Z;
    angulo_manecilla_1 += GIRO_MANECILLA_1;
    if (angulo_manecilla_1 >= 360)
        angulo_manecilla_1 -= 360;
    angulo_manecilla_2 += GIRO_MANECILLA_2;
    if (angulo_manecilla_2 >= 360)
        angulo_manecilla_2 -= 360;
}
```

5. ¿Cómo dibujarías el reloj (como un octógono) y las agujas mediante GL_TRIANGLES, GL_TRIANGLE_STRIP y GL_TRIANGLE_FAN?

El dibujo que se va a hacer será el siguiente:



Tenemos como variables globales vectores en los que almacenamos la posición de cada vértice que necesitamos dibujar.

```
GLdouble centro[3] = {0.0, 0.0, 0.0};
GLdouble vertice1[3] = {0.0, 0.75, 0.0};
GLdouble vertice2[3] = {0.5, 0.5, 0.0};
GLdouble vertice3[3] = {0.75, 0.0, 0.0};
GLdouble vertice4[3] = {0.5, -0.5, 0.0};
GLdouble vertice5[3] = {0.0, -0.75, 0.0};
GLdouble vertice6[3] = {-0.5, -0.5, 0.0};
GLdouble vertice7[3] = {-0.75, 0.0, 0.0};
GLdouble vertice8[3] = {-0.5, 0.5, 0.0};
GLdouble base_manecilla_izq[3] = {-0.1, 0.0, 0.0};
GLdouble base_manecilla_der[3] = {0.1, 0.0, 0.0};
GLdouble punta_manecilla_1[3] = {0.0, 0.65, 0.0};
GLdouble punta_manecilla_2[3] = {0.25, 0.25, 0.0};
```

En nuestra función de inicialización utilizamos la línea siguiente, para que sólo se dibujen las aristas de los polígonos (en lugar de rellenarlos de color).

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

La función de dibujado utilizando *GL_TRIANGLES* sería la siguiente:

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor4d(0.0, 0.0, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
        //Reloj.
        glVertex3dv(centro); glVertex3dv(vertice2); glVertex3dv(vertice1);
        glVertex3dv(centro); glVertex3dv(vertice3); glVertex3dv(vertice2);
        glVertex3dv(centro); glVertex3dv(vertice4); glVertex3dv(vertice3);
        glVertex3dv(centro); glVertex3dv(vertice5); glVertex3dv(vertice4);
        glVertex3dv(centro); glVertex3dv(vertice6); glVertex3dv(vertice5);
        glVertex3dv(centro); glVertex3dv(vertice7); glVertex3dv(vertice6);
        glVertex3dv(centro); glVertex3dv(vertice8); glVertex3dv(vertice7);
        glVertex3dv(centro); glVertex3dv(vertice1); glVertex3dv(vertice8);
        //Manecillas.
        glColor4d(0.0, 0.0, 1.0, 0.0);
        glVertex3dv(base_manecilla_izq); glVertex3dv(base_manecilla_der);
        glVertex3dv(punta_manecilla_1);
        glVertex3dv(base_manecilla_izq); glVertex3dv(base_manecilla_der);
        glVertex3dv(punta_manecilla_2);
    glEnd();
    glutSwapBuffers();
    glFlush();
}
```

En el caso de usar *GL_TRIANGLE_STRIP*, sería la siguiente²:

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor4d(0.0, 0.0, 0.0, 0.0);
    glBegin(GL_TRIANGLE_STRIP);
        //Reloj.
        glVertex3dv(vertice1); glVertex3dv(vertice8); glVertex3dv(centro);
        glVertex3dv(vertice7); glVertex3dv(vertice6); glVertex3dv(centro);
        glVertex3dv(vertice5); glVertex3dv(vertice4); glVertex3dv(centro);
        glVertex3dv(vertice3); glVertex3dv(vertice2); glVertex3dv(centro);
        glVertex3dv(vertice1);
    glEnd();
    glColor4d(0.0, 0.0, 1.0, 0.0);
    glBegin(GL_TRIANGLE_STRIP);
        //Manecilla 1.
        glVertex3dv(base_manecilla_izq);
        glVertex3dv(base_manecilla_der);
        glVertex3dv(punta_manecilla_1);
    glEnd();
    glBegin(GL_TRIANGLE_STRIP);
        //Manecilla 1.
        glVertex3dv(base_manecilla_izq);
        glVertex3dv(base_manecilla_der);
        glVertex3dv(punta_manecilla_2);
    glEnd();
    glutSwapBuffers();
    glFlush();
}
```

² Si se activa el *GL_CULL_FACE*, dos de los lados no se ven, no sé por qué. Para el caso de *GL_TRIANGLES* y de *GL_TRIANGLE_FAN* no hay problema aunque se active.

Por último, utilizando *GL_TRIANGLE_FAN*, sería:

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor4d(0.0, 0.0, 0.0, 0.0);
    glBegin(GL_TRIANGLE_FAN);
        //Reloj.
        glVertex3dv(centro);
        glVertex3dv(vertice1);
        glVertex3dv(vertice8);
        glVertex3dv(vertice7);
        glVertex3dv(vertice6);
        glVertex3dv(vertice5);
        glVertex3dv(vertice4);
        glVertex3dv(vertice3);
        glVertex3dv(vertice2);
        glVertex3dv(vertice1);
    glEnd();
    glColor4d(0.0, 0.0, 1.0, 0.0);
    glBegin(GL_TRIANGLE_FAN);
        //Manecillas.
        glVertex3dv(base_manecilla_izq);
        glVertex3dv(base_manecilla_der);
        glVertex3dv(punta_manecilla_1);
        glVertex3dv(base_manecilla_der); //Aux.
        glVertex3dv(punta_manecilla_2);
    glEnd();
    glutSwapBuffers();
    glFlush();
}
```

En este último código, la línea marcada con el comentario «Aux» crea un triángulo extra igual al de la manecilla grande, pero con la cara trasera hacia nosotros y la delantera hacia el interior de la pantalla. De todas formas, puesto que se crea exactamente en la misma posición que la propia manecilla grande, no nos estorba, y es necesario hacerlo para poder dibujar ambas manecillas sin tener que abrir otro *glBegin()*, de forma que se una la punta de la segunda manecilla al vértice base izquierdo y luego al derecho.