Problemas resueltos

Informática Gráfica

Grado en Informática y Matemáticas. Curso 2018-19.

UNIVERSIDAD
DE GRANADA

ETSI Informática y de Telecomunicación.
Departamento de Lenguajes y Sistemas Informáticos.

Índice general.

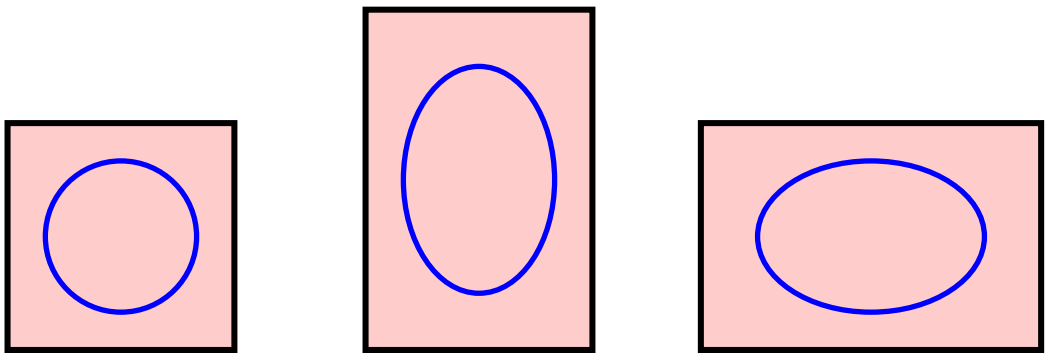
1. Introducción..

1

Supongamos que en la función gestora del evento de cambio de tamaño de la ventana se usa `glViewport` con una llamada de la siguiente forma:

```
glViewport( 0, 0, ancho, alto ) ;
```

donde `ancho` y `alto` son las nuevas dimensiones de la ventana. En estas condiciones, el plano de visión es cuadrado (aunque se puede cambiar, esta es la opción por defecto en OpenGL), pero la ventana no tiene porque serlo (`ancho` y `alto` no tienen porque coincidir), y por tanto el *viewport* tampoco es cuadrado (ya que ocupa toda la ventana). Como consecuencia, los objetos se visualizarán deformados (ya que una zona cuadrada del espacio se visualiza en una *viewport* que no es cuadrado):

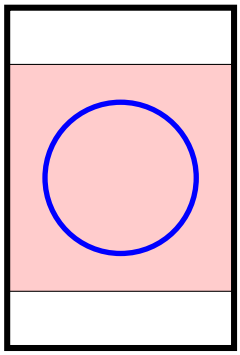


situación original
`ancho == alto`

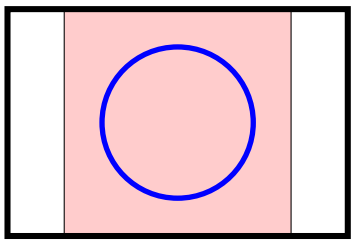
deformado
`ancho < alto`

deformado
`alto < ancho`

Para evitar esto hay varias opciones, las más simple (y restrictiva) es hacer que el *viewport* siempre sea cuadrado, ocupando el cuadrado más grande posible dentro de la ventana (centrado en el centro de la ventana), y dejando el resto (bandas a los lados o arriba y abajo) sin dibujar:



no deformado
`ancho < alto`



no deformado
`alto < ancho`

Escribe una nueva versión de la función gestora del cambio de tamaño que funcione como se describe en el párrafo anterior. Puedes usar el código fuente para ejemplos 2D sencillos que hay en la página web.

Respuesta:

Hay tres posibilidades:

- Si `ancho` es menor que `alto`, el origen del viewport debe desplazarse en vertical, un número de filas igual a la mitad de la diferencia `alto-ancho` (y hacer su lado y su ancho iguales a `ancho`). Se puede hacer con

```
glViewport( 0, (alto-ancho)/2, ancho, ancho );
```

- Si `alto` es menor que `ancho`, entonces el origen del viewport debe desplazarse en horizontal, un número de columnas igual a la mitad de la diferencia `alto-ancho` (y hacer su ancho y alto igual a `alto`).

```
glViewport( (ancho-alto)/2, 0, alto, alto );
```

- Si `ancho` y `alto` coinciden, entonces no hay que hacer nada (se queda como estaba el código original del enunciado).

```
glViewport( 0, 0, ancho, alto );
```

Estas tres posibilidades se pueden combinar de forma sencilla con este código:

```
int min = ancho < alto ? ancho : alto ; // min = mínimo de alto y ancho
glViewport( (ancho-min)/2, (alto-min)/2, min, min );
```

2

Documéntate sobre GLFW y averigua que función o funciones debes usar para obtener el tamaño en pixels del escritorio (filas y columnas). Usando dicha función, describe como habría que modificar el ejemplo básico visto en teoría para que la ventana de la aplicación esté sea al inicio cuadrdad, esté centrada en el escritorio, y su alto sea 4/5 del alto dicho escritorio.

Respuesta:

Para lograr esto, se puede recuperar el monitor primario, y después consultar el modo de vídeo actual de dicho monitor. Finalmente, se consulta la resolución de dicho modo de vídeo (son atributos públicos de la clase). El código quedaría como sigue:

```
const GLFWvidmode *     //modo actual del mon. primario.
 modo       = glfwGetVideoMode( glfwGetPrimaryMonitor() );
const int
 alto_tot   = modo->height,              // alto total del escritorio en el modo actual
 ancho_tot  = modo->width,               // ancho total del escritorio en el modo actual
 alto_ven   = (alto_tot*4)/5,            // alto de la ventana
 ancho_ven  = alto_ven,                  // ancho de la ventana
 pos_y      = (alto_tot-alto_ven)/2,     // posición Y de la ventana
 pos_x      = (ancho_tot-ancho_ven)/2;   // posicion X de la ventana


// Crear ventana del tamaño adecuado y luego posicionarla donde queremos
// (glfw_window debe ser de tipo: GLFWwindow *
glfw_window = glfwCreateWindow( ancho_ven, alto_ven,
                                "titulo", nullptr, nullptr );
glfwSetWindowPos( glfw_window, pos_x, pos_y );
```

3

Escribe el código de una función que permita dibujar un polígono 2D regular de n lados (n es un parámetro de tipo **int**, obviamente $n > 2$), centrado en el origen, y cuyo diámetro sea igual a 3/4 del alto del *viewport* actual. Se supone que la zona visible en el *viewport* del espacio de coordenadas es la que hay por defecto al empezar un programa OpenGL, es decir, entre -1 y $+1$ en los tres ejes.

Las aristas deben ser de color azul oscuro, con dos pixels de ancho, y el interior debe aparecer relleno de color azul claro (celeste). Puedes usar el código fuente para ejemplos 2D sencillos que hay en la página web.

4

Averigua (o verifica mediante un programa sencillo), si la función **glClear** inicializa el color de todos los pixels de la ventana o solamente los de los pixels del viewport.

Respuesta:

La llamada **glClear** limpia toda la ventana actual (el *framebuffer* completo), no solamente el *viewport*

5

Imagina una aplicación que siempre use el viewport cuadrado más grande posible, centrado en la

ventana, según se ha descrito en el enunciado del problema 1. Supongamos que queremos que el fondo del *viewport* sea de color blanco y la parte de la ventana externa al viewport (las bandas sin ocupar), de color gris.

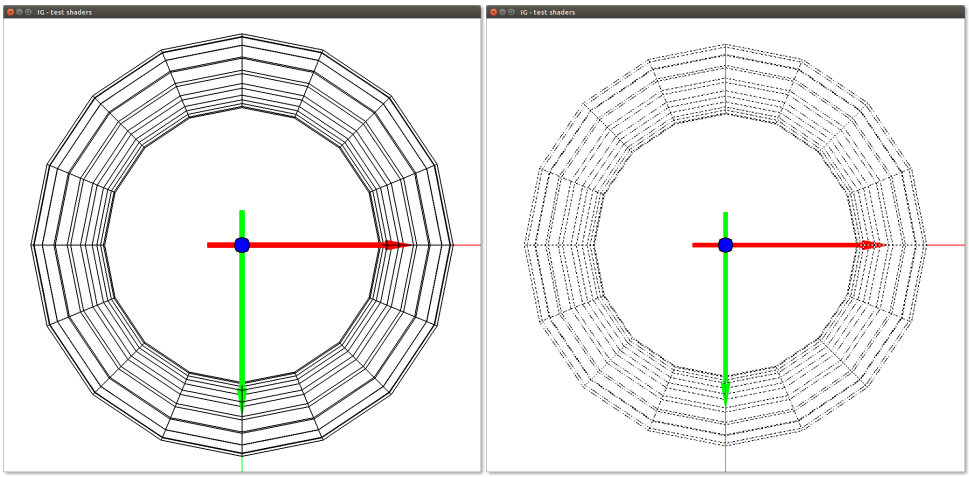
Esto se puede hacer primero limpiando la ventana completa con el color gris, y luego dibujando un polígono rectangular de color blanco que coincida con el viewport. El dibujo del rectángulo se puede simplificar usando la función **`glRect`**. Averigua como funciona y úsala en tu respuesta. Ten en cuenta que esto requiere que en la función gestora de redibujado se conozca el tamaño (ancho y alto) de la ventana.

6

En este problema se intenta ilustrar la utilidad del cauce programable, incluso usando un *fragment shader* muy sencillo. Para hacerlo, debes dar los siguientes pasos:

1. Documéntate sobre la función **`glutWireTorus`**, de la librería GLUT, úsala para producir una imagen de un toro en modo alambre, puedes usar el mismo código que se os proporciona para las prácticas, completando la función que dibuja los objetos (usando el cauce de funcionalidad fija).
2. Usa las rutinas para cargar, compilar y activar un par *fragment/vertex shader* sencillo (las incluidas en las transparencias de teoría), de forma que puedas visualizar el toro exactamente con la misma apariencia que antes, solo que ahora usando el cauce programable.
3. Modifica el *fragment shader* sencillo, de forma que las lineas aparezcan punteadas en lugar de continuas. Para ello, se pueden usar en el *fragment shader* dos variables enteras (`gl_FragCoord.x` y `gl_FragCoord.y`, del tipo **`int`**), que contienen las coordenadas (número de columna y fila, respectivamente) del pixel (fragmento) al que se le quiere asignar color. En función de estas variables, se puede decidir asignar al pixel el color actual o bien el color de fondo, haciendolo con un patrón tal que las aristas aparezcan punteadas. (para esto es útil usar el operador módulo o resto (%), este operador requiere GLSL versión 1.3, se puede conseguir poniendo como primera línea del *fragment shader* esta directiva: **#**`version 130`)

Aqui aparece el toro con línea continua y el toro con líneas de puntos. Ten en cuenta que el dibujo de los ejes no se ve afectado por esto (se debe seguir haciendo con el cauce fijo, es decir, con línea continua)



Respuesta:

7

Usando las propiedades que definen el producto escalar, **demuestra** que si $\vec{a} = \mathcal{R}(x_o, y_0, z_0)^t$ y $\vec{b} = \mathcal{R}(x_1, y_1, z_1)^t$ (en un marco cartesiano $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \mathbf{o}]$), entonces su producto escalar se puede calcular como:

$$\vec{a} \cdot \vec{b} \;=\; x_0x_1 \;+\; y_0y_1 \;+\; z_0z_1$$

Responde a esta pregunta: ¿ sigue siendo esto válido para vectores en 2D ?

Respuesta:

Es trivial sin más que expresar $\vec{a}$ como $x_0\vec{x} + y_0\vec{y} + z_0\vec{z}$ e igualmente $\vec{b}$ como $x_1\vec{x} + y_1\vec{y} + z_1\vec{z}$. Expandiendo estas expresiones en $\vec{a}\cdot\vec{b}$, y usando las propiedades, obtendremos la demostración de forma sencilla:

$$\begin{aligned}
\vec{a} \cdot \vec{b} \;&=\; (x_0\vec{x} + \; y_0\vec{y} \; + \; z_0\vec{z}) \cdot (x_1\vec{x} \; + \; y_1\vec{y} \; + \; z_1\vec{z}) \\
&=\; x_0x_1(\vec{x}\cdot\vec{x}) \; + \; x_0y_1(\vec{x}\cdot\vec{y}) \; + \; x_0z_1(\vec{x}\cdot\vec{z}) \; + \\
&\quad\; y_0x_1(\vec{y}\cdot\vec{x}) \; + \; y_0y_1(\vec{y}\cdot\vec{y}) \; + \; y_0z_1(\vec{y}\cdot\vec{z}) \; + \\
&\quad\; z_0x_1(\vec{z}\cdot\vec{x}) \; + \; z_0y_1(\vec{z}\cdot\vec{y}) \; + \; z_0z_1(\vec{z}\cdot\vec{z}) \\
&=\; x_0x_1(\vec{x}\cdot\vec{x}) \; + \; y_0y_1(\vec{y}\cdot\vec{y}) \; + \; z_0z_1(\vec{z}\cdot\vec{z}) \\
&=\; x_0x_1 \; + \; y_0y_1 \; + \; z_0z_1
\end{aligned}$$

Para vectores en 2D la respuesta es igualmente válida ya que el producto escalar cumple los mismos axiomas.

8

Demuestra, usando las propiedades que hemos visto del producto vectorial, que si $\vec{a} = \mathcal{R}(x_0, y_0, z_0)^t$ y $\vec{b} = \mathcal{R}(x_1, y_1, z_1)^t$ entonces el producto escalar de estos dos vectores se puede obtener como:

$$\vec{a} \times \vec{b} \;=\; \mathcal{R}(\; y_0z_1 - z_0x_1 \;,\; z_0x_1 - x_0z_1 \;,\; x_0y_1 - y_0x_1 \;)$$

Respuesta:

De forma parecida a como hicimos para el producto escalar, en este caso expandimos las expresiones de $\vec{a}$ y $\vec{b}$ como combinación lineal de los versores, y después usamos los axiomas que definen el

producto vectorial:

$$\begin{aligned}
\vec{a}\times\vec{b} &\equiv (x_0\vec{x}+\ y_0\vec{y}\ +\ z_0\vec{z})\times(x_1\vec{x}\ +\ y_1\vec{y}\ +\ z_1\vec{z})\\
&= x_0x_1(\vec{x}\times\vec{x})\ +\ x_0y_1(\vec{x}\times\vec{y})\ +\ x_0z_1(\vec{x}\times\vec{z})\ +\\
&\quad y_0x_1(\vec{y}\times\vec{x})\ +\ y_0y_1(\vec{y}\times\vec{y})\ +\ y_0z_1(\vec{y}\times\vec{z})\ +\\
&\quad z_0x_1(\vec{z}\times\vec{x})\ +\ z_0y_1(\vec{z}\times\vec{y})\ +\ z_0z_1(\vec{z}\times\vec{z})\\
&= \quad x_0y_1(\vec{x}\times\vec{y})\ -\ x_0z_1(\vec{z}\times\vec{x})\\
&\quad -\ y_0x_1(\vec{x}\times\vec{y})\ +\ y_0z_1(\vec{y}\times\vec{z})\\
&\quad +\ z_0x_1(\vec{z}\times\vec{x})\ -\ z_0y_1(\vec{y}\times\vec{z})\\
&= x_0y_1\vec{z}\ -\ x_0z_1\vec{y}\ -\ y_0x_1\vec{z}\ +\ y_0z_1\vec{x}\ +\ z_0x_1\vec{y}\ -\ z_0y_1\vec{x}\\
&= \mathcal{R}(y_0z_1-z_0y_1)\vec{x}\ +\ (z_0x_1-x_0z_1)\vec{y}\ +\ (x_0y_1-y_0x_1)\vec{z}\\
&\equiv (\ y_0z_1-z_0y_1\ ,\ z_0x_1-x_0z_1\ ,\ x_0y_1-y_0x_1\ )
\end{aligned}$$

c.q.d.

9

Sabemos que, para cualquier dos vectores libres no nulos, $\vec{a}$ y $\vec{b}$ en un espacio afín 2D o 3D, se cumple la siguiente propiedad del producto escalar:

$$\vec{a}\cdot\vec{b} \ = \ \|\vec{a}\|\,\|\vec{b}\|\cos(\phi) \tag{1.1}$$

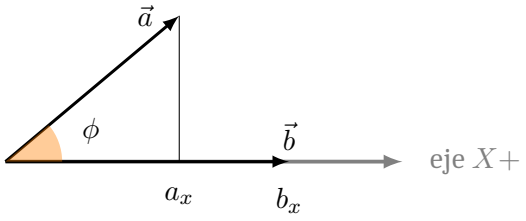
donde ϕ es el ángulo que forman los dos vectores.

Demuestra que esta igualdad se puede demostrar a partir del hecho de que, en un triángulo rectángulo, si consideramos el ángulo agudo ϕ que forman la hipotenusa y un cateto, entonces el coseno de ϕ es igual a la longitud de dicho cateto dividida por la longitud de la hipotenusa.

Respuesta:

Consideramos, en el citado espacio afín, un marco cartesiano tal que su eje X está alineado con el vector $\vec{b}$. Recordemos que en los marcos cartesianos, cualquiera de sus dos ejes son perpendiculares, por definición. Suponemos que las coordenadas de $\vec{a}$ en $\mathcal{R}$ son $(a_x, a_y, a_z, 0)^t$, y las coordenadas de $\vec{b}$ son $(b_x, 0, 0, 0)^t$.

Consideremos los vectores $\vec{a}$ y $\vec{b}$, dibujándolos ambos de forma que el plano que los contiene es el plano de la figura. Al ser el marco cartesiano, se forma un triángulo rectángulo, uno de cuyos ángulos agudos es ϕ:



En el triángulo rectángulo de la figura la longitud de la hipotenusa es $\|\vec{a}\|$. Respecto a la longitud del cateto contiguo, vemos que es igual a la componente X de $\vec{a}$, es decir a_x. Por tanto la propiedad del triángulo rectángulo que consideramos como cierta de partida se escribe como:

$$\cos\phi \;=\; \frac{a_x}{\|\vec{a}\|} \tag{1.2}$$

Por otro lado podemos expandir el valor de $\vec{a}\cdot\vec{b}$ usando las coordenadas cartesianas de $\vec{a}$ y $\vec{b}$, obtenemos:

$$\vec{a}\cdot\vec{b} \;\equiv\; a_x b_x \,+\, a_y 0 \,+\, a_z 0 \;=\; a_x b_x \;=\; a_x\|\vec{b}\|$$

y por tanto podemos escribir a_x como sigue:

$$a_x \;=\; \frac{\vec{a}\cdot\vec{b}}{\|\vec{b}\|}$$

Si suponemos que es cierta la ecuación 1.2, podemos sustituir esta expresión de a_x en dicha ecuación, obtenemos:

$$\cos\phi \;=\; \frac{a_x}{\|\vec{a}\|} \;=\; \left(\frac{\vec{a}\cdot\vec{b}}{\|\vec{b}\|}\right)\frac{1}{\|\vec{a}\|} \;=\; \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

es decir:

$$\vec{a}\cdot\vec{b} \;=\; \|\vec{a}\|\,\|\vec{b}\|\cos\phi$$

que es exactamente la igualdad 1.1 que queríamos demostrar.

2. Modelado de Objetos..

10

Supongamos que queremos codificar una esfera de radio $1/2$ y centro en el origen de dos formas:

- Por enumeración espacial, dividiendo el cubo que engloba a la esfera en celdas, de forma que haya k celdas por lado del cubo, todas ellas son cubos de $1/k$ de ancho. Cada celda ocupa un bit de memoria (si su centro está en la esfera, se guarda un 1, en otro caso un 0).
- Usando un modelo de fronteras (una malla indexada de triángulos), en el cual se usa una rejilla de triángulos y aristas que siguen los meridianos y paralelos, habiendo en cada meridiano y en cada paralelo un total de k vértices (se guarda únicamente la tabla de vértices y la de triángulos).

Asumiendo que un **`float`** y un **`int`** ocupan 4 bytes cada uno, contesta a estas cuestiones:

- Expresa el tamaño de ambas representaciones en bytes como una función de k.
- Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

Compara los tamaños de ambas representaciones en ambos casos ($k = 16$ y $k = 1024$).

Respuesta:

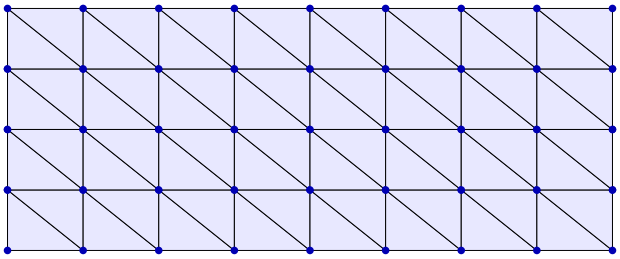
- Expresa el tamaño de ambas representaciones en bytes como una función de k.
  - La enumeración espacial requiere k^3 bits, es decir: $(1/8)k^3$ bytes.
  - La representación mediante una malla requiere k^2 vértices (cada vértice son $3*4$ bytes) y $2k^2$ caras (cada cara son también $3*4$ bytes). En total son $36k^2$ bytes.
- Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
  - La enumeración espacial requiere $(16^3)/8 = 512$ bytes, es decir $= 1/2$ KB.
  - Mediante una malla, se ocupan: $36(16)^2 = 36 * 256 = 9 * 1024$ bytes, es decir: 9 KB.

  (la enumeración espacial ocupa 18 veces menos memoria)
- Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)
  - La enumeración espacial requiere $(1/8)1024^3 = (1/2^3)(2^{10})^3 = 2^{27}$ bytes, es decir $2^7 = 128$ MB.
  - Mediante una malla, se ocupan: $36(2^{10})^2 = 36(2^{20})$ bytes, es decir 36 MB.

  (la enumeración espacial ocupa algo más de tres veces más que la representación de fronteras).

11

Considera una malla indexada (tabla de vértices y caras, esta última con índices de vértices) con topología de rejilla como la de la figura, en la cual hay n columnas de pares de triángulos y m filas (es decir, hay $n + 1$ filas de vértices y $m + 1$ columnas de vértices, con $n, m > 0$, en el ejemplo concreto de la figura, $n = 8$ y $m = 4$).



En relación a este tipo de mallas, responde a estas dos cuestiones:

(a) Supongamos que un **`float`** ocupa 4 bytes (igual un **`int`**) ¿ que tamaño en memoria ocupa la malla completa, en bytes ? (tener en cuenta únicamente el tamaño de la tabla de vértices y triángulos, suponiendo que se almacenan usando los tipos **`float`** e **`int`**, respectivamente). Expresa el tamaño como una función de m y n.
(b) Escribe el tamaño en KB suponiendo que $m = n = 128$.
(c) Supongamos que m y n son ambos grandes (es decir, asumimos que $1/n$ y $1/m$ son prácticamente 0). deduce que relación hay entre el número de caras n_C y el número de vértices n_V en este tipo de mallas.

Respuesta:

(a) En total hay claramente $n_V = (n+1)(m+1)$ vértices, cada uno de ellos supone tres valores **`float`**, es decir, 12 bytes, luego el tamaño de la tabla de vértices es $12(n+1)(m+1)$ bytes. Por otro lado, hay $n_C = 2nm$ caras (triángulos), ocupando cada uno de ellos también 12 bytes, así que el tamaño de la tabla de vértices es $24nm$ bytes.
En total, el número de bytes será la suma de ambas expresiones:

$$12(n+1)(m+1) \; + \; 24nm \;\; = \;\; 36nm \; + \; 12n \; + \; 12m \; + \; 12 \quad \text{bytes}$$

(b) El tamaño total será: $36 \cdot 128^2 + 12 \cdot 257 \; = \; 592908$ bytes. Dividiendo por 2^{10} obtenemos casi 580 KB.
(c) Podemos ver cual es el valor de n_V/n_C, asumiendo que $1/n$ y $1/m$ son iguales a 0. Según lo anterior:

$$\frac{n_V}{n_C} \;\; = \;\; \frac{(n+1)(m+1)}{2nm} \;\; = \;\; \frac{1}{2} + \frac{1}{2n} + \frac{1}{2m} + \frac{1}{2nm} \;\; \approx \;\; \frac{1}{2}$$

es decir $n_C \; \approx \; 2n_V$, y hay casi el doble de caras que de vértices.

12

Imagina de nuevo una malla como la del problema anterior, supongamos que usamos una representación como tiras de triángulos, de forma que cada fila de triángulos (con $2n$ triángulos) se almacena en una tira, habiendo un total de m tiras. La tabla de punteros a tiras tiene un entero (el número de tiras) y m punteros, cada puntero suponemos que tiene 8 bytes de tamaño. De nuevo, asume que las coordenadas son de tipo **`float`** (4 bytes). Responde a estas cuestiones:

(a) Indica que cantidad de memoria ocupa este representación, en estos dos casos:
  (a.1) Como función de n y m, en bytes.
  (a.2) Suponiendo $m = n = 128$, en KB.

(b) Para m y n grandes ¿ que relación hay entre el tamaño en memoria de la malla indexada del problema anterior y el tamaño de la malla almacenada como tiras de triángulos ?

(c) Si suponemos que la transformación de cada vértice se hace en un tiempo constante igual a la unidad ¿ que relación hay entre los tiempos de procesamiento de vértices para esta malla cuando se representa como una malla indexada y como tiras de triángulos ?

Respuesta:

Hay que tener en cuenta que una tira con k triángulos requerirá $k+2$ vértices, y por tanto $12k+24$ bytes.

(a.1) Tamaño como función de n y m, en bytes
Cada tira tiene $2n$ triángulos, por tanto, el número de vértices en total será de $2+2n$ (los dos primeros vértices del primer triángulo, más un vértice adicional por cada triángulo). Es decir, tenemos $2(n+1)$ vértices. Cada vértice ocupa tres flotantes, y cada flotante cuatro enteros, luego tenemos un total de $4 \cdot 3 \cdot 2(n+1)$ bytes, es decir $24n+24$ bytes. Sumándole los 8 bytes del puntero, tenemos $24n+32$ bytes por tira. Puesto que hay m tiras (y un entero: el número de tiras), tenemos un total de

$$(24n+32)m \; + \; 8 \; + \; 4 \;\; = \;\; 24mn \; + \; 32m \; + \; 12 \quad \text{bytes}$$

(a.2) Tamaño suponiendo $m = n = 128$, en KB. Según la fórmula anterior, el tamaño para $m = n = 2^7$ es 397320 bytes, dividiendo por 2^{10} obtenemos un poco más de $388KB$

(b) Comparación con mallas indexadas
Vistas las expresiones de este problema (a.1) y el anterior (a), es evidente que las tiras de triángulos ocupan aproximadamente $2/3$ de la memoria que ocupan las mallas indexadas ($24nm$ frente a $36nm$).

(c) Comparación de tiempos
En una malla indexada, cada vértice es procesado una vez. En la representación con tiras de triángulos, la mayoría de los vértices se procesan dos veces. Esto se debe a que la mayoría de los vértices son vértices interiores que pertenecen a dos tiras. Por tanto, el tiempo de procesamiento de vértices es casi el doble para las tiras de esta forma que para la malla indexada.

13

Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un *poliedro* simplemente conexo de caras triangulares). Considera el número de vértices n_V, el número de aristas n_A y el número de caras n_C en este tipo de mallas. Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

$$\begin{aligned} n_A &= 3\,(n_V - 2) \\ n_C &= 2\,(n_V - 2) \end{aligned}$$

(nótese que, al igual que en el problema anterior, sigue siendo cierto que el número de caras es aproximadamente el doble que el de vértices).

Respuesta:

Para demostrar esto podemos considerar el poliedro topológicamente más simple posible (en el

sentido de grafo de adyacencia más simple), que es un tetraedro. En un tetraedro se tiene $n_A = 6$, $n_C = 4$ y $n_V = 4$, luego las dos igualdades son ciertas.

Supongamos que se inserta, en una arista cualquiera del tetraedro, un nuevo vértice. Después se insertan dos nuevas aristas, que unen dicho nuevo vértice con cada uno de sus dos vértices opuestos en los triángulos originales. De esta forma se dividen esos triángulos en dos. En total hay un vértice más, tres aristas más, y dos triángulos más. Por tanto, vemos que se siguen cumpliendo las igualdades después de las inserciones, suponiendo que se cumplían antes.

Esta inserción de nuevos elementos se puede repetir tantas veces como se quiera, hasta llegar a una malla con una topología exactamente igual a la de cualquier malla de las condiciones descritas en el enunciado. Por tanto, esas igualdades siempre se cumplen, ya que no dependen de la geometría (las posiciones de los vértices).

Alternativamente, este problema se puede resolver de forma más directa usando el hecho de que el número de aristas n_A cumple:

$$n_A \;=\; \frac{3}{2}\, n_C$$

ya que, si contamos tres aristas por cada cara, habremos contado cada arista exactamente dos veces. Por otro lado, sabemos que la *característica de Euler* de estos poliedros es 2, es decir, se cumple:

$$n_V \;-\; n_A \;+\; n_C \;=\; 2$$

de estas dos igualdades anteriores se deducen fácilmente las dos del enunciado.

14

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un puntero a un vector **ari**, que en cada entrada tendrá una estructura (de tipo **Tupla2i**) con un vector **ind** de dos enteros (los índices en la tabla de vértices de los dos vértices en los extremos de la arista). El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

```
std::vector<Tupla2i> ari ;
```

Escribe el código de una función C++ para crear y calcular la tabla de aristas a partir de la tabla de triángulos. Intenta encontrar una solución con la mínima complejidad en tiempo y memoria posible. Suponer que el número de vértices adyacentes a uno cualquiera de ellos es como mucho un valor constante $k > 0$, valor que no depende del número total de vértices, que llamamos n. Considerar dos casos:

1. Los triángulos se dan con orientación *no coherente*: esto quiere decir que si un triángulo está formado por los vértices i, j, k, estos tres índices pueden aparecer en cualquier orden en la correspondiente entrada de la tabla de triángulos (puede aparece como i, j, k o como i, k, j, o como k, j, i, etc....)
2. Los triángulos se dan con orientación *coherente*: esto quiere decir que si dos triángulos comparten una arista entre los vértices i y j, entonces en uno de los triángulos la arista aparece como (i, j) y en el otro aparece como (j, i) (decimos que en el triángulo a, b, c *aparecen* las tres aristas (a, b), (b, c) y (c, a)). Además, asumimos que la malla es *cerrada*, es decir, que cada arista es compartida por exactamente dos triángulos.

Respuesta:

Caso 1 (orientación no coherente)

Para implementar el algoritmo, debemos de recorrer la lista de triangulos. Para cada triángulo entre los vértices i,j y k, añadimos a la tabla de aristas las tres aristas de ese triángulo, es decir, las aristas (i, j), (j, k) y (k, i). Esto tiene una complejidad en tiempo lineal con el número de triángulos. Si asumimos que dicho número es aproximadamente el doble del de vértices n, vemos que la complejidad es lineal con dicho número de vértices.

El problema de esta solución es que las aristas compartidas entre dos triángulos se insertarán dos veces. Para evitarlo, podemos comprobar, antes de insertar una arista, si esa arista ya ha sido insertada en la tabla **`ari`**. Para eso habría que recorrer toda la lista de aristas ya insertadas. Esto produciría una complejidad en tiempo en el orden del número de vértices por el número de aristas. Puesto que el número total de aristas está entre n y $(k/2)n$, la complejidad en tiempo está en $O(n^2)$.

Para encontrar una solución mejor, podemos tener un vector con una entrada por vértice. Le llamamos **`ver_ady`** y tendrá en cada entrada un vector, con los índices vértices adyacentes al vértice de dicha entrada. En concreto, la entrada número i corresponde al vértice i, y tiene un vector con los índices j (con $i < j$) tales que hay una arista entre i y j. Al recorrer la tabla de triángulos añadimos entradas en **`ver_ady`**, es decir, para cada arista (i, j) encontrada, añadimos j al vector correspondiente a i. Para saber si una arista entre (i, j) (donde $i < j$) ya ha sido añadida, consultamos si j está en el vector de índices adyacentes i. Finalmente, tras recorrer los triángulos, se crea la tabla **`ari`** a partir de la tabla **`ver_ady`**.

Puesto que cada entrada en **`ver_ady`** tiene a lo sumo k índices, la consulta mencionada tiene complejidad $O(1)$, y por tanto la creacion de **`ari`** tiene complejidad $O(n)$, tanto en tiempo como en memoria.

Solo quedaría escribir el código usando vectores de la librería estándard de C++.

Caso 2 (malla cerrada, orientación coherente)

En este caso, la creación con complejidad óptima se simplifica. El algoritmo procesa todos los triángulos, y cuando encuentra uno con los vértices i, j, k, introduce las aristas (i, j), (j, k) y (k, i) en la tabla de aristas. Al final, por cada arista entre los vértices i y j habrá exactamente dos entradas en la tabla de aristas, una con (i, j) y la otra con (j, i). Para evitar este duplicación, basta con insertar solo las aristas (i, j) cuando se cumple $i < j$. Esto produce un algoritmo sencillo y correcto con complejidad lineal con el número de vértices.

Esto dejaría la tabla con exactament dos entradas por cada arista, una con la arista

15

Escribe el código de la función para calcular el área total de una malla indexada de triángulos (en el sistema de referencia local del objeto). Será una función que acepta un puntero a una **`MallaInd`** y

devuelve un número real.

16

Escribe el código con las declaraciones necesarias para incorporar a una **`MallaInd`** una estructura de aristas aladas. Se necesitan tres nuevas tablas: la de aristas aladas **`tari`**, la de aristas de vértice **`taver`** y la de aristas de triángulo **`tatri`**.

17

Supongamos que tenemos una malla indexada de triangulos, almacenada en las tablas **`vertices`** (tuplas de 3 flotantes) y `triangulos` (tuplas de 3 enteros). **Escribe** el código de la función para calcular la tabla de normales de triángulos (**`nor_tri`**), (asumir que se diponen de los operadores usuales de tuplas o vectores, es decir $+, -, \cdot, \times, \| \ \|, \ldots$). Estas normales deben quedar normalizadas. Escribe también el código (los más eficiente posible en tiempo y memoria) para calcular la tabla de normales de vértice en **`nor_ver`** (recuerda que la normal a cada vértice se puede calcular, si no hay otra forma de conocerla, como el vector promedio de las normales de las caras adyacentes al vértice, normalizado).

18

Escribe el código para crear una copia de una malla de triángulos, de forma que en la copia los vértices estén cada uno desplazado en la dirección de su normal una distancia ϵ, donde ϵ es un parámetro real de la función.

19

Escribe el pseudocódigo para visualizar una malla con los triángulos de un color sólido y las aristas en modo alambre ¿ que problemas se pueden encontrar ? . Describe razonadamente como se puede usar el código del problema anterior para resolver dichos problemas.

20

Tal y como se ha visto en teoría, en 2D, la transformación T de rotación ϕ radianes entorno al origen de un marco cartesiano tiene asociada una matriz R_ϕ definida así:

$$R_\phi \ = \ \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

donde $c \equiv \cos\phi$ y $s \equiv \sin\phi$.

Demuestra que el producto escalar es invariante por rotación, es decir, que para cualquier ángulo ϕ y vectores $\vec{a} = \mathcal{R}\mathbf{a}$ y $\vec{b} = \mathcal{R}\mathbf{b}$ se cumple:

$$(T\vec{a}) \cdot (T\vec{b}) \;=\; \vec{a} \cdot \vec{b}$$

Respuesta:

Sean $\mathbf{a} = (x_0, y_0, 0)^t$ y $\mathbf{b} = (x_1, y_1, 0)^t$ las coordenadas de $\vec{a}$ y $\vec{b}$, respectivamente. La demostración se puede hacer usando estas coordenadas, es decir, debemos de demostrar que se cumple esta igualdad:

$$(R_\phi \mathbf{a}) \cdot (R_\phi \mathbf{b}) \;=\; \mathbf{a} \cdot \mathbf{b}$$

usando la propiedad $c^2 + s^2 = 1$, de esta forma:

$$\begin{aligned}
(R_\phi \mathbf{a}) \cdot (R_\phi \mathbf{b}) &= (R_\phi (x_0, y_0, 0)^t) \cdot (R_\phi (x_1, y_1, 0)^t) \\
&= (\, cx_0 - sy_0 \,,\; sx_0 + cy_0 \,) \cdot (\, cx_1 - sy_1 \,,\; sx_1 + cy_1 \,) \\
&= (cx_0 - sy_0)(cx_1 - sy_1) \;+\; (sx_0 + cy_0)(sx_1 + cy_1) \\
&= (\, c^2 x_0 x_1 \;+\; s^2 y_0 y_1 \;-\; cs(x_0 y_1 + y_0 x_1) \,) \;+ \\
&\quad (\, c^2 y_0 y_1 \;+\; s^2 x_0 x_1 \;+\; cs(y_0 x_1 + x_0 y_1) \,) \\
&= (c^2 + s^2) x_0 x_1 \;+\; (c^2 + s^2) y_0 y_1 \\
&= x_0 x_1 \;+\; y_0 y_1 \\
&= \mathbf{a} \cdot \mathbf{b}
\end{aligned}$$

21

Demuestra que, en 2D, la matriz de rotación por un ángulo ϕ (que llamamos R_ϕ) tiene como inversa la matriz de rotación por un ángulo $-\phi$. Es decir:

$$(R_\phi)^{-1} \;=\; R_{-\phi}$$

Indica si el resultado también es cierto en 3D para rotaciones cuyo eje son los ejes de coordenadas, y para rotaciones arbitrarias.

Respuesta:

Para esto, usamos las propiedades de simetría y anti-simetría de la funciones seno y coseno, es decir, las igualdades $\cos(-\phi) = \cos\phi$ y $\sin(-\phi) = -\sin\phi$, que son ciertas para cualquier ϕ.

Obtenemos explícitamente la matriz $R_{-\phi}$. Por simplicidad, usamos matrices 2x2, aunque la demostración se extiende trivialmente a matrices de rotación 3x3 en coordenadas homogéneas:

$$R_\phi \;=\; \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \quad \Longrightarrow \quad R_{-\phi} \;=\; \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

donde $c \equiv \cos\phi$ y $s \equiv \sin\phi$.

Ahora podemos verificar directamente que $R_\phi R_{-\phi}$ es la matriz identidad

$$R_\phi R_{-\phi} \;=\; \begin{pmatrix} c & -s \\ s & c \end{pmatrix}\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \;=\; \begin{pmatrix} c^2+s^2 & cs-sc \\ sc-cs & s^2+c^2 \end{pmatrix} \;=\; \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \;\equiv\; I$$

lo cual equivale a $(R_\phi)^{-1} = R_{-\phi}$, c.q.d.

En 3D, el resultado también es cierto. Para una rotación cuyo eje es un eje de coordenadas, dicha rotación solo afecta a dos coordenadas y deja invariante la tercera, por tanto la demostración vista aquí sigue siendo válida, añadiendo una componente que no cambia.

Para rotaciones arbitrarias en torno a cualquier eje, dichas rotaciones pueden verse como las rotaciones anteriores, sin más que expresarlas en un sistema de coordenadas adecuado (rotado) que tenga un eje paralelo al eje de rotación.

22

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo ϕ y vector $\mathbf{v} = (x, y)^t$, se cumple $\|R_\phi \mathbf{v}\| = \|\mathbf{v}\|$ (aquí R_ϕ es la matriz de rotación un ángulo igual a ϕ radianes). Describe razonadamente si este resultado puede generalizarse a 3D.

Respuesta:

Como en otros problemas, definimos $c = \cos\phi$ y $s = \sin\phi$. Usando la definición de rotación en 2D, si el punto original tiene como coordendas $\mathbf{v} = (x, y)^t$ entonces las coordenadas rotadas $R_\phi(x, y)^t$ son $(cx - sy, sx + cy)$, por tanto:

$$\|R_\phi \mathbf{v}\| \;=\; \sqrt{(cx-sy)^2 + (sx+cy)^2}$$

el término bajo la raiz cuadrada se puede simplificar

$$\begin{aligned} \|R_\phi \mathbf{v}\|^2 &= (cx-sy)^2 \,+\, (sx+cy)^2 \\ &= (c^2x^2 + s^2y^2 - 2cxsy) \,+\, (s^2x^2 + c^2y^2 + 2sxcy) \\ &= c^2x^2 \,+\, s^2y^2 \,+\, s^2x^2 \,+\, c^2y^2 \\ &= (c^2+s^2)x^2 \,+\, (c^2+s^2)y^2 \\ &= x^2 \,+\, y^2 \\ &= \|\mathbf{v}\|^2 \end{aligned}$$

de donde se deduce directamente $\|R_\phi \mathbf{v}\| = \|\mathbf{v}\|$, c.q.d.

Esto se puede también demostrar de forma más sencilla viendo que, en general, $\|\mathbf{u}\| = \sqrt{\mathbf{u}\cdot\mathbf{u}}$ para cualquier $\mathbf{u}$, y aplicando la invarianza del producto escalar ante rotaciones :

$$\|R_\phi \mathbf{v}\| \;=\; \sqrt{(R_\phi \mathbf{v})\cdot(R_\phi \mathbf{v})} \;=\; \sqrt{\mathbf{v}\cdot\mathbf{v}} \;=\; \|\mathbf{v}\|$$

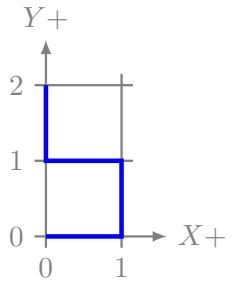
c.q.d.

23

(problema eliminado de esta relación)

24

Escribe una función en OpenGL, llamada **gancho** para dibujar con OpenGL la figura azul de la izquierda (en la cual cada segmento recto tiene longitud unidad, y el extremo inferior está en el origen). La función debe ser neutra respecto de la matriz modelview, el color o el grosor de la línea, es decir, usará la modelview, el color y grosor del estado de OpenGL en el momento de la llamada.



Respuesta:

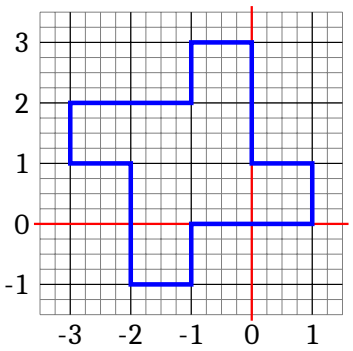
La implementación de esta función es muy sencilla, únicamente hay que tener en cuenta que en **glBegin** debemos usar la constante **GL_LINE_STRIP**, de forma que OpenGL conecta cada vértice con el siguiente que se le envía.

```
void gancho()
{
   // definir vértices
   const Tupla2f vertices =
   {  { 0.0, 0.0 } ,
      { 1.0, 0.0 } ,
      { 1.0, 1.0 } ,
      { 0.0, 1.0 } ,
      { 0.0, 2.0 }
   }

   // dibujar:
   glEnableClienteState( GL_VERTEX_ARRAY );
   glVertexPointer( 2, GL_FLOAT, 0, vertices.data() );
   glDrawArrays( GL_LINE_STRIP, 0, vertices.size() );
   glDisableClientState( GL_VERTEX_ARRAY );
}
```

25

Usando (exclusivamente) la función **gancho** del problema anterior, construye otra función (**gancho_x4**) para dibujar con OpenGL el polígono que aparece en la figura:



Hacer dos variantes: En esta función no se podrá usar **glPushMatrix** ni **glPopMatrix**, se usará exclusivamente **glTranslatef** y **glRotate** (la longitud de cada segmento seguirá siendo la unidad). Hay que tener en cuenta que la figura se puede obtiener exclusivamente usando rotaciones de la original, pero en esas rotaciones el centro no es el origen.

Respuesta:

Para implementar esta función usaremos un bucle, en cada iteración dibuja un gancho. El primer gancho coincide con el original, así que puede dibujarse sin introducir transformación alguna. Siguiendo en el sentido contrario de las agujas del reloj, el segundo gancho se obtiene rotando 90^o grados (positivos, es decir, en el sentido contrario de las agujas del reloj) el primero y luego desplazando dos unidades en Y (positivas, es decir, hacia arriba). A partir de este segundo gancho, el tercero se obtiene de nuevo con una rotación de 90^o seguida de traslación de 2 unidades en Y. Por tanto, el código puede quedar como sigue aquí:

El código quedaría así:

```
void gancho_x4()
{
   glMatrixMode( GL_MODELVIEW );
   for( unsigned i = 0 ; i < 4 ; i++ )
   {
      gancho();
      glTranslatef( 0.0, +2.0, 0.0 );   // (2) traslación +2 en Y
      glRotatef( 90.0, 0.0,0.0,1.0 );   // (1) rotar
   }
}
```

26

Escribe el pseudocódigo OpenGL otra función (**gancho_2p**) para dibujar esa misma figura, pero

escalada y rotada de forma que sus extremos coincidan con dos puntos arbitrarios disintos $\dot{p}_0$ y $\dot{p}_1$, puntos cuyas coordenadas de mundo son $\mathbf{p}_0 = (x_0, y_0, 1)^t$ y $\mathbf{p}_1 = (x_1, y_1, 1)^t$. Estas coordenadas se pasan como parámetro a dicha función.

Escribe una solución (a) acumulando matrices de rotación, traslación y escalado en la matriz *modelview* de OpenGL. Escribe otra solución (b) en la cual la matriz *modelview* se calcula directamente sin necesidad de usar funciones trigonometricas (como lo son el arcotangente, el seno, coseno, arcoseno o arcocoseno).

Respuesta:

El problema se puede solucionar usando un escalado seguido de una rotación y finalmente una traslación. Puesto que el escalado es uniforme, es conmutativo con la rotación, es decir, se puede hacer primero el escalado y luego la rotación, o al revés.

Respecto al escalado, debe de convertir una figura de alto igual a 2 unidades en una figura de alto igual a la distancia entre $\mathbf{p}_0$ y $\mathbf{p}_1$, luego el factor de escala es f, definido como:

$$f \;=\; \frac{\|\mathbf{p}_1 - \mathbf{p}_0\|}{2} \tag{2.1}$$

Respecto a la rotación, debe ser una rotación de forma que alinee el eje Y con la línea que va desde $\mathbf{p}_0$ a $\mathbf{p}_1$. Consideramos el triángulo rectángulo formado por $\mathbf{p}_0 = (x_0, y_0, 1)^t$, el punto $(x_0, y_1, 1)^t$ y $\mathbf{p}_1 = (x_1, y_1, 1)^t$. En dicho triángulo la hipotenusa es la línea de $\mathbf{p}_0$ a $\mathbf{p}_1$, de longitud $\|\mathbf{p}_1 - \mathbf{p}_0\|$. El ángulo de rotación ϕ es por tanto el arco cuya tangente es igual a $x_1 - x_0$ dividido por $y_1 - y_0$ (cateto opuesto dividido por cateto contiguo), pero cambiado de signo, pues para valores positivos de divisor y dividendo, la rotación es en sentido horario, es decir, un ángulo negativo. Por tanto, tenemos:

$$\phi \;=\; -\arctan\left(\frac{x_1 - x_0}{y_1 - y_0}\right) \;=\; \arctan\left(\frac{x_0 - x_1}{y_1 - y_0}\right) \tag{2.2}$$

Sin embargo, esta expresión produce el mismo ángulo cuando divisor y dividendo cambian ambos de signo, además si $y_1 = y_0$ la división produce un valor infinito que podría no ser gestionado bien por arctan. Por tanto debemos usar la función atan2:

$$\phi \;=\; \text{atan2}(\, x_0 - x_1 \,,\, y_1 - y_0 \,) \tag{2.3}$$

Finalmente, respecto al desplazamiento, debe ser claramente un desplazamiento por $\mathbf{p}_0$.

Por tanto, podemos construir la matriz como sigue:

$$M \;=\; \mathrm{D}[\mathbf{p}_0] \cdot \mathrm{R}[\phi] \cdot \mathrm{E}[f] \tag{2.4}$$

Por tanto, en código OpenGL:

```
#include <cmath> // para 'atan2', M_PI
.....

void gancho_2p( const Tupla2f & p0, const Tupla2f & p1 )
{
   const float
      dx  = p1(X)-p0(X),
      dy  = p1(Y)-p0(Y),
      phi = std::atan2( -dx, dy )*180.0/M_PI,
```

```
      d   = std::sqrt( dx*dx, dy*dy ),
      f   = d/2.0;

   glMatrixMode( GL_MODELVIEW );
   glTranslatef( p0(X), p0(Y), 0.0 );   // (3) traslación por p0
   glRotatef( phi, 0.0, 0.0, 1.0 );     // (2) rotación por φ
   glScalef( f, f, 1.0 );               // (1) escalado por f
   gancho();
}
```

En el caso (b), si queremos evitar el arcotangente (y también la raíz cuadrada), podemos calcular directamente la matriz de cambio de base

Se puede usar un marco de coordenadas $\mathcal{R} = [\vec{x}_R, \vec{y}_R, \dot{p}_0]$, que es un marco ortogonal respecto del marco de coordenadas del mundo, pero no es un marco necesariamente ortonormal, es decir, aunque los dos ejes son perpendiculares entre si, la longitud de cada uno no es necesariamente la unidad. El marco tiene origen en el punto de coordenadas $\mathbf{p}_0$. El eje Y (vector $\vec{y}_R$) es la mitad del vector desde $\mathbf{p}_0$ hacia $\mathbf{p}_1$), luego las coordenadas de $\vec{y}_R$ son:

$$\mathbf{y}_R \;=\; (a,b,0)^t \;=\; \frac{1}{2}\,(\mathbf{p}_1 - \mathbf{p}_0) \;=\; ((x_1-x_0)/2, (y_1-y_0)/2, 0)^t \tag{2.5}$$

Por otro lado, el vector $\vec{x}_R$ tiene como coordenadas $\mathbf{x}_R$, las cuales se obtienen rotando -90^o el vector $\mathbf{y}_R$, es decir:

$$\mathbf{x}_R \;=\; \mathrm{R}[-90^o]\,\mathbf{y}_R \;=\; \mathrm{R}[-90^o]\,(a,b,0)^t \;=\; (b,-a,0)^t \tag{2.6}$$

Podemos construir directamente la matriz de cambio de base desde las coordenadas relativas a $\mathcal{R}$ hacia las coordenadas relativas al marco de coordenadas del mundo (en coordenadas homogéneas en 2D). Esto se hace poniendo en las columnas las coordenadas $\vec{x}_R$ y $\vec{y}_R$, y haciendo los términos de traslación iguales a las coordenadas de $\dot{p}_0$, es decir:

$$M \;=\; \begin{pmatrix} b & a & x_0 \\ -a & b & y_0 \\ 0 & 0 & 1 \end{pmatrix}$$

En el código se usa **glMultMatrix**, especificando la matriz traspuesta de M (y extendida a 3D, pero de forma que no cambia la Z), queda así:

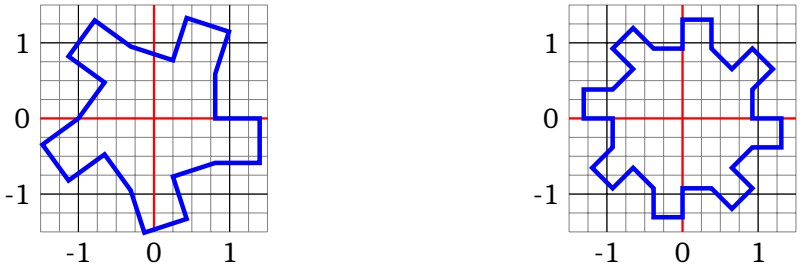
```
void gancho_2p( const Tupla2f & p0, const Tupla2f & p1 )
{
   const float
      a = 0.5*(p1(X)-p0(X)),
      b = 0.5*(p1(Y)-p0(Y));

   const GLfloat mat[16] =    // matriz 4x4 M (puesta por columnas)
   {  b    , a     , 0.0, 0.0,
      -a   , b     , 0.0, 0.0,
      0.0  , 0.0   , 1.0, 0.0,
      p0(X), p0(Y) , 0.0, 1.0
   } ;

   glMatrixMode( GL_MODELVIEW );
   glMultMatrix( mat );
   gancho();
}
```

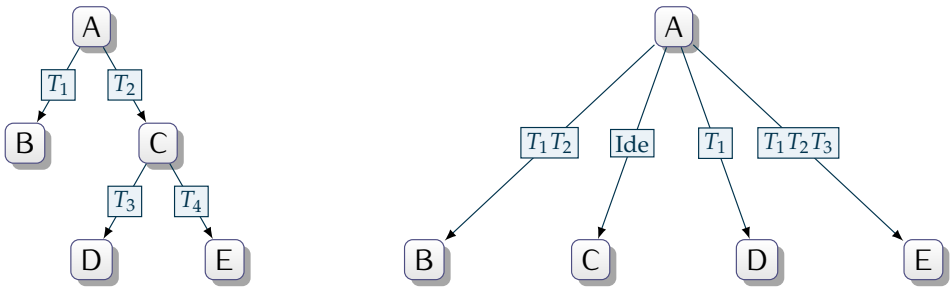
27

Usa la función del problema anterior para construir estas dos nuevas figuras, en las cuales hay un número variable de instancias de la figura original, dispuestas en circulo (vemos los ejemplos para 5 y 8 instancias, respectivamente).



28

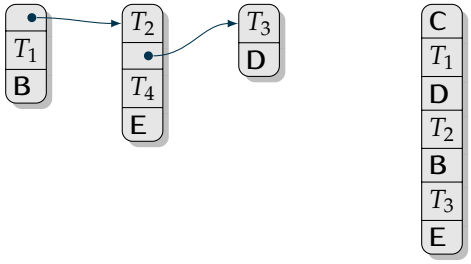
Dados los dos siguientes grafos de escena sencillos:



Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones: T_1,T_2 y T_3.

Respuesta:

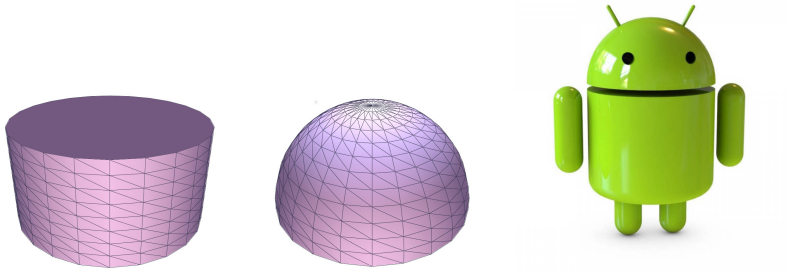
El grafo de la izquierda requiere tres nodos, se da una solución aquí (hay otra). El grafo de la derecha requiere únicamente un nodo, ya que las transformaciones son compuestas y no independientes



29

Supon que dispones de dos funciones para dibujar dos mallas u objetos simples: **`Semiesfera`** y **`Cilindro`**. La semiesfera (en coordenadas maestras) tiene radio unidad, centro en el origen y el eje vertical en el eje Y. Igualmente el cilindro tiene radio y altura unidad, el centro de la base está en el origen, y su eje es el eje Y. Con estas dos primitivas queremos construir la figura símbolo de Android (ver las figuras)

- Diseña el grafo de escena (tipo PHIGS) correspondiente, ten en cuenta que hay objetos compuestos que se pueden instanciar más de una vez (cada brazo o pierna se puede construir con un objeto compuesto de dos semiesferas en los extremos de un cilindro).
- Escribe el código OpenGL para visualizarlo, usando transformaciones y push/pop de la matriz modelview. Engloba el código en una función C llamada **`Android`**.



`Cilindro` **`Semiesfera`** **`Android`**

30

Escribe una segunda versión del grafo de escena del problema anterior, de forma que las transformaciones estén parametrizadas por dos valores reales (α y β) que expresan el ángulo de rotación del brazo izquierdo y derecho (respectivamente), en torno al eje que pasa por los centros de las dos semiesferas superiores de los brazos. Asimismo, habrá otro parámetro (ϕ) que es el ángulo de rotación de la cabeza (completa: con los ojos y antenas) entorno al eje vertical que pasa por su centro (cuando estos ángulo valen 0, el androide está en reposo y tiene exactamente la forma de la figura del problema anterior).

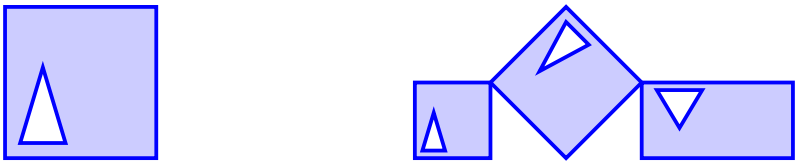
Escribe el código en C para visualizar el androide parametrizado de esta forma. Usa parámetros (valores reales) en las funciones que lo rquieran. En concreto, la función **`Android`** debe aceptar

como parámetros los tres ángulos.

31

Supón que dispones de una función llamada **`figura_simple`** que dibuja con OpenGL la figura que aparece a la izquierda (un cuadrado de lado unidad con la esquina inferior izquierda en el origen, con un triángulo inscrito). Esta función llama a **`glVertex`** para especificar las posiciones de los vértices de los polígonos de dicha figura.

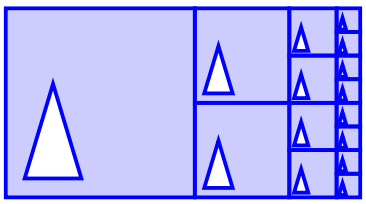
Usando exclusivamente llamadas a dicha función, construye otra función llamada **`figura_compleja`** que dibuja la figura de la derecha. Para lograrlo puedes usar manipulación de la pila de la matriz modelview (**`glPushMatrix`** y **`glPopMatrix`**), junto con **`glTranslatef`** y **`glScalef`**.



Respuesta:

32

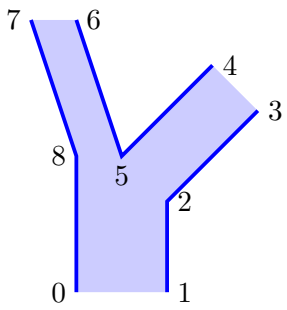
Usando de nuevo la función **`figura_simple`**, escribe otra función llamada **`figura_compleja_rec`**. Esta nueva función dibuja la figura que aparece aquí.



El código incluirá una única llamada a **`figura_simple`**, que se ejecutará 15 veces. Resuelve el problema con una función recursiva. **Respuesta:**

33

Escribe el código OpenGL de una función (llamada **`tronco`**) que dibuje la figura que aparece a aquí. El código dibujará el polígono relleno de color azul claro, y las aristas que aparecen de color azul oscuro (ten en cuenta que no todas las aristas del polígono relleno aparecen).



Índice de vértice	Coordenadas de vértice
0	$(+0.0, +0.0)$
1	$(+1.0, +0.0)$
2	$(+1.0, +1.0)$
3	$(+2.0, +2.0)$
4	$(+1.5, +2.5)$
5	$(+0.5, +1.5)$
6	$(+0.0, +3.0)$
7	$(-0.5, +3.0)$
8	$(+0.0, +1.5)$

A la derecha aparecen las coordenadas de los 9 vértices, empezando en el vértice en el extremo inferior izquierdo, y continuando en el sentido contrario a las agujas del reloj.

Respuesta:

Este código es simple, y basta con usar a las primitivas adecuadas de OpenGL, pasando los vértices con **`glVertex2f`**, por ejemplo. Solo hay un par de detalles no obvios a tener en cuenta para implementarlo correctamente:

- Al dibujar las aristas con la misma profundidad que el polígono interior, ocurre el problema que se suele nombrar como *z-fighting*, que consiste en que, al hacerse eliminación de partes ocultas (EPO) con Z-buffer (y tener las líneas la misma Z que el relleno), las líneas pueden quedar por detras del relleno en algunos pixels o en todos y no verse bien.
  Para solucionarlo, se pueden dibujar las líneas con unz Z ligeramente más cercana al observador, o bien deshabilitar EPO por Z-buffer (con **`glDisable`**(**`GL_DEPTH_TEST`**)), y dubujar las líneas después del relleno.
- Otro problema aparece si se intenta visualizar el relleno usando un único polígono. En el estándar se indica explicitamente que el polígono debe ser convexo, pero este no lo es. Por tanto, se debe rellenar usando varios polígonos convexos que cubran todo el original. Por ejemplo, podemos usar un polígono con los vértices: 0,1,2,5,8, otro con: 5,2,3,4, y otro con: 7,8,0

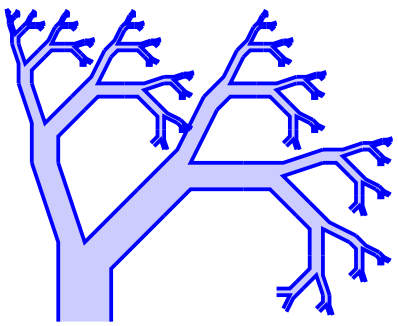
A continuación se incluye el código de la versión final:

```
void tronco()
{
   // relleno (tres polígonos)
   glColor3f(0.5,0.8,1.0);
   glBegin(GL_POLYGON);
      glVertex2f(0.0,0.0);   // 0
      glVertex2f(1.0,0.0);   // 1
      glVertex2f(1.0,1.0);   // 2
      glVertex2f(0.5,1.5);   // 5
      glVertex2f(0.0,1.5);   // 8
   glEnd();
   glBegin(GL_POLYGON);
      glVertex2f(0.5,1.5);   // 5
      glVertex2f(1.0,1.0);   // 2
      glVertex2f(2.0,2.0);   // 3
      glVertex2f(1.5,2.5);   // 4
   glEnd();
   glBegin(GL_POLYGON);
      glVertex2f(0.5,1.5);   // 5
      glVertex2f(0.0,3.0);   // 6
      glVertex2f(-0.5,3.0);  // 7
      glVertex2f(0.0,1.5);   // 8
   glEnd();
   ........
```

```
........
// aristas (tres polilíneas)
glColor3f(0,0,1);
glBegin(GL_LINE_STRIP);
   glVertex2f(1.0,0.0);  // 1
   glVertex2f(1.0,1.0);  // 2
   glVertex2f(2.0,2.0);  // 3
glEnd() ;
glBegin(GL_LINE_STRIP);
   glVertex2f(1.5,2.5);  // 4
   glVertex2f(0.5,1.5);  // 5
   glVertex2f(0.0,3.0);  // 6
glEnd() ;
glBegin(GL_LINE_STRIP);
   glVertex2f(-0.5,3.0); // 7
   glVertex2f(0.0,1.5);   // 8
   glVertex2f(0.0,0.0);   // 0
glEnd();
/ fin de 'tronco'
```

34

Usando el código del problema anterior, escribe una función **arbol** que, llamando a **tronco** (y manipulando la pila de matrices modelview), dibuje el árbol que aparece en la figura de la derecha.
En este árbol cada tronco conecta con dos copias del mismo más pequeñas situadas en sus dos ramas. El tronco raíz es semejante al del problema anterior. En total hay 63 troncos de 6 tamaños o niveles distintos. Diseña el código usando recursividad, de forma que el número de niveles sea un parámetro modificable en dicho código.



Respuesta:

El código recursivo dibuja el **tronco**, y después (si no se ha agotado el número de niveles), se llama a si mismo recursivamente para los dos subárboles izquierdo y derecho (el número de niveles restantes se reduce en una unidad).

Las dos transformaciones que se aplican a los sub-árboles son:

- sub-árbol derecho: el subárbol derecho debe ser semejante a un árbol completo, pero aplicando una transformación que lleve la arista 0-1 a coincidir con la arista 4-3. Dicha transformación se obtiene por composición:
  1. rotación de 45^o en sentido antihorario (es una rotación entorno al eje z, con un ángulo -45^o), no varía el vértice 0, ya que está en el origen.

2. escalado uniforme en X e Y, usando un factor que es la distancia entre los vértices 3 y 4, dividida por la distancia entre 0 y 1 (que es la unidad). La distancia entre 4 y 3 es $\sqrt{2}/2$, por tanto este es el factor de escala
3. traslación que lleva el vértice 0 hasta el vértice 4, por tanto es la traslación por el vector igual a las coordenadas de 4 (ya que 0 está en el origen), es decir $(1.5, 2.5, 0.0)$.

- sub-árbol izquierdo: en este caso, la arista 0-1 debe hacerse coincidir con la arista 7-6. para ello, usamos estas transformaciones:
  1. escalado igual a la distancia entre los vértices 7 y 6, que es $1/2$
  2. traslación que lleva el vértice 0 a vértice 7, es decir, traslación por las coordenadas del vértice 7, que son $(-0.5, 3.0)$

con todo esto, el código puede quedar así:

```
const GLfloat mr2 = 0.5*sqrtf(2.0);   // aprox. 0.7, mitad raiz de 2

void arbol( unsigned resto_niveles )
{
   tronco(); // dibujar el tronco central
   if ( resto_niveles > 0 ) // si quedan más niveles, dibujarlos:
   {  glPushMatrix();
        glTranslatef( 1.5, 2.5, 0.0 );
        glScalef( mr2, mr2, 1.0 );
        glRotatef( -45.0, 0.0, 0.0, 1.0 );
        arbol( resto_niveles-1 );   // sub-árbol a la derecha
      glPopMatrix();
      glPushMatrix();
        glTranslatef( -0.5, 3.0, 0.0 );
        glScalef( 0.5, 0.5, 0.5 );
        arbol( resto_niveles-1 ); // sub-árbol la izquierda
      glPopMatrix();
   }
}
```

35

Supongamos que se usa una proyección perspectiva, con un view-frustum determinado por las dos distancias (n y f) a los planos de recorte (con $0 < n < f$). La coordenada Z de un vértice es z_c en coordenadas de cámara y z_n en coordenadas normalizadas de dispositivo. Llamamos p a la función real de variable real tal que $z_n = p(z_c)$ (obviamente, la función p depende de f y n). Según los apuntes, la función p se puede definir como:

$$p(z) = \frac{-az-b}{-z} = a + \frac{b}{z} \qquad \text{donde:} \qquad a \equiv \frac{f+n}{f-n} \quad b \equiv \frac{2fn}{f-n}$$

Demuestra que se cumplen estas propiedades:

(a) La función p no está definida en $z = 0$
(b) Los valores a y b son ambos positivos y además $1 < a$.
(c) La función p es estrictamente decreciente.
(d) Se cumple $p(-f) = 1$ y $p(-n) = -1$ (es decir: transforma el intervalo $[-f, -n]$ en el intervalo $[-1, +1]$, invirtiendo el orden)
(e) Si $z < 0$ entonces $p(z) < a$ (y si $0 < z$, entonces $a < p(z)$)

Respuesta:

(a) La función p no está definida en $z = 0$:
Esto es evidente, puesto que la función es una fracción con z en el denominador.

(b) Los valores a y b son ambos positivos y además $1 < a$:
Tambien evidente, puesto que se cumple $0 < n < f$, y por tanto los valores $f + n$, $f - n$ y fn son todos positivos, como consecuencia también lo son a y b. Además, se cumple $f - n < f + n$ y por tanto $1 < (f + n)/(f - n) = a$.

(c) La función p es estrictamente decreciente :
La derivada (p') de la función p se puede expresar como sigue:

$$p'(z) = \left(\frac{b}{z}\right)' = \frac{-b}{z^2}$$

luego p será decreciente si su derivada p' es negativa para cualquier z, es decir, si se cumple $0 < b$. Por la pregunta anterior, sabemos que esto es cierto.

(d) Se cumple $p(-f) = 1$ y $p(-n) = -1$:
Para probar esto basta con sustituir z por $-f$ y $-n$ y verificar los resultados:

$$p(-f) = a - \frac{b}{f} = \frac{f+n}{f-n} - \frac{2fn}{f(f-n)} = \frac{f+n}{f-n} - \frac{2n}{f-n} = \frac{f-n}{f-n} = 1$$

$$p(-n) = a - \frac{b}{n} = \frac{f+n}{f-n} - \frac{2fn}{n(f-n)} = \frac{f+n}{f-n} - \frac{2f}{f-n} = \frac{n-f}{f-n} = -1$$

(e) Si $z < 0$ entonces $p(z) < a$:
También evidente, ya que si $z < 0$ se cumple $b/z < 0$ (ya que sabemos que $0 < b$), y por tanto el valor: $p(z) = a + b/z$ es estrictamente menor que a.

36

Demuestra que una recta en coordenadas de mundo se transforma en una recta en coordenadas de dispositivo, incluso aunque se use una proyección perspectiva.

Respuesta:

La transformación de puntos desde coordenadas de mundo hasta coordenadas de recortado es una transformación afín. Igualmente, es afín la transformación desde coordenadas normalizadas de dispositivo hasta coordenadas de dispositivo. Por tanto todas esas transformaciones, al ser afines, transforman lineas rectas en lineas rectas, por definición.

Solo nos quedaría por verificar la conservación de líneas para la transformación desde coordenadas de recortado a coordenadas normalizadas de dispositivo. Supongamos una recta formada por los puntos de coordenadas de recortado (x, y, z) tales que:

$$(x, y, z) \;=\; (o_x, o_y, o_z) \;+\; t(v_x, v_y, v_z)$$

Según los apuntes cada punto de dicha recta (con $z \neq 0$) es transformado en otro con coordenadas normalizadas de dispositivo (x', y', z'), en función de varias constantes a, b, c, d y f como se indica aquí:

$$x' \;=\; \frac{ax + bz}{-z} \qquad y' \;=\; \frac{cy + dz}{-z} \qquad z' \;=\; \frac{f + ez}{-z}$$

Si la recta original es paralela al plano $z = 0$ (es decir, si $v_z = 0$) entonces todos los puntos tienen la misma z, ya que $z = o_z$ independientemente de t, por tanto la división por la constante $-z$ produce una transformación que también es afín, y transforma líneas rectas en líneas rectas.

Por tanto, solo tenemos que verificar el caso de que $v_z \neq 0$. En ese caso la recta en coordenadas de recortado debe intersecar en algún punto con el plano $z = 0$. Por tanto, consideramos dicho punto como punto origen de la recta y mantenemos el vector director (v_x, v_y, v_z), o, lo que es lo mismo, asumimos, sin pérdida de generalidad, que la recta pasa por $(p_x, p_y, 0)$, donde:

$$p_x \;\equiv\; o_x \,-\, o_z \,\frac{v_x}{v_z} \qquad p_y \;\equiv\; o_y \,-\, o_z \,\frac{v_y}{v_z}$$

Por tanto, ahora podemos reescribir las expresiones del punto transformado expandiendo x,y y z según la ecuación de la recta (que ahora pasa por $(p_x, p_y, 0)$). Con la recta expresada de esta forma, el valor de t no puede ser 0, ya que no podemos proyectar el punto origen $(p_x, p_y, 0)$ al no poder

dividir por su z, que es nula. Las expresiones quedan así:

$$\begin{aligned} x' &= \frac{ax+bz}{-z} = \frac{a(p_x+tv_x)+btv_z}{-tv_z} = \left(\frac{av_x}{-v_z}-b\right) + \left(\frac{-1}{tv_z}\right)ap_x \\ y' &= \frac{cy+dz}{-z} = \frac{c(p_y+tv_y)+dtv_z}{-tv_z} = \left(\frac{cv_y}{-v_z}-d\right) + \left(\frac{-1}{tv_z}\right)cp_y \\ z' &= \frac{f+ez}{-z} = \frac{etv_z+f}{-tv_z} = (-e) + \left(\frac{-1}{tv_z}\right)f \end{aligned}$$

Lo anterior es equivalente a:

$$(x', y,', z') = (o'_x, o'_y, o'_z) + f(t)(v'_x, v'_y, v'_z)$$

donde $f(t) \equiv -1/(tv_z)$ y :

$$\begin{aligned} o'_x &= \frac{av_x}{-v_z} - b & v'_x &= a\,p_x \\ o'_y &= \frac{cv_y}{-v_z} - d & v'_y &= c\,p_y \\ o'_z &= -e & v'_z &= f \end{aligned}$$

Esto demuestra que los puntos transformados están todos en una misma línea recta, la que pasa por (o'_x, o'_y, o'_z) y tiene como vector director (v'_x, v'_y, v'_z). Sin embargo, dado que f no es lineal en t, la transformación no es afín, y por tanto no se conserva el paralelismo entre las líneas rectas.

37

Escribe las ecuaciones implícitas de los 6 planos que delimitan el *view-frustum* (para el caso de la proyección perspectiva), en el espacio (4D) de las coordenadas de recortado (CC) $(x_{cc}, y_{cc}, z_{cc}, w_{cc})$. Para esto, usa las ecuaciones implícitas de esos mismos planos en el espacio de las coordenadas normalizadas de dispositivo (NDC) $(x_n, y_n, z_n, 1)$, y la relación que hay entre las coordenadas de recortado y las normalizadas de dispositivo.

Las ecuaciones implicitas en CC ¿ se puden generalizar a cualquier tipo de proyección (perspectiva u ortografica ? si crees que sí es posible, indica como)

Respuesta:

Las ecuaciones de los planos, en el espacio de coordenadas normalizadas de dispositivo, se deducen del hecho de que en ese espacio el view-frustum es un cubo de lado dos y centro en el origen. Por tanto, son las siguientes:

Plano	Ecuación
delantero	$z_n = -1$
trasero	$z_n = +1$
izquierdo	$x_n = -1$
derecho	$x_n = +1$
inferior	$y_n = -1$
superior	$y_n = +1$

Las coordenadas normalizadas de dispositivo se definen a partir de las de recortado según esta relación:

$$(x_n, y_n, z_n, 1) = \frac{1}{w_{cc}} (x_{cc}, y_{cc}, z_{cc}, w_{cc})$$

Por tanto, podemos simplemente sustituir las coords. normalizadas por sus expresiones en términos de las coordenadas de recortado. Por ejemplo, para el plano izquierdo tenemos $x_{cc}/w_{cc} = -1$, o, lo que es lo mismo, $x_{cc} + w_{cc} = 0$. Haciendo lo mismo para el resto de planos, deducimos:

Plano	Ecuación
delantero	$z_{cc} + w_{cc} = 0$
trasero	$z_{cc} - w_{cc} = 0$
izquierdo	$x_{cc} + w_{cc} = 0$
derecho	$x_{cc} - w_{cc} = 0$
inferior	$y_{cc} + w_{cc} = 0$
superior	$y_{cc} - w_{cc} = 0$

Respecto a la generalización para cualquier tipo de proyección (incluida la proyección ortográfica), vemos que en el caso de la proyección ortográfica el valor de w_{cc} es 1, ya que la matriz O siempre produce coordenadas W iguales a la unidad cuando se aplica a cualquier punto que ya tiene dicha coordenada W a 1, como es el caso de las coordenadas del ojo. Por tanto, las ecuaciones dadas para los planos siguen siendo válidas, ya que $w_{cc} = 1$ y las coordenadas NDC coinciden con las coordenadas CC, según la fórmula de arriba (se divide por la unidad).

38

Supongamos que queremos visualizar una escena que se encuentra totalmente incluida en un cubo de lado s y centrado en el punto (c_x, c_y, c_z). Queremos que la escena se visualize de forma que dicho cubo (de dibujarse) se visualizaría exactamente en el *viewport*, que suponemos que es cuadrado (igual ancho que alto, medido en pixels). Queremos usar una proyección ortográfica paralela al eje Z (el observador está en la rama positiva del eje Z, a distancia infinita).

(a) Describe razonadamente como deben ser las matrices de vista V y de proyección P para este caso.

(b) Escribir una función con esta declaración:

```
// asignar matrices M y P usando proyección ortográfica alineada
void AsignarMatrices_OA( const float c[3], const float s ) ;
```

que fije las matrices del cauce fijo de OpenGL, usando para ello llamadas a **gluLookAt** y **glOrtho**.

Respuesta:

Apartado (a)

Por simplicidad, vamos a suponer que el origen del sistema de referencia de la cámara lo situamos en (c_x, c_y, c_z), con sus ejes paralelos a los ejes del marco de coordenadas del mundo.

En estas condiciones, la matriz de vista V es simplemente una traslación:

$$V \;=\; \mathrm{D}[-c_x, -c_y, -c_z]$$

No son necesarias rotaciones dado que los ejes de los dos distemas de referencia son paralelos.

Respecto a la matriz de proyección, se trata de hacer simplemente una normalización, que nos escale el cubo de lado s (ahora centrado en el origen, ya que M lo ha trasladado), en un cubo de lado 2. Es decir, la proyección es simplemente un escalado uniforme con factor de escala igual a $2/s$, esto es:

$$P \;=\; \mathrm{E}[2/s] \;=\; \mathrm{E}[2/s, 2/s, 2/s]$$

Apartado (b)

Se puede fijar la matriz de vista V en el cauce fijo usando la función **gluLookAt**. Para ellos fijamos:

- el punto origen con coordenadas de mundo $\mathbf{o} = (c_x, c_y, c_z)$
- el punto de atención desplazado una unidad (en el eje Z negativo) desde **o**, es decir, tendrá unas coordenadas de mundo $\mathbf{a} = (c_x, c_y, c_z - 1)$
- el vector VUP tiene coordenadas de mundo $\mathbf{u} = (0, 1, 0)$ (alineado con el eje Y del m.cc. del mundo)

Para fijar la matriz de proyección se puede usar una llamada a **glOrtho**, en ese caso es necesario dar los valores de los 6 parámetros (l, r, t, b, n, f) que delimitan el view-frustum, que en este caso es un cubo. El parámetro l es $-s/2$, y r será $+s/2$, ya que el cubo está centrado en el origen (iguales valores tendrán el par t y b). Respecto a n y f, debemos de tener en cuenta que determinan el rango en Z del frustum, de forma que dicho frustum queda entre $-n$ y $-f$, con $n < f$. Por tanto, si queremos que en coordenadas de cámara el frustum quede entre $z = -n = +s/2$ (detrás del observador) y $z = -f = -s/2$ (delante del observador), debemos obviamente de hacer $n = -s/2$ y $f = s/2$. Así que las instruciones deben ser:

Por tanto, el código de la función puede ser el siguiente:

```
// asignar matrices V y P usando proyección ortográfica alineada
void AsignarMatrices_OA( const float c[3], const float s )
{
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity() ;
   const float ms = s/2.0 ;   // mitad del ancho del cubo
   glOrtho( -ms,+ms, -ms,+ms, -ms,+ms );

   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity() ;
   gluLookAt( c[X],c[Y],c[Z],   c[X],c[Y],c[Z]-1,  0.0,1.0,0.0 );
}
```

39

Repite el problema anterior, con los mismos datos, pero ahora para una proyección perspectiva (usando exclusivamente **gluLookAt** y **glFrustum**). En este caso, el observador está en la línea paralela a Z que pasa por el centro del cubo, a una distancia d (> 0) del plano de recorte delantero, en la dirección de la rama positiva del eje Z (es decir, la posición del observador es exactamente $(c_x, c_y, c_z + d)$). No es necesario que escribas las matrices, ahora basta con que se escriban las instrucciones para cambiar M y P en el cauce fijo, instruciones que forman el cuerpo de esta función:

```
// asignar matrices V y P usando proyección perspectiva alineada:
void AsignarMatrices_PA( const float c[3], const float s, const float d ) ;
```

Respuesta:

La matriz de vista V se fija igual que en el problema anterior, pero ahora el origen del m.cc. de cámara (**o**) ya no está en (c_x, c_y, c_z), sino desplazado una distancia desde ese punto hacia la parte positiva del eje Z. Dicha distancia debe ser igual a la suma de $s/2$ (la mitad del lado del cubo inscrito en el frustum) y d (la distancia que separa el plano de recorte delantero del la posición del observador). Por tanto, hacemos $\mathbf{o} = (c_x, c_y, c_z + s/2 + d)$.

El punto de atención lo podemos dejar en el centro del frustum, es decir ahora $\mathbf{a} = (c_x, c_y, c_z)$, y el vector VUP es igual que antes, $\mathbf{u} = (0.0, 1.0, 0.0)$, ya que los marcos de cámara y mundo siguen alineados.

Respecto a la matriz de proyección, ahora los valores de l, r, t y b coinciden con los de antes, pero los valores n y f no. El valor de n es igual a d (distancia del observador al plano de recorte delantero), y el valor de f debe ser igual a n más el lado del cubo l (ambos valores n y f deben ser ahora positivos, por ser la proy. perspectiva).

Con todo lo dicho, el código debe quedar así:

```
// asignar matrices V y P usando proyección perspectiva alineada:
void AsignarMatrices_PA( const float c[3], const float s, const float d )
{
   const float ms = s/2.0 ;   // ancho de la mitad del cubo

   glMatrixMode( GL_PROJECTION );
   glLoadIdentity() ;
   glFrustum( -ms,+ms, -ms,+ms, d,d+s );

   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity() ;
   gluLookAt( c[X],c[Y],c[Z]+d+ms, c[X],c[Y],c[Z],  0.0,1.0,0.0 );
}
```

40

Teniendo en cuenta el problema anterior, vamos a suponer ahora que queremos modificar los requisitos, añadiendo estos dos nuevos:

(a) El punto de atención sigue siendo el centro del frustum, pero ahora el vector VPN (vector de dirección desde dicho punto hacia el observador) no está alineado con el eje Z, sino que se expresa en coordenadas polares usando dos parámetros α (rotación en torno a Y) y β (rotación en torno a X). Cuando α y β valen ambas 0, dicho vector VPN coincide con el eje Z (igual que antes), pero si no es así, VPN se obtiene rotando dicho eje Z, primero β grados en torno al eje X y luego α grados en torno al eje Y (α y β se pueden interpretar como longitud y latitud, respectivamente).

(b) Ahora no suponemos que el viewport es cuadrado, sino que tiene una relación de aspecto (alto dividio por ancho, en pixels) igual a un valor r, en general no necesariamente igual a 1 (aunque siempre positivo). Los objetos deben proyectarse no deformados, de forma que el frustum debe ser el más pequeño posible que contenga el cubo completamente en su interior cuando α y β valgan ambias 0.0 (cuando no es así, puede que se recorten algunas esquinas del cubo).
Esto es parecido al problema del tema 1 relativo a ventanas no cuadradas, pero ahora lo que hacemos es ocupar todo el *viewport* (no cuadrado) usando un frustum no cuadrado (en el otro problema, usábamos un viewport cuadrado que no ocupa la ventana).

Escribe una nueva versión de la función para calcular las matrices, con esta declaración:

```
// asignar matrices V y P usando proyección perspectiva rotada con ratio arbitrario:
void AsignarMatrices_PRR( const float c[3], const float s, const float d,
                          const float a, const float b, const float r );
```

Respuesta:

Respecto a la matriz de vista, los puntos del modelo en primer lugar serán transformados usando una matriz de cambio de sistema de referencia: desde el s.r. del mundo hacia el s.r. de la cámara original (no rotada), esto se hace igual que en el problema anterior (con **gluLookAt** y los mismos parámetros). Después, cada punto debe ser rotado entorno al centro del view frustum, que en el sistema de referencia de la cámara original (no rotada) es el punto de coordenadas $(0,0,-d-s/2)$. Respecto de ese punto, se encadenan dos rotaciones: los puntos primero se rotan un ángulo $-\alpha$ entorno a X, y luego un ángulo $-\beta$ entorno a Y. La implementación se hace, por tanto, usando tralación seguida de dos rotaciones y finalmente dehacer la traslación.

Para la matriz de proyección debemos de considerar la razón de aspecto del viewport, para implementar el requisito hay que distinguir dos casos: si $r \leq 1$, el viewport es más ancho que alto, en ese caso el frustum se ajusta al cubo por el alto, y quedan dos bandas laterales a izquierda y derecha, fuera del cubo pero visibles en el viewport. Cuando $1 < r$, el viewport es más alto que ancho, en ese caso el frustum se ajusta al cubo por el ancho, y quedan dos bandas horizontales arriba y abajo, también fuera del cubo pero en el viewport.

Con todo esto, el código puede quedar así:

```
// asignar matrices M y P usando proyección perspectiva rotada con ratio arbitrario:
void AsignarMatrices_PRR( const float c[3], const float s, const float d,
                          const float a, const float b, const float r)
{
   const float
      ms = s/2.0 ,                    // ancho de la mitad del cubo
      rx = std::max( 1.0, 1.0/r ),    // vale 1 si se hace ajuste por ancho (si no, vale 1/r)
      ry = std::max( 1.0, r );        // vale 1 si se hace ajuste por alto (si no, vale r)

   glMatrixMode( GL_PROJECTION );
```

```
   glLoadIdentity() ;
   glFrustum( -ms*rx,+ms*rx, -ms*ry,+ms*ry, d,d+s );

   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity() ;

   // (2) cambio de sistema de referencia:
   // (desde el s.r. de la cámara original hacia el s.r. de la camara rotada:)
   glTranslatef( 0.0,0.0, -d-ms );// (4) traslación desde origen (z=0) hacia centro (z=-d-ms)
   glRotatef(  b, 1.0,0.0,0.0 );   // (3) rotación β eje X
   glRotatef( -a, 0.0,1.0,0.0 );   // (2) rotación −α eje Y
   glTranslatef( 0.0,0.0, d+ms );  // (1) traslación desde centro (z=-d-ms) hacia origen (z=0)

   // (1) cambio sistema de referencia:
   // (desde s.r. del mundo al s.r. de la cámara original (no rotada))
   gluLookAt( c[X],c[Y],c[Z]+d+ms,  c[X],c[Y],c[Z],  0.0,1.0,0.0 );
}
```

Otra posibilidad sería usar una única llamada a **gluLookAt** usando como coordenadas del punto de atención la tupla $\mathbf{a} = (c_x, c_y, c_z)$, calcular el punto del observador como $\mathbf{o} = \mathbf{a} + A(0,0,1,0)^t$, y calcular VUP como $\mathbf{u} = A(0,1,0,0)^t$, donde A es la matriz de rotación compuesta entorno a los eje X e Y:

$$A \;=\; \mathrm{R}[y,\alpha] \cdot \mathrm{R}[x,-\beta]$$

es decir:

```
// asignar matrices M y P usando proyección perspectiva rotada con ratio arbitrario:
void AsignarMatrices_PRR( const float c[3], const float s, const float d,
                          const float a, const float b, const float r)
{
   const float
      ms = s/2.0 ,                    // ancho de la mitad del cubo
      rx = std::max( 1.0, 1.0/r ),  // vale 1 si se hace ajuste por ancho (si no, vale 1/r)
      ry = std::max( 1.0, r );      // vale 1 si se hace ajuste por alto (si no, vale r)

   glMatrixMode( GL_PROJECTION );
   glLoadIdentity() ;
   glFrustum( -ms*rx,+ms*rx, -ms*ry,+ms*ry, d,d+s );

   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity() ;

   const Matriz4f
      mrot = MAT_Rotacion( a, 0,1,0 )*
             MAT_Rotacion( -b, 1,0,0 ); // matriz de rotacion A
   const Tupla4f
      aten = { c[X], c[Y], c[z], 1 },          // punto de atencion = c
      obs  = aten + mrot*Tupla4f( 0,0,1, 0),   // observador = aten. + A* ejeZ
      vup  = mrot*Tupla4f( 0,1,0, 0);          // vup = A * ejeY

   gluLookAt( obs[X],  obs[Y],  obs[Z],
              aten[X], aten[Y], aten[Z],
              vup[X],  vup[Y],  vup[Z]   );
}
```

41

El valor de *apertura de campo horizontal* (*horizontal field of view*, γ_x) es el ángulo entre los planos laterales del frustum (en proyección perspectiva), y la *apertura de campo vertical* (*vertical field of view*, γ_y) es el ángulo entre los planos superior e inferior. En los problemas anteriores, los valores de d y r determinan implíctamente los valores de γ_x y γ_y (estos ángulos cumplen $0 < \gamma < \pi$)

Teniendo esto en cuenta escribe una nueva versión de la función anterior, que no acepte d como parámetro, y que incluya un nuevo parámetro (γ_m), que es el mínimo de γ_x y γ_y. Si $r \leq 1$ (ajuste por alto), entonces γ_y debe ser el valor γ_m, mientras que si $1 < r$ (ajuste por ancho) entonces γ_x debe ser γ_m (el otro ángulo estará implicitamente determinado por el que es igual a γ_m).

```
// asignar matrices M y P usando proyección perspectiva rotada con ratio arbitrario y apertura de
// campo mínima (el parámetro g es γm).
void AsignarMatrices_PRRF( const float c[3], const float s, const float g,
                           const float a, const float b, const float r );
```

Respuesta:

El código será similar al del problema anterior, excepto que ahora calcularemos el valor de d antes de crear las matrices. Para ello, hay que tener en cuenta el triángulo rectángulo cuyos tres vértices son: el observador (**o**), el centro de la cara delantera del frustum (**p**), y el centro de una arista mayor de dicha cara delantera (**e**). En este triángulo, el ángulo entre $\mathbf{p}-\mathbf{o}$ y $\mathbf{e}-\mathbf{o}$ es $\gamma_m/2$. La longitud de $\mathbf{e}-\mathbf{p}$ (cateto opuesto) es conocida e igual a $s/2$, mientras que la longitud de $\mathbf{p}-\mathbf{o}$ (cateto contiguo) es d. Por tanto, sabemos que:

$$\tan\left(\frac{\gamma_m}{2}\right) \;=\; \frac{s/2}{d}$$

luego, obviamente, podemos despejar d de esta igualdad, y obtenemos:

$$d \;=\; \frac{s}{2\tan\left(\frac{\gamma_m}{2}\right)}$$

Asi que el código puede quedar como sigue:

```
// asignar matrices M y P usando proyección perspectiva rotada con ratio arbitrario y apertura de
// campo mínima (el parámetro g es γm).
void AsignarMatrices_PRRF( const float c[3], const float s, const float g,
                           const float a, const float b, const float r )
{
   const float
      ms = s/2.0 ,                    // ancho de la mitad del cubo
      rx = std::max( 1.0, 1.0/r ),    // vale 1 si se hace ajuste por ancho (si no, vale 1/r)
      ry = std::max( 1.0, r ),        // vale 1 si se hace ajuste por alto (si no, vale r)
      d  = ms/tan(g/2.0) ;            // calcular d a partir de s y γ

   glMatrixMode( GL_PROJECTION );
   glLoadIdentity() ;
   glFrustum( -ms*rx,+ms*rx, -ms*ry,+ms*ry, d,d+s );

   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity() ;
```

```
    // (2) cambio de sistema de referencia:
    // (desde el s.r. de la cámara original hacia el s.r. de la camara rotada:)
    glTranslatef( 0.0,0.0, -d-ms );// (4) traslación desde origen (z=0) hacia centro (z=-d-ms)
    glRotatef(  b, 1.0,0.0,0.0 );   // (3) rotación β eje X
    glRotatef( -a, 0.0,1.0,0.0 );   // (2) rotación −α eje Y
    glTranslatef( 0.0,0.0, d+ms );  // (1) traslación desde centro (z=-d-ms) hacia origen (z=0)

    // (1) cambio sistema de referencia:
    // (desde s.r. del mundo al s.r. de la cámara original (no rotada))
    gluLookAt( c[X],c[Y],c[Z]+d+ms,   c[X],c[Y],c[Z],   0.0,1.0,0.0 );
}
```

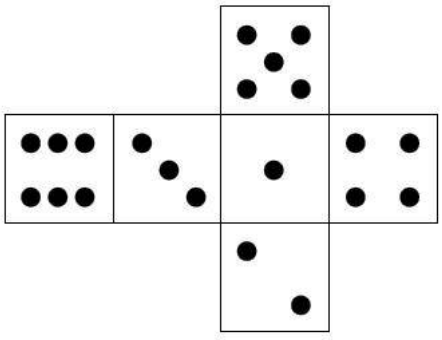
42

Supongamos de nuevo el problema anterior. En la solución adoptada, si la escena ocupa completamente el cubo, puede haber partes de la escena (cerca de las esquinas) las cuales, al rotar la vista, queden fuera del frustum, y por tanto no se vean. Por tanto, queremos una nueva versión de la función, con los mismos parámetros, pero ahora interpretamos s como el diámetro de una esfera centrada en (c_x, c_y, c_z), de forma que sabemos que ningún punto de la escena está a una distancia superior a $s/2$ del centro. Ahora queremos que el frustum sea el más pequeño posible que englobe a dicha esfera, de forma que al rotar la vista entorno al centro, ningún punto puede quedar fuera del frustum. En esta versión, los dos planos de ajuste del frustum son tangentes a la esfera, y también lo son las caras delantera y trasera de dicho frustum (ver figura).

Respuesta:

43

Supongamos que se desea crear una malla indexada (tabla de vértices y triángulos), para un cubo, de forma que deseamos aplicar un textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3 (es decir el ancho en pixels, dividido por el alto, es igual a $4/3$). La imagen aparece aquí:



Cuestiones:

(a) Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
(b) Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de textura, y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en $(1/2, 1/2, 1/2)$. Dibuja un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.

Respuesta:

Es necesario duplicar algunos vértices.....

44

Considera de nuevo el cubo y la textura del problema anterior. Ahora supón que queremos visualizar con OpenGL el cubo usando sombreado de Gouroud o de Phong, para lo cual debemos de asignar normales a los vértices. Responde a estas cuestiones

- Describe razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que has escrito en en el problema anterior, o es necesario usar otra tabla distinta.
- Si has respondido que no es posible usar las mismas tablas, escribe la nueva tabla de vértices, la nueva tabla de coordenadas de textura. Asimismo, escribe como sería la tabla de normales.

45

Considera un cubo (de nuevo de lado unidad, y con centro en $(1/2, 1/2, 1/2)$) que se quiere visualizar con una textura a partir de una única imagen (cuadrada) que se replicará en las 6 caras de dicho cubo. Asume que no se va a usar iluminación (no es necesario calcular la tabla de normales). Escribe ahora la tabla de coordenadas de vértices y la tabla de coordenadas de textura.

46

Suponemos que visualizamos una esfera de radio unidad centrada en el origen. Se ilumina con una fuente de luz puntual situada en el punto $\mathbf{p} = (0, 2, 0)$. El observador está situado en $\mathbf{o} = (2, 0, 0)$. En estas condiciones:

- Describe razonadamente en que punto de la superficie de la esfera el brillo será máximo si el material es puramente difuso ($M_D = (1, 1, 1)$, y el resto de reflectividades a 0) ¿ es ese punto visible para el observador ?
- Repite el razonamiento anterior asumiendo ahora que el material es puramente pseudo-especular ($M_S = (1, 1, 1)$, resto a cero). Indica si dicho punto es visible para el observador.

47

Supongamos que en una aplicación interactiva 2D se usa una llamada a 'glOrtho' para visualizar un rectángulo en coordenadas del mundo en el viewport. La llamada a 'glOrtho' usa estos valores como argumentos:

```
glOrtho( l, r, b, t, -1, 1 );
```

donde los argumentos (números reales) cumplen: $l < r$ y $b < t$.

Además, se ha hecho la siguiente llamada a 'glViewport'

```
glViewport( x_l, y_b, w, h ) ;
```

donde los argumentos son números enteros positivos (en unidades de pixels), y donde x_l e y_b son relativos a la esquina inferior izquierda de la ventana (la Y crece hacia arriba)

Suponemos ahora que usando un callback de 'glut' sabemos que el ratón ha sido pulsado en la posición (x_m, y_m), donde estos valores son enteros positivos, en unidades de pixels, pero ahora relativos a la esquina superior izquierda de la ventana (esto es frecuente en las implementaciones de 'glut', ahora la Y crece hacia arriba). La ventana tiene un alto, en pixels, igual a h_w.

Describe razonadamente como podemos calcular las coordenadas del punto donde se ha pulsado el ratón, en diversos sistemas de coordenadas:

- Coordenadas de dispositivo: x_d,y_d (se suponen relativas al viewport, es decir, valen $0, 0$ en la esquina inferior izquierda del viewport.
- Coordenadas normalizadas de dispositivo: x_{ndc},y_{ndc}
- Coordenadas de mundo x_w,y_w.

Respuesta:

- Las coordenadas de dispositivo coinciden con las que proporciona el sistema de ventanas, excepto por la inversiónm de la Y, y el hecho de que las queremos calcular relativas al viewport. Se calculan así:

$$\begin{aligned} x_d &= x_m - x_l \\ y_d &= h_w - y_b - y_m \end{aligned}$$

- Las coordenadas normalizadas de dispositivo se obtienen a partir de las anteriores, simplemente dividiendo por el ancho y el alto del viewport:

$$\begin{aligned} x_{ndc} &= x_d/w \\ y_{ndc} &= y_d/h \end{aligned}$$

- Finalmente, las coordenadas de mundo se obtienen también linealmente respecto de las anteriores:

$$\begin{aligned} x_w &= l + x_{ndc}(r-l) \\ y_w &= b + y_{ndc}(t-b) \end{aligned}$$

5. Realismo en rasterización. Ray-tracing..

48

Supongamos que un *rayo* (una semirecta en 3D) tiene como origen o extremo el punto **o**, y como vector de dirección **d** (normalizado). Además conocemos las coordenadas de los tres vértices de un triángulo arbitrario (las llamamos $\mathbf{v}_0$,$\mathbf{v}_1$ y $\mathbf{v}_2$). Supuesto que conocemos las coordenadas del mundo de **o**,**d**,$\mathbf{v}_0$,$\mathbf{v}_1$ y $\mathbf{v}_2$, describe razonadamente como sería una función para determinar si hay algún punto de intersección entre el rayo y el triángulo, y en caso afirmatico, calcular:

- La distancia t entre **o** y el punto de intersección ($0 < t$).
- Las coordenadas del punto de intersección (coordenadas del mundo).

Si el rayo está está incluido en el plano infinito Π que contiene al triángulo, se supone que no hay intersección. Para resolver este problema con un algoritmo sencillo, puedes usar estas recomendaciones:

- Considera un marco de coordenadas 3D que tenga el origen en un vértice del triángulo, cuyos ejes X e Y sean las dos aristas que parten de dicho vértice, y cuyo eje Z sea igual a la normal del triángulo (normalizada). Resuelve el problema en ese marco de coordenadas, es más fácil.
- Calcula primero el punto de intersección del rayo con el plano infinito Π que contiene al triángulo, si hay alguno, y en caso afirmativo, determina si dicho punto está en el triángulo o no lo está.

49

En el problema anterior, supón que conoces los tres vectores normales en los tres vértices del triángulo, llamemosles $\mathbf{n}_0$,$\mathbf{n}_1$ y $\mathbf{n}_2$. Describe como se calcularía la normal interpolada en el punto de intersección. Extiende tu razonamiento a cualquier tipo de tuplas con atributos de vértice, como pueden ser las coordenadas de textura, el color u otras.

50

Supongamos de nuevo un rayo que tiene como origen el vector **o**, y como vector de dirección **d** (normalizado). Queremos calcular si hay o no hay algún punto intersección entre dicho rayo y una esfera cuyo centro es **c** y cuyo radio es r. Describe razonadamente como sería el algoritmo para ello. En caso de que haya al menos una intersección, el algoritmo debe de producir:

- La distancia t desde **o** al punto de intersección más cercano a **o**
- Las coordenadas (del mundo) de dicho punto de intersección.

Se supone que si el rayo es tangente a la esfera, sí hay intersección entre ambos (en el punto de

tangencia).

Para hacerlo sencillo, considera el campo escalar F asociado a la esfera, tal que, para un punto **p**, se cumple:

$$F(\mathbf{p}) \begin{cases} < 0 & \text{si } \mathbf{p} \text{ está en el interior de la esfera} \\ = 0 & \text{si } \mathbf{p} \text{ está en la superficie de la esfera} \\ > 0 & \text{si } \mathbf{p} \text{ está en el exterior de la esfera} \end{cases}$$

Una vez que se conoce la expresión del campo escalar (en términos de las coordenadas del mundo de **p**), el problema de calcular la intersección se reduce a encontrar los puntos del rayo con valor 0 de dicho campo F.

51

Extiende tu solución al problema anterior para otros tipos de objetos, usando los mismos principios. En concreto, piensa en estos objetos:

- Un cono, con radio de la base unidad, vértice en el origen, eje Y, y altura unidad (entre $y = 0$ e $y = 1$). el cono se considera como un objeto sólido, cerrado por un disco en su base.
- Un cilindro, con radio unidad, eje Y, y altura unidad (también entre $y = 0$ e $y = 1$). También se considera como un sólido, cerrado por dos discos en los extremos.

52

Extiende el algoritmo del problema anterior para conos y cilindros sólidos con radios y alturas arbitrarios, y con eje arbitrario. Extiende también el problema de una esfera al caso genérico de elipsoides de tamaño y orientación arbitrarios.

Para hacerlo sencillo, piensa una solución que use en cada caso el algoritmo ya descrito para cada tipo de objeto (algoritmo que solo funciona con tamaño, posición y orientación *normalizados*).

53

Describe razonadamente como extendrías todos los algoritmos para intersecciones de los problemas anteriores para poder producir no solo el punto de intersección más cercano (si lo hay), sino también el vector normal a dicho punto (normalizado, y apuntando hacia el exterior del objeto).

54

Describe razonadamente cuales son los órdenes de complejidad en tiempo del algoritmo de visualización por rasterización y del algoritmo de visualización por ray-tracing, con estos supuestos:

- Todas las primitivas son triángulos.
- Estos órdenes se expresan en función del número de pixels p y del número de triángulos de

la escena n.

- La rasterización de triángulos conlleva un tiempo fijo por triángulo que es despreciable frente al tiempo que conlleva dicha rasterización por cada pixel donde cada triángulo es visible.
- El ray-tracing conlleva un tiempo fijo por pixel que es despreciable frente al tiempo que conlleva calcular si hay o no hay intersección entre un rayo y un triángulo.