

### Problema 1.9.

Demuestra que el producto escalar de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano como la suma del producto componente a componente, a partir de las propiedades que definen dicho producto escalar.

Sean  $\vec{a}$  en un marco cartesiano,  $\vec{a} = C(ax, ay, az, 0)^t$   
 $\vec{b} = C(bx, by, bz, 0)^t$

Propiedades del producto escalar:

- i) Comutativo:  $\vec{a}\vec{b} = \vec{b}\vec{a}$
- ii) Distributiva:  $\vec{a}(\vec{b} + \vec{c}) = \vec{a}\vec{b} + \vec{a}\vec{c}$
- iii)  $\vec{a}\vec{a} = |\vec{a}|^2$
- iv)  $\vec{a}\vec{b} = |\vec{a}||\vec{b}|\cos\theta$

$$\begin{aligned}\vec{a} &= ax\hat{i} + ay\hat{j} + az\hat{k} \\ \vec{b} &= bx\hat{i} + by\hat{j} + bz\hat{k}\end{aligned}$$

$$\begin{aligned}\hat{i}\hat{i} &= \hat{j}\hat{j} = \hat{k}\hat{k} = 1 \\ \hat{i}\hat{j} &= \hat{j}\hat{k} = \hat{k}\hat{i} = \hat{i}\hat{k} = \hat{k}\hat{j} = 0\end{aligned}$$

$$\begin{aligned}\vec{a}\vec{b} &= (ax\hat{i} + ay\hat{j} + az\hat{k})(bx\hat{i} + by\hat{j} + bz\hat{k}) = axbx(\hat{i}\hat{i}) + axby(\hat{i}\hat{j}) + axbz(\hat{i}\hat{k}) + \\ &\quad aybx(\hat{j}\hat{i}) + ayby(\hat{j}\hat{j}) + aybz(\hat{j}\hat{k}) + \\ &\quad azbx(\hat{k}\hat{i}) + azby(\hat{k}\hat{j}) + azbz(\hat{k}\hat{k}) \\ &= axbx + ayby + azbz\end{aligned}$$

### Problema 1.10.

Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se indica en la transparencia anterior, a partir de las propiedades que definen dicho producto vectorial.

Sean  $\vec{a} = C(ax, ay, az, 0)$ ,  $\vec{b} = C(bx, by, bz, 0)$

Propiedades producto vectorial:

- i)  $a \times b = -b \times a$
- ii)  $a \times (b+c) = a \times b + a \times c$
- iii)  $|a \times b| = |a||b|\sin\theta$

$$\begin{aligned}\vec{a} &= ax\hat{i} + ay\hat{j} + az\hat{k} \\ \vec{b} &= bx\hat{i} + by\hat{j} + bz\hat{k}\end{aligned}$$

$$\begin{aligned}a \times b &= \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ ax & ay & az \\ bx & by & bz \end{vmatrix} = \hat{i} \begin{vmatrix} ay & az \\ by & bz \end{vmatrix} - \hat{j} \begin{vmatrix} ax & az \\ bx & bz \end{vmatrix} + \hat{k} \begin{vmatrix} ax & ay \\ bx & by \end{vmatrix} = \\ &= (aybz - azby)\hat{i} - (axbz - azbx)\hat{j} + (axby - aybx)\hat{k}\end{aligned}$$

### Problema 1.11.

Demuestra que el producto vectorial de dos vectores es perpendicular a cada uno de esos dos vectores.

Tenemos que demostrar que  $(a \times b) \cdot a = 0$  y  $(a \times b) \cdot b = 0$

$$a(a \times b) = \det(a, a, b) = 0$$

dos filas iguales

$$b(a \times b) = \det(a, b, b) = 0$$

dos filas iguales

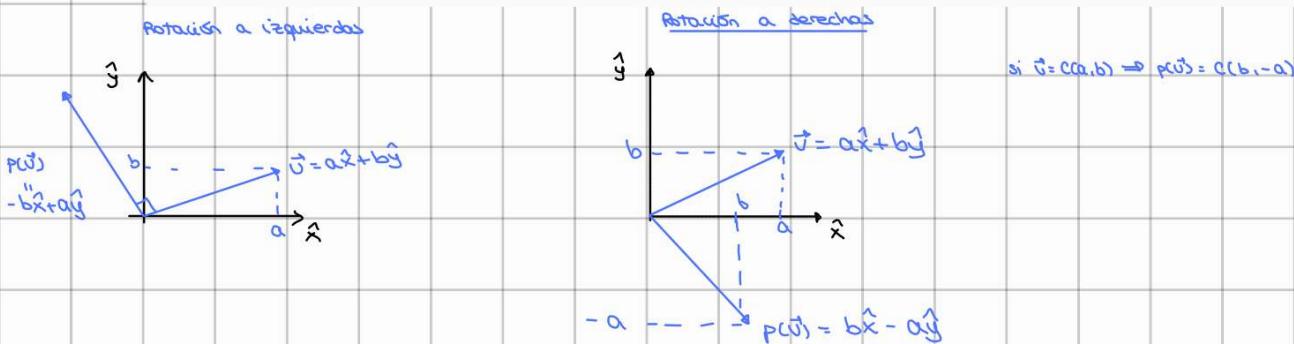
### Problema 2.7.

Demuestra que  $\vec{u}$  y  $P(\vec{u})$  son siempre perpendiculares según la definición anterior (es decir, siempre  $\vec{u} \cdot P(\vec{u}) = 0$ ).

$$\begin{aligned} \vec{u} \cdot P(\vec{u}) &= 0 & \vec{v} = c(a, b) & P(\vec{v}) = c(-b, a) \\ \vec{v} &= a\hat{x} + b\hat{y} & \Rightarrow \vec{v} \cdot P(\vec{v}) = (a\hat{x} + b\hat{y}) \cdot (-b\hat{x} + a\hat{y}) = -ab \frac{(a\hat{x})}{\hat{x}} + a^2 \frac{(a\hat{y})}{\hat{y}} - b^2 \frac{(b\hat{x})}{\hat{x}} + ba \frac{(b\hat{y})}{\hat{y}} = -ab + ba = -ab + ab \\ P(\vec{v}) &= -b\hat{x} + a\hat{y} & & = 0 \end{aligned}$$

### Problema 2.8.

Describe como se podría definir una rotación hacia la derecha (en el sentido de las agujas del reloj) en lugar de a izquierdas.



### Problema 2.9.

Demuestra que la transformación afín  $P$  (cuando se aplica a vectores, no a puntos) no depende del marco cartesiano  $C$  con respecto al cual expresamos las coordenadas  $(a, b)$  (en el caso de aplicarla a puntos, la rotación de  $90^\circ$  es entorno al punto origen  $o$  de  $C$ ).

$$\begin{aligned} \vec{v} &= c(a, b) = c'(a', b') & \text{Matriz de paso de } C \text{ a } C': \begin{pmatrix} a' \\ b' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \Rightarrow a' = a\cos\theta + b\sin\theta \\ P(\vec{v}) &= c(-b, a) & b' = -a\sin\theta + b\cos\theta \\ & \downarrow & P(\vec{v}') = c'(-b', a') = (-(-a\sin\theta + b\cos\theta), a\cos\theta + b\sin\theta) = (a\sin\theta - b\cos\theta, a\cos\theta + b\sin\theta) \\ \text{pasamos a } C' \rightarrow \begin{pmatrix} a' \\ b' \end{pmatrix} &= \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} -b \\ a \end{pmatrix} \Rightarrow a' = -b\cos\theta + a\sin\theta & \xrightarrow{\quad} \\ & & b' = b\sin\theta + a\cos\theta \end{aligned}$$

### Problema 2.10.

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo  $\theta$  y vectores  $\vec{a}$  y  $\vec{b}$  se cumple:

$$R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) = \vec{a} \cdot \vec{b}$$

(usa las coordenadas de  $\vec{a}$  y  $\vec{b}$  en un marco cartesiano cualquiera)

$$\text{e} \text{ d} \text{g} \text{lo}, \vec{a} \text{ y } \vec{b} \text{ vectores} \Rightarrow R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) = \vec{a} \cdot \vec{b} = (ax_1^2 + ay_1^2)(bx_2^2 + by_2^2) = ax_1bx_2 + ay_1by_2$$

$$\vec{a} = c(x_a, y_a) \quad \vec{b} = c(x_b, y_b)$$

matriz de rotación:  $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$

$$R_\theta(\vec{a}) = (\vec{a}_x \cos\theta - \vec{a}_y \sin\theta, \vec{a}_x \sin\theta + \vec{a}_y \cos\theta)$$

$$R_\theta(\vec{b}) = (\vec{b}_x \cos\theta - \vec{b}_y \sin\theta, \vec{b}_x \sin\theta + \vec{b}_y \cos\theta)$$

$$\begin{aligned} R_\theta(\vec{a}) R_\theta(\vec{b}) &= (\vec{a}_x \cos\theta - \vec{a}_y \sin\theta)(\vec{b}_x \cos\theta - \vec{b}_y \sin\theta) + (\vec{a}_x \sin\theta + \vec{a}_y \cos\theta)(\vec{b}_x \sin\theta + \vec{b}_y \cos\theta) \\ &= \vec{a}_x \vec{b}_x \cos^2\theta - \vec{a}_x \vec{b}_y \cos\theta \sin\theta - \vec{a}_y \vec{b}_x \cos\theta \sin\theta + \vec{a}_y \vec{b}_y \sin^2\theta + \vec{a}_x \vec{b}_x \sin\theta \cos\theta + \vec{a}_x \vec{b}_y \cos\theta \cos\theta + \vec{a}_y \vec{b}_x \sin\theta \cos\theta + \vec{a}_y \vec{b}_y \cos^2\theta \\ &= \vec{a}_x \vec{b}_x (\cos^2\theta + \sin^2\theta) + \vec{a}_y \vec{b}_y (\sin^2\theta + \cos^2\theta) \\ &\stackrel{!}{=} \vec{a}_x \vec{b}_x + \vec{a}_y \vec{b}_y = \vec{a} \cdot \vec{b} \end{aligned}$$

### Problema 2.11.

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo  $\theta$  y vector  $\vec{v}$ , se cumple:

$$\|R_\theta(\vec{v})\| = \|\vec{v}\|$$

$$\text{Si } \vec{v} \text{ es un vector, } \vec{v} \Rightarrow \|R_\theta(\vec{v})\| = \|\vec{v}\|$$

$$\vec{v} = c(x, y)$$

$$R_\theta(\vec{v}) = (\cos\theta x - \sin\theta y, \sin\theta x + \cos\theta y)$$

$$\therefore \|\vec{v}\| = \sqrt{x^2 + y^2}$$

$$\begin{aligned} \therefore \|R_\theta(\vec{v})\| &= \sqrt{(\cos\theta x - \sin\theta y)^2 + (\sin\theta x + \cos\theta y)^2} = \sqrt{\cos^2\theta x^2 + \sin^2\theta y^2 - 2\cos\theta \sin\theta xy + \sin^2\theta x^2 + \cos^2\theta y^2 + 2\sin\theta \cos\theta xy} \\ &= \sqrt{x^2 (\cos^2\theta + \sin^2\theta) + y^2 (\sin^2\theta + \cos^2\theta)} = \sqrt{x^2 + y^2} \end{aligned}$$

### Problema 2.12.

Demuestra que el producto escalar de vectores en 3D es invariante por rotaciones elementales (usa tu solución al problema 10)

$$\text{Problema 10: es cierto en 2D, } R_\theta(\vec{a}) R_\theta(\vec{b}) = \vec{a} \cdot \vec{b}$$

• Rotación alrededor del eje  $z$ : es una rotación en  $xy$  y  $z$  se queda fijo

$$R_\theta(\vec{a}) R_\theta(\vec{b}) = \vec{a}_x \vec{b}_x + \vec{a}_y \vec{b}_y + \vec{a}_z \vec{b}_z = \vec{a} \cdot \vec{b}$$

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Igual con el resto de ejes

### Problema 2.13.

Demuestra que las rotaciones elementales en 3D no modifican la longitud de un vector (usa tu solución al problema 11)

$$\text{Problema 11: } \|R_\theta(\vec{v})\| = \|\vec{v}\|$$

• Rotación alrededor del eje  $z \rightarrow z$  no cambia

$$\|R_\theta(\vec{v})\| = \sqrt{(\vec{v}_x \cos\theta - \vec{v}_y \sin\theta)^2 + (\vec{v}_x \sin\theta + \vec{v}_y \cos\theta)^2 + \vec{v}_z^2} = \sqrt{x^2 + y^2 + z^2} = \|\vec{v}\|$$

Igual con los otros ejes

### Problema 2.14.

Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualesquier dos vectores  $\vec{a}$  y  $\vec{b}$  y un ángulo  $\theta$ , se cumple:

$$R_\theta(\vec{a} \times \vec{b}) = R_\theta(\vec{a}) \times R_\theta(\vec{b})$$

$$\vec{a}, \vec{b}, \theta : \text{Ro}(\vec{a} \times \vec{b}) = \text{Ro}(\vec{a}) \times \text{Ro}(\vec{b})$$

$$\vec{a} = (ax, ay, az), \vec{b} = (bx, by, bz)$$

$$\text{Sup } \text{Ro}(c\vec{a}) = (ax', ay', az') \quad y \quad \text{Ro}(b\vec{a}) = (bx', by', bz')$$

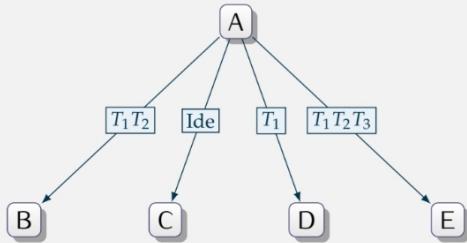
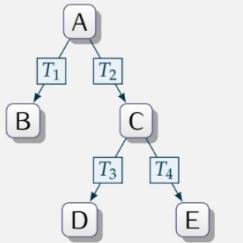
$$\vec{a} \times \vec{b} = \begin{pmatrix} aybz - azby \\ azbx - axbz \\ axby - aybx \end{pmatrix} \Rightarrow \text{Ro}(\vec{a} \times \vec{b}) = \text{Ro}\left(\begin{pmatrix} ay'bz' - az'by' \\ az'bx' - ax'bz' \\ ax'ybz' - ay'bx' \end{pmatrix}\right) \rightarrow R$$

$$\text{Ro}(\vec{a}) \times \text{Ro}(\vec{b}) = \begin{pmatrix} ax'by' - az'by' \\ az'bx' - ax'bz' \\ ax'ybz' - ay'bx' \end{pmatrix} \rightarrow R$$

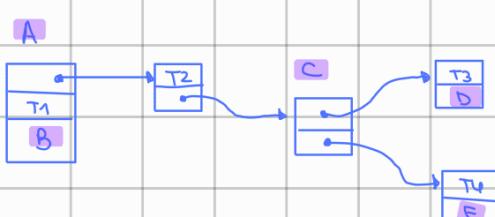
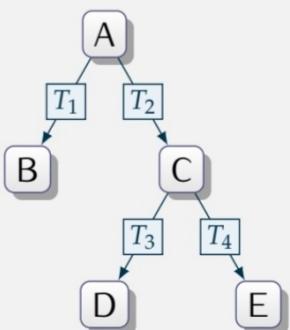
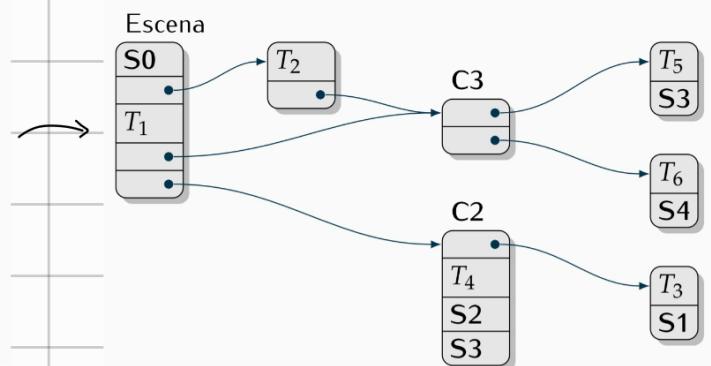
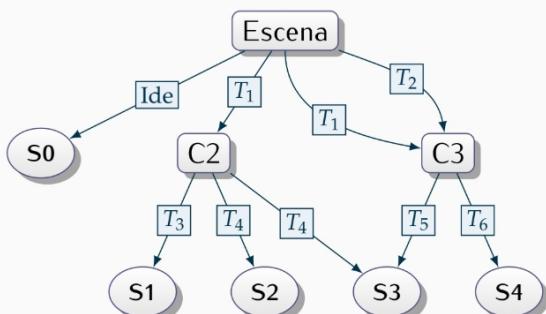
Como las rotaciones preservan las longitudes, ángulos y relaciones de perpendicularidad, A y B coinciden.

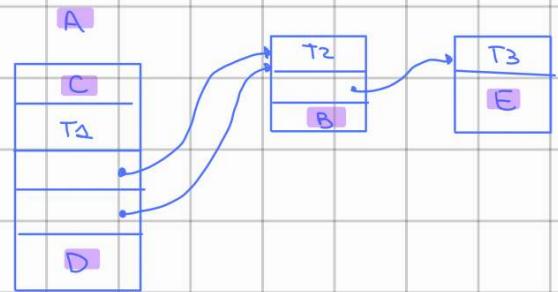
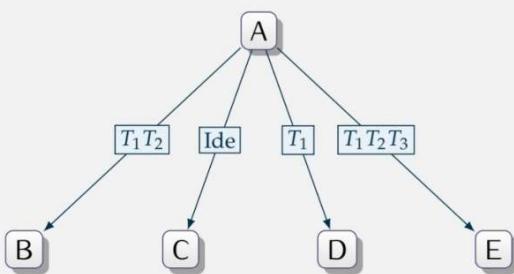
### Problema 2.19.

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  $T_3$ .



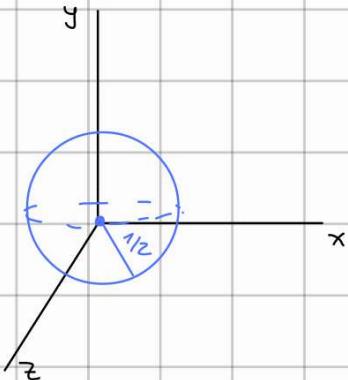


### Problema 2.1.

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).

(continua en la siguiente transparencia)

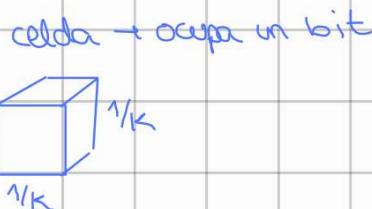
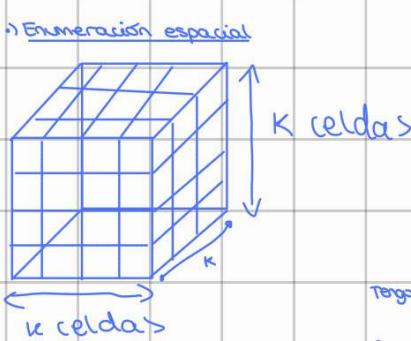


### Problema 2.1. (continuación)

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  $k = 1024$ ).

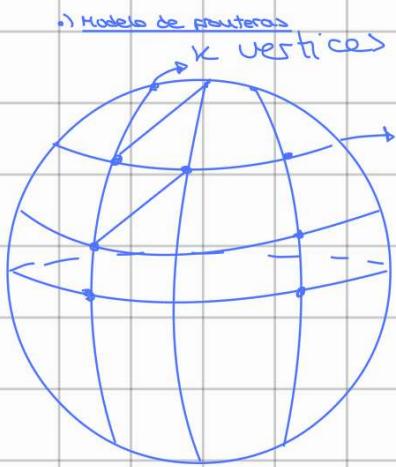


$$\text{Tengo } K \cdot K \cdot K \text{ celdas} \rightarrow K^3 \text{ bits}$$

Como un byte son 8 bits  $\rightarrow K^3 \text{ bits} \text{ son } \frac{K^3}{8} \text{ bytes} \Rightarrow TCK = \frac{K^3}{8} \text{ (redondeando hacia arriba)}$

$$k=16 \Rightarrow TCK = \frac{16^3}{8} = \frac{4096}{8} = 512 \text{ bytes} \cdot \frac{1 \text{ KB}}{1024 \text{ B}} = 0.5 \text{ KB}$$

$$k=1024 \Rightarrow TCK = \frac{1024^3}{8} = 134217728 \text{ B} \cdot \frac{1 \text{ MB}}{1048576 \text{ B}} = 128 \text{ MB}$$



$\leftarrow$  vértices

nº vértices totales  $\rightarrow k(k-2) + 2$   $\xrightarrow{\text{polar norte y sur}}$   
 sin costur  
 polo norte  
 y sur

nº triángulos  $\rightarrow$  Cada rectángulo  
 se divide en 2 triángulos

hay  $k-1$  segmentos en cada meridiano y paralelo  
 $\Rightarrow$  hay  $(k-1)^2$  rectángulos  
 $\Rightarrow$  hay  $2(k-1)^2$  triángulos

Un vértice es  $(x,y,z)$   $\Rightarrow$  ocupa  $3 \cdot 4 = 12B$   $\Rightarrow$  vértices ocupan  $12(k^2 - 2k + 2)$   
 un triángulo ocupa  $12B \Rightarrow$  triángulos ocupan  $24(k-1)^2$

$$\text{Tamaño total} \rightarrow 12k^2 - 24k + 24 + 24k^2 - 48k + 24 = 36k^2 - 72k + 48$$

$T(k) = 36k^2 - 72k + 48$  Bytes

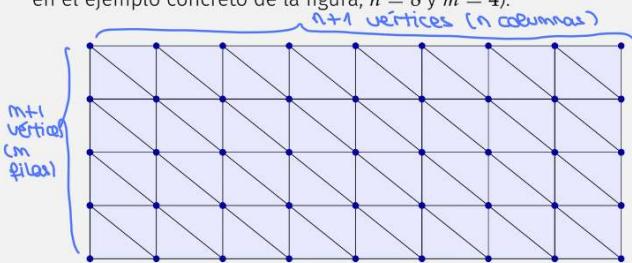
$$\text{si } k=16 \Rightarrow T(k) = 8142B \cdot \frac{1kB}{1024B} = 7.92 kB$$

$$\text{si } k=1024 \Rightarrow T(k) = 32675056B \cdot \frac{1MB}{1048576B} = 35.93 MB$$

Para  $k$  pequeños ocupa menos la primera y para  $k$  grandes ocupa menos la segunda.

### Problema 2.2.

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  $m = 4$ ).



(continua en la siguiente transparencia)

GIMI-GIANT informática y ciencias de datos en la era del big data. 13 de noviembre de 2020 - transparencia 6/64

### Problema 2.2. (continuación)

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

- Supongamos que un **float** ocupa 4 bytes (igual a 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  $n$ .
- Escribe el tamaño en KB suponiendo que  $m = n = 128$ .
- 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.

### Problema 2.3.

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:

(continua en la siguiente transparencia)

a)  $n_V$  vértices  $\rightarrow (n+1)(m+1)$

$$\downarrow (x,y) \rightarrow \text{ocupa } 2 \cdot 4 = 8B$$

$$Vértices \rightarrow 8(n+1)(m+1)B$$

nº triángulos  $\rightarrow 2nm$

$$\downarrow 3 \cdot 4B \rightarrow 3 \cdot 4 = 12B$$

$$Triángulos \rightarrow 24nmB$$

$T(m,n) = 8(n+1)(m+1) + 24nm$

b)  $m=n=128$

$$T(m,n) = 526344B = 514 KB$$

c) ¿Relación entre el  $n_V$  vértices  $n_V$  y  $n_C$ ?

$$n_V = (n+1)(m+1) = nm + n + m + 1$$

$$n_C = 2nm$$

para  $n$  y  $m$  muy grandes  $n_V \approx nm$   
 $n_C \approx 2nm$

$$\Rightarrow n_C \approx 2n_V$$

### Problema 2.3. (continuación)

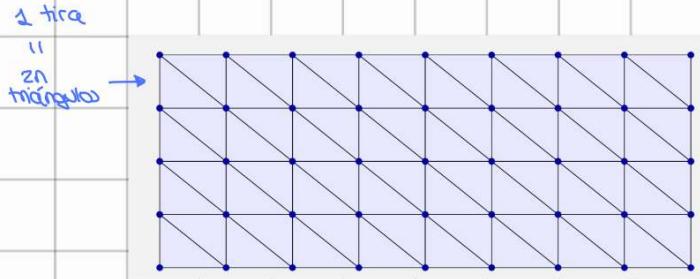
a) Indica que cantidad de memoria ocupa esta representación, en estos dos casos:

- Como función de  $n$  y  $m$ , en bytes.
- Suponiendo  $m = n = 128$ , en KB.

b) Para  $m$  y  $n$  grandes (es decir, cuando  $1/n$  y  $1/m$  son casi nulos), describe 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, describe 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.

*Cargar n vértices*



$$\text{TOTAL} \rightarrow 32n(2n+2) + 4 = 64mn + 64m + 4$$

$$\Rightarrow m=n=128 \rightarrow 4272688 = 417,264 \text{ KB}$$

$$c) nV_1 = (m+1)(m+1) = mn+m+n+1$$

$$nV_2 = (2n+2)m = 2mn+2m \rightarrow \text{del doble approx?}$$

n tiras (int)  $\rightarrow 48$   
 Tabla puertos  
 4+8mB  
 o) 11

Cada tira de  $2n$  triángulos necesita  $\rightarrow 3$  vértices para cada triángulo  $\rightarrow 3 + 2n - 1 = 2n + 2$  vértices  
 $n$  vértices totales  $\rightarrow (2n+2)m \cdot 3 \cdot 4$

$$b) T_1(m,n) = 8(m+1)(m+1) + 24nm = 8(mn+m+n+m+1) + 24nm \\ = 8mn + 8m + 8n + 8 + 24mn$$

para  $n$  y  $m$  grandes  $\rightarrow T_1(m,n) \approx 36mn$

$$T_2(m,n) = 26mn + 10m + 4$$

para  $n$  y  $m$  grandes  $\rightarrow T_2(m,n) \approx 26mn$

$$T_1 = 36 \text{ mn} \quad T_2 = 26 \text{ mn} \Rightarrow \frac{T_1}{T_2} = \frac{36}{26} = \frac{18}{13}$$

$$T_1 = \frac{18}{13} T_2$$

### Problema 2.4.

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:

$$n_A = 3(n_V - 2)$$

$$n_C = 2(n_V - 2)$$

(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).

$$\text{Fórmula de Euler para poliedros} \rightarrow n_V - n_A + n_C = 2$$

$$\text{a) Aristas y caras} \rightarrow 1 \text{ cara triángulo (3 aristas)} \\ \text{cada arista es compartida por 2 caras} \Rightarrow n_A = \frac{3n_C}{2}$$

$$\text{b) Aristas y vértices} \rightarrow \text{cada vértice conecta a 3 aristas en promedio} \\ \text{cada arista conecta a 2 vértices} \Rightarrow n_A = \frac{3n_V}{2}$$

$$n_V - n_A + n_C = 2 \Rightarrow n_V - \frac{3n_C}{2} + n_C = 2 \Rightarrow 2n_V - 3n_C + 2n_C = 4$$

$$2n_V - n_C = 4 \rightarrow n_C = 2n_V - 4 = 2(n_V - 2)$$

$$n_A = \frac{3n_C}{2} = \frac{3}{2}(2(n_V - 2)) = 3(n_V - 2)$$

### Problema 2.5.

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector **ari**, que en cada entrada tendrá una tupla de tipo **uvec2** (contiene dos **unsigned**) con 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.

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  $n$ .

(continua en la transparencia siguiente)

### Problema 2.5. (continuación)

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 aparecer 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.

### Problema 2.6.

Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función que acepta un puntero a una **MallaInd** y devuelve un número real (asumir que se dispone del tipo **vec3** y de los operadores usuales de tuplas o vectores, es decir suma +, resta -, producto escalar ., producto vectorial  $\times$ , módulo  $\| \cdot \|$ , etc ...).

Área total malla indexada

## EJERCICIOS TEMA 1

### Problema 1.1.

Crea una copia del repositorio **opengl3-minimo** y en esa copia escribe una función que genera una tabla de coordenadas de posición de vértices con las coordenadas de los vértices de un polígono regular de  $n$  lados o aristas (es una constante del programa), con centro en el origen y radio unidad.

Los vértices se almacenan como flotantes de doble precisión (**double**), con 2 coordenadas por vértice (usa una tabla de tipo **vector<dvec2>** para esto).

Adicionalmente, en esa función escribe el código que crea el correspondiente descriptor de VAO (**DescrVAO**) a partir de esta tabla (el vao queda guardado como una variable global del programa).

(el enunciado continua en la siguiente transparencia)

**APARTADO A** La tabla solo tiene a

```

void problema1 (const unsigned num)
{
    assert (glGetError () == GL_NO_ERROR);
    assert (num > 2);

    std::vector<glm::dvec2> vertices;
    for (unsigned int i = 0; i < num; ++i) {
        double angulo = 2.0 * M_PI * i / num,
               x = cos (angulo),
               y = sin (angulo);
        vertices.push_back (dvec2 (x, y));
    }
}

```

```
DescriVao VAO * vaoaristas = new DescriVBOAtribs (cource → und_atrib_posiciones, vertices  
assert (glGetError () == GL_NO_ERROR);
```

// visualizar aristas  
glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE)  
cauce → fijar color Plano (blue)  
cauce → fijar color (0, 0, 0)  
vaAristas → draw(GLline\_LOOP)

**APARTADO B** para cada segmento se da el punto inicial y el final

```

void problema1 (const unsigned num)
{
    assert (glGetError () == GL_NO_ERROR);
    assert (num > 2);

    std::vector<glm::dvec2> vertices;
    vertices.pushback (vec2 (cos 0, sin 0));
    for (unsigned int i = 1; i < num + 1) {
        double angulo = 2.0 * M_PI * i / num;
        double x = cos (angulo);
        double y = sin (angulo);
        vertices.pushback (dvec2 (x, y));
        vertices.pushback (dvec2 (x, y));
    }
}

```

vertices.pushback(vec2(cos 0, sin 0))

```
assert (glGetError () == GL_NO_ERROR);
```

// visualizar aristas

- glPolygonMode (GL\_FRONT\_AND\_BACK, GL\_LINE)
  - cauce → fijar color plano (true)
  - cauce → fijar color (0, 0, 0)
  - vista a tristos → draw(GL\_LINES)

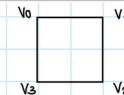
### Problema 1.1. (continuación)

El valor de  $n$  ( $> 2$ ) es un parámetro (un entero sin signo). El VAO sería la base para visualizar el polígono (únicamente las aristas), considerando la tabla de vértices como una **secuencia de vértices no indexada**.

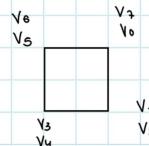
Escribe dos variantes del código, de forma que la tabla se debe diseñar para representar el polígono usando distintos tipos de primitivas:

- (a) tipo de primitiva **GL\_LINE\_LOOP**
  - (b) tipo de primitiva **GL\_LINES**.

En este problema y el siguiente se pide únicamente generar las tablas y el VAO, en ningún caso se pide visualizar nada.



índice vector posiciones



## Problema 1.2.

Escribe otra función para generar una tabla de vértices y una tabla de índices (y el correspondiente descriptor de VAO en una variable global), que permitiría visualizar el mismo polígono regular del problema anterior pero ahora como un conjunto de  $n$  triángulos iguales llenados que comparten un vértice en el centro del polígono (el origen). Usa ahora datos de simple precisión **float** para los vértices, con tres valores por vértice, siendo la Z igual a 0 en todos ellos.

La tabla está destinada a ser visualizada con el tipo de primitiva **GL\_TRIANGLES**.

Escribe dos variantes del código: una variante (a) usa una secuencia no indexada (con  $3n$  vértices), y otra (b) usa una secuencia indexada (sin vértices repetidos), con  $n + 1$  vértices y  $3n$  índices.

### APARTADO A

No indexada. La tabla tiene los 3 vértices de cada triángulo

```
void problema1.2 (const unsigned num) {
    assert (glGetError () == GL_NO_ERROR);
    assert (num > 2);
    std::vector<glm::vec3> vertices;
    double angulo_anterior = 0;

    for (unsigned int i=1; i < num_vertices; ++i)
        float angulo = 2.0 * M_PI * i / num_vertices;
        vertices.pushback (vec3(0,0));
        vertices.pushback (vec3 (cos (angulo_anterior), sen (angulo_anterior)));
        vertices.pushback (vec3 (cos (angulo), sen (angulo)));
        angulo_anterior = angulo;
    }

    DescriVAO VAO * vaosAristas = new DescriVBOAtribs (cauce → una_atrib_posiciones, índice, vector posiciones)
    assert (glGetError () == GL_NO_ERROR);

    // visualizar aristas
    glPolygonMode (GL_FRONT_AND_BACK, GL_LINE)
    cauce → fijar color Plano (hue)
    cauce → fijar color (0,0,0)
    vaosAristas → draw(GL_TRIANGLES)
```

### APARTADO B

Indexada. Como en las prácticas

```
void problema1.2 (const unsigned num) {
    assert (glGetError () == GL_NO_ERROR);
    assert (num > 2);
    std::vector<glm::vec3> vertices;
    vertices.pushback (vec3(0,0));
    for (unsigned int i=0; i < num_vertices; ++i)
        float angulo = 2.0 * M_PI * i / num_vertices;
        vertices.pushback (vec3 (cos (angulo), sen (angulo)));
    }

    for (unsigned int i=1; i < num)
        indices.pushback (0, i, (i+1)% (num+1))
```

DescriVAO VAO \* vaosAristas = new DescriVBOAtribs (cauce → una\_atrib\_posiciones, índice, vector posiciones)
vaosAristas → agregar (new DescriVBOInds (indices))

```
// visualizar aristas
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
cauce → fijar color Plano (hue);
cauce → fijar color (0,0,0);
vaosAristas → draw(GL_TRIANGLES);
```

### Problema 1.3.

Crea una copia del repositorio `opengl3-minimo`, y modifica el código de ese repositorio para incluir una nueva función para visualizar las aristas y el relleno del polígono regular de  $n$  lados (donde  $n$  es una constante de tu programa), usando los dos descriptores de VAO que se mencionan en

- ▶ el enunciado del problema 1.1 (variante (a), con `GL_LINE_LOOP`) para las aristas,
- ▶ el enunciado del problema 1.2 (variante (b), secuencia indexada) para el relleno.

(el enunciado continua en la siguiente transparencia)

### Problema 1.3. (continuación)

El polígono se verá relleno de un color plano y las aristas con otro color (también plano), distintos ambos del color de fondo. Debes usar un VAO para las aristas y otro para el relleno.

No uses una tabla de colores de vértices para este problema, en su lugar usa la función `glVertexAttrib` para cambiar el color antes de dibujar.

Incluye todo el código en una función de visualización (nueva), esa función debe incluir tanto la creación de tablas y VAOs (en la primera llamada), como la visualización (en todas las llamadas).

```
void problema1.3 (const unsigned num) {  
  
    assert (glGetError () == GL_NO_ERROR);  
    assert (num > 2)  
    std::vector<glm::vec2> vertices  
    for (unsigned int i=0; i < num +1) {  
        double angulo = 2.0 * M_PI * i/num;  
        double x = cos (angulo);  
        double y = sin (angulo);  
        vertices.pushback (vec2 (x,y));  
    }  
  
    std::vector<glm::vec2> vertices  
    vertices.pushback (vec2 (0,0))  
    for (unsigned int i=0; i < num-vertices; ++i)  
        float angulo = 2.0 * M_PI * i/num_<vertices;  
        vertices.pushback (vec2 (cos (angulo), sen (angulo)));  
    }  
  
    for (unsigned int i=1; i < num)  
        indices.pushback (0,i,(i+1)% (num+1))  
  
    DescriVAO * vaoAristas = new DescriVAO (0, new DescriVBO Atribs (cauce → ind_atrib - posiciones, posiciones)  
    vaoAristas → agregar (new DescriVBO Inds (índices));  
  
    DescriVAO * vaorelleno = new DescriVAO (0, new DescriVBO Atribs (cauce → ind_atrib - posiciones, posiciones)  
    vaorelleno → agregar (new DescriVBO Inds (índices));  
  
    //visualizar relleno  
    glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);  
    cauce → fijarColor Plano (true)  
    cauce → fijar Color (3 0.0, 0.0, 1.0); //usa glVertexAttrib  
    vaorelleno → draw (GL_TRIANGLES)  
  
    //visualizar aristas  
    glPolygonMode (GL_FRONT_AND_BACK, GL_LINE)  
    cauce → fijarColor Plano (true)  
    cauce → fijar Color (3 0.0, 0.0, 0.0); //usa glVertexAttrib  
    vaorelleno → draw (GL_LINES LOOP)  
}
```

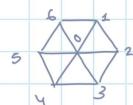
### Versión Usando directamente `glVertexAttrib`

```
// visualizar relleno  
glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);  
glVertexAttrib3f (cauce.ind_atrib_colores, 0,0,1);  
vaorelleno → draw (GL_TRIANGLES)  
  
// visualizar aristas  
glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);  
glVertexAttrib3f (cauce.ind_atrib_colores, 1,1,1);  
vaorelleno → draw (GL_LINES )
```

#### Problema 1.4.

Repite el problema anterior (problema 1.3), pero ahora intenta usar el mismo VAO para las aristas y los triángulos rellenos. Para eso puedes usar una única tabla de  $n + 1$  posiciones de vértices, esa tabla sirve de base para el relleno, usando índices. Para las aristas, puedes usar **GL\_LINE\_LOOP**, pero teniendo en cuenta únicamente los  $n$  vértices del polígono (sin usar el vértice en el origen).

Notas: puedo usar la misma tabla para relleno y aristas, hay que usar funciones OpenGL para decirle que no entrece en 0



```

void problema1_4 (const unsigned num)
{
    assert (n >= 2);
    assert (glGetError () == GL_NO_ERROR);
    std::vector<glm::vec2> vertices;
    std::vector<glm::ivec3> indices;

    vertices.pushback (vec2 (0,0));
    for (int i = 0; i < num; ++i)
        vertices.pushback (vec2 (cos (2 * M_PI * i / num), sin (2 * M_PI * i / num)));

    for (int i = 1; i < num; ++i)
        indices.pushback (0, i, (i + 1) % (num + 1));

    if (array == 0) {
        // Crear y activar VAO
        glGenVertexArrays (1, &array);
        glBindVertexArray (array);

        // Crear y activar VBO
        GLuint buffer;
        glGenBuffers (1, &buffer);
        glBindBuffer (GL_ARRAY_BUFFER, buffer);

        // Transferir datos
        glBufferData (GL_ARRAY_BUFFER, tot_size, data, GL_STATIC_DRAW);

        // Registrar formato de datos, dirección y memoria
        glVertexAttribPointer (cauce >> ind_atributos -> posiciones, 3, GL_FLOAT, GL_FALSE, 0, 0);

        // Activar atributo posiciones
        glEnableVertexAttribArray (cauce >> ind_atrib -> posiciones);

        // Crear y activar VBO
        GLuint bufferInd;
        glGenBuffers (1, &bufferInd);
        glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, bufferInd);
        glBufferData (GL_ELEMENT_ARRAY_BUFFER, tot_size, indices, GL_STATIC_DRAW);

        assert (glGetError == GL_NO_ERROR);
    }

    glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
    cauce >> FijarUsarColorPlano (true);
    cauce >> FijarColor (.1f, 0.0f, 0.0f);
    glEnable (GL_DEPTH_TEST);
    glDrawElements (GL_TRIANGLES, indices.size () * 3, GL_UNSIGNED_INT, 0);
    cauce >> FijarColor (.1f, 0.0f, 0.0f);

    glDrawArrays (GL_LINE_LOOP, 1, num);
    ↑           ↑           ↑ n: de vértices que queremos visualizar
    TYPE        índice del primero a visualizar
}

```

NOTAS : No tapar aristas con relleno . Opciones

- 1) desactivar eliminación de partes ocultas
- 2) dejar eliminación de partes ocultas . 2 del relleno más lejos del observador que las aristas

### Problema 1.5.

Repite el problema anterior (problema 1.4), pero ahora el relleno, en lugar de hacerse todo del mismo color plano, se hará mediante interpolación de colores (las aristas siguen estando con un único color). Para eso se debe generar una tabla de colores de vértices (tipo `vector<vec3>`), inicializada con colores aleatorios para cada uno de los  $n + 1$  vértices.

Ten en cuenta que para visualizar las aristas, debes de deshabilitar antes el uso de la tabla de colores.

```
void ejercicio_1_5 (const unsigned int num) {
    assert (glGetError == GL_NO_ERROR);
    assert (num > 2);

    std::vector<glm::vec3> posiciones;
    std::vector<glm::ivec3> indices;
    std::vector<glm::vec3> colores;

    posiciones.pushback ({0,0,0});
    colores.pushback ({0,0,0});

    for (unsigned int i = 0; i < num; ++i) {
        double angulo = 2 * M_PI * i / num;
        double x = cos (angulo);
        double y = sin (angulo);
        posiciones.pushback (vec3 (x,y,0));
        colores.pushback (vec3 (rand() % 2, rand() % 2, rand() % 2));
    }

    for (unsigned int i = 1; i < num; ++i)
        indices.pushback ({0, i, (i+1) % (num+1)});
```

//Crear VAO

```
DescrVAO * VAO_Ind = new DescrVAO (2, newDescrVBOAtribs (cause > Ind_Atrib_posiciones, vertices));
VAO_Ind > agregar (new DescrVBODAttribs (cause > Ind_Atrib_colores, colores));
VAO_Ind > agregar (new DescrVBODInds (GL_UNSIGNED_INT, num * 3, indices.data()));

glDisable (GL_DEPTH_TEST);
//Visualizar relleno:
glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
cause > fijarColorPlano (true);
glDrawElements (GL_TRIANGLES, num * 3, GL_UNSIGNED_INT, 0);
assert (glGetError () == GL_NO_ERROR)
```

//Visualizar aristas

```
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
VAO_Ind -> habilitarAtribs (cause > Ind_Atrib_colores, false);
cause > fijarColorPlano (false);
glDrawArrays (GL_LINE_LOOP, 1, num);
assert (glGetError () == GL_NO_ERROR);
```

### Problema 1.6.

Repite el problema anterior (problema 1.5), pero ahora, para guardar las posiciones y los vértices, se usará una estructura tipo AOS, es decir, se usa un array de estructuras o bloques de datos, en cada estructura o bloque se guardan 3 flotantes para la posición y 3 flotantes para el color RGB.

La estructura completa se debe alojar en un único VBO de atributos con todos los flotantes de las posiciones y colores, así que no uses la clase **DescrVAO**, sino que escribe el código OpenGL directamente.

```
struct VertexData {
    glm::vec3 posicion;
    glm::vec3 color
};

void ejercicio_1_6 (const unsigned num) {
    assert (glGetError () == GL_NO_ERROR);
    assert (num > 2);
    std::vector<VertexData> VectorDatos;
    VectorDatos.pushback ({glm::vec3 (0,0,0); glm::vec3 (0,0,0)});

    for (unsigned int i = 0; i < num; ++i) {
        double angulo = 2.0 * M_PI * i / num;
        double x = cos (angulo);
        double y = sin (angulo);
        VectorDatos.pushback ({glm::vec3 (x,y,0), glm::vec3 (rand()%2, rand()%2, rand()%2)});
    }

    //Creamos y activamos VAO
    GLuint vao;
    glGenVertexArrays (1, &vao);
    glBindVertexArray (vao);

    //Crear VBO, activar y pasar datos
    GLuint vbo;
    glGenBuffers (1, &vbo);
    glBindBuffer (GL_ARRAY_BUFFER, vbo);
    glBufferData (GL_ARRAY_BUFFER, VectorDatos.size() * sizeof (VertexData), VectorDatos.data(), GL_STATIC_DRAW);

    //Atributos posición y color
    glEnableVertexAttribArray (0);
    glEnableVertexAttribArray (1);

    //Registrar en el VAO activo los parámetros descriptores de ubicación de atributo
    glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, sizeof (VertexData), (void *) offset (VectorDatos, color));
    ↑           ↑          ↑
    índice     tamaño      tipo
    atributo   distancia entre atributos → desplazamiento
                           de un vértice y otro

    glEnable (GL_DEPTH_TEST);

    //relleno
    glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
    glDrawElements (GL_TRIANGLES, num * 3, GL_UNSIGNED_INT, 0);

    //restos
    glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
    glDrawArrays (GL_LINE_LOOP, 1, num);

    //Limpieza
    glBindVertexArray (0);
    glBindBuffer (GL_ARRAY_BUFFER, 0);
    glDeleteBuffer (1, &vbo);
    assert (glGetError () == GL_NO_ERROR)
```

### Problema 1.7.

Modifica el código del ejemplo `opengl3-minimo` para añadir a la clase **Cauce** un método que permita especificar la región visible, el método puede tener esta declaración:

```
void fijarRegionVisible( const float x0, const float x1,
                        const float y0, const float y1,
                        const float z0, const float z1 ) ;
```

El método debe de fijar el valor del parámetro uniform con la matriz de proyección, según lo visto en las transparencias anteriores.

```
void fijarRegionVisible (const float x0, const float x1,
                        const float y0, const float y1,
                        const float z0, const float z1) {  
  
    float sx = 2.0 / (x1 - x0)
    float sy = 2.0 / (y1 - y0)
    float sz = 2.0 / (z1 - z0)
    float cx = (x0 + x1) / 2
    float cy = (y0 + y1) / 2
    float cz = (z0 + z1) / 2  
  
    const GLfloat matriz_proyeccion [16] = {sx, 0, 0, -cx * sx,
                                            0, sy, 0, -cy * sy,
                                            0, 0, sz, -cz * sz,
                                            0, 0, 0, 1
                                         };  
  
    glUniformMatrix4f (loc_proyeccion, 1, GL_TRUE, matriz_proyeccion);}
```

### Problema 1.8.

Modifica el código del ejemplo **opengl3-minimo** para que no se introduzcan deformaciones cuando la ventana se redimensiona y el alto queda distinto del ancho. El código original del repositorio presenta los objetos deformados (escalados con distinta escala en vertical y horizontal) cuando la ventana no es cuadrada, ya que visualiza en el viewport (no cuadrado) una cara (cuadrada) del cubo de lado 2.

Para evitar este problema, en cada cuadro se deben de leer las variables que contienen el tamaño actual de la ventana y en función de esas variables modificar la zona visible, que ya no será siempre un cubo de lado 2 unidades, sino que será un ortoedro que contendrá dicho cubo de lado 2, pero tendrá unas dimensiones superiores a 2 (en X o en Y, no en ambas), adaptadas a las proporciones de la ventana (el ancho en X dividido por el alto en Y es un valor que debe coincidir en el ortoedro visible y en el viewport).



Idea: mapear cuadrado



) Si ancho ≤ alto

`glviewport (0,  $\frac{\text{alto} - \text{ancho}}{2}$ , ancho, ancho)`



) si ancho > alto

`glviewport ( $\frac{\text{ancho} - \text{alto}}{2}$ , 0, alto, alto)`



) caso general

$$m = \min \{ \text{ancho}, \text{alto} \}$$

`glviewport ( $\frac{\text{ancho} - m}{2}$ ,  $\frac{\text{alto} - m}{2}$ , m, m);`

### Problema 1.9.

Demuestra que el producto escalar de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano como la suma del producto componente a componente, a partir de las propiedades que definen dicho producto escalar.

$$\vec{U} = U_x \hat{x} + U_y \hat{y} + U_z \hat{z}$$

$$\vec{V} = V_x \hat{x} + V_y \hat{y} + V_z \hat{z}$$

$$\vec{U} \cdot \vec{V} = (U_x \hat{x} + U_y \hat{y} + U_z \hat{z}) \cdot (V_x \hat{x} + V_y \hat{y} + V_z \hat{z}) =$$

$$U_x V_x (\hat{x} \cdot \hat{x}) + U_y V_x (\hat{y} \cdot \hat{x}) + U_z V_x (\hat{z} \cdot \hat{x}) + U_x V_y (\hat{x} \cdot \hat{y}) + U_y V_y (\hat{y} \cdot \hat{y}) + U_z V_y (\hat{z} \cdot \hat{y}) +$$

$$U_x V_z (\hat{x} \cdot \hat{z}) + U_y V_z (\hat{y} \cdot \hat{z}) + U_z V_z (\hat{z} \cdot \hat{z}) = U_x V_x + U_y V_y + U_z V_z$$

### Problema 1.10.

Demuestra que el producto vectorial de dos vectores se puede calcular usando sus coordenadas en cualquier marco cartesiano según se indica en la transparencia anterior, a partir de las propiedades que definen dicho producto vectorial.

$$\text{Sean } \vec{U} = U_x \hat{x} + U_y \hat{y} + U_z \hat{z}$$

$$\vec{V} = V_x \hat{x} + V_y \hat{y} + V_z \hat{z}$$

$$\vec{U} \times \vec{V} = (U_x \hat{x} + U_y \hat{y} + U_z \hat{z}) \times (V_x \hat{x} + V_y \hat{y} + V_z \hat{z}) =$$

$$U_x V_x (\hat{x} \times \hat{x}) + U_y V_x (\hat{y} \times \hat{x}) + U_z V_x (\hat{z} \times \hat{x}) + U_x V_y (\hat{x} \times \hat{y}) + U_y V_y (\hat{y} \times \hat{y}) + U_z V_y (\hat{z} \times \hat{y}) +$$

$$U_x V_z (\hat{x} \times \hat{z}) + U_y V_z (\hat{y} \times \hat{z}) + U_z V_z (\hat{z} \times \hat{z}) =$$

$$(U_y U_z - U_z U_y) \hat{x} + (U_z V_x - U_x V_z) \hat{y} + (U_x V_y - U_y V_x) \hat{z}$$

### Problema 1.11.

Demuestra que el producto vectorial de dos vectores es perpendicular a cada uno de esos dos vectores.

$$\vec{U} = U_x \hat{x} + U_y \hat{y} + U_z \hat{z}$$

$$\vec{V} = V_x \hat{x} + V_y \hat{y} + V_z \hat{z}$$

$$\vec{U} \times \vec{V} = (U_y V_z - U_z V_y) \hat{x} + (U_z V_x - U_x V_z) \hat{y} + (U_x V_y - U_y V_x) \hat{z}$$

$$(\vec{U} \times \vec{V}) \cdot \vec{U} = (U_y V_z - U_z V_y) U_x + (U_z V_x - U_x V_z) U_y + (U_x V_y - U_y V_x) U_z =$$

$$\cancel{U_y V_z U_x} - \cancel{U_z V_y U_x} + \cancel{U_z V_x U_y} - \cancel{U_x V_z U_y} + \cancel{U_x V_y U_z} - \cancel{U_x V_y U_z} - \cancel{U_y V_x U_z} = 0$$

$$(\vec{U} \times \vec{V}) \cdot \vec{V} = (U_y V_z - U_z V_y) V_x + (U_z V_x - U_x V_z) V_y + (U_x V_y - U_y V_x) V_z =$$

$$\cancel{U_y V_z V_x} - \cancel{U_z V_y V_x} + \cancel{U_z V_x V_y} - \cancel{U_x V_z V_y} + \cancel{U_x V_y V_z} - \cancel{U_x V_y V_z} - \cancel{U_y V_x V_z} = 0$$

# PROBLEMAS TEMA 2

## Problema 2.1.

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).

(continua en la siguiente transparencia)

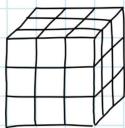
## Problema 2.1. (continuación)

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  $k = 1024$ ).

### Enumeración espacial



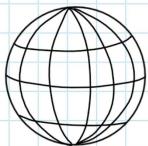
$$\rightarrow k \text{ celdas por lado} \rightarrow k^3 \text{ celdas}$$

$$\rightarrow \text{memoria : } Ce \cdot k^3 = k^3 \text{ bits} = \frac{k^3}{8} \text{ bytes}$$

$$\rightarrow k = 16 \rightarrow \frac{16^3}{8} \text{ bytes} = 512 \text{ bytes}$$

$$\rightarrow k = 1024 \rightarrow \frac{1024^3}{8} \text{ bytes} = 2^{28} \text{ bytes}$$

### Modelo de fronteras :



$$\rightarrow k^2 \text{ vértices}, 2k^2 \text{ triángulos}$$

$$\rightarrow \text{cada vértice } 3 \times 4 \text{ bytes} = 12 \text{ bytes}$$

$$\rightarrow \text{cada triángulo } 3 \times 4 = 12 \text{ bytes}$$

$$12k^2 + 24k^2 = 36k^2 \text{ bytes}$$

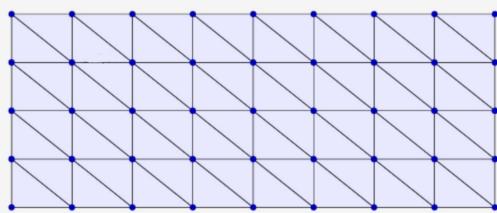
$$\rightarrow k = 16 \rightarrow 36 \cdot 16^2 \text{ bytes}$$

$$\rightarrow k = 1024 \rightarrow 36 \cdot 1024^2 \text{ bytes}$$

Conclusión: para  $k$  pequeños enumeración espacial. Cuando  $k$  grande, llega un punto en el que es más eficiente el de fronteras

### Problema 2.2.

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  $m = 4$ ).



### Problema 2.2. (continuación)

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

- Supongamos que un **float** ocupa 4 bytes (igual a 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  $n$ .
- Escribe el tamaño en KB suponiendo que  $m = n = 128$ .
- 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.

a) Hay  $2 \cdot n \cdot m$  triángulos :

$$\text{1 triángulo : } 3 \cdot 4 \text{ bytes} = 12 \text{ bytes}$$

$$\left. \begin{array}{l} \\ \end{array} \right\} 24 \cdot n \cdot m \text{ bytes}$$

Hay  $(n+1)(m+1)$  vértices

$$\text{1 vértice : } 3 \times 4 = 12 \text{ bytes}$$

$$\left. \begin{array}{l} \\ \end{array} \right\} 12(n+1)(m+1) \text{ bytes}$$

$$\left. \begin{array}{l} \\ \end{array} \right\} 24nm + 12(n+1)(m+1) \text{ bytes}$$

Tamaño total de la malla :  $24nm + 12(n+1)(m+1)$

b)  $m = n = 128$

$$\text{TAMAÑO : } 24 \cdot 128 \cdot 128 + 12 \cdot 129 \cdot 129 = 592908 \text{ bytes} = 592'908 \text{ KB}$$

$$c) \frac{n \text{ caras}}{n \text{ vértices}} = \frac{24(n+1)(m+1)}{12nm} \xrightarrow{n, m \rightarrow \infty} 2 \Rightarrow n_C = 2n_V$$

### Problema 2.3.

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:

(continua en la siguiente transparencia)

→ cada tira tiene  $2n$  triángulos . necesita  $2n + 2$  vértices

→ Cada vértice :  $3 \times 4$  bytes

→ tabla :  $4 + 8m$  bytes

$$\text{Memoria : } 4 + 8m + m(2n+2) \cdot 3 \cdot 4 = 4 + 8m + 24mn + 24m$$

$$n = m = 128 \rightarrow 397824 \text{ bytes} \approx 388,5 \text{ KB}$$

$$b) \frac{\text{tamaño malla indexada}}{\text{tamaño tiras triángulos}} = \frac{24nm + 12(n+1)(m+1)}{24nm + 32m + 4} \xrightarrow{n, m \rightarrow \infty} \frac{36}{24} = 1,5$$

Malla indexada ocupa 1.5 veces más

c) malla ind :  $(n+1)(m+1)$  vértices

tira: en el peor de los casos, vértices duplicados  $\Rightarrow (2n+2)2 \cdot m \rightarrow 4nm + 4m$  (4 veces más)

### Problema 2.3. (continuación)

a) Indica que cantidad de memoria ocupa esta representación, en estos dos casos:

- Como función de  $n$  y  $m$ , en bytes.
- Suponiendo  $m = n = 128$ , en KB.

b) Para  $m$  y  $n$  grandes (es decir, cuando  $1/n$  y  $1/m$  son casi nulos), describe 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, describe 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.

### Problema 2.4.

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).

$$n_A = \frac{3n_C}{2} = \frac{3(n_A + 2 - n_V)}{2} \Rightarrow 2n_A = 3n_A + 6 - 2n_V \Rightarrow -n_A = 6 - 2n_V \Rightarrow n_A = 3(n_V - 2)$$

$$n_C = n_A + 2 - n_V = \frac{3n_C}{2} + 2 - n_V \Rightarrow 2n_C = 3n_C + 4 - 2n_V \Rightarrow -n_C = 4 - 2n_V \Rightarrow n_C = 2(n_V - 2)$$

### Problema 2.5.

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector *ari*, que en cada entrada tendrá una tupla de tipo *uvec2* (contiene dos *unsigned*) con 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.

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  $n$ .

(continua en la transparencia siguiente)

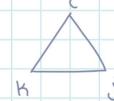
$$\text{Euler : } n_V + n_C = n_A + 2$$

• Todas las caras son triángulos

• Cada arista comparte la para 2 caras

$$\left. \begin{array}{l} 3n_C \\ \hline 2 \end{array} \right\} = n_A$$

Tabla de aristas: tiene los dos vértices que forman la arista.



aristas: ij, jk, ik

No usar contenedores. Restricción adicional: Sabemos que cada vértice está conectado a un  $n_i$  máximo de otros vértices

Caso 1 para cada entrada de la tabla de triángulos hay 3 posibles aristas. Para comprobar si hemos añadido ya la arista, usamos la matriz triangular (*pares*) donde cada fila corresponde a un vértice y contiene los vértices con mayor índice adyacentes a este y para los que ya hemos almacenado la arista. Complejidad  $O(n)$

```
void generarAri() {
    vector<vector<uvec2>> pares (n-1)
    int mayor, menor
    for (int i=0; i<ki.size(); ++i) {
        for (int j=0; j<3; ++j) {
            menor = min (tri[i][j], tri[i][j+1] % 3);
            mayor = max (tri[i][j], tri[i][j+1] % 3);
            if (find (pares[menor].begin(), pares[menor].end(), mayor) == pares[menor].end())
                ari.pushback ({menor, mayor}); //añadir arista
            pares[menor].pushback (mayor);
        }
    }
}
```

arista no almacenada

Caso 2: cada arista aparece 2 veces, en una nos la dan como (a,b) y en otra como (b,a). Solo la metemos una vez, cuando la primera componente es mayor que la segunda.

```
void generarAri() {
    int a,b;
    for (int i=0; i<tri.size(); ++i) {
        for (int j=0; j<3; j++) {
            a = tri[i][j];
            b = tri[i][(j+1)%3];
            if (a>b) {
                ari.pushback({a,b});
            }
        }
    }
}
```

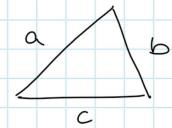
### Problema 2.6.

Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función que acepta un puntero a una

**MallaInd** y devuelve un número real (asumir que se dispone del tipo **vec3** y de los operadores usuales de tuplas o vectores, es decir suma +, resta -, producto escalar ., producto vectorial  $\times$ , módulo  $\| \cdot \|$ , etc ...).

propiedad del producto vectorial : Área del triángulo =

$$\frac{\|\vec{a} \times \vec{b}\|}{2}$$



```
double Area ( const mallaInd & malla ) {
    double area = 0;
    vec3 p, q, r, u, v;
    for (int i=0; i<malla.triangulos.size(); ++i) {
        // vértices
        p = malla.vertices[malla.triangulos[i][0]];
        q = malla.vertices[malla.triangulos[i][1]];
        r = malla.vertices[malla.triangulos[i][2]];
        // dos aristas
        u = q - p;
        v = r - p;
        a = sqrt(u.cross(v).length_sq());
        area += a / 2.0;
    }
    return area;
}
```

# EXAMEN

## Problema 2.7.

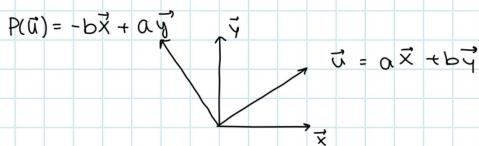
Demuestra que  $\vec{u}$  y  $P(\vec{u})$  son siempre perpendiculares según la definición anterior (es decir, siempre  $\vec{u} \cdot P(\vec{u}) = 0$ ).

$$\left. \begin{array}{l} \vec{u} = a\vec{x} + b\vec{y} \\ P(\vec{u}) = -b\vec{x} + a\vec{y} \end{array} \right\} \vec{u} \cdot P(\vec{u}) = (a\vec{x} + b\vec{y}) \cdot (-b\vec{x} + a\vec{y}) = -ab(\vec{x} \cdot \vec{x}) - bb(\vec{y} \cdot \vec{x}) + aa(\vec{x} \cdot \vec{y}) + ab(\vec{x} \cdot \vec{x}) = ab - ba = 0$$

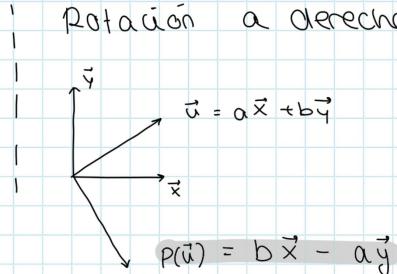
## Problema 2.8.

Describe como se podría definir una rotación hacia la derecha (en el sentido de las agujas del reloj) en lugar de a izquierdas.

Rotación a izquierdas:



Rotación a derechas



## Problema 2.9.

Demuestra que la transformación afín  $P$  (cuando se aplica a vectores, no a puntos) no depende del marco cartesiano  $C$  con respecto al cual expresamos las coordenadas  $(a, b)$  (en el caso de aplicarla a puntos, la rotación de  $90^\circ$  es entorno al punto origen  $o$  de  $C$ ).

Una transformación afín se puede expresar como  $P(v) = Av + b$ , donde  $A$  es una matriz y  $b$  un vector de traslación. Cuando aplicamos  $P$  a un vector  $v$ , obtenemos un nuevo vector  $v'$

Supongamos que tenemos dos marcos cartesianos  $C$  y  $C'$ . Las coordenadas de  $v$  en  $C$  son  $(a, b)$  y las coordenadas de  $v$  en  $C'$  son  $(a', b')$ . La relación entre ambas es una transformación lineal  $Tv_C = v_{C'}$ , donde  $T$  es la matriz de transformación de  $C$  a  $C'$

Aplicamos la transformación afín en el marco  $C$ :  $P(v) = Av + b$

Aplicamos la transformación afín en marco  $C'$ :  $P(Tv) = ATv + b$ . Observamos que la forma de la transformación no ha cambiado, solo que  $A$  se ha multiplicado por  $T \Rightarrow$  la transformación no depende del marco cartesiano

Caso matriz de rotación :  $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$

### Problema 2.10.

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo  $\theta$  y vectores  $\vec{a}$  y  $\vec{b}$  se cumple:

$$R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) = \vec{a} \cdot \vec{b}$$

(usa las coordenadas de  $\vec{a}$  y  $\vec{b}$  en un marco cartesiano cualquiera)

En la rotación se conservan las escalas y se conservan los ángulos

sean  $\vec{a}, \vec{b}$  dos vectores en el plano

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

$$\left\{ \begin{array}{l} \|R_\theta(\vec{a})\| = \|\vec{a}\| \\ \|R_\theta(\vec{b})\| = \|\vec{b}\| \\ \cos(R_\theta(\vec{a}), R_\theta(\vec{b})) = \cos(\vec{a}, \vec{b}) \end{array} \right.$$

$$\Rightarrow \vec{a} \cdot \vec{b} = \|\vec{a}\| \cdot \|\vec{b}\| \cdot \cos(\vec{a}, \vec{b}) = \|R_\theta(\vec{a})\| \|R_\theta(\vec{b})\| \cdot \cos(R_\theta(\vec{a}), R_\theta(\vec{b})) = \overrightarrow{R_\theta(\vec{a})} \cdot \overrightarrow{R_\theta(\vec{b})}$$

$$\begin{aligned} \vec{a} &= a_x \hat{x} + a_y \hat{y} \\ \vec{b} &= b_x \hat{x} + b_y \hat{y} \end{aligned} \quad \left\{ \quad \vec{a} \cdot \vec{b} = a_x b_x + a_y b_y \right.$$

$$\begin{aligned} R(\vec{a}) &= (a_x \cos(\theta) - a_y \sin(\theta)) \hat{x} + (a_y \cos(\theta) + a_x \sin(\theta)) \hat{y} \\ R(\vec{b}) &= (b_x \cos(\theta) - b_y \sin(\theta)) \hat{x} + (b_y \cos(\theta) + b_x \sin(\theta)) \hat{y} \end{aligned} \quad \left\{ \Rightarrow \right.$$

$$\begin{aligned} R(\vec{a}) \cdot R(\vec{b}) &= (a_x \cos(\theta) - a_y \sin(\theta)) (b_x \cos(\theta) - b_y \sin(\theta)) + (a_y \cos(\theta) + a_x \sin(\theta)) (b_y \cos(\theta) + b_x \sin(\theta)) = \\ &= \cancel{a_x b_x \cos^2 \theta} - \cancel{a_y b_x \sin \theta \cos \theta} - \cancel{a_x b_y \cos \theta \sin \theta} + \cancel{a_y b_y \sin^2 \theta} + \cancel{a_y b_x \cos^2 \theta} + \cancel{a_x b_y \sin \theta \cos \theta} + \cancel{a_y b_x \cos \theta \sin \theta} + \cancel{a_x b_x \sin^2 \theta} = \\ &= a_x b_x + a_y b_y \end{aligned}$$

### Problema 2.11.

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo  $\theta$  y vector  $\vec{v}$ , se cumple:

$$\|R_\theta(\vec{v})\| = \|\vec{v}\|$$

$$\text{sea } \vec{u} = a \hat{x} + b \hat{y} \quad \|\vec{u}\| = \sqrt{a^2 + b^2}$$

$$R_\theta(\vec{u}) = (a \cos(\theta) - b \sin(\theta), b \cos(\theta) + a \sin(\theta))$$

$$\begin{aligned} \|R_\theta(\vec{u})\| &= \sqrt{(a \cos(\theta) - b \sin(\theta))^2 + (b \cos(\theta) + a \sin(\theta))^2} = \sqrt{\cancel{a^2 \cos^2 \theta} + \cancel{b^2 \sin^2 \theta} - 2ab \sin \theta \cos \theta + \cancel{b^2 \cos^2 \theta} + \cancel{a^2 \sin^2 \theta} + 2ab \cos \theta \sin \theta} \\ &= \sqrt{a^2 + b^2} \end{aligned}$$

### Problema 2.12.

Demuestra que el producto escalar de vectores en 3D es invariante por rotaciones elementales (usa tu solución al problema 10)

$$\vec{a} = a_x \hat{x} + a_y \hat{y} + a_z \hat{z}$$

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$$

$$\vec{b} = b_x \hat{x} + b_y \hat{y} + b_z \hat{z}$$

$$\text{Rot}_{(x, \theta)}(\vec{a}) = (a_x, a_y \cos \theta - a_z \sin \theta, a_y \sin \theta + a_z \cos \theta)$$

$$\text{Rot}_{(x, \theta)}(\vec{b}) = (b_x, b_y \cos \theta - b_z \sin \theta, b_y \sin \theta + b_z \cos \theta)$$

$$\text{Rot}_{(x, \theta)}(\vec{a}) \cdot \text{Rot}_{(x, \theta)}(\vec{b}) = a_x b_x + (a_y \cos \theta - a_z \sin \theta)(b_y \cos \theta - b_z \sin \theta) + (a_y \sin \theta + a_z \cos \theta)(b_y \sin \theta + b_z \cos \theta) =$$

$$\cancel{a_x b_x} + \cancel{a_y b_y \cos^2 \theta} - \cancel{a_z b_z \sin \theta \cos \theta} - \cancel{a_y b_z \cos \theta \sin \theta} + \cancel{a_x b_z \sin^2 \theta} + \cancel{a_y b_y \sin^2 \theta} + \cancel{a_z b_x \cos \theta \cos \theta} + \cancel{a_y b_z \sin \theta \cos \theta} + \cancel{a_z b_z \cos^2 \theta} =$$

$$a_x b_x + a_y b_y + a_z b_z$$

$$\text{Rot}_{(y,\theta)}(\vec{a}) = (ax\cos\theta + az\sin\theta, ay, -ax\sin\theta + az\cos\theta)$$

$$\text{Rot}_{(y,\theta)}(\vec{b}) = (bx\cos\theta + bz\sin\theta, by, -bx\sin\theta + bz\cos\theta)$$

$$\begin{aligned} \text{Rot}_{(y,\theta)}(\vec{a}) \cdot \text{Rot}_{(y,\theta)}(\vec{b}) &= (ax\cos\theta + az\sin\theta)(bx\cos\theta + bz\sin\theta) + ayby + (-ax\sin\theta + az\cos\theta)(-bx\sin\theta + bz\cos\theta) = \\ &\cancel{axbx\cos^2\theta + azbx\sin\theta\cos\theta + axbz\sin\theta\cos\theta + azbz\sin^2\theta + ayby} + \cancel{axbx\sin^2\theta - azbx\sin\theta\cos\theta - azbz\sin\theta\cos\theta + azbz\cos^2\theta} = \\ &axbx + ayby + azbz \end{aligned}$$

$$\text{Rot}_{(z,\theta)}(\vec{a}) = (ax\cos\theta - ay\sin\theta, ax\sin\theta + ay\cos\theta, az)$$

$$\text{Rot}_{(z,\theta)}(\vec{b}) = (bx\cos\theta - by\sin\theta, bx\sin\theta + by\cos\theta, bz)$$

$$\begin{aligned} \text{Rot}_{(z,\theta)}(\vec{a}) \cdot \text{Rot}_{(z,\theta)}(\vec{b}) &= (ax\cos\theta - ay\sin\theta, ax\sin\theta + ay\cos\theta, az)(bx\cos\theta - by\sin\theta, bx\sin\theta + by\cos\theta, bz) = \\ &= (ax\cos\theta - ay\sin\theta)(bx\cos\theta - by\sin\theta) + (ax\sin\theta + ay\cos\theta)(bx\sin\theta + by\cos\theta) + azbz = \\ &\cancel{axax\cos^2\theta - aybx\sin\theta\cos\theta - axbz\sin\theta\cos\theta + ayby\sin^2\theta + \cancel{axbx\sin^2\theta + aybx\sin\theta\cos\theta + axbz\sin\theta\cos\theta + ayby\cos^2\theta + azbz}} = \\ &axbx + ayby + azbz \end{aligned}$$

#### Problema 2.14.

Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualquiera dos vectores  $\vec{a}$  y  $\vec{b}$  y un ángulo  $\theta$ , se cumple:

$$R_\theta(\vec{a} \times \vec{b}) = R_\theta(\vec{a}) \times R_\theta(\vec{b})$$

$$\rightarrow \vec{a} \times \vec{b} = \|a\| \cdot \|b\| \cdot \sin(a,b)$$

• Módulos se conservan por rotaciones

• Ángulos se conservan por rotaciones

$$\vec{a} \times \vec{b} = \|a\| \cdot \|b\| \cdot \sin(a,b) = \|R_\theta(a)\| \cdot \|R_\theta(b)\| \cdot \sin(R_\theta(a) \cdot R_\theta(b)) = R_\theta(a) \times R_\theta(b)$$

$$\vec{a} \times \vec{b} = (aybz - azby)\hat{x} + (azbx - axbz)\hat{y} + (axby - aybx)\hat{z}$$

$$R_\theta(\vec{a} \times \vec{b}) = \begin{pmatrix} (aybz - azby)\cos\theta - (azbx - axbz)\sin\theta \\ (aybz - azby)\sin\theta + (azbx - axbz)\cos\theta \\ axby - aybx \end{pmatrix}$$

$$R_\theta(\vec{a}) = (ax\cos\theta - ay\sin\theta)\hat{x} + (ay\cos\theta + ax\sin\theta)\hat{y} + bz\hat{z}$$

$$R_\theta(\vec{b}) = (bx\cos\theta - by\sin\theta)\hat{x} + (by\cos\theta + bx\sin\theta)\hat{y} + bz\hat{z}$$

$$R_\theta(\vec{a}) \times R_\theta(\vec{b}) = \begin{vmatrix} x & y & z \\ ax\cos\theta - ay\sin\theta & ay\cos\theta + ax\sin\theta & az \\ bx\cos\theta - by\sin\theta & by\cos\theta + bx\sin\theta & bz \end{vmatrix} =$$

$$\begin{aligned} &x(ay\cos\theta - ax\sin\theta)bz + (ax\cos\theta - ay\sin\theta)(by\cos\theta + bx\sin\theta)z + (bx\cos\theta - by\sin\theta)az = \\ &-(bx\cos\theta - by\sin\theta)(ay\cos\theta + ax\sin\theta)z - (ax\cos\theta - ay\sin\theta)bz - (by\cos\theta + bx\sin\theta)az = \end{aligned}$$

Primeras coordenadas

$$\rightarrow (aybz\cos\theta - azby\sin\theta) - (azbx\cos\theta + bzaz\sin\theta) = \checkmark$$

$$(aybz\cos\theta - azby\sin\theta) - (azbx\cos\theta + bzaz\sin\theta) =$$

Segunda coordenada:

$$(bxaz\cos\theta - byaz\sin\theta) - (axbz\cos\theta + bybz\sin\theta) = (bxaz - axbz)\cos\theta + (bybz - byaz)\sin\theta \checkmark$$

Tercera coordenada:

$$(ax\cos\theta - ay\sin\theta)(by\cos\theta + bx\sin\theta) - (bx\cos\theta - by\sin\theta)(ay\cos\theta + ax\sin\theta) =$$

$$\cancel{axby\cos^2\theta - ayby\sin\theta\cos\theta + axbx\sin\theta\cos\theta - aybx\sin^2\theta - bxay\cos^2\theta + byay\cos\theta\sin\theta - axbx\sin\theta\sin\theta + byax\sin^2\theta} =$$

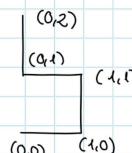
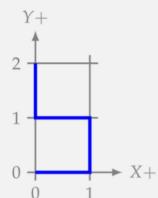
$$axby - aybx \quad \checkmark$$

### Problema 2.15.

Escribe una función llamada **gancho** para dibujar con OpenGL en modo diferido la polilínea de la figura (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 matriz *modelview*, el color y grosor del estado de OpenGL en el momento de la llamada (y no los cambia).

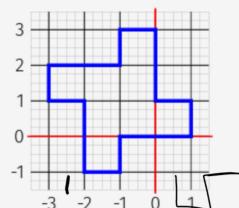
Usa la plantilla en el repositorio **opengl3-minimo** para esto.



```
void gancho () {
    vec3 vertices = {(0,0,0), (1,0,0), (1,1,0), (0,1,0), (0,2,0)};
    //generar y enlazar VAO
    glGenVertexArrays(1 & array)
    glBindVertexArray (array)
    //generar y enlazar VBO
    glGenBuffers(1 & buffer)
    glBindBuffer (GL_ARRAY_BUFFER, buffer)
    glBufferData (GL_ARRAY_BUFFER, sizeof(vertices), 3 * sizeof(float), GL_STATIC_DRAW)
    glEnableVertexAttribArray(0)
    glDrawArrays (GL_LINE_STRIP, 0, 4)
    glBindVertexArray (0)
}
```

### Problema 2.16.

Usando (exclusivamente) la función **gancho** del problema anterior, construye otra función (**gancho\_x4**) para dibujar con OpenGL, usando el **cauce fijo**, el polígono que aparece en la figura:



Usa exclusivamente **compMM**, **translate** y **rotate**.

```
void gancho_x4 () {
    glPushMatrix();
    //primer gancho
    gancho();
    //segundo gancho (rotado 90% y trasladado)
    glPushMatrix()
    glTranslate ((0,2,0))
    glRotate (PI/2, (0,0,1))
    gancho();
    glPopMatrix()
    //tercer gancho : rotación 180 y translación
    glPushMatrix()
    glTranslate (2,2,0);
    glRotate (PI, (0,0,1))
    gancho();
    glPopMatrix()
    //cuarto gancho rotación 270
    glPushMatrix()
    glTranslate (-2,0,0)
    glRotate (3PI/4, (0,0,1))
    gancho();
    glPopMatrix()
```

```
//tercer gancho : rotación 180 y translación
glPushMatrix()
glTranslate (2,2,0);
glRotate (PI, (0,0,1))
gancho();
glPopMatrix()
//cuarto gancho rotación 270
glPushMatrix()
glTranslate (-2,0,0)
glRotate (3PI/4, (0,0,1))
gancho();
glPopMatrix()
```

glPopMatrix()

### Problema 2.17.

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 distintos  $p_0$  y  $p_1$ , 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 parámetros a dicha función (como **vec3**)

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 trigonométricas (como lo son el arctangente, el seno, coseno, arcoseno o arcocoseno).

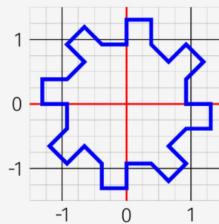
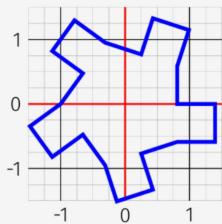
```
gancho_2p ( vec3 p0, Vec3 p1 ) {  
    //guardar matriz actual  
    glPushMatrix();  
    //trasladar origen a p0:  
    glTranslate( p0.x, p0.y, 0.0 );  
  
    //calcular ángulo de rotación a partir de p0 y p1  
    float dx = p1.x - p0.x;  
    float dy = p1.y - p0.y;  
    float angulo = atan2(dy, dx) * 180 / M_PI  
        //devuelve en radianes, transformo  
  
    //distancia entre p0 y p1 (norma)  
    float distancia = sqrt( (dx * dx) + (dy * dy) );  
  
    //escalar figura para que encaje:  
    glScalef( distancia, distancia, 1 );  
  
    gancho();  
  
    //restaurar matriz:  
    glPopMatrix();
```

Construir directamente ModelView:

```
void gancho_2p ( vec3 p0, Vec3 p1 ) {  
    //vector dirección  
    Vec3 dir = Vec3( p1.x - p0.x, p1.y - p0.y, 0 );  
  
    //norma  
    float distancia = sqrt( dir.x * dir.x + dir.y * dir.y + dir.z * dir.z );  
  
    //normalizar dir  
    dir = (dir.x / distancia, dir.y / distancia, 0 );  
  
    //ortogonal a v:  
    Vec3 orto = Vec3( -dir.y, dir.x, 0 );  
  
    //matriz:  
    float modelview[16] = {  
        dir.x, orto.x, 0, 0,  
        dir.y, orto.y, 0, 0,  
        0, 0, 1, 0,  
        p0.x, p0.y, 0, 0 }  
  
    //cargar matriz  
    glPushMatrix();  
    glLoadMatrix( modelview );  
  
    //dibujar gancho  
    gancho();  
  
    //Restaurar matriz original  
    glPopMatrix();
```

### Problema 2.18.

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 círculo (vemos los ejemplos para 5 y 8 instancias, respectivamente).



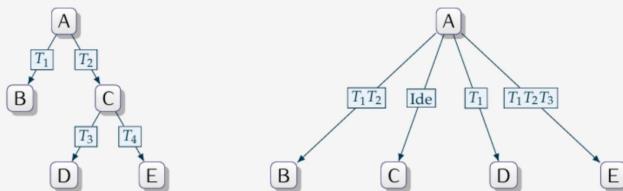
Para pintar una figura con  $n$  ganchos, los vértices (puntos iniciales y finales) son los radíos  $n$ -ésimos de la unidad. Como el primer gancho no empieza en  $(1,0,0)$ , sino que empieza a la mitad, metemos un desfase al círculo mitad.

```
void engronaje (int n) {
    float alpha, beta;
    for (int i=0; i < n; ++i) {
        alpha = -(i - 0.5) * 2.0 * M_PI / n
        beta = (i + 0.5) * 2.0 * M_PI / n
        gancho - 2p = ( vec3 (cos(alpha) sen(alpha), 0) , vec3 (cos(beta), sin(beta), 0));
    }
}
```

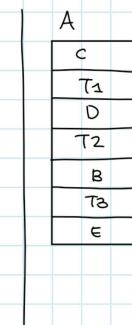
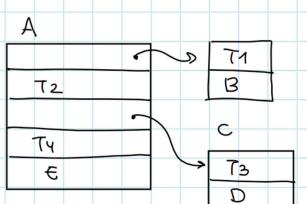
### Problema 2.19.

conocido en clase (muy apurado)

Dados los dos siguientes grafos de escena sencillos:

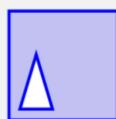


Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones:  $T_1, T_2$  y  $T_3$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



### Problema 2.20.

Escribe una función llamada **FiguraSimple** que dibuje con OpenGL en modo diferido la figura que aparece aquí (un cuadrado de lado unidad, relleno de color, con la esquina inferior izquierda en el origen, con un triángulo inscrito, relleno del color de fondo).



(usa el repositorio **opengl3-minimo**)

**FiguraSimple()**

```
vector<vec3> vert_cuadrado = {(-1,-1,0), (1,-1,0), (1,1,0), (-1,1,0)};
vector<vec3> vert_triangulo = {(-0.9,-0.9,0), (-0.7,-0.9,0), (-0.8,0,0)}
```

```
DescriptorVAO * vaoCuadrado = new DescriptorVAO(0, new DescriptorVBOAttribs({cauce -> ind_atrib_posiciones, vert_cuadrado}),
vaoCuadrado->agregar(new DescriptorVBOInds(indices));
```

```
DescriptorVAO * vaoTriangulo = new DescriptorVAO(0, new DescriptorVBOAttribs({cauce -> ind_atrib_posiciones, vert_triangulo}),
vaoTriangulo->agregar(new DescriptorVBOInds(indices));
```

// visualizar relleno

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

cauce -> fijarUserColor Plano (true)

cauce -> fijar Color ({0.0, 0.0, 0.5});

vaoCuadrado -> draw(GL\_TRIANGLES)

cauce -> fijarUserColor ({0.0, 0.0, 0.9});

vaoTriangulo -> draw(GL\_TRIANGLES)

// visualizar aristas

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINES);
```

cauce -> fijarUserColor Plano (true)

cauce -> fijar Color ({0.0, 0.0, 1.0});

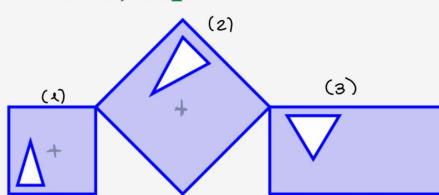
vaoCuadrado -> draw(GL\_LINE\_LOOP)

// usa glVertex Attrib

vaoTriangulo -> draw(GL\_LINES\_LOOP)

### Problema 2.21.

Usando exclusivamente llamadas a la función del problema 21, construye otra función llamada **FiguraCompleja** que dibuja la figura de aquí. Para lograrlo puedes usar manipulación de la pila de la matriz modelview (**pushMM** y **popMM**), junto con **MAT\_Traslacion** y **MAT\_Escalado**:

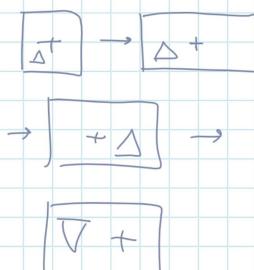


**void FiguraCompleja()**

```
pushMM() // guarda original
glScale (0.5, 0.5, 1) // cada 1
figura_simple() // }
popMM()
```

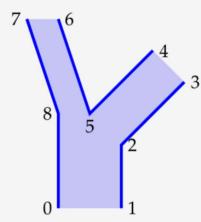
```
pushMM()
glTranslate (0,0,0)
glRotate (pi/4, (0,0,1))
glScale (sqrt(2), sqrt(2), 1)
glScale (-1,1,0)
figura_simple();
popMM();
```

```
pushMM();
glTranslate (4,0,0)
glRotate (pi, (0,0,1))
glScale (0.5, 3, 1)
glScale (0.5, -1, 1)
```



**Problema 2.22.**

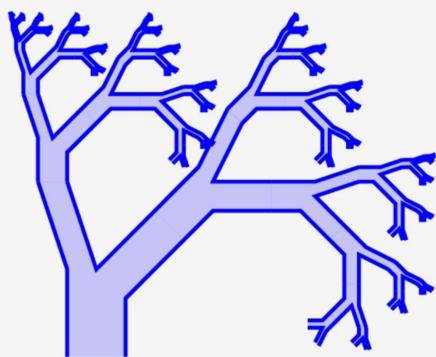
Escribe el código OpenGL de una función (llamada **Tronco**) que dibuje la figura que aparece a aquí. El código dibujará el polígono relleno de color azul claro, y las aristas que aparecen de color azul oscuro (ten en cuenta que no todas las aristas del polígono relleno aparecen).



Índice	Coordenadas
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)

### Problema 2.23.

Escribe una función **Arbol** la cual, mediante múltiples llamadas a **Tronco** del problema 2.22, dibuje el árbol que aparece en la figura de abajo. Diseña el código usando recursividad, de forma que el número de niveles sea un parámetro modifiable en dicho código (en la figura es 6)



# EJERCICIOS TEMA 3

## Problema 3.1.

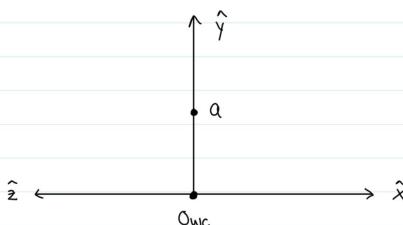
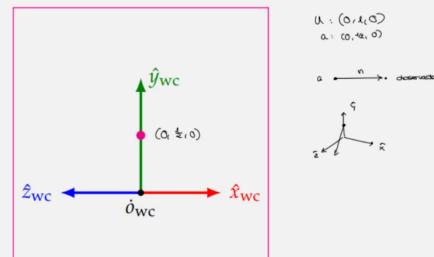
Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja, verde y azul), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

- El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
- El punto de coordenadas  $(0, 0.5, 0)$  (aparece como un disco de color morado en la figura) debe aparecer en el centro del viewport
- El observador (foco de la proyección) estará a 3 unidades de distancia del punto  $(0, 0.5, 0)$

(continua en la siguiente transparencia).

## Problema 3.1. (continuación)

Escribe unos valores que podríamos usar para  $\mathbf{a}$ ,  $\mathbf{u}$  y  $\mathbf{n}$  de forma que se cumplan estos requisitos. En la figura se observa una vista esquemática de como quedaría la figura en un viewport cuadrado, no necesariamente a escala.



→  $\mathbf{a}$ : punto en el eje óptico o punto de atención  $(0, 0.5, 0)$

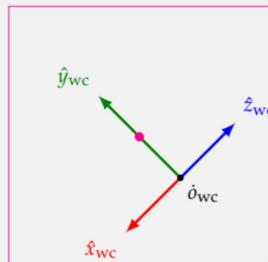
→  $\mathbf{oec}$ : punto del espacio de la proyección. Darse situaría observador ficticio  $(0, 0.5, 3)$

→  $\mathbf{n}$  vector perpendicular al plano de visión donde se proyecta la imagen  
 $\mathbf{n} = \mathbf{oec} - \mathbf{a} = (0, 0, 3)$

→  $\mathbf{u}$  vector libre que indica una dirección que el observador ve proyectada en vertical (hacia arriba)  
 $(0, 1, 0) - (0, 0, 0) = (0, 1, 0)$

## Problema 3.2.

Repite el problema anterior, pero ahora para esta vista:



$$\mathbf{a}: (0, 0.5, 0)$$

$$\mathbf{oec} = (0, 0.5, -3)$$

$$\mathbf{n} = \mathbf{oec} - \mathbf{a} = (0, 0, -3)$$

$$\mathbf{u} = (0, 0, 1)$$

"estamos situados por detrás, por eso vemos el eje z a la derecha y el x a la izquierda"

## Problema 3.3.

Escribe el código para calcular los vectores de coordenadas  $\mathbf{x}_{\text{ec}}$ ,  $\mathbf{y}_{\text{ec}}$ ,  $\mathbf{z}_{\text{ec}}$  y  $\mathbf{o}_{\text{ec}}$  que definen el marco de vista a partir de los vectores de coordenadas  $\mathbf{a}$ ,  $\mathbf{u}$  y  $\mathbf{n}$  (todos estos vectores de coordenadas son de tipo `vec3`).

```
using namespace glm;
vec3 oec, xec, yec, zec;

void funcion(vec3 a, vec3 u, vec3 m){
    oec = n+a;
    zec = Normalize(n);
    xec = Normalize(ProductoVectorial(u,n));
    yec = ProductoVectorial(zec,xec);
}
```

## Problema 3.4.

Partiendo de los vectores de coordenadas  $\mathbf{x}_{\text{ec}}$ ,  $\mathbf{y}_{\text{ec}}$ ,  $\mathbf{z}_{\text{ec}}$  y  $\mathbf{o}_{\text{ec}}$  que se calculan en el problema anterior, escribe el código que calcula explícitamente las 16 entradas de la matriz de vista (crea una **Matriz4f** llamada **V** y luego asigna valor a  $V(i,j)$  para cada fila **i** y columna **j**, ambas entre 0 y 3).

La función **MAT\_Vista** construye la matriz de vista a partir del origen y los tres ejes del marco de coordenadas de vista:

```
glm::mat4 MAT_Vista( const glm::vec3 ejes[3], const glm::vec3 &origen )
{
    // declaramos rot (inicialmente la matriz identidad)
    glm::mat4 rot = mat4(1.0);

    // copiamos cada eje en una fila de la matriz rot
    for( unsigned i = 0 ; i < 3 ; i++ )
        for( unsigned j = 0 ; j < 3 ; j++ )
            rot[i][j] = ejes[j][i]; // en GLM, M[i][j] es la columna i, fila j de M

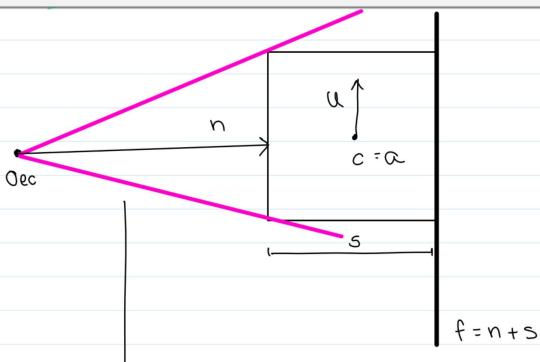
    // devolver la traslación seguida de la matriz de rotación
    return rot * translate( -origen );
}
```

### Problema 3.5.

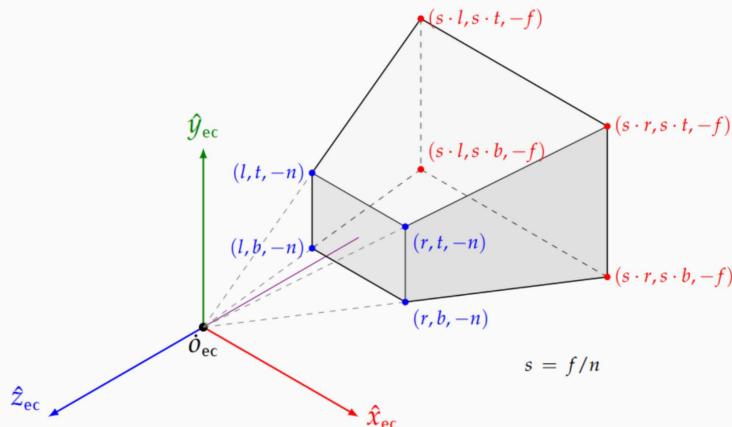
Queremos visualizar una escena con mallas indexadas de la cual sabemos que tiene todos los vértices dentro de un cubo de lado  $s$  unidades cuyo centro es el punto de coordenadas del mundo  $\mathbf{c} = (c_x, c_y, c_z)$ .

Para construir la matriz de vista, se sitúa el observador en el punto  $\mathbf{o}_{\text{ec}} = (c_x, c_y, c_z + s + 2)$ , el punto de atención  $\mathbf{a}$  se hace igual a  $\mathbf{c}$  (el centro del cubo se ve en el centro de la imagen), y el vector  $\mathbf{u}$  es  $(0, 1, 0)$ . Se visualizará en un viewport cuadrado.

(continua en la siguiente página)



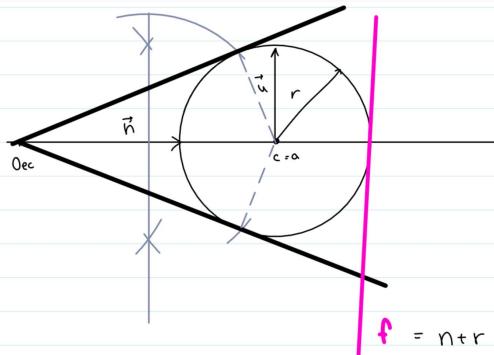
quiero las cosas lo más grandes posibles  $\rightarrow$  cierro el plano lo máximo posible



$$\begin{aligned} l &= s/2 \\ r &= s/2 \\ n &= s+z \\ f &= n+s \\ t &= s/2 \\ b &= s/2 \end{aligned}$$

### Problema 3.6.

Repite el problema anterior 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora la escena, en lugar de estar contenida en un cubo de lado  $s$  unidades, está contenida en una esfera de radio  $r$  unidades (con centro igualmente en  $\mathbf{c}$ ).



### Problema 3.5. (continuación)

Queremos construir la matriz de proyección perspectiva  $Q$  de forma que se cumplan estos requerimientos:

1. No se recorta ningún triángulo.
2. El tamaño aparente de los objetos (proyectados en pantalla) es el mayor posible.
3. El valor del parámetro  $n$  es el mayor posible.
4. El valor del parámetro  $f$  es el menor posible.
5. Los objetos no aparecen deformados.

Con estos requerimientos, indica como calcular los valores  $l, r, t, b, n$  y  $f$  (para obtener la matriz  $Q$  de proyección), en función de  $s$  y  $(c_x, c_y, c_z)$ .

- 1) no se recorta ningun triángulo  $\Rightarrow$  TODO el cubo dentro
- 2) tamaño mayor posible  $\Rightarrow$  planos a los extremos
- 3)  $n$  mayor posible : hasta la cara delantera
- 4)  $f$  menor posible : pegado a la cara trasera
- 5) objetos no deformados : perspectiva

$$\begin{aligned} n &= S+2 \\ f &= n+s \end{aligned}$$

Los demás se calculan mediante trigonometría

### Problema 3.7.

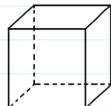
Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora, en lugar de suponer que el *viewport* es cuadrado, sabemos que tiene  $w$  columnas de pixels y  $h$  filas de pixels, y no podemos suponer que  $w = h$ .

coordenadas normalizadas de dispositivo (entre 0 y 1)

$$x_{ndc} = x_d / w$$

$$y_{ndc} = y_d / h$$

coordenadas de mundo



El *viewport* es la cara delantera del view frustum, que está en el mundo. Como las coordenadas normalizadas de dispositivo son la proporción donde está el punto si el ancho y el alto fuera 1, tenemos que:

$$x_w = l + x_{ndc} (r - l)$$

$$y_w = b + y_{ndc} (t - b)$$

sumamos (l,b) porque no medimos respecto del (0,0)

### Problema 3.8.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo  $\beta$  en grados que debe ser la apertura de campo vertical de la proyección perspectiva. Para ello, ahora tenemos libertad para situar al observador en la línea paralela al eje Z que pasa por  $c$ , de forma que la apertura de campo vertical sea exactamente  $\beta$ .

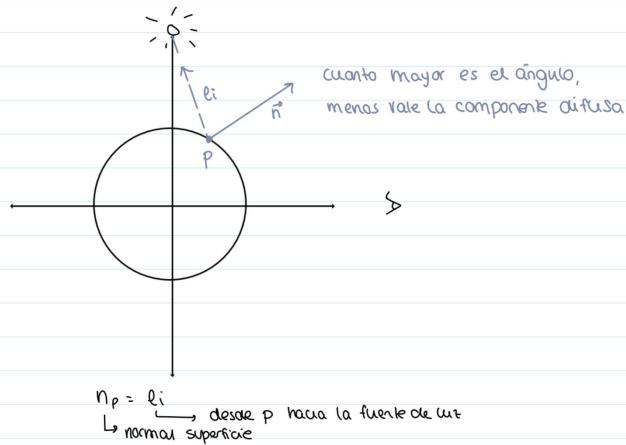
Indica como calcular la coordenada Z que debemos usar ahora para  $\mathbf{o}_{oc}$  (la X y la Y son las mismas que antes), de forma que se cumpla lo dicho, también indica como debemos de calcular ahora los valores de  $l, r, t, b, n$  y  $f$  (todo ello en función de  $\beta$ ,  $s$  y  $\mathbf{c} = (c_x, c_y, c_z)$ ).

Demasiado complicado para examen

### Problema 3.9.

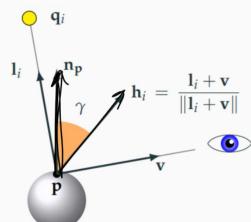
Suponemos que visualizamos una esfera de radio unidad centrada en el origen. Se ilumina con una fuente de luz puntual situada en el punto  $\mathbf{p} = (0, 2, 0)$ . El observador está situado en  $\mathbf{o} = (2, 0, 0)$ . En estas condiciones:

- ▶ Describe razonadamente en que punto de la superficie de la esfera el brillo será máximo si el material es puramente difuso ( $k_d = 1$  en todos los puntos, y  $k_a$  y  $k_s$  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.



1) Material puramente difuso, el mayor brillo será en  $(0, 1, 0)$   
punto no visible por el observador, solo puede ver hasta el cono tangente a la esfera

2) Material puramente especular



brillo proporcional al coseno del ángulo  $\gamma$

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = k_s(\mathbf{p}) d_i [\mathbf{n}_p \cdot \mathbf{h}_i]^e$$

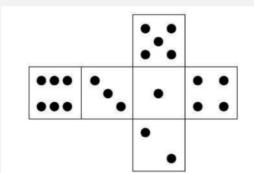
mayor brillo cuando maximizo  $\mathbf{n}_p \cdot \mathbf{h}_i \Rightarrow$  cuando forman un ángulo de 0°  
 $\mathbf{n}_p = \mathbf{h}_i$   
 $\downarrow$  bisectriz  $\mathbf{l}_i, \mathbf{v}$   
normal a la superficie }       $\left. \begin{array}{l} \mathbf{l}_i = (0, 1, 0) \\ \mathbf{v} = (0, 0, 2) \end{array} \right\}$  bisectriz  $(1, 1, 0)$

$$\text{Normalizado} \rightarrow \left( \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right)$$

Luego  $\mathbf{p} = \left( \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right)$  visible para el espectador

### Problema 3.10.

Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar una textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3. La imagen aparece aquí:



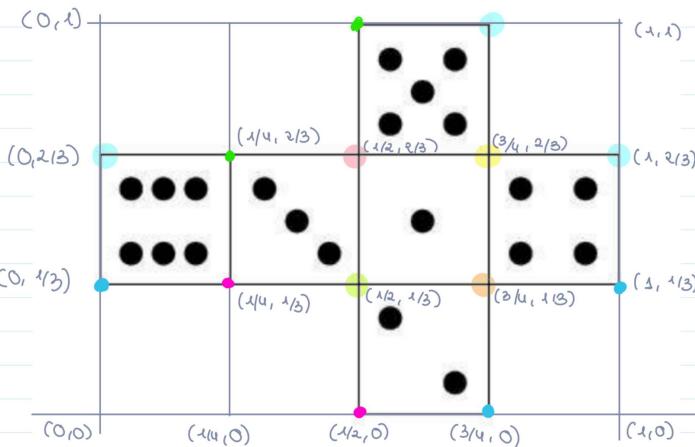
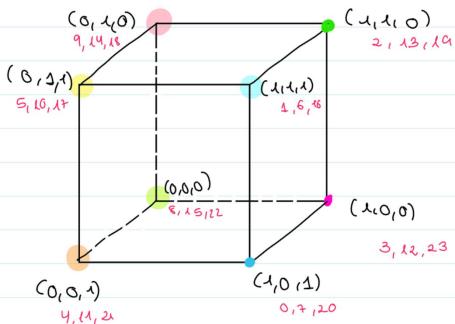
(continua en la siguiente transparencia)

### Problema 3.10. (continuación)

Responde a estas cuestiones:

- Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
- 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.

a) Necesitaremos como mínimo 24 vértices (3 por cada vértice del cubo)



```

vertices = [
    //cara derecha
    {1,0,1}, // 0
    {1,1,1}, // 1
    {1,1,0}, // 2
    {1,0,0}, // 3

    //cara frente
    {0,0,1}, // 4
    {0,1,1}, // 5
    {1,1,1}, // 6
    {1,0,1}, // 7

    //cara izquierda
    {0,0,0}, // 8
    {0,1,0}, // 9
    {0,1,1}, // 10
    {0,0,1}, // 11

    //atrás
    {1,0,0}, // 12
    {1,1,0}, // 13
    {0,1,0}, // 14
    {0,0,0}, // 15

    //techo
    {1,1,1}, // 16
    {0,1,1}, // 17
    {0,1,0}, // 18
    {1,1,0}, // 19

    //suelo
    {1,0,1}, // 20
    {0,0,1}, // 21
    {0,0,0}, // 22
    {1,0,0} // 23
];

```

$cc\_t = \{ \}$   
//cara derecha  
 $\{0, 1/3\}, \{0, 2/3\}, \{1/4, 2/3\}, \{1/4, 1/3\}$

//cara frente  
 $\{3/4, 1/3\}, \{3/4, 2/3\}, \{1, 2/3\}, \{1, 1/3\}$

//cara izquierda  
 $\{1/2, 1/3\}, \{1/2, 2/3\}, \{3/4, 2/3\}, \{3/4, 1/3\}$

//atrás  
 $\{1/4, 1/3\}, \{1/4, 2/3\}, \{1/2, 2/3\}, \{1/2, 1/3\}$

//techo  
 $\{3/4, 1/4\}, \{3/4, 2/3\}, \{1/2, 2/3\}, \{1/2, 1/4\}$

//suelo  
 $\{1/2, 0\}, \{3/4, 1/3\}, \{1/2, 1/3\}, \{3/4, 0\}$

### Problema 3.11.

Considera de nuevo el cubo y la textura del problema anterior. Ahora supón que queremos visualizar con OpenGL el cubo usando sombreado de Gouraud 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 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.

Necesitamos escribir otra distinta para las normales ya que estas son vec3, no vec 2.

se usa el siguiente algoritmo:

$$\text{Para cada vértice } \vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \quad \vec{s} = \sum_{k=0}^{k=2} \vec{m}_i$$

donde  $\vec{m}_i$  las normales de las caras adyacentes al vértice

$$\text{normales de caras: } \vec{n} = \frac{\vec{a} \times \vec{b}}{\|\vec{a} \times \vec{b}\|}$$

### Problema 3.12.

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.

Mismo que cubo 24 con la diferencia de que el cubo 24 está centrado en  $(0, 0, 0)$

# EJERCICIOS TEMA 4

## Problema 4.1.

Supongamos que queremos visualizar una secuencia de frames, en los cuales la cámara va cambiando. Para ellos queremos escribir el código de una función que fija la matriz de vista en el cauce. La función acepta como parámetro un valor real  $t$ , que es el tiempo en segundos transcurrido desde el inicio de la animación. Suponemos que la animación dura  $s$  segundos en total.

En ese tiempo el observador de cámara se desplaza con un movimiento uniforme desde un punto de coordenadas de mundo  $\mathbf{o}_0$  (para  $t = 0$ ) hasta un punto destino  $\mathbf{o}_1$  (para  $t = 1$ ). Además el punto de atención de la cámara también se desplaza desde  $\mathbf{a}_0$  hasta  $\mathbf{a}_1$ . Durante toda la animación, el vector VUP es  $(0, 1, 0)$ .

Escribe el pseudo-código de la citada función.

```
void FijarMatrizVista (float t, float s, vec3 o0, vec3 o1, vec3 a0, vec3 a1){

    //cálculo del origen
    vec3 oec = o0*(1-t/s) + o1*t/s;

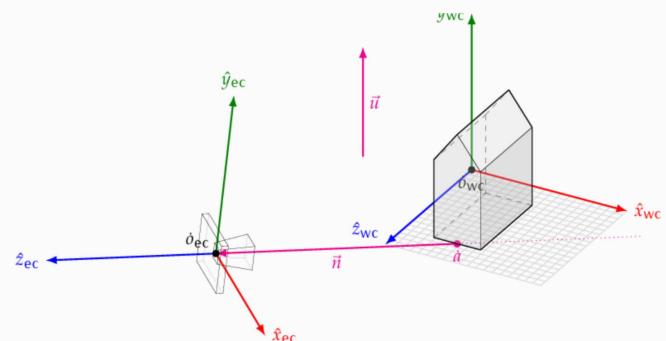
    //cálculo punto de atención de la cámara
    at = a0*(1-t/s) + a1*t/s;

    //construir Matriz vista
    mat4 V = LookAt (oec,at,VUP);

    //fijar matriz vista en cauce
    FijarMatrizVistaEnCauce (V);
}
```

Se puede construir explícitamente usando las funciones de creación de matrices. Podemos usar la función **lookAt** de GLM, que tiene como parámetros las tuplas  $\mathbf{o}_{\text{ec}}$ ,  $\mathbf{a}$  y  $\mathbf{u}$ :

```
glm::mat4 lookAt( const glm::vec3 & oec, const glm::vec3 & a,
                  const glm::vec3 & u );
```



Podemos expresar el marco de cámara en función de  $\vec{n}$  y  $\vec{a}_t$ :

$$\begin{aligned} \vec{z}_{\text{ec}} &= \vec{n} / \|\vec{n}\| & \vec{y}_{\text{ec}} &= \vec{z}_{\text{ec}} \times \vec{x}_x \\ \vec{x}_{\text{ec}} &= (\vec{y}_w \times \vec{z}_{\text{ec}}) / \|\vec{y}_w \times \vec{z}_{\text{ec}}\| & \dot{o}_{\text{ec}} &= \vec{a}_t + \vec{n} \end{aligned}$$

( $\vec{z}_{\text{ec}}$  es paralelo a  $\vec{n}$ , y el vector VUP siempre es  $\vec{y}_w$ ).

```
void FijarMatrizVista (float t, float s, vec3 o0, vec3 o1, vec3 a0, vec3 a1){

    //cálculo del origen
    vec3 oec = o0*(1-t/s) + o1*t/s;

    //cálculo punto de atención de la cámara
    at = a0*(1-t/s) + a1*t/s;

    //vector n, desde origen hasta punto de atención
    vec3 n = a0-oec;

    //construcción del marco de coordenadas de la cámara
    vec3 zec = Normalizar(n);
    vec xec = Normalizar(ProductoVectorial (VUP, zec));
    vec3 yec = ProductoVectorial(zec, xec);

    //construir Matriz vista
    mat4 V = LookAt (oec,at,VUP);

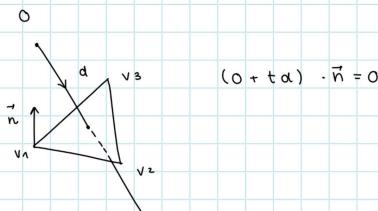
    //fijar matriz vista en cauce
    FijarMatrizVistaEnCauce (V);
```

### Problema 4.2.

Una posibilidad para hacer selección en mallas de triángulos es usar cálculo de intersecciones entre un rayo (una semirrecta que pasa por el centro de un pixel) y cada uno de los triángulos de la malla. Diseña un algoritmo en pseudo-código para el cálculo de intersecciones entre un rayo y un triángulo:

- ▶ El rayo tiene como origen o extremo el punto cuya coordenadas del mundo es la tupla  $\mathbf{o}$ , y como vector de dirección la tupla  $\mathbf{d}$  (la suponemos normalizada).
- ▶ Las coordenadas del mundo de los vértices del triángulo son  $\mathbf{v}_0, \mathbf{v}_1$  y  $\mathbf{v}_2$ .
- ▶ El algoritmo debe de indicar si hay intersección o no, y, en caso de que la haya, calcular las coordenadas del mundo del punto de intersección.

(ver la siguiente transparencia).



```
bool InterseccionRayoTriangul(o,d,v0,v1,v2,&p){
```

```
//vectores del triángulo
e1 = v1-v0;
e2 = v2-v0

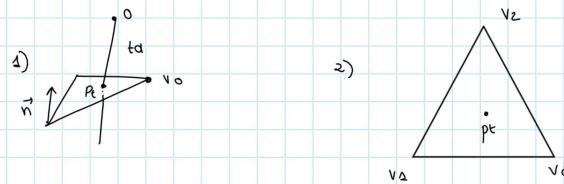
//vector normal al triángulo
n = Normalize(productoVectorial(e1,e2));

//ver si el rayo es paralelo al triángulo (perpendicular a la nor
if (ProductoEscalar(n,d)< epsilon){
    return false;
}

//vector desde v0 al origen del rayo
A0 = o -v0

//calcular t
vec3 t = -productoEscalar(escalar n, A0) / productoEscalar(n,d)
```

CASOS EN LOS QUE HAY INTERSECCIÓN :



```
if(t<0){
    return false;
}

else{
    //calculamos p
    p = o + t *d;

    //calculamos coordenadas baricéntricas
    AP = p - v0;
    d00 = dot(e1, e1);
    d01 = dot(e1, e2);
    d11 = dot(e2, e2);
    d20 = dot(AP, e1);
    d21 = dot(AP, e2);
    denom = d00 * d11 - d01 * d01;
    u = (d11 * d20 - d01 * d21) / denom;
    v = (d00 * d21 - d01 * d20) / denom

    // Verificar si el punto de intersección está dentro del triángulo
    if((u >= 0) && (v >= 0) && ((u + v) <= 1)
        return true;
    else
        return false;
}
```

### Problema 4.3.

Para implementar la selección usando intersecciones es necesario calcular el rayo que tiene como origen el observador y pasa por centro del pixel donde se ha hecho click. Escribe el pseudo-código del algoritmo que calcula el rayo a partir de las coordenadas del pixel donde se ha hecho click:

- ▶ Tenemos una vista perspectiva, y conocemos los 6 valores  $l, r, t, b, n, f$  usados para construir la matriz de proyección.
- ▶ También conocemos el marco de coordenadas de vista, es decir, las tuplas  $x_{ec}, y_{ec}$  y  $\mathbf{o}_{ec}$  con los versores y la tupla  $\mathbf{o}_{ec}$  con el punto origen (todos en coordenadas del mundo).
- ▶ El viewport tiene  $w$  columnas y  $f$  filas de pixels. Se ha hecho click en el pixel de coordenadas enteras  $x_p$  e  $y_p$

El algoritmo debe producir como salida las tuplas  $\mathbf{o}$  y  $\mathbf{d}$  (normalizado) que definen el rayo.

```
funcionCalcularRayoClic(l,r,t,b,n,f,xec,yec,oec,xd,yd,w,h){
```

```
//coordenadas normalizadas de dispositivo
xnd = l + (r-1) * (xp+0.5)/w;
ynd = t - (t-b) * (yp+0.5)/h;

//punto en el plano de proyección está a distancia near de la cámara
point_on_plane= oec* n;

//punto viewport
point_on_viewport = xec*xnd + yec*ynd + point_on_plane;

//vector dirección del rayo
direccion_rayo = Normalizar(punto_viewport-oec)

return(oec, dirección_rayo)
```

## PROBLEMAS T5

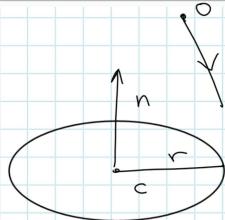
## Problema 5.1.

Supongamos que un *rayo* (una semirecta en 3D) tiene como origen o extremo el punto cuyas coordenadas del mundo es la tupla **o**, y como vector de dirección la tupla **d** (la suponemos normalizada).

Además sabemos que un disco de radio  $r$  tiene como centro el punto de coordenadas de mundo  $\mathbf{c}$  y está en el plano perpendicular al vector  $\mathbf{n}$

Con estos datos de entrada, diseña un algoritmo para calcular si hay intersección entre el rayo y el disco.

(ten en cuenta las indicaciones que hay en la siguiente transparencia).



```

bool InterseccionRayoCirculo(o,d,n,r,c,&p){

    //ver si el rayo es paralelo al círculo (perpendicular a la normal)
    if (ProductoEscalar(n, d)< epsilon){
        return false;
    }

    //calcular parámetro de intersección
    t = ((c-o)*n)/(d*n);

    //si t negativo, está por detrás y no hay intersección
    if(t< 0)
        return false;

    //calcula la intersección
    p = o + t*d;
    //comprobar que el punto está dentro del disco
    if (sqrt((c.x-p.x)*(c.x-p.x) + (c.y-p.y)*(c.y-p.y))<r)
        return true;
    else
        return false;
}

```

### Problema 5.1. (continuación)

Ten en cuenta que habrá intersección si y solo si se cumplen cada una de estas dos condiciones:

1. El rayo interseca con el plano que contiene al triángulo, es decir, existe  $t > 0$  tal que el punto  $\mathbf{p}_t \equiv \mathbf{o} + t\mathbf{d}$  está en dicho plano. Equivale a decir que el vector  $\mathbf{p}_t - \mathbf{v}_0$  es perpendicular a la normal al plano  $\mathbf{n}$ .
  2. El punto  $\mathbf{p}_t$  citado arriba está dentro del disco, es decir, su distancia a  $\mathbf{c}$  es inferior al radio.

1. **Calcular el punto de intersección entre el rayo y el plano:** Sabemos que el disco está en un plano perpendicular a  $n$ . El primer paso es determinar si el rayo intersecta el plano en el que reside el disco.

La ecuación general del plano es:

$$(P - c) \cdot n = 0$$

Donde  $P$  es cualquier punto en el plano,  $c$  es el centro del disco y  $n$  es el vector normal del plano.

La ecuación paramétrica del rayo es:

$$P(t) = o + t \cdot d$$

Donde  $o$  es el origen del rayo y  $d$  es la dirección del rayo (un vector normalizado).  $t$  es un parámetro que indica cuán lejos está un punto del origen a lo largo del rayo.

Sustituyendo  $P(t)$  en la ecuación del plano, obtenemos:

$$(o + t \cdot d - c) \cdot n = 0$$

2. **Resolver para  $t$ :** Resolviendo la ecuación anterior, podemos encontrar el valor de  $t$ , el parámetro de la intersección del rayo con el plano:

$$t = \frac{(c - o) \cdot n}{d \cdot n}$$

### Problema 5.2.

Diseña un algoritmo para calcular la primera intersección entre un rayo (con origen en  $\mathbf{o}$  y vector  $\mathbf{d}$ , normalizado) y una esfera de radio unidad y centro en el origen, si hay alguna.

Ten en cuenta que un punto cualquiera  $\mathbf{p}$  está en esfera si y solo si el módulo de  $\mathbf{p}$  es la unidad, es decir, si y solo si  $F(\mathbf{p}) = 0$ , donde  $F$  es el campo escalar definido así:

$$F(\mathbf{p}) \equiv \mathbf{p} \cdot \mathbf{p} - 1$$

Describe como podría usarse ese mismo algoritmo para calcular la intersección con una esfera con centro y radio arbitrarios (este problema puede reducirse al anterior si el rayo se traslada a un espacio de coordenadas donde la esfera tiene centro en el origen y radio unidad).

$$\text{Ecuación esfera con radio 1} \quad F(\mathbf{p}) = \mathbf{p} \cdot \mathbf{p} - 1 = 0$$

$$\text{Ecuación del rayo} \quad \mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

$$\text{Intersección rayo - esfera: } F(\mathbf{r}(t)) = (\mathbf{o} + t\mathbf{d})(\mathbf{o} + t\mathbf{d}) - 1 \Rightarrow \mathbf{o}^2 + t^2\mathbf{d}^2 + 2\mathbf{o}\mathbf{d}t - 1 \Rightarrow t^2 + 2\mathbf{o}\mathbf{d}t + (\mathbf{o}^2 - 1) = 0$$

$$t = \frac{-2\mathbf{o}\mathbf{d} \pm \sqrt{4\mathbf{o}^2\mathbf{d}^2 - 4(\mathbf{o}^2 - 1)}}{2} = -\mathbf{o}\mathbf{d} \pm \sqrt{\mathbf{o}^2\mathbf{d}^2 - \mathbf{o}^2 - 1}$$

→ si el discriminante negativo, no hay intersección.

→ si  $\Delta_{disc} = 0 \Rightarrow 1$  solución

→ si  $\Delta_{disc} > 0 \Rightarrow 2$  soluciones. La más pequeña es la que nos quedamos

función IntersecciónRayoEsfera( $\mathbf{o}$ ,  $\mathbf{d}$ ):

# Paso 1: Calcular los coeficientes de la ecuación cuadrática

A = 1

B = 2 \* ( $\mathbf{o} \cdot \mathbf{d}$ )

C = ( $\mathbf{o} \cdot \mathbf{o}$ ) - 1

# Paso 2: Calcular el discriminante

discriminante = B^2 - 4 \* A \* C

si discriminante < 0:

retornar "No hay intersección"

# Paso 3: Calcular las dos soluciones de la ecuación cuadrática

t1 = (-B - sqrt(discriminante)) / (2 \* A)

t2 = (-B + sqrt(discriminante)) / (2 \* A)

# Paso 4: Determinar la primera intersección

si t1 >= 0:

retornar t1 # Primer punto de intersección

si t2 >= 0:

retornar t2 # Segundo punto de intersección, si el primero es negativo

retornar "No hay intersección" # Ambos puntos están detrás del origen

En el caso de centro y radio arbitrario,

```
función IntersecciónRayoEsferaArbitraria(o, d, c, r):
    # Paso 1: Trasladar el origen del rayo
    o' = o - c

    # Paso 2: Escalar la dirección del rayo
    d' = d / r

    # Paso 3: Llamar al algoritmo de intersección con la esfera de radio 1 y centro en el origen
    retornar IntersecciónRayoEsfera(o', d')
```

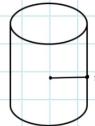
### Problema 5.3.

Describe como podemos definir el campo escalar cuyos ceros son los puntos en un cilindro con altura unidad y radio unidad (sin considerar los discos que forman la base ni la tapa).

Usando esa definición diseña el algoritmo para calcular la intersección rayo-cilindro.

Describe asimismo el campo escalar y el algoritmo correspondientes a un cono de altura unidad y radio de la base unidad (sin considerar el disco de la base).

#### Caso cilindro



$$\begin{aligned} f(p) &= f(x, y, z) \\ f(p) &= x^2 + y^2 - 1 = 0 \end{aligned}$$

$$\begin{aligned} x &= o_x + t dx \\ z &= o_z + t dz \end{aligned} \quad \left. \begin{array}{l} \text{Intersección rayo - cilindro extensión infinita} \\ \text{Si es extensión finita, ver si la otra se queda dentro de las coordenadas y} \end{array} \right.$$

$$r(t) = o + t d$$

$$\begin{aligned} f(r(t)) &= (o_x + t dx)^2 + (o_y + t dy)^2 - 1 = 0 \\ 0^2_x + t^2 dx^2 + 2t o_x dy + 0^2_y + t^2 dy^2 + 2o_y t dy &= \\ (d_x^2 + d_y^2) t^2 + 2(o_x dx + o_y dy) t + (o_y^2 + o_z^2 - 1) &= \end{aligned}$$

$$\begin{aligned} A &= d_x^2 + d_y^2 \\ B &= o_x dx + o_y dy \\ C &= o_y^2 + o_z^2 - 1 \end{aligned}$$

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

función IntersecciónRayoCilindro(o, d):

```
# Paso 1: Calcular los coeficientes de la ecuación cuadrática
A = d_x^2 + d_y^2
B = 2 * (o_x * d_x + o_y * d_y)
C = o_x^2 + o_y^2 - 1
```

[Copiar](#)

# Paso 2: Calcular el discriminante

```
discriminante = B^2 - 4 * A * C
```

si discriminante < 0:

```
    retornar "No hay intersección" # El rayo no intersecta la superficie lateral
```

# Paso 3: Calcular las soluciones de la ecuación cuadrática

```
t1 = (-B - sqrt(discriminante)) / (2 * A)
t2 = (-B + sqrt(discriminante)) / (2 * A)
```

# Paso 4: Verificar que la intersección esté dentro del cilindro ( $0 \leq z \leq 1$ )

```
z1 = o_z + t1 * d_z
z2 = o_z + t2 * d_z
```

si  $0 \leq z1 \leq 1$ :

```
    retornar t1 # Primer punto de intersección
```

si  $0 \leq z2 \leq 1$ :

```
    retornar t2 # Segundo punto de intersección
```

```
retornar "No hay intersección" # Ninguna intersección está dentro del cilindro
```

## Cono

$$\frac{x^2 + y^2}{z^2} = 1 \quad 0 \leq z \leq 1$$

$$F(x, y, z) = x^2 + y^2 - z^2 = 0$$

$$F(r(t)) = (0x + tdx)^2 + (0y + tdy)^2 - (0z + tdz)^2$$

$$A = dx^2 + dy^2 - dz^2$$

$$B = 2(0x \cdot dx + 0y \cdot dy - 0z \cdot dz)$$

$$C = 0^2x + 0^2y - 0^2z$$

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

función IntersecciónRayoCono(o, d):

# Paso 1: Calcular los coeficientes de la ecuación cuadrática

$$A = d_x^2 + d_y^2 - d_z^2$$

$$B = 2 * (o_x * d_x + o_y * d_y - o_z * d_z)$$

$$C = o_x^2 + o_y^2 - o_z^2$$

# Paso 2: Calcular el discriminante

$$\text{discriminante} = B^2 - 4 * A * C$$

si discriminante < 0:

retornar "No hay intersección" # El rayo no intersecta el cono

# Paso 3: Calcular las soluciones de la ecuación cuadrática

$$t1 = (-B - \sqrt{\text{discriminante}}) / (2 * A)$$

$$t2 = (-B + \sqrt{\text{discriminante}}) / (2 * A)$$

# Paso 4: Verificar que la intersección esté dentro del cono ( $0 \leq z \leq 1$ )

$$z1 = o_z + t1 * d_z$$

$$z2 = o_z + t2 * d_z$$

si  $0 \leq z1 \leq 1$ :

retornar t1 # Primer punto de intersección

si  $0 \leq z2 \leq 1$ :

retornar t2 # Segundo punto de intersección

retornar "No hay intersección" # Ninguna intersección está dentro del cono