

Problema 1.1.

Escribe el código 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. Escribe el código que crea el correspondiente VAO a esta secuencia de vértices.

En estos problemas, puedes usar las funciones **CrearVBOAtrib**, **CrearVBOInd** y **CrearVAO**.

(el enunciado continua en la siguiente transparencia)

Problema 1.1. (continuación)

El valor de n (> 2) es un parámetro (un entero sin signo). La tabla 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**.

// Ejercicio 1

```
// A) GL_LINE_LOOP

// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos;

std::vector<Tupla2d> posiciones;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n)});
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_DOUBLE, 2,
posiciones.size(), posiciones.data());
    }
} else{
    glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa
}
```

// B) GL_LINES

```
// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos;

std::vector<Tupla2d> posiciones;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n)});
    posiciones.push_back({cos(2*M_PI*(i+1)/n),
sin(2*M_PI*(i+1)/n)})
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_DOUBLE, 2,
posiciones.size(), posiciones.data());
    }
} else{
    glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa
}
```

Problema 1.2.

Escribe el código para generar una tabla de vértices y una tabla de índices, que permitiría visualizar el mismo polígono regular del problema anterior pero ahora como un conjunto de n triángulos iguales rellenos 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.

// Ejercicio 1.2

```
// A) NO INDEXADA (3n vertices)

// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos;

std::vector<Tupla3f> posiciones;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n), 0.0});
    posiciones.push_back({cos(2*M_PI*(i+1)/n),
sin(2*M_PI*(i+1)/n), 0.0});
    posiciones.push_back({0.0, 0.0, 0.0});
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_FLOAT, 3,
posiciones.size(), posiciones.data());
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO ya
creado, simplemente se activa
```

// B) INDEXADA (3n indice y n+1 vertices)

```
// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_posiciones, ind_pos,
nombre_vbo_ind;

std::vector<Tupla3f> posiciones;
std::vector<Tupla3u> indices;

for(int i=0; i<n; i++){
    posiciones.push_back({cos(2*M_PI*i/n),
sin(2*M_PI*i/n), 0.0});
}
posiciones.push_back({0.0, 0.0, 0.0});

for(int i=0; i<n; i++){
    indices.push_back({i, (i+1)%n, n});
}

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
CrearVBOAtrib(ind_pos, GL_FLOAT, 3,
posiciones.size(), posiciones.data());
    }
    if(indices.size() > 0){
        nombre_vbo_ind = CrearVBOInd( indices );
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO ya
creado, simplemente se activa
}
```

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 de polígono regular de n lados (donde n es una constante de tu programa), usando las tablas, VBOs y VAOs de coordenadas que codifican dicho polígono regular, según se describe 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, VBOs y ambos VAOs (en la primera llamada), como la visualización (en todas las llamadas).

```
// Ejercicio 1.3
```

```
    // A) GL_LINE_LOOP
```

```
    // Poligono de n lados
```

```
    int n=5;
```

```
    int nombre_vao = 0;
```

```
    int nombre_vbo_posiciones, ind_pos,  
    ind_colores;
```

```
    std::vector<Tupla2d> posiciones;
```

```
    for(int i=0; i<n; i++){
```

```
        posiciones.push_back({cos(2*M_PI*i/n),  
        sin(2*M_PI*i/n)});
```

```
    }
```

```
    if(nombre_vao == 0){
```

```
    //A partir de ahora debemos crear un nuevo  
    VAO donde le vamos a volver a pasar los  
    datos necesarios para formar  
    // un polígono relleno con triángulos
```

```
    // Poligono de n lados
```

```
    std::vector<Tupla3u> indices;
```

```
    int nombre_vao_2, nombre_vbo_ind;
```

```
    posiciones.push_back({0.0, 0.0, 0.0});
```

```
    for(int i=0; i<n; i++){  
        indices.push_back({i, (i+1)%n, n});
```

```
    }
```

```
    if(nombre_vao_2 == 0){
```

<pre> nombre_vao = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_DOUBLE, 2, posiciones.size(), posiciones.data()); } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } //Con esto le decimos que el color por defecto es el negro para los vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,0.0,1.0); glDrawArrays(GL_LINE_LOOP, 0, posiciones.size()); glBindVertexArray(0); //Hasta aquí hemos dibujado las aristas del polígono. </pre>	<pre> nombre_vao_2 = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } else{ glBindVertexArray(nombre_vao_2); // VAO ya creado, simplemente se activa } glVertexAttrib3f(ind_colores, 1.0,0.0,0.0,1.0); glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0); glBindVertexArray(0); //Ahora hemos dibujado el relleno del polígono(mostrando los triángulos en vez de las aristas). </pre>
---	--

Problema 1.4.

Repite el problema anterior (problema 1.3), pero ahora intenta usar el mismo VAO para las aristas y los triángulos llenos. 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).

<pre> // Ejercicio 1.4 // A) GL_LINE_LOOP // Poligono de n lados int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla2d> posiciones; for(int i=0; i<n; i++){ posiciones.push_back({cos(2*M_PI*i/n), sin(2*M_PI*i/n)}); } posiciones.push_back({0.0, 0.0, 0.0}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; for(int i=0; i<n; i++){ indices.push_back({i, (i+1)%n, n}); } if(nombre_vao == 0){ nombre_vao = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } </pre>	<pre> //Con esto le decimos que el color por defecto es el negro para los vertices del VAO activo ahora mismo //El Último 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,0.0); // Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices glDrawArrays(GL_LINE_LOOP, 0, (posiciones.size()-1)); glVertexAttrib3f(ind_colores, 1.0,0.0,0.0); glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0); glBindVertexArray(0); </pre>
---	--

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, 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.

<pre>// Ejercicio 1.5 // A) GL_LINE_LOOP // Poligono de n lados int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla3d> posiciones; std::vector<Tupla3f> colores; for(int i=0; i<n; i++){ posiciones.push_back({cos(2*M_PI*i/n), sin(2*M_PI*i/n), 0.0}); colores.push_back({rand()/float(RAND_MAX), rand()/float(RAND_MAX), rand()/float(RAND_MAX)}); posiciones.push_back({0.0, 0.0, 0.0}); colores.push_back({rand()/float(RAND_MAX), rand()/float(RAND_MAX), rand()/float(RAND_MAX)}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; for(int i=0; i<n; i++){ indices.push_back({i, (i+1)%n, n}); } if(nombre_vao == 0){ nombre_vao = crearVAO();</pre>	<pre>if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } if(colores.size()>0){ nombre_vbo_col = CrearVBOAtrib(ind_colores, GL_FLOAT, 3, colores.size(), colores.data()); } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } //El color por defecto es el negro para los vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,0.0); glDisableVertexAttribArray(ind_colores); //Quito la tabla de colores para pintar las aristas // Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices glDrawArrays(GL_LINE_LOOP, 0, (posiciones.size()-1)); glEnableVertexAttribArray(ind_colores); //Como vamos a pintar el relleno, habilito la tabla de colores glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0); glBindVertexArray(0);</pre>
---	--

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 las funciones dadas para creación de VAOs y VBOs (asumen una tabla por VBO con estructura SOA).

// Ejercicio 1.6

```
struct VStruct{  
    Tupla3f pos;  
    Tupla3f col;  
};  
  
int crearVBOAOS(std::vector<VStruct> v){  
    int nombre_vbo;  
    int tamano = v.size()*sizeof(VStruct);  
    glGenBuffers(1, &nombre_vbo);  
    glBindBuffer(GL_ARRAY_BUFFER,  
    nombre_vbo);  
  
    glBufferData(GL_ARRAY_BUFFER,tamani  
o,v.data(), GL_STATIC_DRAW);  
  
    glVertexAttribPointer(ind_posiciones, 3,  
    GL_FLOAT, GL_FALSE,sizeof(VStruct), 0);  
  
    glVertexAttribPointer(ind_colores, 3,  
    GL_FLOAT, GL_FALSE, sizeof(VStruct),  
    3*sizeof(float));  
  
    glEnableVertexAttribArray(ind_posiciones);  
    glEnableVertexAttribArray(ind_colores);  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
    return nombre_vbo;
```

// Poligono de n lados

```
std::vector<Tupla3u> indices;  
int nombre_vbo_ind;  
  
for(int i=0; i<n; i++){  
    indices.push_back(i, (i+1)%n, n);  
}  
  
if(nombre_vao == 0){  
    nombre_vao = crearVAO();  
  
    if(vectorAOS.size() > 0) {  
        nombre_vbo_aos =  
crearVBOAOS(vectorAOS);  
    }  
    if(indices.size() > 0){  
        nombre_vbo_ind = CrearVBOInd(  
indices );  
    }  
}  
else{  
    glBindVertexArray(nombre_vao); // VAO  
ya creado, simplemente se activa  
}  
  
//Con esto le decimos que el color por  
defecto es el negro para los vértices del  
VAO activo ahora mismo
```

```

};

// A) GL_LINE_LOOP

// Poligono de n lados
int n=5;
int nombre_vao = 0;
int nombre_vbo_aos, ind_pos,
ind_colores;

std::vector<VStruct> vectorAOS;

for(int i=0; i<n; i++){
    VStruct v;
    v.pos = {cos(2*M_PI*i/n),
    sin(2*M_PI*i/n), 0.0};
    v.col = {rand()/float(RAND_MAX),
    rand()/float(RAND_MAX),
    rand()/float(RAND_MAX)};
    vectorAOS.push_back(v);

}

VStruct v;
v.pos = {0.0, 0.0, 0.0};
v.col = {rand()/float(RAND_MAX),
rand()/float(RAND_MAX),
rand()/float(RAND_MAX)};
vectorAOS.push_back(v);

```

//El ultimo 1 se refiere a la transparencia
`glVertexAttrib3f(ind_colores, 0.0,0.0,0.0);`
`glDisableVertexAttribArray(ind_colores);`
//Quito la tabla de colores para pintar las aristas

// Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices
`glDrawArrays(GL_LINE_LOOP, 0,`
`(vectorAOS.size()-1));`

`glEnableVertexAttribArray(ind_colores);`
//Como vamos a pintar el relleno, habilito la tabla de colores
`glDrawElements(GL_TRIANGLES,`
`indices.size()*3, GL_UNSIGNED_INT, 0);`
`glBindVertexArray(0);`

Problema 1.7.

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).

// Ejercicio 1.7

```
float r = float(ancho_actual)/alto_actual;

float x0, x1, y0, y1;
float z0 = -1.0, z1 = 1.0;

if (r >= 1){
    x0 = -r, x1 = r;
    y0 = -1.0, y1 = 1.0;
}
else{
    x0 = -1.0, x1 = 1.0;
    y0 = -1/r, y1 = 1/r;
}

float sx = 2 / (x1-x0), sy = 2 / (y1-y0), sz = 2/(z1-z0);
float cx = (x0+x1) / 2, cy = (y0+y1) / 2, cz = (z0+z1) / 2;

GLfloat matriz_proyeccion[16] = {
    sx, 0, 0, -cx*sx,
    0, sy, 0, -cy*sy,
    0, 0, sz, -cz*sz,
    0, 0, 0, 1
};
glUniformMatrix4fv( loc_mat_proyeccion, 1, GL_TRUE, matriz_proyeccion );
///////////////////////////////
```

Sea $C = [\hat{e}_x, \hat{e}_y, \hat{e}_z]$ un marco de referencia cartesiano. Sean \vec{a}, \vec{b} dos vectores cualesquiera. Si sus coordenadas en el marco C son $\vec{a} = ((a_x, a_y, a_z, 0))^T$ y $\vec{b} = ((b_x, b_y, b_z, 0))^T$, entonces:

$$\vec{a} = a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z \quad \vec{b} = b_x \hat{e}_x + b_y \hat{e}_y + b_z \hat{e}_z$$

$$\begin{aligned}\vec{a} \cdot \vec{b} &= (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_x \hat{e}_x + b_y \hat{e}_y + b_z \hat{e}_z) = (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_x \hat{e}_x) + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_y \hat{e}_y) + \\ &\quad + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \cdot (b_z \hat{e}_z) = a_x b_x (\hat{e}_x \cdot \hat{e}_x) + a_x b_y (\hat{e}_x \cdot \hat{e}_y) + a_x b_z (\hat{e}_x \cdot \hat{e}_z) + a_y b_x (\hat{e}_y \cdot \hat{e}_x) + a_y b_y (\hat{e}_y \cdot \hat{e}_y) + \\ &\quad + a_y b_z (\hat{e}_y \cdot \hat{e}_z) + a_z b_x (\hat{e}_z \cdot \hat{e}_x) + a_z b_y (\hat{e}_z \cdot \hat{e}_y) + a_z b_z (\hat{e}_z \cdot \hat{e}_z) = [a_x b_x + a_y b_y + a_z b_z]\end{aligned}$$

$$\begin{aligned}\vec{a} \times \vec{b} &= (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_x \hat{e}_x + b_y \hat{e}_y + b_z \hat{e}_z) = (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_x \hat{e}_x) + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_y \hat{e}_y) + \\ &\quad + (a_x \hat{e}_x + a_y \hat{e}_y + a_z \hat{e}_z) \times (b_z \hat{e}_z) = a_x b_x (\hat{e}_x \times \hat{e}_x) + a_x b_y (\hat{e}_x \times \hat{e}_y) + a_x b_z (\hat{e}_x \times \hat{e}_z) + \\ &\quad + a_y b_x (\hat{e}_y \times \hat{e}_x) + a_y b_y (\hat{e}_y \times \hat{e}_y) + a_y b_z (\hat{e}_y \times \hat{e}_z) + a_z b_x (\hat{e}_z \times \hat{e}_x) + a_z b_y (\hat{e}_z \times \hat{e}_y) + a_z b_z (\hat{e}_z \times \hat{e}_z) = \\ &= [(a_y b_z - a_z b_y) \hat{e}_x + (a_z b_x - a_x b_z) \hat{e}_y + (a_x b_y - a_y b_x) \hat{e}_z].\end{aligned}$$

- ④ Dada un vector \vec{v} , $\vec{v} \times \vec{v} = 0$ pues $\vec{v} \times \vec{v} = -\vec{v} \times \vec{v}$ y el único vector que cumple ser su mismo opuesto es el $\vec{0}$.

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y) \cdot \hat{e}_x + (a_z b_x - a_x b_z) \cdot \hat{e}_y + (a_x b_y - a_y b_x) \cdot \hat{e}_z$$

Para ver que son perpendiculares veamos que el producto escalar es 0:

$$\begin{aligned}1) (\vec{a} \times \vec{b}) \cdot \vec{a} &= (a_y b_z - a_z b_y) \cdot a_x + (a_z b_x - a_x b_z) \cdot a_y + (a_x b_y - a_y b_x) \cdot a_z = \\ &= a_x a_y b_z - a_x a_z b_y + a_y a_z b_x - a_x a_y b_z + a_x a_z b_y - a_y a_z b_x = 0. \checkmark\end{aligned}$$

$$\begin{aligned}2) (\vec{a} \times \vec{b}) \cdot \vec{b} &= (a_y b_z - a_z b_y) \cdot b_x + (a_z b_x - a_x b_z) \cdot b_y + (a_x b_y - a_y b_x) \cdot b_z = \\ &= a_x b_x b_z - a_z b_x b_y + a_z b_x b_y - a_x b_y b_z + a_x b_y b_z - a_y b_x b_z = 0! \checkmark\end{aligned}$$

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).

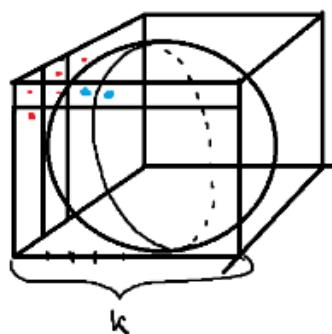
(continua en la siguiente transparencia)

Problema 2.1. (continuación)

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

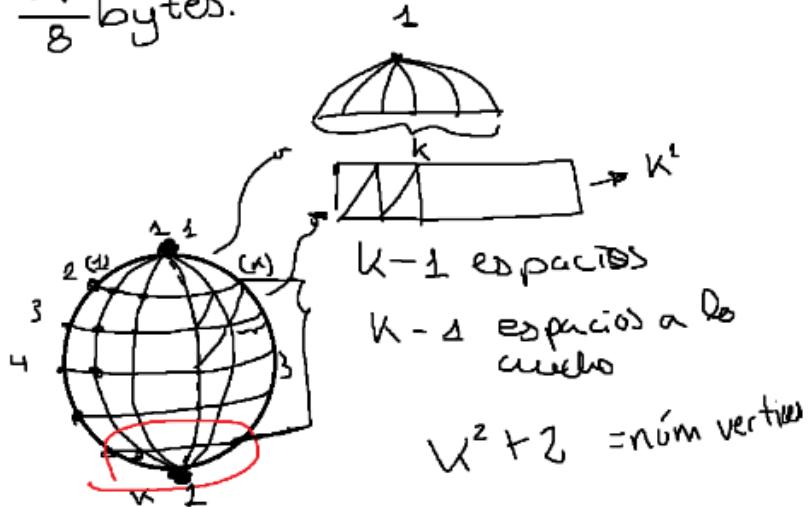
- ▶ Expresa el tamaño de ambas representaciones en bytes como una función de k .
- ▶ Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- ▶ Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

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



Si metemos la esfera en un cubo de lado k (k celdas)
Tenemos en total k^3 celdas

Por tanto, el modelo ocupa k^3 bits ó
 $\frac{k^3}{8}$ bytes.



Tapas son $k-1$ triángulos

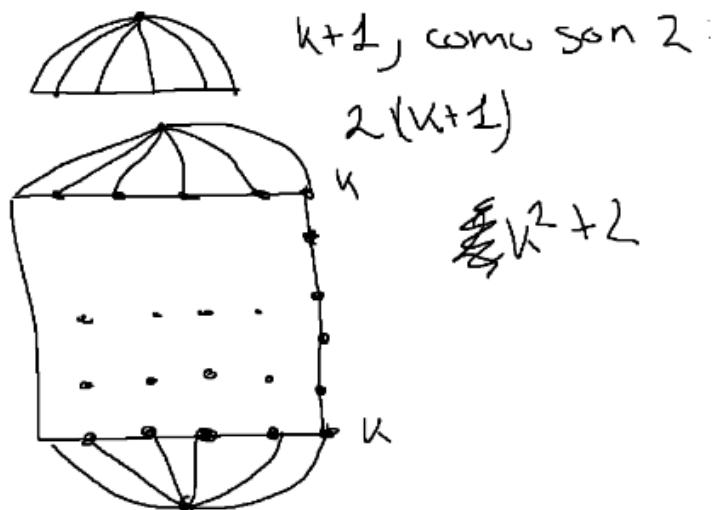
$$2(k-1) + 2(k-3)^2$$

$$2k^2 + 18 - 8k + 2k - 2 = 2k^2 - 4k + 16 - 16$$



trian

$3 \cdot N_t = \text{Num indices}$

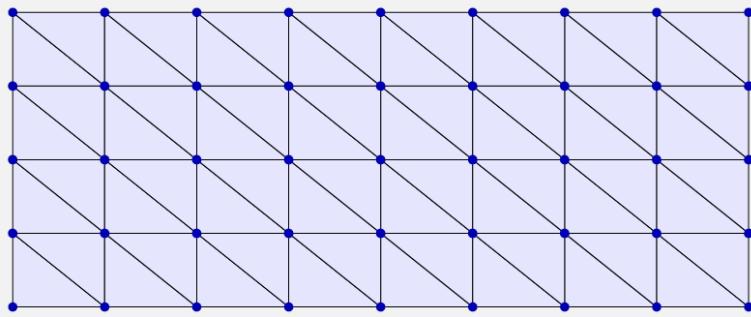


Resumiendo:

$$\begin{aligned} k^2 + 2 \text{ vértices} &\leftrightarrow (k^2 + 2) \cdot 3 \cdot 4 \text{ bytes} \\ 3 \cdot \underbrace{(2k^2 - 4k + 16)}_{n \text{ triángulos}} \cdot 4 &\quad \underbrace{n \text{ bytes para}}_{\text{vértices}} \\ &\quad \underbrace{n \text{ bytes de}}_{\text{tabla triángulos}} \\ \text{num. de enteros para} & \\ \text{rep. todos los triángulos} & \end{aligned}$$

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)

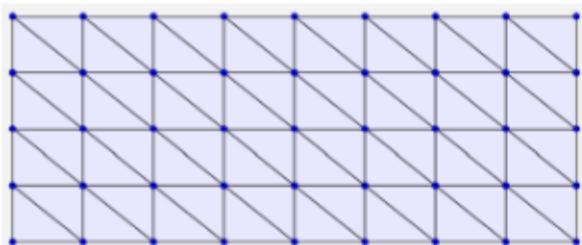
GIM: Informática Gráfica- curso 22-23- creado el 17 de septiembre de 2022 – transparencia 63 de 184

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.

2.2.



Hay $(n+1)(m+1)$ vértices

Hay $2 \cdot n \cdot m$ triángulos B es flat

Para vértices $(n+1)(m+1) \cdot 2 \cdot 4$

Pura triángulos

$$3 \cdot 2 \cdot n \cdot m \cdot 4 = 24n \cdot m$$

c)

$$\lim_{n \rightarrow \infty} \frac{sn + 8nm + 8m + 8}{24n^m} = \frac{0}{24} = \frac{1}{3}$$

\uparrow
 $n = \text{cras}$

Hay 3 veces más triángulos que vértices.

$$\lim_{n \rightarrow \infty} M_n = \frac{8 \text{ dy}}{24 \text{ mm}} + \frac{8 \text{ dy}}{24 \text{ mm}} + \frac{8 \text{ dy}}{24 \text{ mm}} + \frac{8 \text{ dy}}{24 \text{ mm}}$$

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).

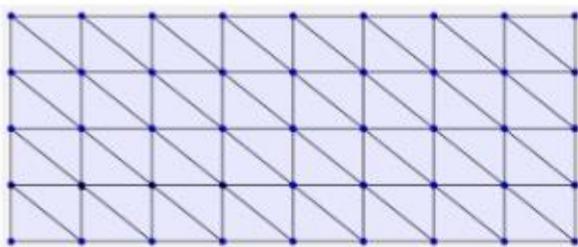
Responde a estas cuestiones:

(continua en la siguiente transparencia)

Problema 2.3. (continuación)

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

2.3.



Hay m tiras, cada tira necesita
 $2n$ vértices

En total guardas $2nm$ vértices

Tienes $8 \cdot m$ Bytes de puntos

Tienes $2nm \cdot 2 \cdot 4$ Bytes para
vértices $\overset{2}{\text{Bytes}}$ $\overset{\text{Byte flat}}{\text{Bytes}}$ $\overset{\text{Bytes del}}{\text{2.2}}$

$$\lim_{n \rightarrow \infty, m \rightarrow \infty} \frac{32nm + 8n + 8m + 8}{16nm + 8m} =$$

= $\boxed{2}$ → las mallas indexadas
ocupan el doble.
(en 2D) ó

2 comp. por vértices)

c) $\lim_{m \rightarrow \infty, n \rightarrow \infty} \frac{n \text{ vértices en 2.2}}{(m-1)(n-1)} = \frac{1}{2}$

$$\lim_{m \rightarrow \infty, n \rightarrow \infty} \frac{(m-1)(n-1)}{2nm} = \frac{1}{2}$$

La malla indexada tardará la mitad
que 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:

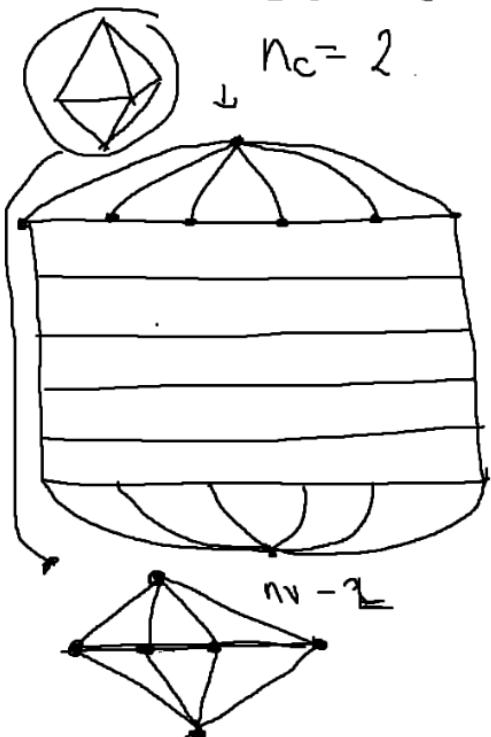
$$n_A = 3(n_V - 2)$$

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

(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).

2. 4.

Si tenemos n_V vértices



$$3 \cdot n_C = 2 n_A$$

fórmula Euler

$$n_V - n_A + n_C = 2$$

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 `Tupla2i` (contiene dos `int`) 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.

```
void calcularAristas(){
    //Ejercicio 2.5 1) NO COHERENCIA
    std::vector<Tupla2i> ari;
    std::map<Tupla2i, Tupla2i> ari_mp;
    for(int i = 0; i < triangulos.size(); i++){
        for(int j=0; j < 3; j++){
            if(!ari_mp.find({triangulos[i][j],
                triangulos[i][(j+1)%3]}) &&
                !ari_mp.find(triangulos[i][(j+1)%3],
                {triangulos[i][j]}))
                ari_mp.insert({{triangulos[i][j],
                    triangulos[i][(j+1)%3]}});
        }
    }
    //Este codigo calcularía en nlog(n)
}
```

```
void calcularAristas(){
    //Ejercicio 2.5 2) COHERENCIA
    std::vector<Tupla2i> ari;
    std::map<Tupla2i, Tupla2i> ari_mp;
    for(int i = 0; i < triangulos.size(); i++){
        for(int j=0; j < 3; j++){
            if(triangulos[i][j]<triangulos[i][(j+1)%3]){
                ari.push_back({{triangulos[i][j],triangulos[i][(j+1)%3]}});
            }
        }
    }
    //Este codigo calcularía en n
}
```

Problema 2.6.

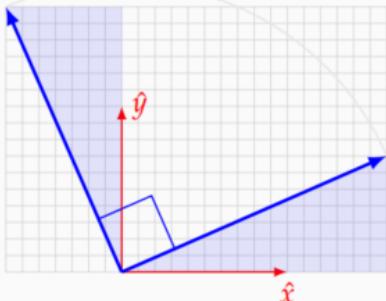
Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función que acepta un puntero a una **MallaInd** y devuelve un número real (asumir que se dispone del tipo **Tupla3f** y de los operadores usuales de tuplas o vectores, es decir suma **+**, resta **-**, producto escalar *****, producto vectorial **×**, módulo **||** **||**, etc ...).

```
// Ejercicio 2.6
void calculaArea(){
    double area = 0.0;
    for(int i=0; i < triangulos.size(); i++){
        Tupla3f v1 = (vertices[triangulos[i][0]] - vertices[triangulos[i][1]]);
        Tupla3f v2 = (vertices[triangulos[i][0]] - vertices[triangulos[i][2]]);
        Tupla3f cros = v1.cross(v2); // Producto vectorial, su módulo será el cuadrado de su
área
        area += sqrt(cros.lengthSq());
    }
    area = area/2;
}
```

Problema 2.7.

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

$$P(\vec{u}) = -b\hat{x} + a\hat{y}$$

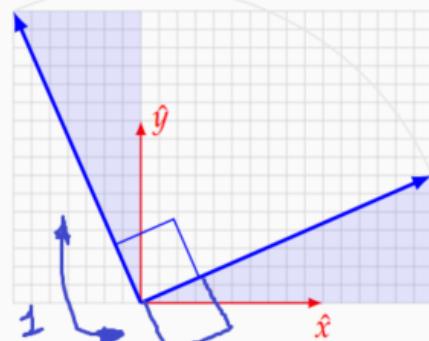


$$\vec{u} = a\hat{x} + b\hat{y}$$

$$P(\vec{u}) = -b\hat{x} + a\hat{y}$$

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

$$P(\vec{u}) = -b\hat{x} + a\hat{y}$$



$$P'(\vec{u}) = b\hat{x} - a\hat{y}$$

Problema 2.12.

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

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

$$\vec{a} = (a_x \cdot \hat{x}, a_y \cdot \hat{y});$$

$$\vec{b} = (b_x \cdot \hat{x}, b_y \cdot \hat{y});$$

$$\text{Rot}[\theta] \vec{a} = \begin{bmatrix} \cos\theta \cdot ax \cdot \hat{x} - \sin\theta \cdot ay \cdot \hat{y} \\ \sin\theta \cdot ax \cdot \hat{x} + \cos\theta \cdot ay \cdot \hat{y} \end{bmatrix}$$

$$\text{Rot}[\theta]_B^A = \begin{bmatrix} \cos\theta \cdot b_x \cdot \hat{x} - \sin\theta \cdot b_y \cdot \hat{y} \\ \sin\theta \cdot b_x \cdot \hat{x} + \cos\theta \cdot b_y \cdot \hat{y} \end{bmatrix}, \quad \begin{matrix} b_x \\ b_y \end{matrix}$$

$$R_\theta(\vec{a}) \cdot R(\vec{b}) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} \cos^2\theta & -\cos\theta\sin\theta & 0 \\ \sin\theta\cos\theta & \cos^2\theta & -\sin\theta\sin\theta \\ 0 & \sin\theta\sin\theta & \cos^2\theta \end{pmatrix}$$

$$\Rightarrow \cos^2\theta ux^2x + \sin^2\theta uy^2y \quad | \quad 2^{\text{nd}} \text{ part}$$

$$\Rightarrow 1 - \sin^2 \theta \alpha x b x + \sin^2 \theta c y b y$$

$$axbx - \sin^2 \theta (axbx + ay by),$$

—

—

$$B_1(\vec{a}) \cdot B(\vec{b}) = a_x b_x + a_y b_y$$

18-1990

Problema 2.13.

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

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

$$\| \vec{a} \| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

P

$$\text{Rot}[\theta] \vec{a} = \left[\underbrace{[\cos \theta \cdot a_x \cdot \hat{x} - \sin \theta \cdot a_y \cdot \hat{y}]}_{a_x}, \underbrace{[\sin \theta \cdot a_x \cdot \hat{x} + \cos \theta \cdot a_y \cdot \hat{y}]}_{a_y} \right]$$

$$\| \text{Rot}[\theta] \vec{a} \| = \sqrt{\underbrace{\cos^2 \theta a_x^2}_{\frac{1}{1}} + \underbrace{\sin^2 \theta a_y^2}_{\frac{1}{1}} + \cancel{\hat{x}^2} + \underbrace{\sin^2 \theta a_x^2}_{\frac{1}{1}} + \underbrace{\cos^2 \theta a_y^2}_{\frac{1}{1}} - \cancel{\hat{x}^2}}$$

$$\underbrace{\cos^2 \theta a_x^2 + \sin^2 \theta a_x^2 + \sin^2 \theta a_y^2 + \cos^2 \theta a_y^2}_{a_x^2 + a_y^2} + a_y^2 (\underbrace{\cos^2 \theta + \sin^2 \theta}_{1})$$

$$a_x^2 + a_y^2$$

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 θ , se cumple:

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

$$2.14 \quad R_\theta(\vec{a} \times \vec{b}) = R_\theta(\vec{a}) \times R_\theta(\vec{b})$$

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

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

$$(a_y b_z - a_z b_y) \hat{x}$$

$$\cos \theta (a_z b_x - a_x b_z) - \sin \theta (a_x b_y - a_y b_x) = y$$

$$\sin \theta (a_z b_x - a_x b_z) + \cos \theta (a_x b_y - a_y b_x) = z$$

$$R_\theta(\vec{a}) = \begin{pmatrix} a_x & x \\ \cos \theta a_y - \sin \theta a_z & y \\ \sin \theta a_y + \cos \theta a_z & z \end{pmatrix}$$

$$R_\theta(\vec{b}) = \begin{pmatrix} b_x & x \\ \cos \theta b_y - \sin \theta b_z & y \\ \sin \theta b_y + \cos \theta b_z & z \end{pmatrix}$$

$$\begin{pmatrix} (\cos \theta a_y - \sin \theta a_z)(\sin \theta b_y + \cos \theta b_z) - (\sin \theta a_y + \cos \theta a_z)b_x \\ \sin \theta a_y + \cos \theta a_z \cdot b_x - a_x(\sin \theta b_y + \cos \theta b_z) \\ a_x(\cos \theta b_y - \sin \theta b_z) - b_x(\cos \theta a_y - \sin \theta a_z) \end{pmatrix}$$

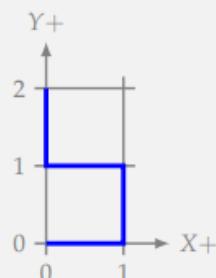
$$\cos^2 a_y b_y + \cos^2 a_y b_z - \sin^2 a_y b_y - \sin^2 a_y b_z - \sin a_y b_x - \cos a_y b_z,$$

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(){
```

```
// A) GL_LINE_LOOP
```

```
// Polígono de n lados
```

```
int nombre_vao = 0;
```

```
int nombre_vbo_posiciones, ind_pos, ind_colores;
```

```
std::vector<Tupla2d> posiciones;
```

```

posiciones.push_back({0,0});
posiciones.push_back({1,0});
posiciones.push_back({1,1});
posiciones.push_back({0,1});
posiciones.push_back({0,2});

if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_DOUBLE, 2, posiciones.size(),
posiciones.data());
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa
}

//Con esto le decimos que el color por defecto es el negro para los vertices del VAO activo
ahora mismo
//El último 1 se refiere a la transparencia
glVertexAttrib3f(ind_colores, 0.0,0.0,1.0,1.0);

glDrawArrays(GL_LINES, 0, posiciones.size());

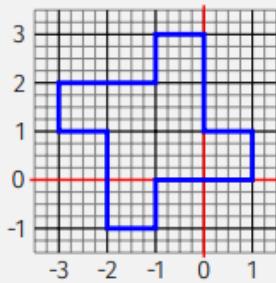
glBindVertexArray(0);

//Hasta aquí hemos dibujado las aristas del polígono.
}

```

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** con **MAT_Traslacion** y **MAT_Rotacion**.

// Ejercicio 2.16

<pre>void gancho_trasladado(float x, float y){ compMM(MAT_Translacion({x,y, 0.0})); gancho(); compMM(MAT_Translacion({-x,-y, 0.0})); } }</pre>	<pre>void gancho_x4(){ compMM(MAT_Translacion({-1,0})); //Esta es la traslación de todos los ganchos gancho_trasladado(1, 0); compMM(MAT_Rotacion(90,{0,0,1})); gancho_trasladado(1, 0); compMM(MAT_Rotacion(90,{0,0,1})); gancho_trasladado(1, 0); compMM(MAT_Rotacion(90,{0,0,1})); gancho_trasladado(1, 0); }</pre>
---	--

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 \vec{p}_0 y \vec{p}_1 , puntos cuyas coordenadas de mundo son $\mathbf{p}_0 = (x_0, y_0, 1)^t$ y $\mathbf{p}_1 = (x_1, y_1, 1)^t$. Estas coordenadas se pasan como parámetro a dicha función (como **Tupla3f**)

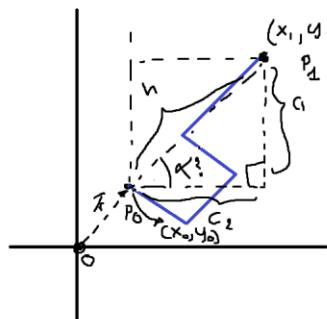
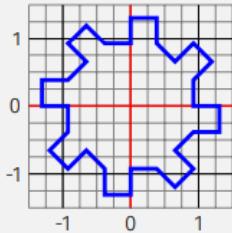
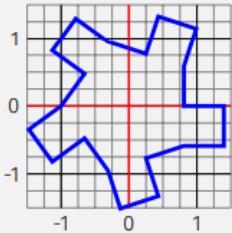
Escribe una solución (a) acumulando matrices de rotación, traslación y escalado en la matriz *modelview* de OpenGL. Escribe otra solución (b) en la cual la matriz *modelview* se calcula directamente sin necesidad de usar funciones trigonométricas (como lo son el arcotangente, el seno, coseno, arcoseno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

// Ejercicio 2.17

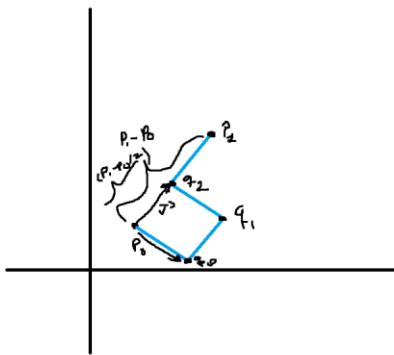
```
void ganchoEntrePuntos(Tupla3f p0, Tupla3f p1){  
  
void ganchoMultiple(int n){  
  
    std::vector<Tupla3f> posiciones;  
  
    for(int i=0; i<n; i++){  
        posiciones.push_back({cos(2*M_PI*i/n), sin(2*M_PI*i/n), 0.0});  
    }  
  
    for(int i=0; i<n; i++){  
        ganchoEntrePuntos(posiciones[i], posiciones[(i+1)%n]);  
    }  
}
```

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).



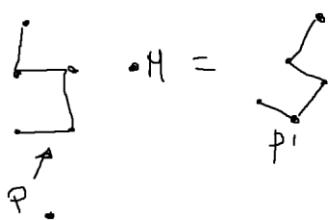
$$\begin{aligned} h^2 &= C_2^2 + C_1^2 \\ h &= \sin \alpha \cdot C_1 \\ \sin \alpha &= \frac{h}{C_1} \\ \alpha &= \arcsin\left(\frac{h}{C_1}\right) \end{aligned}$$



$$\sqrt{(P_1 - P_0) \cdot \text{sq-length}} \rightarrow \text{escalado}$$

$(P_0 - O) \rightarrow \text{traducción}$

$$C_2 = x_1 - x_0 \quad C_1 = y_1 - y_0$$



$$\begin{matrix} P_0 \\ M \\ 3 \times 3 \\ 3 \times 3 \end{matrix} = \begin{matrix} P'_0 \\ P'^{-1} \\ P' \\ 3 \times 3 \end{matrix}$$

$$(P_1 - P_0)/2 = \vec{V} = a\vec{x} + b\vec{y}$$

$$\vec{V}_{q_0} = b\vec{x} - a\vec{y}$$

$$P_0 + \vec{V}_{q_0} = q_0 \quad P_0 + \vec{V}_{q_0} + \vec{V} = q_1$$

$$P_0 + \vec{V} = q_2 \quad P_0 + 2\vec{V} = P_1$$

$$\{P_0, q_1, q_2, q_0, P_1\} = P' \text{ Juntos org para matriz } M$$

$$V_i(a_i, b_i) = a_i \hat{x} + b_i \hat{y}$$

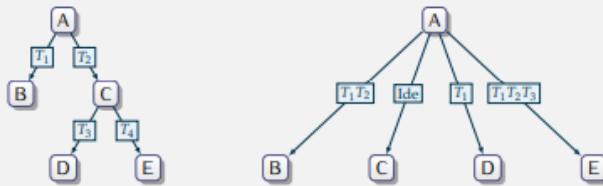
$$V'_i(a_i, b_i) = a_i \underbrace{\hat{x}}_{c_0 \hat{x} + c_1 \hat{y}} + b_i \underbrace{\hat{y}}_{d_0 \hat{x} + d_1 \hat{y}}$$

$$c_0 \hat{x} + c_1 \hat{y} \quad d_0 \hat{x} + d_1 \hat{y}$$

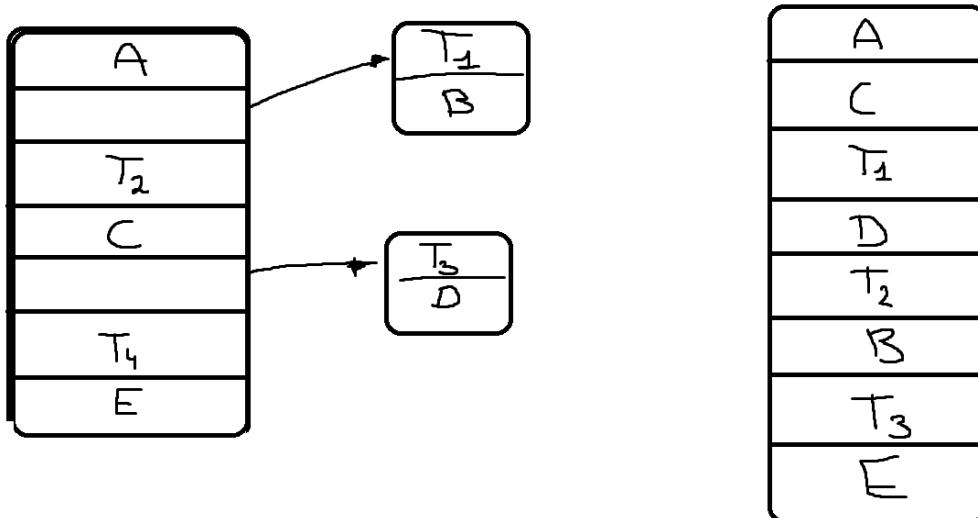
$$\begin{pmatrix} e_i \\ f_i \end{pmatrix} = \begin{pmatrix} c_0 & d_0 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} a_i \\ b_i \end{pmatrix}$$

Problema 2.19.

Dados los dos siguientes grafos de escena sencillos:

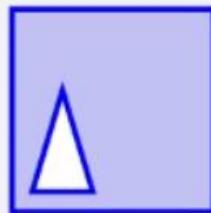


Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones: T_1, T_2 y T_3 .



Problema 2.20.

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**)

// Ejercicio 2.20

<pre>void FiguraSimple(){ int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla3d> posiciones; posiciones.push_back({0,0,0}); posiciones.push_back({1,0,0}); posiciones.push_back({1,1,0}); posiciones.push_back({0,1,0}); posiciones.push_back({0.25,0.25,0}); posiciones.push_back({0.75,0.25,0}); posiciones.push_back({0.5,0.5,0}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; indices.push_back({0,1,2}); indices.push_back({2,3,0}); indices.push_back({4,5,6});</pre>	<pre>//El color por defecto es el negro para los //vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,1.0); // Asi no cogemos el centro, para hacer //las lineas, como es la funcion DrawArrays //no cogera la tabla de indices glDrawArrays(GL_LINE_LOOP, 0, 4); glVertexAttrib3f(ind_colores, 0.0,0.0,0.25); glDrawElements(GL_TRIANGLES, 2*3, GL_FLOAT, indices.data()); glVertexAttrib3f(ind_colores, 0.0,0.0,1.0); glDrawArrays(GL_LINE_LOOP, 4, 3); glVertexAttrib3f(ind_colores, 1.0,1.0,1.0); glDrawElements(GL_TRIANGLES, 3, GL_FLOAT, indices.data() + 6 * sizeof(float)); glBindVertexArray(0); }</pre>
---	---

```

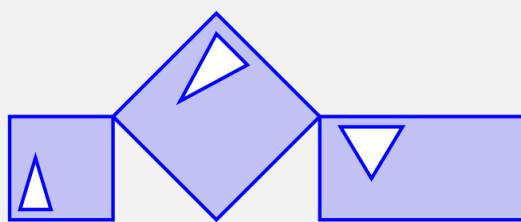
if(nombre_vao == 0){
    nombre_vao = crearVAO();

    if(posiciones.size() > 0) {
        nombre_vbo_posiciones =
        CrearVBOAtrib(ind_pos, GL_FLOAT, 3,
        posiciones.size(), posiciones.data());
    }
    if(indices.size() > 0){
        nombre_vbo_ind = CrearVBOInd(
        indices );
    }
}
else{
    glBindVertexArray(nombre_vao); // VAO
ya creado, simplemente se activa
}

```

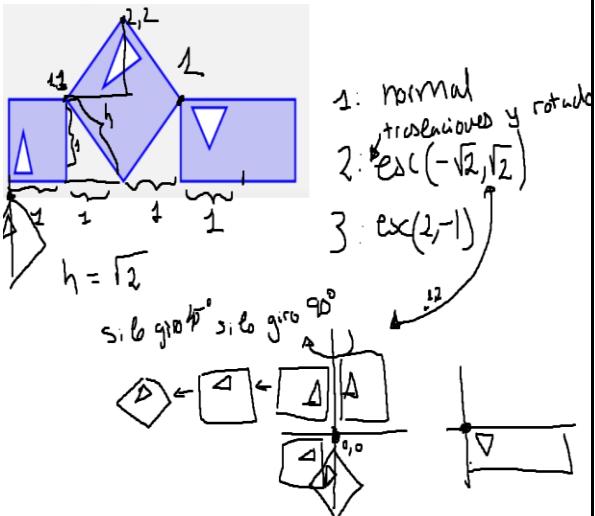
Problema 2.21.

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**:



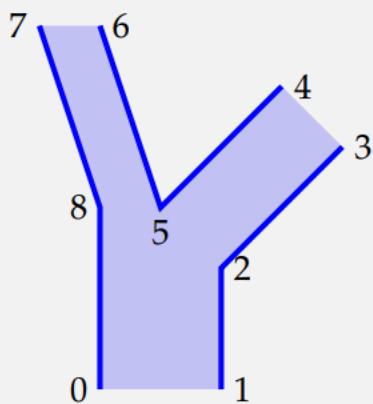
// Ejercicio 2.21

```
void FiguraCompleja(){
    figuraSimple();
    pushMM();
    compMM(MAT_Translacion({2,2,0}));
    compMM(MAT_Rotacion(135,{0,0,1}));
    compMM(MAT_Escalado(-sqrt(2),
    sqrt(2),1));
    figuraSimple();
    popMM();
    pushMM();
    compMM(MAT_Translacion({3,1,0}));
    compMM(MAT_Escalado(2,-1,1));
    figuraSimple();
    popMM();
}
```



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)

CIM - Informática Gráfica - curso 22-23 - creado el 17 de septiembre de 2022 - transparencia 167 de 197

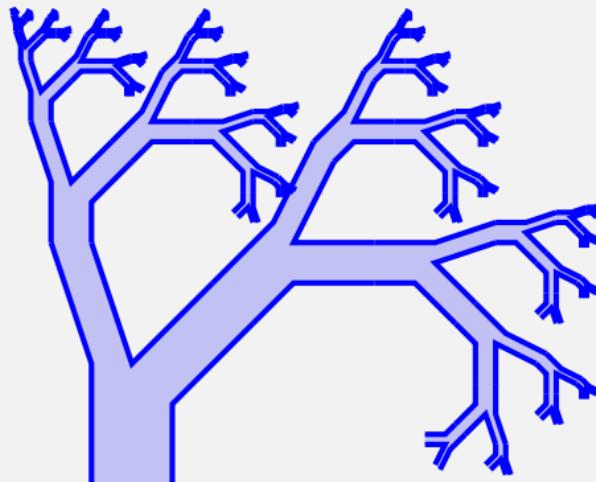
// Ejercicio 2.22

```
if(nombre_vao == 0){
```

<pre> void Tronco(){ int n=5; int nombre_vao = 0; int nombre_vbo_posiciones, ind_pos, ind_colores; std::vector<Tupla3d> posiciones; posiciones.push_back({0,0,0}); posiciones.push_back({1,0,0}); posiciones.push_back({1,1,0}); posiciones.push_back({2,2,0}); posiciones.push_back({1.5,2.5,0}); posiciones.push_back({0.5,1.5,0}); posiciones.push_back({0,0,3,0,0}); posiciones.push_back({-0.5,3,0,0}); posiciones.push_back({0,0,1.5,0}); // Poligono de n lados std::vector<Tupla3u> indices; int nombre_vbo_ind; indices.push_back({0,8,1}); indices.push_back({8,5,2}); indices.push_back({1,2,8}); indices.push_back({7,6,8}); indices.push_back({8,6,5}); indices.push_back({4,3,2}); indices.push_back({4,5,2}); } </pre>	<pre> nombre_vao = crearVAO(); if(posiciones.size() > 0) { nombre_vbo_posiciones = CrearVBOAtrib(ind_pos, GL_FLOAT, 3, posiciones.size(), posiciones.data()); } if(indices.size() > 0){ nombre_vbo_ind = CrearVBOInd(indices); } else{ glBindVertexArray(nombre_vao); // VAO ya creado, simplemente se activa } //El color por defecto es el negro para los vertices del VAO activo ahora mismo //El ultimo 1 se refiere a la transparencia glVertexAttrib3f(ind_colores, 0.0,0.0,1.0); // Asi no cogemos el centro, para hacer las lineas, como es la funcion DrawArrays no cogera la tabla de indices glDrawArrays(GL_LINES, 1, 3); glDrawArrays(GL_LINES, 4, 3); glDrawArrays(GL_LINES, 7, 3); glVertexAttrib3f(ind_colores, 0.0,0.0,0.25); glDrawElements(GL_TRIANGLES, indices.size()*3, GL_FLOAT, indices.data()); glBindVertexArray(0); } </pre>
--	--

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 modificable en dicho código (en la figura es 6)



GIM: Informática Gráfica- curso 22-23- creado el 17 de septiembre de 2022 – transparencia 168 de 184.

```
void Arbol(int n, int exp){  
    Tronco();  
    if(n!=0){  
        pushMM();  
        MAT_Translacion({-  
            0.5*pow(0.5,exp),3*pow(0.5,exp),0});  
        MAT_Escalado({0.5,0.5,0.5});  
        Arbol(n-1,exp+1);  
        popMM();  
        pushMM();  
  
        MAT_Translacion({1.5*pow(sqrt(0.5),exp),2.5  
            *pow(sqrt(0.5),exp),0});  
        MAT_Rotacion(45,{0,0,1});  
  
        MAT_Escalado(sqrt(0.5),sqrt(0.5),sqrt(0.5));  
        Arbol(n-1,exp+1);  
        popMM();  
    }  
}
```

Problema 2.24.

Supón que dispones de dos funciones para dibujar dos mallas u objetos simples: **Semiesfera** y **Cilindro**. La semiesfera (en coordenadas maestras) tiene radio unidad, centro en el origen y el eje vertical en el eje Y. Igualmente el cilindro tiene radio y altura unidad, el centro de la base está en el origen, y su eje es el eje Y.



(continua en la siguiente transparencia)

Problema 2.24. (continuación)

Con estas dos primitivas queremos escribir el código que visualiza la figura Android, usando la plantilla de código de prácticas. Para ello:

- ▶ Diseña el grafo de escena (tipo PHIGS) correspondiente, ten en cuenta que hay objetos compuestos que se pueden instanciar más de una vez (cada brazo o pierna se puede construir con un objeto compuesto de dos semiesferas en los extremos de un cilindro).
- ▶ Escribe el código OpenGL para visualizarlo, usando una clase llamada **Android**, derivada de **NodoGrafoEscena**.

Problema 2.25.

Escribe una segunda versión del grafo de escena del problema 2.24, de forma que las transformaciones estén parametrizadas por dos valores reales (α y β) que expresan el ángulo de rotación del brazo izquierdo y derecho (respectivamente), en torno al eje que pasa por los centros de las dos semiesferas superiores de los brazos.

Asimismo, habrá otro parámetro (ϕ) que es el ángulo de rotación de la cabeza (completa: con los ojos y antenas) entorno al eje vertical que pasa por su centro (cuando estos ángulos valen 0, el androide está en reposo y tiene exactamente la forma de la figura del problema anterior).

Escribe el código de una nueva clase (**AndroidParam**, derivada de **NodoGrafoEscena**) para visualizar el androide parametrizado de esta forma.

```
/*
ObjetoModificable::ObjetoModificable(Tupla
3f traslacion, float angulo_rotacion, Tupla3f
rotacion, Tupla3f escalado, Objeto3D &
ObjetoModificable){

    agregar(MAT_Traslacion(traslacion));
    agregar(MAT_Rotacion(angulo_rotacion,
rotacion));
    agregar(MAT_Escalado(escalado(0),
escalado(1), escalado(2)));

    agregar(& ObjetoModificable);
}
*/
```

```
AndroidCabeza::AndroidCabeza(){

    int indice = agregar(MAT_Rotacion((0.0),
{0.0, 1.0, 0.0}));

    agregar(new ObjetoModificable({1, 2.5,
0}, 135, {1,0,0}, {0.1,0.1,0.1}, * new
SemiEsfera(20,20, {0,0,0})));
    agregar(MAT_Traslacion({0, 2.2, 0}));
    agregar(MAT_Rotacion(180, {0,0,1}));
```

```
class AndroidCabeza : public
NodoGrafoEscena
{ public:

    Matriz4f * rotacionCabeza;
    AndroidCabeza() ; // constructor
    unsigned leerNumParametros() const;
    void actualizarEstadoParametro( const
unsigned iParam, const float tSec );
} ;
```

```
class AndroidCuerpo : public
NodoGrafoEscena
{ public:
```

```
    AndroidCuerpo() ; // constructor
} ;
```

```
class AndroidExtremidad : public
NodoGrafoEscena
{ public:
```

```
    Matriz4f * rotacionExtremidad;
    int identificador;
```

<pre> agregar(new SemiEsfera(20, 20, {0, 1, 0})); rotacionCabeza = leerPtrMatriz(indice); } unsigned AndroidCabeza::leerNumParametros() const{ return 1; } void AndroidCabeza::actualizarEstadoParametro (const unsigned iParam, const float tSec) { assert(iParam < leerNumParametros()); switch (iParam) { case 0: * rotacionCabeza = MAT_Rotacion(sin(tSec)*40, {0,1,0}); break; default: * rotacionCabeza = MAT_Rotacion(sin(tSec)*40, {0,1,0}); break; } } AndroidCuerpo::AndroidCuerpo(){ //agregar(MAT_Traslacion({0, 0, 0})); agregar(MAT_Escalado(1, 2, 1)); agregar(new Cilindro(20,20, {0, 1, 0})); } AndroidTubo::AndroidTubo(){ agregar(new ObjetoModificable({0, 3, 0}, 180, {1,0,0}, {1,1,1}, * new SemiEsfera(20,20, {0,1,0}))); agregar(new ObjetoModificable({0, 0, 0}, 0, {1,0,0}, {1,1,1}, * new SemiEsfera(20,20, {0,1,0}))); agregar(MAT_Escalado(1,3,1)); agregar(new Cilindro(20,20, {0, 1, 0})); } </pre>	<pre> class ObjetoModificable : public NodoGrafoEscena { public: ObjetoModificable(Tupla3f traslacion, float angulo_rotacion, Tupla3f rotacion, Tupla3f escalado, Objeto3D & ObjetoModificable) ; // constructor } ; AndroidExtremidad(Tupla3f traslacion, Tupla3f escalado, int id) ; // constructor unsigned leerNumParametros() const; void actualizarEstadoParametro(const unsigned iParam, const float tSec); } ; //** class AndroidTubo : public NodoGrafoEscena { public: AndroidTubo() ; // constructor } ; class NodoAndroid : public NodoGrafoEscena { public: AndroidCabeza * cabeza; AndroidCuerpo * cuerpo; AndroidExtremidad *brazo1, * brazo2, * pierna1, * pierna2; NodoAndroid() ; // constructor unsigned leerNumParametros() const; void actualizarEstadoParametro(const unsigned iParam, const float tSec); } ; NodoAndroid::actualizarEstadoParametro(const unsigned iParam, const float tSec) { assert(iParam < leerNumParametros()); switch (iParam) { case 0: </pre>
--	---

```

AndroidExtremidad::AndroidExtremidad(Tu
pla3f traslacion, Tupla3f escalado, int id){

    identificador = id;

    agregar(MAT_Traslacion(traslacion));
    agregar(MAT_Escalado(escalado[0],
    escalado[1], escalado[2]));

    agregar(MAT_Traslacion({0,2,0}));
    int indice = agregar(MAT_Rotacion((0.0),
    {0.0, 1.0, 0.0}));
    agregar(MAT_Traslacion({0,-2,0}));

    agregar(new AndroidTubo());

    rotacionExtremidad =
leerPtrMatriz(indice);
}

unsigned
AndroidExtremidad::leerNumParametros()
const{

    return 1;
}

void
AndroidExtremidad::actualizarEstadoParam
etro( const unsigned iParam, const float
tSec )
{
    assert(iParam < leerNumParametros());

    switch (iParam)
    {
        case 0:

            if(identificador == 0){
                * rotacionExtremidad =
MAT_Rotacion(sin(tSec)*40, {1,0,0});
            }
            if(identificador == 1){
                * rotacionExtremidad =
MAT_Rotacion(-1*sin(tSec)*40, {1,0,0});
            }

            break;

        default:
            * rotacionExtremidad =
MAT_Rotacion(sin(tSec)*40, {1,0,0});
    }
}

```

```

    brazo1-
>actualizarEstadoParametro(iParam, tSec);
    brazo2-
>actualizarEstadoParametro(iParam, tSec);
    cabeza-
>actualizarEstadoParametro(iParam, tSec);
    break;

    default:
        brazo1-
>actualizarEstadoParametro(iParam, tSec);
        brazo2-
>actualizarEstadoParametro(iParam, tSec);
        cabeza-
>actualizarEstadoParametro(iParam, tSec);

        break;
    }
}

```

```

        break;
    }

}

NodoAndroid::NodoAndroid(){

    cabeza = new AndroidCabeza();
    agregar(cabeza);

    cuerpo = new AndroidCuerpo();
    agregar(cuerpo);

    brazo1 = new AndroidExtremidad({-
1.4,1,0}, {0.3, 0.3, 0.3}, 0);
    brazo2 = new
AndroidExtremidad({1.4,1,0}, {0.3, 0.3, 0.3},
1);
    agregar(brazo1);
    agregar(brazo2);

    pierna1 = new AndroidExtremidad({-0.5, -0.5,0}, {0.3, 0.3, 0.3}, 2);
    pierna2 = new AndroidExtremidad({0.5, -0.5,0}, {0.3, 0.3, 0.3}, 3);
    agregar(pierna1);
    agregar(pierna2);

}

unsigned
NodoAndroid::leerNumParametros() const{

    return 1;
}

void

```

Problema 3.1.

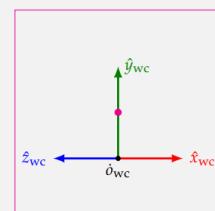
Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja, verde y azul), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

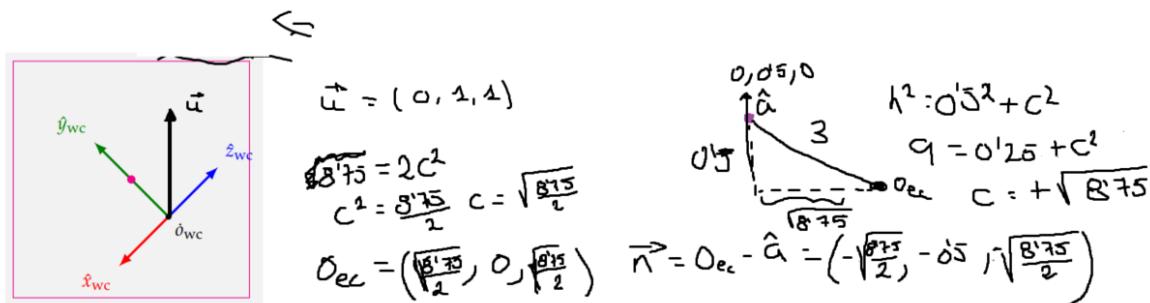
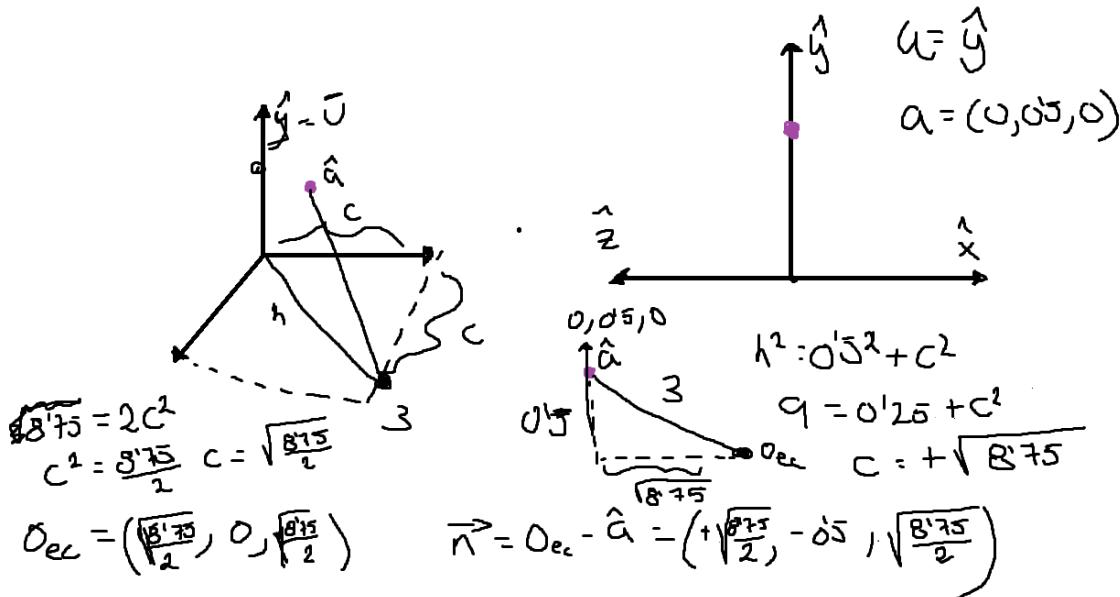
1. El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2. El punto de coordenadas $(0,0.5,0)$ (aparece como un disco de color morado en la figura) debe aparecer en el centro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





Problema 3.3.

Escribe el código para calcular los vectores de coordenadas x_{ec} , y_{ec} , z_{ec} y o_{ec} que definen el marco de vista a partir de los vectores de coordenadas a , u y n (todos estos vectores de coordenadas son de tipo **Tupla3f**).

Problema 3.4.

Partiendo de los vectores de coordenadas x_{ec} , y_{ec} , z_{ec} y o_{ec} que se calculan en el problema anterior, escribe el código que calcula explicitamente 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).

// Ejercicio 3.3

```
void CalcularMarcoCamara(Tupla3f a,
Tupla3f u, Tupla3f n){

    Tupla3f o_ec = a+n;
    Tupla3f z_ec = n.normalized();
    Tupla3f x_ec = (u.cross(n)).normalized();
```

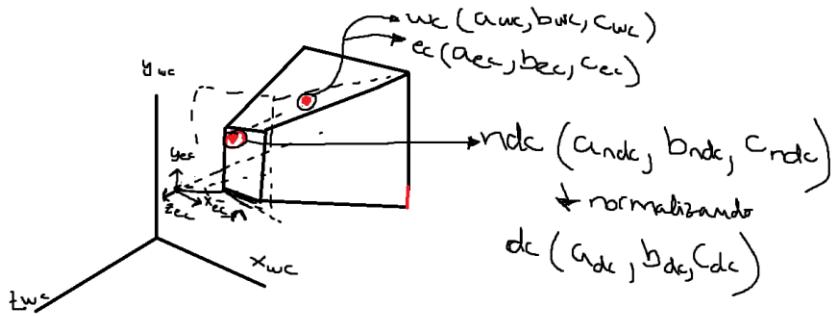
// Ejercicio 3.4

```
void CalcularMarcoCamaraXYZ(Tupla3f x_ec,
Tupla3f y_ec, Tupla3f z_ec, Tupla3f o_ec){

    Matriz4f * V;
    for(int i=0; i<2; i++){
        V[0][i] = x_ec[i];
        V[1][i] = y_ec[i];
        V[2][i] = z_ec[i];
        V[3][i] = -o_ec[i];
```

```
Tupla3f y_ec =  
(z_ec.cross(x_ec)).normalized();  
}  
  
V[1][i] = y_ec[i];  
V[2][i] = z_ec[i];  
}  
V[0][3] = -x_ec.dot(o_ec);  
V[1][3] = -y_ec.dot(o_ec);  
V[2][3] = -z_ec.dot(o_ec);  
V[3][3] = 1;  
}
```

Explicación pasar a coordenadas normalizadas de proyección:



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 $\mathbf{o}_{ec} = (c_x, c_y, c_z + s + 2)$, 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.

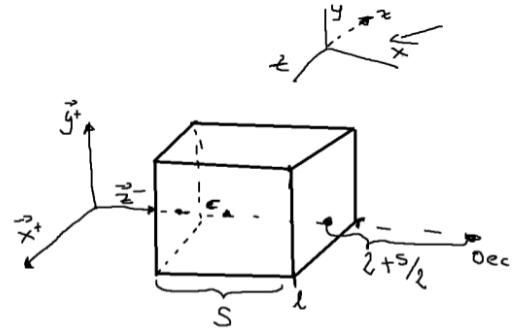
(continua en la siguiente página)

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.

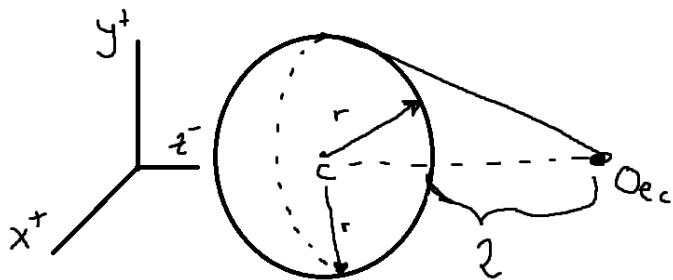
Con estos requerimientos, indica como calcular los valores l, r, t, b, n y f (para obtener la matriz Q de proyección), en función de s y (c_x, c_y, c_z) .



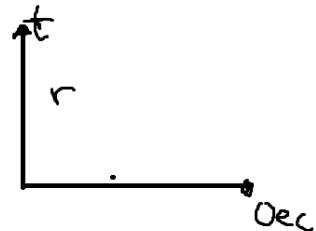
$$\begin{aligned}
 n &= 2 + \frac{s}{2} & f &= 2 + \frac{3s}{2} \\
 r &= -\frac{s}{2} & t &= \frac{s}{2} \\
 l &= +\frac{s}{2} & b &= -\frac{s}{2}
 \end{aligned}$$

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}).



$$\mathbf{o}_{es} = (c_x, c_y, c_z + 2 + r)$$



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

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 $w = h$.

Este problema es parecido al del tema 1. Debemos modificar t/b o r/e según el ratio del *viewport*

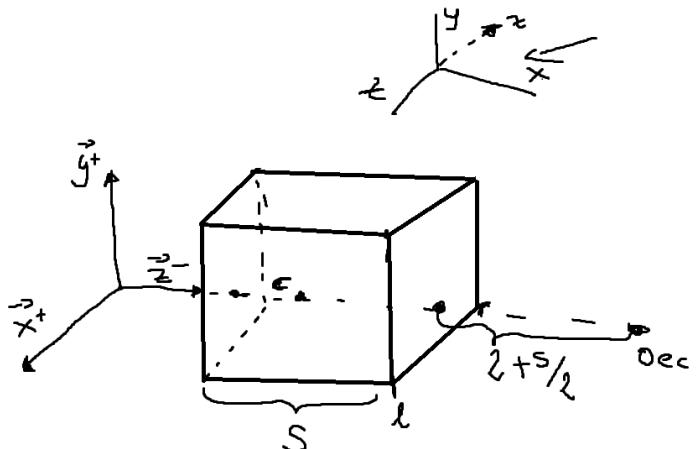
Si $w/h > 1$

$$r = r \cdot \frac{w}{h} \quad e = e \cdot \frac{w}{h}$$

Si $w/h < 1$

$$t = t \cdot \frac{h}{w} \quad b = b \cdot \frac{h}{w}$$

Así no deformaremos los objetos.

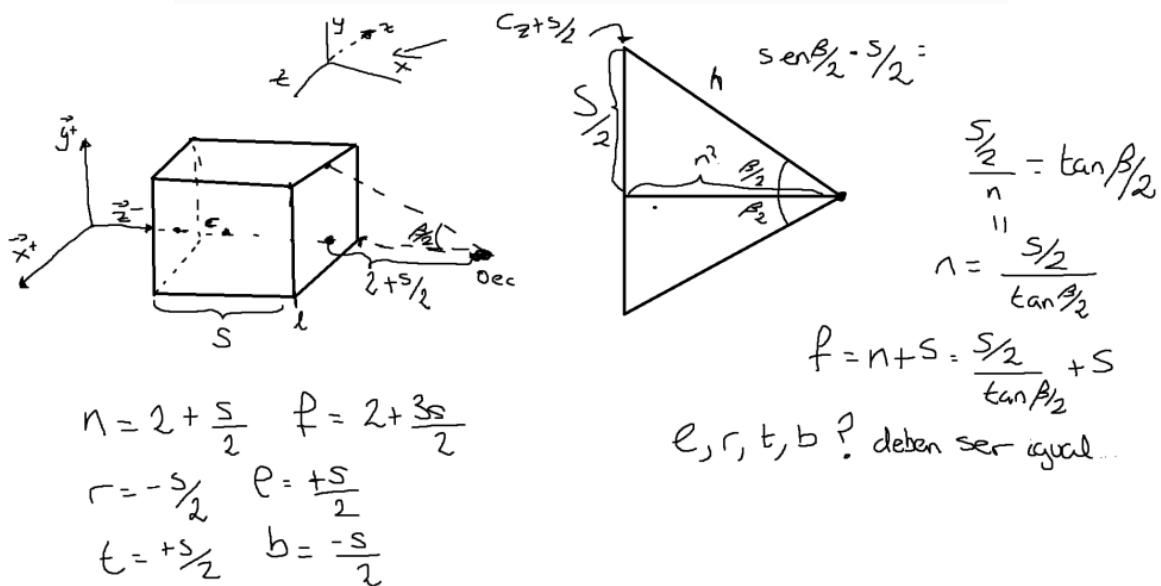


$$\begin{aligned} n &= 2 + \frac{s}{2} & f &= 2 + \frac{3s}{2} \\ r &= -\frac{s}{2} & e &= \frac{s}{2} \\ t &= +\frac{s}{2} & b &= -\frac{s}{2} \end{aligned}$$

Problema 3.8.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo β 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 β .

Indica como calcular la coordenada Z que debemos usar ahora para \mathbf{o}_{ec} (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 β, s y $\mathbf{c} = (c_x, c_y, c_z)$).



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.

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 $p = (0, 2, 0)$. El observador está situado en $\mathbf{o} = (2, 0, 0)$. En estas condiciones:

- Describe razonadamente en qué 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.

• material puramente difuso.

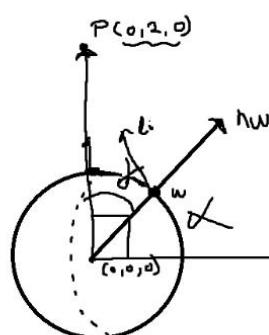
El máximo es cuando $n_p \cdot \mathbf{l}_i = \mathbf{n}_p$

$$\therefore \alpha = 0$$

Por tanto, el punto debe ser el $(0, 2, 0)$

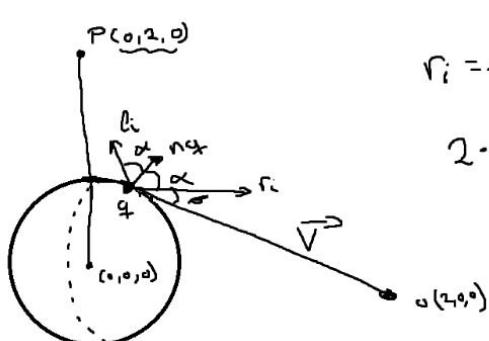
¿Es visible para el observador? No, porque la esfera no es el \mathbf{p}' es un anterior. La tg en el \mathbf{p} es paralela a \mathbf{o} .

(Para calcular de una esfera centrada en $(0, 0, 0)$ su punto de máximo brillo difuso respecto a una luminaria en $\mathbf{l}(l_x, l_y, l_z)$ se tendría que calcular $\frac{\mathbf{l} - (0, 0, 0)}{\|\mathbf{l} - (0, 0, 0)\|}$)



• Si n_p y \mathbf{l}_i coinciden, el brillo es máximo

con $\beta = 0^\circ$ respecto a \mathbf{v}



$$r_i = l_i (\mathbf{n}_p \cdot \mathbf{n}_p) \mathbf{n}_p - \mathbf{l}_i = \mathbf{v}$$

$$2 \cdot ((0, 2, 0) - \mathbf{p})(\mathbf{p} - (0, 0, 0)) - ((0, 2, 0) - \mathbf{p}) = \\ = (2, 0, 0) - \mathbf{p}$$

$$h_i = \frac{(\mathbf{p}_{\text{fuente}} - \text{centro}) + (0 - \text{centro})}{\|\text{mod del vector}\|}$$

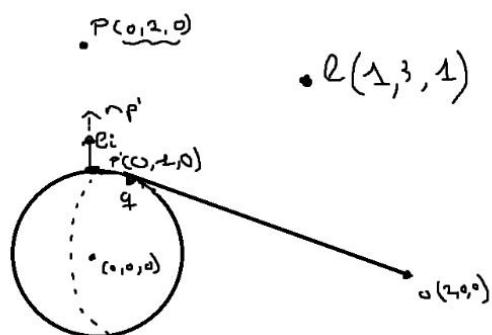
$$h_i = \frac{0, 2, 0 + 2, 0, 0}{\|(2, 2, 0)\|} = \frac{(2, 2, 0)}{\sqrt{2^2 + 2^2}} = \frac{(2, 2, 0)}{\sqrt{8}} = \frac{(2, 2, 0)}{2\sqrt{2}}$$

$$h_i = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0 \right) = \mathbf{p} \text{ (porque el centro es el } 0, 0, 0\text{)}$$

$$+ \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0 \right) = \mathbf{p}$$

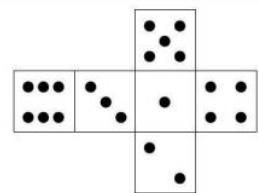
Aplicar el cálculo de h_i
suponiendo que el punto es el centro

Si \mathbf{l}_i sera visible. $\exists \alpha$: tg en la esfera de la recta desde el o con círculo de $r_{\text{p'}}$



Problema 3.10.

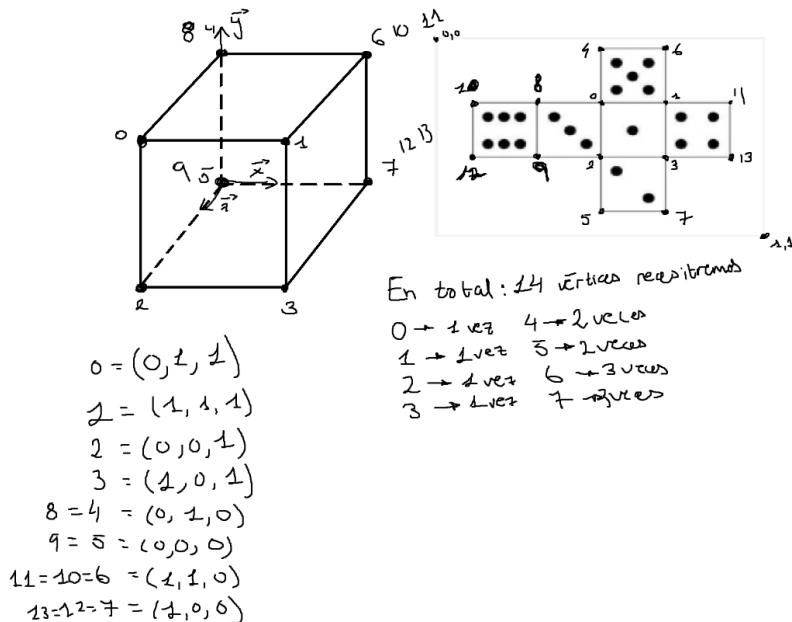
Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar un texture que incluya las caras de un dado. Para ello disponemos de una imagen de texture que tiene una relación de aspecto 4:3. La imagen aparece aquí:



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 texture, 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 texture en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de texture.



Triángulos:
 $\{(0,6,4), (0,1,6), (0,2,1), (2,3,1)$
 $(1,11,3), (3,14,13), (2,7,5),$
 $(3,7,2), (2,8,9), (2,0,8)\}$
 $(4,8,10), (4,10,12)\}$

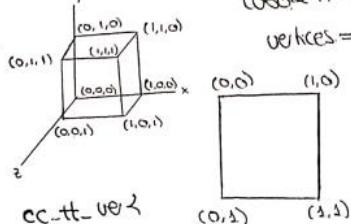
Cord - text =
 $(0.5, 0.33) = 0 \quad (0.75, 0.33) = 7$
 $(0.75, 0.33) = 1 \quad (0.25, 0.33) = 8$
 $(0.5, 0.66) = 2 \quad (0.25, 0.66) = 9$
 $(0.75, 0.66) = 3 \quad (0, 0.33) = 10$
 $(0.5, 0) = 4 \quad (1, 0.33) = 11$
 $(0.5, 1) = 5 \quad (0, 0.66) = 12$
 $(0.75, 0) = 6 \quad (1, 0.66) = 13$

Problema 3.11.

Considera de nuevo el cubo y la texture 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 texture 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 texture. Asimismo, escribe como sería la tabla de normales.

(45) (Cube 24)



cc_tt_ver2

$\{0,0,0\} // 0$
 $\{0,0,1\} // 1$
 $\{0,1,0\} // 2$
 $\{0,1,1\} // 3$
 $\{1,0,0\} // 4$
 $\{1,0,1\} // 5$
 $\{1,1,0\} // 6$
 $\{1,1,1\} // 7$

 $\{0,0,0\} // 8$
 $\{0,0,1\} // 9$
 $\{0,1,0\} // 10$
 $\{0,1,1\} // 11$
 $\{1,0,0\} // 12$
 $\{1,0,1\} // 13$
 $\{1,1,0\} // 14$
 $\{1,1,1\} // 15$

 $\{0,0,0\} // 16$
 $\{0,0,1\} // 17$
 $\{0,1,0\} // 18$
 $\{0,1,1\} // 19$
 $\{1,0,0\} // 20$
 $\{1,0,1\} // 21$
 $\{1,1,0\} // 22$
 $\{1,1,1\} // 23$

Cube24::Cube24() : Mainland ("Cube 24") {

vertices = {
 $\{0,0,0\} // 0$
 $\{0,0,1\} // 1$
 $\{0,1,0\} // 2$
 $\{0,1,1\} // 3$
 $\{1,0,0\} // 4$
 $\{1,0,1\} // 5$
 $\{1,1,0\} // 6$
 $\{1,1,1\} // 7$

 $\{0,0,0\} // 8$
 $\{0,0,1\} // 9$
 $\{0,1,0\} // 10$
 $\{0,1,1\} // 11$
 $\{1,0,0\} // 12$
 $\{1,0,1\} // 13$
 $\{1,1,0\} // 14$
 $\{1,1,1\} // 15$

 $\{0,0,0\} // 16$
 $\{0,0,1\} // 17$
 $\{0,1,0\} // 18$
 $\{0,1,1\} // 19$
 $\{1,0,0\} // 20$
 $\{1,0,1\} // 21$
 $\{1,1,0\} // 22$
 $\{1,1,1\} // 23$

};

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 Camara3Modos::Problema41(float t){  
    //Se desplaza con un movimiento uniforme, por tanto el o_ec (oe_x + a*t, oe_y + b*t, oe_z + c*t)  
    //El punto de atención también cambia de la misma forma a = (a_x + q*t, a_y + r*t, a_z + l*t)  
    //Por tanto para cada segundo t, tendremos un punto de atención, un origen y el vector VUP  
    //que nos dicen que será siempre (0,1,0). Con esta información podemos obtener fácilmente  
    //x_ec, y_ec, y z_ec con los que sabemos calcular la nueva matriz de proyección (hecho en un  
    problema anterior)  
    Tupla3f o_ec = {oe_x + a*t, oe_y + b*t, oe_z + c*t};  
    Tupla3f a = {a_x + q*t, a_y + r*t, a_z + l*t};  
    Tupla3f n = o_ec - a;  
    Tupla3f z_ec = n.normalized();  
    Tupla3f x_ec = (u.cross(n)).normalized();  
    Tupla3f y_ec = (z_ec.cross(x_ec)).normalized();  
  
    //Sabiendo x_ec, y_ec y z_ec podemos construir nuestra nueva matriz de vista para cada  
    segundo t  
    Matriz4f * V;  
  
    for(int i=0; i<2; i++){  
        V[0][i] = x_ec[i];  
        V[1][i] = y_ec[i];  
        V[2][i] = z_ec[i];  
    }  
  
    V[0][3] = -x_ec.dot(o_ec);  
    V[1][3] = -y_ec.dot(o_ec);  
    V[2][3] = -z_ec.dot(o_ec);  
    V[3][3] = 1;  
  
    cauce.fijarMatrizVista(V);  
    //Alternativamente el método a emplear sería este:  
    glUniformMatrix4fv( loc_mat_vista, 1, GL_FALSE, *V );  
}
```

$$\begin{aligned}
 o_x + d_x \cdot t &= V_{12}^x \cdot a + V_{2z}^x \cdot b \\
 o_y + d_y \cdot t &= V_{12}^y \cdot a + V_{2z}^y \cdot b \\
 o_z + d_z \cdot t &= V_{12}^z \cdot a + V_{2z}^z \cdot b
 \end{aligned}
 \quad \left. \begin{aligned}
 o_y + \frac{dy \cdot V_{12}^x \cdot a + dy \cdot V_{2z}^x \cdot b + dy \cdot o_x}{dx} &= V_{12}^y \cdot a + V_{2z}^y \cdot b \\
 o_z + \frac{dz \cdot V_{12}^x \cdot a + dz \cdot V_{2z}^x \cdot b + dz \cdot o_x}{dx} &= V_{12}^z \cdot a + V_{2z}^z \cdot b
 \end{aligned} \right\}$$

$$t = \frac{V_{12}^x \cdot a + V_{2z}^x \cdot b + o_x}{dx}$$

$$dy \cdot V_{12}^x \cdot a + dy \cdot V_{2z}^x \cdot b + dz \cdot o_x = V_{12}^y \cdot a \cdot dx + V_{2z}^y \cdot b \cdot dx + o_y \cdot dx$$

$$o_x = \frac{V_{12}^y \cdot b \cdot dx + o_y \cdot dx - dy \cdot V_{2z}^x \cdot b - dy \cdot o_x}{dy \cdot V_{12}^x - V_{12}^y \cdot dx}$$

* Despejamos b en la segunda:

$$dz \cdot V_{12}^x \cdot a + dz \cdot V_{2z}^x \cdot b + dz \cdot o_x = o_x \cdot o_z + dz \cdot V_{12}^z \cdot a + dz \cdot V_{2z}^z \cdot b$$

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 cuyas 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).

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 65 de 81

Problema 4.2. (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.
2. El punto \mathbf{p}_t , citado arriba está dentro del triángulo. Es decir, hay dos valores reales no negativos a y b (con $0 \leq a + b \leq 1$) tales que el vector $\mathbf{p}_t - \mathbf{v}_0$ es igual a $a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$.

(a los tres valores a , b y $c \equiv 1 - a - b$ se les llama *coordenadas baricéntricas* de \mathbf{p}_t en el triángulo, se usan en ray-tracing).

//Ejercicio 4.2

```

void calcularInterseccion(){
    Tupla3f o, d, v1, v2, v0;
    //o+t*d= (v1-v2)*a + (v0-v2)*b tal que 0<=a+b<=1
    /*
    Habría que resolver el siguiente sistema de ecuaciones (3 incógnitas y 3 ecuaciones, además tenemos en cuenta
    la restricción 0<=a+b<=1):
    ox+dx*t = (v1-v2)_x*a + (v0-v2)_x*b
    oy+dy*t = (v1-v2)_y*a + (v0-v2)_y*b
    oz+dz*t = (v1-v2)_z*a + (v0-v2)_z*b
    */
    if(d.dot((v1-v2).cross((v0-v2)))!=0){

        // Resuelvo el sistema y saco a,b y t
        t = blabla;
        a = blabla;
        b = blabla;

        if(0<=a+b<=1 && t>0)
            return true;
    }
}

```

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 $\mathbf{x}_{ec}, \mathbf{y}_{ec}$ y \mathbf{o}_{ec} con los versores y la tupla $\mathbf{\omega}_{ec}$ 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.

GIM: Informática Gráfica- curso 22-23- creado el 20 de diciembre de 2022 – transparencia 67 de 81

// Ejercicio 4.3

```
void calcularRayo(int x_dc, int y_dc, int w, int h, float t, float b, float n, float f, float r, float l){  
    // Asumimos que xl = yb = 0 ya que solo afectan al posicionamiento de la ventana  
    // respecto a la pantalla al completo  
    // int xl=0, yb = 0;  
    // float x_ndc = (2*(x_dc-xl))/w, y_ndc= (2*(y_dc-yb))/h, z_ndc = 0;  
  
    // float x_cc = x_ndc * (r-l), y_cc = y_ndc * (t-b);  
    // Calculamos la matriz Q con t,b,r,l,n,f  
    // Matriz4f Q = MAT_Perspectiva(l,r,b,t,n,f);  
    // Tupla3f t = {x_cc,y_cc,0};  
    // Tupla3f t_ec = MAT_Inversa(Q)*t;  
    // Para construir la matriz V debemos tener en cuenta los vectores x_ec, y_ec, z_ec y  
    o_ec que nos dan  
    // este fue un ejercicio ya realizado anteriormente.  
    // Tupla3f t_wc = MAT_Inversa(V)*t_ec;  
  
    // El punto o será el o_ec en coordenadas de mundo (que es como nos la dan), y el  
    vector d es el resultante  
    // de hacer t_wc - o_ec  
}
```

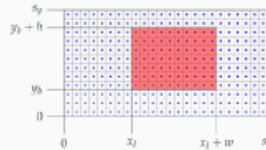
$$(x_{dc}, y_{dc}, z_{dc}, 1)^T = D(x_{ndc}, y_{ndc}, z_{ndc}, 1)^T$$

$$(x_{dc}, y_{dc}, z_{dc})^T \cdot (x_{ndc}, y_{ndc}, z_{ndc})^{-1} = D$$

D

$$D = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x_f + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y_b + \frac{h}{2} \\ 0 & 0 & \frac{z_f - z_{ndc}}{2} & \frac{z_f + z_{ndc}}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Suponemos que la ventana tiene s_x columnas y s_y filas, y que el gestor de ventanas acepta coordenadas de píxeles enteras no negativas.



$$\text{se deben cumplir estas desigualdades: } \begin{cases} 0 \leq x_l < x_f + w \leq s_x \\ 0 \leq y_b < y_b + h \leq s_y \end{cases}$$

Vamos a usarlo
 $x_l = y_b = 0$ ya que no interfiere
 $\Rightarrow z_{ndc} = \emptyset$ (supone que grande la prof)

$$x_{dc} - x_l = \frac{(x_{ndc} + 1)w}{2}$$

$$\frac{2(x_{dc} - x_l) - l}{w} = x_{ndc}$$

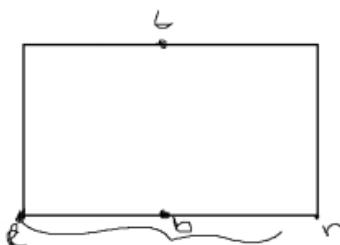
$$\frac{2(y_{dc} - y_b)}{w} = y_{ndc}$$

$$(x_{ndc}, y_{ndc}, 1) = Q \begin{pmatrix} x_{ec} \\ y_{ec} \\ z_{ec} \\ 1 \end{pmatrix}$$

$$x_{ec} = (f - b) \cdot x_{ndc}$$

$$y_{ec} = (r - l) \cdot y_{ndc}$$

Teniendo f, r, n, f, b podemos obtener Q



Teniendo $Q, (x_{ec}, y_{ec}, 0)$ y

$$\text{Sabiendo } (x_{ec}, y_{ec}, 0) = Q (x_{ndc}, y_{ndc}, z_{ndc})$$

$$Q^{-1}(x_{ec}, y_{ec}, 0) = (x_{ndc}, y_{ndc}, z_{ndc})$$

Sabiendo las eje coordinadas

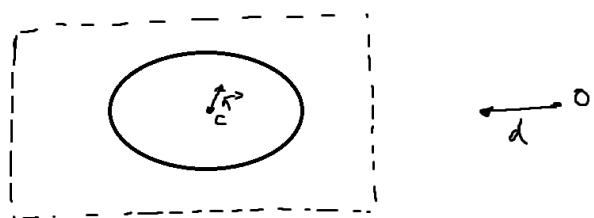
podemos obtener los $x_{ndc}, y_{ndc}, z_{ndc}$
que corresponden ya sea de tener

$x_{ec}, y_{ec}, 0$ transformarlos en matriz
de vista y (no olvidar que
 $z_{ndc} = x_{ec} \cdot y_{ec}$)

$$\text{Al hacer } V^{-1}(x_{ec}, y_{ec}, 0) = (x_{ndc}, y_{ndc}, z_{ndc})$$

Ese es el punto al que proyectamos y
a su vez el vector que va desde del a ese punto
es el \vec{x}

5.1.



Hay que calcular intersección del plano dado por c y \vec{n}
y el rayo \vec{d} y o ($o + t \cdot \vec{d}$)

Ver si ese punto l está a una distancia
menor que r de c ($\|l - c\| \leq r$)
(Siempre y cuando $t \geq 0$)

*Cómo calcular el punto del plano

Supongamos un punto l del plano, el vector
 $\vec{l} - \vec{c}$ debe ser perpendicular a \vec{n}

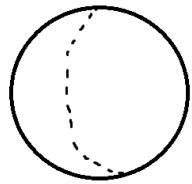
$(l - c) \cdot \vec{n} = 0$ Si ese punto también lo podemos
calcular como $l = o + t \cdot \vec{d}$

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

$$(o_x + t \cdot d_x - c_x) \cdot n_x + (o_y + t \cdot d_y - c_y) \cdot n_y + (o_z + t \cdot d_z - c_z) \cdot n_z = 0$$

Sólo t es incógnita.

5.2.



$$\| \vec{p} \| = 1$$

$$\| \vec{p} - \vec{r} \| \leq \| \vec{p} \|$$

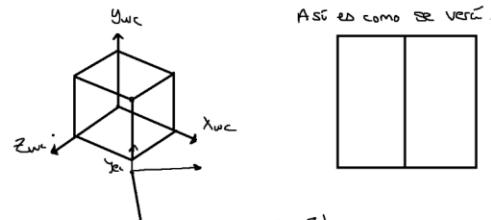
$$P = t \cdot \vec{d} + o \quad (t \cdot \vec{d} + o) \cdot (t \cdot \vec{d} + o) - 1 = 0$$

$$(td_x + o_x)(td_x + o_x) + (td_y + o_y)(td_y + o_y) + (td_z + o_z)(td_z + o_z) - 1 = 0$$

Calcular t y debe ser $t > 0$.

- B) Dadas las siguientes configuraciones, dibujar como se vería un cubo unitario (vértices en $(0,0,0)$ y en $(1,1,1)$). Se supone que hay un plano de recorte delantero (PLANO FRONTAL), que coincide con el plano de proyección. El PRP está en el eje z a una distancia d. Mostrar también los ejes. El sistema es coordenado derecho.

VRP	VPN	VUP
$(0,0,2)$	$(1,0,1)$	$(0,1,0)$
$(0,10,0)$	$(0,2,0)$	$(0,0,-1)$
$(-1,-1,-1)$	$(-1,-1,-1)$	$(0,1,0)$
$(-1,0,-1)$	$(-1,0,-1)$	$(1,1,0)$
$(2,0,0)$	$(-1,0,-1)$	$(0,-1,0)$



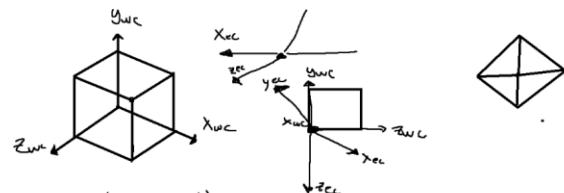
$$VUPXVPN = x_{ec} = \begin{pmatrix} x & y & z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$\Rightarrow x + 0y + 0z - 1z = 1x - 1z$$

$$VPN \times x_{ec} = \begin{pmatrix} x & y & z \\ 0 & 0 & 1 \\ 1 & 0 & -1 \end{pmatrix} =$$

$$= 0x + 1y + 0z - 0z + 1y - 0x =$$

$$2y$$



$$z_{ec} = (-1, -1, 1)$$

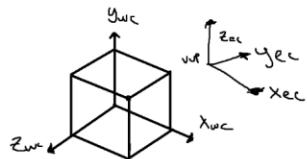
$$VPN \begin{vmatrix} x & y & z \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{vmatrix} = -x + 0y + 0z + z - 0y + 0x =$$

$$= -x + z = (-1, 0, 1)$$

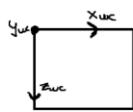
$$VPN \begin{vmatrix} x & y & z \\ -1 & -1 & 1 \\ -1 & 0 & 1 \end{vmatrix} = -x + y + 0z - z + y_1 0x$$

$$= -x + 2y - z$$

$$y_{ec} = (-1, 2, -1)$$



$$VPN \begin{vmatrix} x & y & z \\ 0 & 0 & -1 \\ 0 & 2 & 0 \end{vmatrix} = 0x + 0y + 0z - 0z - 0y + 2x$$



|| (0,0,0) | (-1,0,-1) | (0,-1,0) ||

C) Implementar en seudocódigo la rotación con respecto a un eje arbitrario, definido por dos puntos.

```
Tupla4f rotar(float grados, Tupla4f e, Tupla4f punto){
    Matriz4f *R[4][4];
    float s = sin(grados*M_PI/180), c = cos(grados*M_PI/180);
    float a[3][3];
    for(int i=0; i < 3; i++){
        for(int j=0; j < 3; j++){
            a[i][j] = (1-c)*e[i]*e[j];
        }
    }
}
```

```

    }

R[0][0] = a[0][0]+c; R[0][1] = a[0][1]-s*e[2]; R[0][2] = a[0][2]+s*e[1]; R[0][3] = 0;
R[1][0] = a[1][0]+s*e[2]; R[1][1] = a[1][1]+c; R[1][2] = a[1][2]-s*e[0]; R[1][3] = 0;
R[2][0] = a[2][0]-s*e[1]; R[2][1] = a[2][1]+s*e[0]; R[2][2] = a[2][2]-c; R[2][3] = 0;
for(int i=0; i < 3 ; i++)
    R[3][i] = 0;
R[3][3] = 1;

return R*punto;
}

```

Diferencia entre geometria y topologia

Geometria se basa en la posicion de los vertices.

Topologia se basa en la relación entre aristas y triangulos.

Geometria almacena vertices

Topologia almacena triangulos y aristas.

Geometría y topología de las mallas

Una malla tiene una geometría y una topología:

- ▶ **Geometría:** conjunto de puntos que están en alguna cara (eso incluye los puntos que están en alguna arista y las posiciones de los vértices).
- ▶ **Topología:** conjunto de relaciones de adyacencia entre vértices, aristas y caras (sin tener en cuenta la geometría, es decir, considerando únicamente los índices de los vértices).

Esta definición permite que dos mallas distintas

- ▶ tengan la misma topología pero distinta geometría (p.ej., partimos de una malla y cambiamos las posiciones de sus vértices, pero mantenemos las adyacencias).
- ▶ tengan la misma geometría pero distinta topología (p.ej., partimos de una malla con caras de cuatro aristas y dividimos cada cara en dos caras triángulares coplanares).

¿Por qué no varía el objeto al ensanchar la pantalla?

Cambiamos la matriz de proyección y deformando la imagen, si no la cambiaríamos se ensancharían las dos cosas y se distorsionaría todo

- K) Al cambiar la ventana de una aplicación su tamaño pasa a ser AnchoxAlto. Dada la distancia d1 al plano delantero y d2 al plano trasero, indicar como quedaría la llamada a glFrustum ()

La función **MAT_Perspectiva** permite generar una matriz de proyección perspectiva de forma más fácil en ocasiones:

```
MAT_Perspectiva( GLdouble β, GLdouble a, // calcula Q a partir de β,a,n,f
GLdouble n, GLdouble f ) ;
```

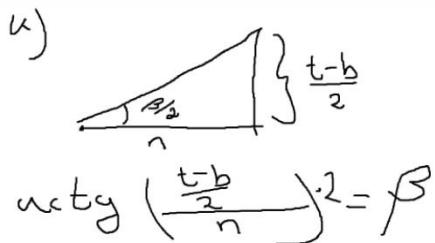
esta función equivale a **MAT_Frustum** centrado con:

$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

donde:

β ≡ es la apertura vertical del campo de visión (fovy), es el ángulo en grados (entre 0 y 180.0) que hay entre la cara superior y la inferior del frustum

a ≡ relación de aspecto (aspect ratio) de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).



- L) ¿Qué diferencia hay entre modo inmediato y modo retenido en OpenGL.

Hay varios formas de visualizar secuencias de vértices y sus atributos:

- ▶ Envío en **modo inmediato**: cada vez que queremos visualizar, se envían los atributos e índices a la GPU por el bus del sistema.
De dos formas:
 - ▶ Usando una llamada a una función por cada vértice o atributo.
 - ▶ Usando una única llamada para enviar tablas completas
 Este modo de visualización es muy ineficiente en tiempo (requiere transferir muchos datos por cada cuadro o frame), y no se usa en OpenGL moderno.
- ▶ Envío en **modo diferido**: los datos de la secuencia de vértices se envían a la GPU una sola vez. Es el modo que usaremos.

M) Se desea obtener una relación 1:1 en un espacio 2D sin deformación en OpenGL. ¿Cuales deberían ser los valores de glOrtho?

La matriz de proyección **ortográfica** O se obtiene por tanto como composición de T seguido de S , es decir $O = S \cdot T$, o lo que es lo mismo:

$$O = \begin{pmatrix} a'_0 & 0 & 0 & a'_1 \\ 0 & b'_0 & 0 & b'_1 \\ 0 & 0 & c'_0 & c'_1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ donde: } \begin{cases} a'_0 \equiv \frac{2}{r-l} & a'_1 \equiv -\frac{r+l}{r-l} \\ b'_0 \equiv \frac{2}{t-b} & b'_1 \equiv -\frac{t+b}{t-b} \\ c'_0 \equiv \frac{-2}{f-n} & c'_1 \equiv -\frac{f+n}{f-n} \end{cases}$$

de forma que ahora hacemos:

$$(x_{cc}, y_{cc}, z_{cc}, w_{cc})^t = O(x_{ec}, y_{ec}, z_{ec}, w_{ec})^t$$

donde w_{cc} sí vale 1 con seguridad.

L y R, t y b deben estar relacionados de manera de que uno sea el negativo del otro.

O) Hemos importado un modelo PLY y queremos comprobar que es cerrado. Escribir el seudocódigo (indicar las estructuras de datos que se usan)

Tienes que comprobar que no existe ninguna arista que solo está anexa a un triángulo.

- R) Calcular la matriz de transformación resultante de aplicar las siguientes instrucciones OpenGL:

```
glLoadIdentity();
glRotatef(30,0,1,0);
glScalef(1.0,0.5,2.0);
glTranslatef(30,10,20);
```

$$\text{Esc} [s_x, s_y, s_z] = \quad \text{Tra} [d_x, d_y, d_z] = \quad \text{Ciz}_{xy}[a] =$$

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rot}_x[\alpha] = \quad \text{Rot}_y[\alpha] = \quad \text{Rot}_z[\alpha] =$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Primero multiplicas por la de rotacion, luego la de escalado a la derecha y luego la de translación a la derecha.

- S) Indicar que hace cada una de estas primitivas de OpenGL. Poner un ejemplo dibujado, numerando los vértices. (GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_POLYGON, GL_QUADS, GL_QUAD_STRIP, GL_TRIANGLES, GL_TRIANGLE, GL_TRIANGLE_FAN)

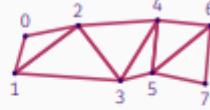
Puntos GL_POINTS	Segmentos GL_LINES	Polilínea abierta GL_LINE_STRIP	Polilínea cerrada GL_LINE_LOOP

Tira de triángulos

GL_TRIANGLE_STRIP

Polygonos:

(0,1,2), (2,1,3), (2,3,4),
(4,3,5), (4,5,6), (6,5,7), ...

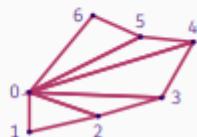


Abanico de triángulos

GL_TRIANGLE_FAN

Polygonos:

(0,1,2), (0,2,3), (0,3,4),
(0,4,5), (0,5,6), ...

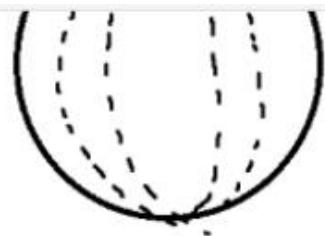


Polígonos de más de tres vértices

Respecto a la posibilidad de visualizar primitivas de más de tres vértices:

- ▶ En versiones de OpenGL anteriores a la 3.0 existían las primitivas tipo cuadrilátero y polígono
- ▶ Las constantes eran: **GL_POLYGON**, **GL_QUADS** y **GL_QUAD_STRIP**.
- ▶ En la versión 3.0 de OpenGL (2008), se declararon obsoletas este tipo de primitivas, y en posteriores versiones se eliminaron
- ▶ En OpenGL moderno, para visualizar estas primitivas en modo relleno hay que descomponerlas en triángulos.
- ▶ En cualquier caso, en versiones antiguas de OpenGL lo que se hacía internamente es descomponerlas en triángulos.

W) Se han generado los puntos que representan a una esfera (m puntos para la curva generatriz, n divisiones). Se han guardado en un vector, por meridianos. Implementar un procedimiento que almacene las caras.



$V = [v_0, v_1, \dots, v_m, v_{m+1}, \dots, v_{n+m}]$

$\underbrace{v_0, v_1, \dots, v_m}_{\text{1er merid}}$ en totale ring
 n meridianen

Erianguler $(V_{k+i} V_{k+i}, V_{(k+1) \cdot j + i}) \}$

Método Inicializar MallaRevol:

```

void MallaRevol::inicializar
(
    const std::vector<Tupla3f> & perfil, // tabla de vértices del perfil original
    const unsigned num_copias // número de copias del perfil
)
{
    // COMPLETAR: Práctica 2: completar: creación de la malla.....

    float x,y;
    vertices.clear();
    for(int i=0; i <= num_copias-1; i++){
        for(int j=0; j <= perfil.size()-1; j++){
            Tupla3f p = perfil.at(j);
            Tupla3f q;

            q =
MAT_Rotacion(360*i/(num_copias-1),
{0.0,1.0, 0.0})*p;
            vertices.push_back(q);
        }
    }

    // Añadimos los triangulos
    triangulos.clear();
    float k;
    for(int i = 0; i <= num_copias - 2; i++){
        for(int j=0; j <= perfil.size() - 2; j++){
            k = i*perfil.size() + j;
            triangulos.push_back({k,
k+perfil.size(), k+perfil.size()+1});
            triangulos.push_back({k,
k+perfil.size()+1, k+1});

        }
    }
}

```

```

// Practica 4
//-----// COMPLETAR: Práctica 4: cálculo normales y coordenadas de textura
///////////////////////////////
// Calculamos las normales de las aristas
std::vector<Tupla3f> normales_m;
for (unsigned int i = 0; i < perfil.size()-1; i++){
    float v_1 = (perfil[i+1] - perfil[i])(0);
    float v_2 = (perfil[i+1] - perfil[i])(1);
    Tupla3f m_i({v_2, -v_1, 0.0}); // giro de un vector -90°
    if (m_i.lengthSq() != 0) // por si hubiera dos puntos seguidos iguales
        m_i = m_i.normalized();
    normales_m.push_back(m_i);
}

// Calculamos las normales de los vertices
std::vector<Tupla3f> normales_n;
normales_n.push_back(normales_m[0]);
for (unsigned int i = 1; i < perfil.size()-1; i++){
    normales_n.push_back( (normales_m[i-1] + normales_m[i]).normalized());
    normales_n.push_back(
normales_m[perfil.size()-2] );

    // Calculamos los vectores d y t, junto con sumas_parciales (vector auxiliar)
    std::vector<float> d;
    std::vector<float> t;
    std::vector<float> sumas_parciales;
    float suma_total;

    for (unsigned int i = 0; i < perfil.size()-1; i++)
        d.push_back( sqrt( (perfil[i+1] - perfil[i]).lengthSq() ) );

```

```

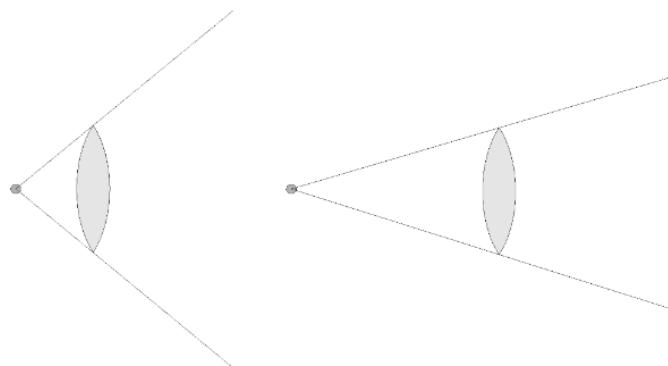
sumas_parciales.push_back(0.0);
for (unsigned int i = 1; i < perfil.size(); i++)

sumas_parciales.push_back(sumas_parciales[i-1] + d[i-1]);

suma_total =
sumas_parciales[perfil.size()-1];
t.push_back(0.0);
for (unsigned int i = 1; i < perfil.size(); i++)
    t.push_back( sumas_parciales[i] /
suma_total );
///////////////////////////////
for(int i=0; i <= num_copias-1; i++){
    for(int j=0; j <= perfil.size()-1; j++){
        nor_ver.push_back(
MAT_Rotacion(2.0*180.0*i / (num_copias-1), {0.0, 1.0, 0.0}) * normales_n[j] );
        cc_tt_ver.push_back({float(i) /
(num_copias-1), 1-t[j]}); // i y j están
cambiados
    }
}
}

```

3. Las lentes de las cámaras con zoom permiten cambiar la zona visible, desde ángulos más grandes (gran angular) a más pequeños (tele). ¿Cómo se podría conseguir el mismo efecto con los parámetros del glFrustum? Explicarlo y poner ejemplos de valores. (2)



Tenemos los parámetros l,r,b,t,n,f:

La función **MAT_Perspectiva** permite generar una matriz de proyección perspectiva de forma más fácil en ocasiones:

```
MAT_Perspectiva( GLdouble β, GLdouble a, // calcula Q a partir de β,a,n,f  
                  GLdouble n, GLdouble f ) ;
```

esta función equivale a **MAT_Frustum** centrado con:

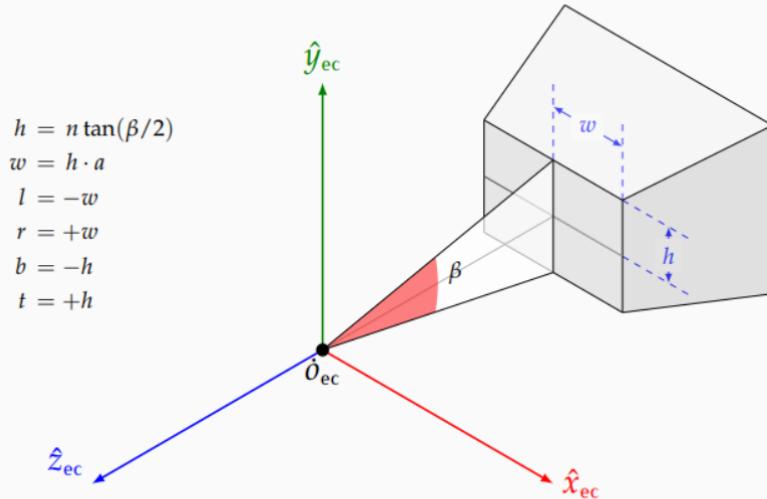
$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

donde:

$\beta \equiv$ es la **apertura vertical del campo de visión (fovy)**, es el ángulo en grados (entre 0 y 180.0) que hay entre la cara superior y la inferior del frustum

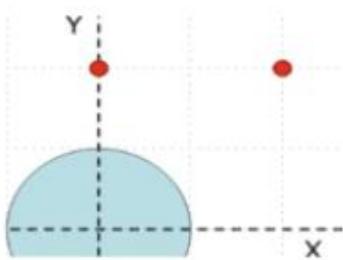
$a \equiv$ **relación de aspecto (aspect ratio)** de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).

El significado de los parámetros se aprecia en esta figura:



Esta perspectiva es *centrada*, ya que $r = -l$ y $t = -b$

- b) Supongamos que tenemos una esfera difusa de material blanco de radio unidad centrada en el origen, una fuente de luz puntual en $(0, 2, 0)$ de color azul $(0, 0, 1)$ y otra fuente de luz puntual en $(2, 2, 0)$ de color rojo $(1, 0, 0)$. ¿Dónde tendremos el máximo valor de iluminación difusa en la esfera para las dos luces? ¿En qué zona de la esfera se producirá una mezcla de color tal que las intensidades difusas producidas por las dos luces sean semejantes. (2)



Ej 3. 2019

$$\frac{n_p \cdot l_b}{P} = \frac{n_{\bar{p}} \cdot l_r}{P}$$

$0,2,0-p$ $2,2,0-p$

$$P \cdot (-P_x, 2-P_y, -P_z) = P \cdot (2-P_x, 2-P_y, -P_z)$$

$$-P_x^2 + 2P_y - P_y^2 - P_z^2 = 2P_x - P_x^2 + 3P_y - 2P_y^2 - P_z^2$$

$$-2p_x + p_y^2 = 0$$

$$p_y^2 = 2p_x \Rightarrow p_y = \sqrt{2p_x}$$

$$P_x^2 + P_y^2 = 1$$

$$P_x^2 + 2P_x - 1 = 0$$

$$P_y = \sqrt{2(-1 + \sqrt{2})} = 0.9101$$

Colapsar Aristas para minimización de caras

```

void colapsarArista(){
    //Tenemos el punto inicial pi y final pf de la arista y la lista de triangulos
    v = pf-pi;
    nuevo_punto = pi + v/2;
    for(todos los triangulos)
        IF triangulo contiene o el punto final o el inicial, lo quitamos y lo formamos con el nuevo
        punto
    }

void calcularVerticesMasRepetidosConAristaCompartida(){
    //1. Primero debemos calcular las aristas que no forman parte del contorno para que no se
    pierda
    // la forma de la malla

    for(cada_arista)
        comprobar que la arista aparezca en 2 o mas triangulos

        IF no aparece en otro triangulo
            quito el triangulo de la lista de candidatos a colapsar

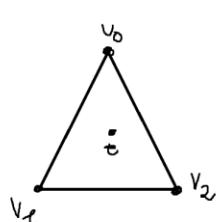
    for(triangulos_candidatos)
        colapsar_algunas de las aristas
}

```

//-----

una ocupado por sus vecinos.

3. Describir como se pueden seleccionar los vértices de una malla en un programa que visualiza mallas de triángulos.

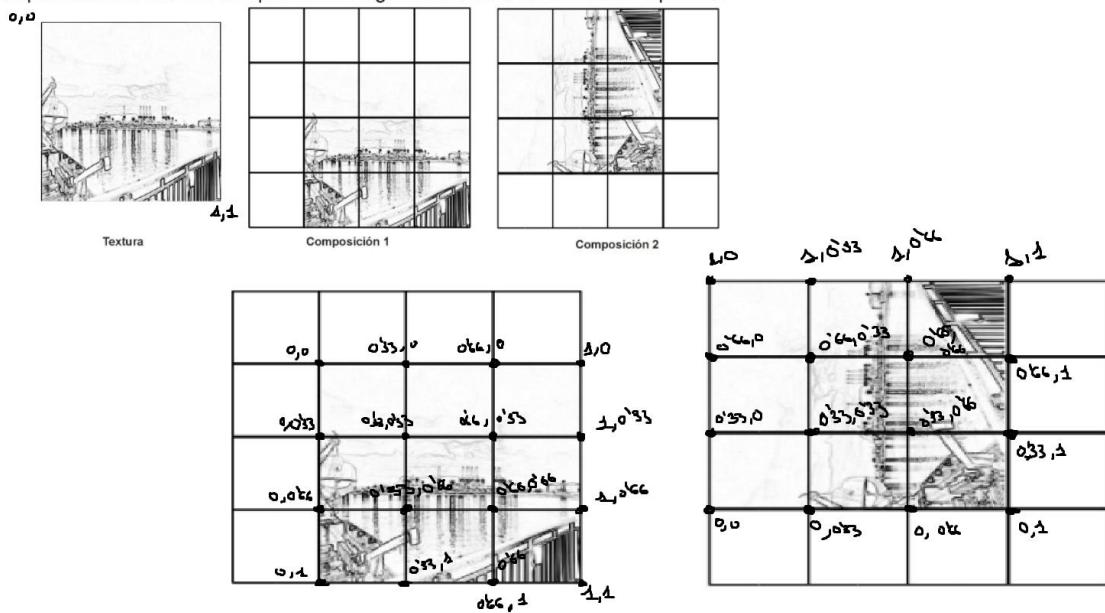


En el 4.3 sabemos reflejar el rayo.
En el problema 4.2 sabemos
calcular la intersección entre un
rayo y un triángulo.

Una vez tenemos el pto t:

1. Calcular $\|\vec{t} - \vec{v}_0\|$
2. Calcular $\|\vec{t} - \vec{v}_1\|$
3. Calcular $\|\vec{t} - \vec{v}_2\|$
4. El vértice asociado al módulo menor
es el que entendemos como clic

6. Se ha creado una retícula de 4x4 cuadrados. Indicar las coordenadas de textura de las esquinas de cada uno de ellos para obtener las dos composiciones siguientes con la textura de la izquierda.



5. Indicar cual será la posición de la luz en los casos siguientes:

- | | | | |
|----|---|----|---|
| a) | <code>GLfloat Position[4]={1,1,1,0};</code> | b) | <code>GLfloat Position[4]={1,1,1,0};</code> |
| - | <code>glMatrixMode(GL_MODELVIEW);</code> | - | <code>glMatrixMode(GL_MODELVIEW);</code> |
| - | <code>glPushMatrix();</code> | - | <code>glPushMatrix();</code> |
| - | <code>glLoadIdentity();</code> | - | <code>glLoadIdentity();</code> |
| - | <code>glLightfv(GL_LIGHT0, GL_POSITION, Position);</code> | - | <code>glRotate(45,0,1,0);</code> |
| - | <code>glPopMatrix();</code> | - | <code>glLightfv(GL_LIGHT0, GL_POSITION, Position);</code> |
| - | <code>glPopMatrix();</code> | - | <code>glPopMatrix();</code> |

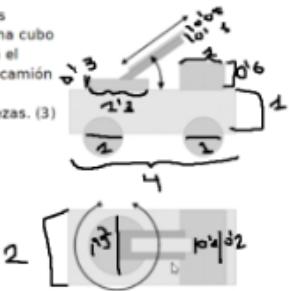
- a) La luz va a estar en la posición indicada $(1,1,1,0)$
 b) Hay que rotar la posición 45° en el eje y en $(\sqrt{2}, 1, 0, 0)$

$$\text{Nuevo_pos} = \text{MAT_Rot}(45, (0, 1, 0)) * \text{pos}$$

$$\begin{pmatrix} \cos(45) & 0 & \sin(45) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(45) & 0 & \cos(45) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} =$$

$$\begin{aligned} x' &= \cos 45 \cdot 1 + \sin 45 \cdot 1 = \frac{\sqrt{2}}{2} \\ y' &= 1 \\ z' &= -\sin 45 \cdot 1 + \cos 45 \cdot 1 = 0 \end{aligned}$$

1. Generar el grafo de escena incluyendo las transformaciones, tal que partiendo de una cubo unidad y un cilindro unidad centrados en el origen permite obtener un modelo de un camión con escalera. Hacer un dibujo del posicionamiento y dimensiones de las piezas. (3)



Escalera(α)

Trans($1, 0, 0$)
Esc($1/2, 0.05, 0.1$)

Cubo

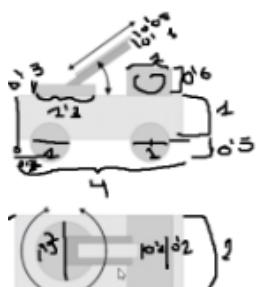
Base Escalera

Trans($1.5, 1, 0$)
Esc($1.5, 0.1, 0.2$)
Cubo

Cilindro Esc
Esc($-1/3, 0.3, 1/3$)
Cilindro

Esc Comp(γ, α, Tr)
Rot($\gamma, 0, 1, 0$)
Cilindro Esc
Trans($0, 0.3, 0$)
Base Esc(α, Tr)

B3 Esc Anim(α, Tr)
Rot($\alpha, 0, 1$)
Base Esc
Trans($0, 0.1, 0$)
Escalera(Tr)



Cabin
Trans($3, 1.5, 0$)
Esc($1, 0.6, 2$)
Cubo

Chasis
Trans($2, 0.15, 0$)
Esc($1, 4, 2$)
Cubo

Rueda
Trans($0.5, 0, 0$)
Rot($90, 4, 0, 0$)
Esc($0.2, 0$)
Cilindro

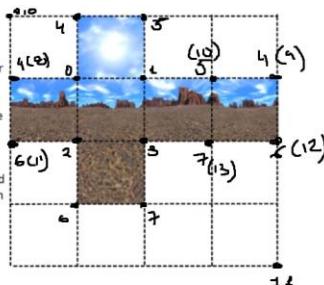
Ruedas Delanteras
Trans($3.3, 0, 0$)
Rueda

Base(γ, α, Tr)
Rueda
Ruedas Delanteras
Chasis
Cabin
Trans($0, 1.5, 0$)
Esc Comp(γ, α, Tr)

Distintas formas para modelar un sólido definido por fronteras como revolución, etc.

Revolution, barrido el del eje Y o X o Z, aristas aladas.

2. Sean seis imágenes que se han agrupado como una sola y se quieren usar para un skybox (caja que engloba la escena). Considerar un cubo creado como lista de vértices y caras (8 vértices y 12 caras inicialmente). Indica como se han de asignar las coordenadas de textura a las caras de dicho cubo (nota: hay que ser coherente con las imágenes del skybox y recordad que en OpenGL las imágenes están normalizadas con el origen en la posición superior izquierda). (2)



Caras : $(0,1,3), (0,2,3) \quad (9,12,10)$
 $(4,0,4), (4,5,1) \quad (10,13,12)$
 $(2,3,6), (6,7,3)$
 $(8,11,0), (8,2,0)$
 $(-1,10,3), (10,3,13)$

0	una vez	4	3 veces
1	una vez	5	2 veces
2	una vez	6	3 veces
3	una vez	7	2 veces

$p_4 \rightarrow 8, 9$
 $p_5 \rightarrow 10$
 $p_6 \rightarrow 11, 12$
 $p_7 \rightarrow 13$

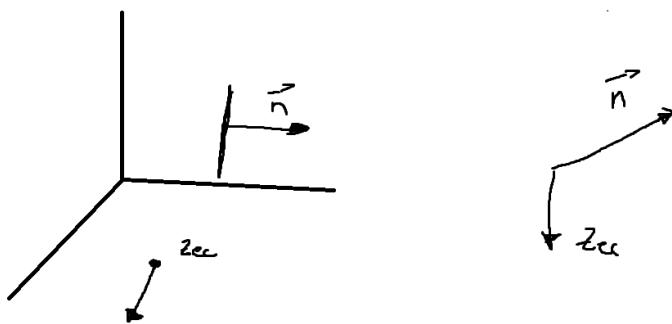
Texturas :

$(0^{\circ}25, 0^{\circ}25)$	$/ / \circ$	$(6^{\circ}75, 0^{\circ})$	$(0^{\circ}75, 0^{\circ}25)$
$(0^{\circ}5, 0^{\circ}25)$		$(5^{\circ}25, 0^{\circ}25)$	$(0^{\circ}10, 0^{\circ}25)$
$(5^{\circ}25, 0^{\circ}25)$		$(5^{\circ}25, 0^{\circ}75)$	$(0^{\circ}10, 0^{\circ})$
$(0^{\circ}5, 0^{\circ}5)$		$(0^{\circ}25, 0^{\circ}5)$	$(4, 0^{\circ}25)$
$(0^{\circ}25, 0^{\circ})$		$(0, 0^{\circ}25)$	$(5^{\circ}75, 0^{\circ}25)$

6. Explique los pasos que se siguen en OpenGL para utilizar una imagen como textura (1)

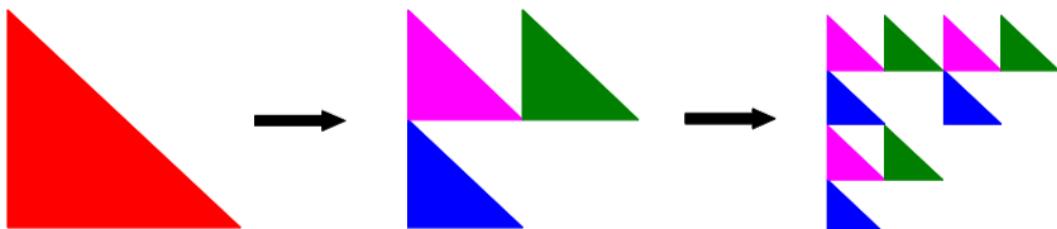
La cargas la manda a la gpu,(matriz de texels) la normaliza, y luego la interpola y asignas a cada uno de los vértices.

2. Supongamos un objeto descrito con listas de vértices y caras (`vector<vertex3f> Vertices; vector<vertex3i> Triangulos`) y un punto donde se sitúa el observador. Implementar en pseudocódigo el algoritmo de eliminación de las caras traseras (back face culling) que permite seleccionar las caras visibles a un observador. ¿Qué listas ha de devolver el algoritmo? (2)



1. Sacamos con la lista de vértices y de triángulos una lista de normales asociados a esos triángulos
2. Si un vector de esa lista y Zer forman un ángulo mayor o igual que 90° y menor o igual que 270° lo borramos
3. Desolvemos la lista con los triángulos que no cumplen la condición anterior.

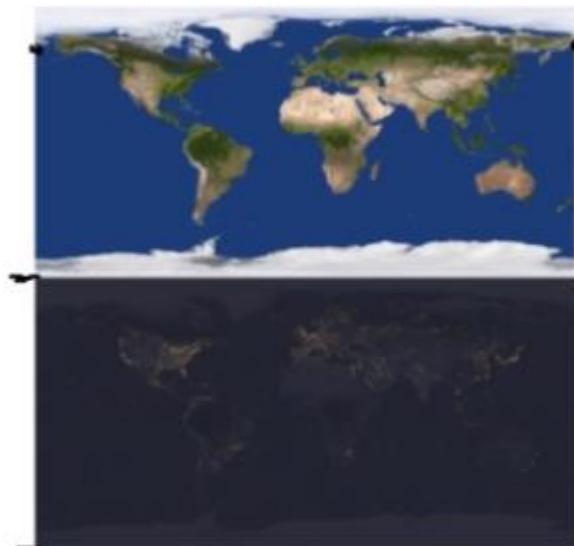
1) Programar usando C++ y OpenGL un procedimiento que de forma recursiva permita generar el siguiente patrón. Se indicará el número de niveles de la recursión mediante un parámetro. (2.5 pt)



```
void calculaTriangulo(unsigned n, unsigned exp){
    if(n==1)
        dibujaTriangulo();
    else{
        MAT_Escalado(0.5,0.5);
        calculaTriangulo(n-1, exp+1);
        MAT_Translacion(0,0.5*exp);
        calculaTriangulo(n-1,exp+1);
        MAT_Translacion(0.5*exp,0);
        calculaTriangulo(n-1,exp+1);
    }
}
```

}

3) Dada la siguiente imagen que representa simultáneamente el planeta tierra de día y de noche, escriba mediante código C++ la asignación de texturas para una hora h del día a la malla poligonal que representa una esfera `Esfera::Esfera(int nmeridianos, int nparalelos)` creada con la siguiente llamada `Esfera=new Esfera(24,24)`, suponiendo que el eje de rotación de la Tierra permanece en todo momento perpendicular al plano orbital con respecto al sol. ((2.5 pt)



```
void calculaTexturaPlaneta(){
    for(int i=0; i < 24; i++){
        for(int j=0; j<24; j++){
            if(i<12){
                coord_tex.push_back({i*1/23,j*(0.5/23)});
            }
            else{
                coord_tex.push_back({i*1/23,0.5+j*(0.5/23)});
            }
        }
    }
}
```

Ejercicio 1. Considera una malla de n triángulos almacenada en memoria con un vector caras (con n entradas), de forma que $caras[i][j]$ es un entero, en concreto el índice del vértice número j de la cara número i (con $0 \leq i \leq n$ y $0 \leq j < 3$).

1. Con esta definición, escribe el código de una función con esta declaración:

```
bool comparten_vertice(int c1, int c2);
```

que devuelve true cuando las caras número $c1$ y $c2$ comparten un vértice (devuelve false si esto no es así).

```
bool comparten_vertice(int c1, int c2){  
    const int NUMERO_VERTICES = 3;  
    bool comparten_vertice = false;  
  
    for (int j = 0; j < NUMERO_VERTICES && !comparten_vertice; ++j){  
        for (int k = 0; k < NUMERO_VERTICES && !comparten_vertice; ++k){  
            if (caras[c1][j] == caras[c2][k])  
                comparten_vertice = true;  
        }  
    }  
  
    return comparten_vertice;  
}
```

2. Escribe el código de otra función:

```
bool comparten_aristas(int c1, int c2);
```

que devuelve true cuando las caras número $c1$ y $c2$ comparten una arista (devuelve false si esto no es así).

```
bool comparten_arista(int c1, int c2){  
    const int NUMERO_VERTICES = 3;  
    bool comparten_arista = false;  
    bool comparten_vertice = comparten_vertice(c1, c2);  
    int vertice_a;  
    int vertice_b;  
  
    vertice_a = -1;  
    vertice_b = -1;  
  
    if (comparten_vertice){  
        for (int j = 0; j < NUMERO_VERTICES && !comparten_arista; ++j){  
            for (int k = 0; k < NUMERO_VERTICES && !comparten_arista; ++k){  
                if ( caras[c1][j] == caras[c2][k] && vertice_a == vertice_b )  
                    vertice_a = caras[c1][j];  
                else if ( caras[c1][j] == caras[c2][k] && vertice_a != vertice_b )  
                    comparten_arista = true;  
            }  
        }  
    }  
  
    return comparten_arista;  
}
```

2. Considera una esfera representada como una malla de n triángulos, que está almacenada en memoria con un vector cara, un vector vertice y un vector normal , de forma que cara[i][j] es un entero, en concreto el indice del vértice número j de la cara número i en el vector vertice, normal[i] es el vector normal al vértice i, y vertice[i] contiene el vértice i. Los vectores y los puntos se almacenan como una estructura con tres campos (x,y,z). Describe usando pseudocódigo una función para transformar la esfera en un elipsoide. Indica como se podría realizar una animación del proceso.

```

void esferaEclipsoide(){
    //El vector cara se mantiene igual. Los que van a cambiar son vertices y el normal
    //Para hacer el elipsoide vamos a escalarlo en el eje x

    for Vertice v en Vertices:
        v = MAT_Escalado(2,1,1)*v;

    vector listaNormalesTri;
    vector listaNormalesVer;
    for triangulo in Triangulos:
        listaNormalesTri.push_back(calculaNormal(triangulo));
        for vertices in triangulos:
            listaNormalesVer[ver]+=listaNormalesTri(triangulo);

    for normal in listaNormalesVer:
        normal = normal.normalizada;

}

```

```

void Mallalnd::calcularNormales()
{
    // COMPLETAR: en la práctica 4: calculo de las normales de la malla
    // se debe invocar en primer lugar 'calcularNormalesTriangulos'

    /////////////////////////////////
    calcularNormalesTriangulos();

    nor_ver = std::vector<Tupla3f>(vertices.size(), Tupla3f(0.0, 0.0, 0.0));
    for (unsigned int i = 0; i < triangulos.size(); i++){
        for (unsigned int j = 0; j < 3; j++){
            unsigned int indice_vertice = triangulos[i](j);

            nor_ver[indice_vertice] = nor_ver[indice_vertice] + nor_tri[i];
        }
    }

    for (unsigned int i = 0; i < nor_ver.size(); i++)
        if (nor_ver[i].lengthSq() != 0.0)
            nor_ver[i] = nor_ver[i].normalized();
    ///////////////////////////////
}

```

```

void Mallalnd::calcularNormalesTriangulos()
{
    // si ya está creada la tabla de normales de triángulos, no es necesario volver a crearla
    const unsigned nt = triangulos.size() ;
    assert( 1 <= nt );
    if ( 0 < nor_tri.size() )
    {
        assert( nt == nor_tri.size() );
        return ;
    }

    // COMPLETAR: Práctica 4: creación de la tabla de normales de triángulos
    // ....
    Tupla3f vector_a, vector_b;
    Tupla3f vector_mc;

    Tupla3f vector_normal;

    Tupla3f vector_nulo = {0.0,0.0,0.0};

    for(int n=0; n<triangulos.size(); n++){

        vector_a = vertices.at(triangulos.at(n)[1]) - vertices.at(triangulos.at(n)[0]);
        vector_b = vertices.at(triangulos.at(n)[2]) - vertices.at(triangulos.at(n)[0]);

        vector_mc = vector_a.cross(vector_b);

        if(vector_mc.lengthSq() != 0.0){

            vector_normal = vector_mc.normalized();

            nor_tri.push_back(vector_normal);

        }else{

            nor_tri.push_back(vector_nulo);
        }
    }
}

```