

Problema 1.1.

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

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

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

(el enunciado continua en la siguiente transparencia)

Problema 1.1. (continuación)

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

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

- tipo de primitiva **GL_LINE_LOOP**.
- tipo de primitiva **GL_LINES**.

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

a) GL - LINE - LOOP

```
void DibujarFigura2(unsigned int n){  
    assert(n>2);  
    assert( glgetError() == GL_NO_ERROR );  
    std::vector<glm::dvec2> vertices;  
    for(int i=0; i<n; i++){  
        vertices.push_back(glm::dvec2(cos(2*M_PI*i/n),sin(2*M_PI*i/n)));  
    }  
    if(vao_glm == nullptr){  
        vao_glm = new DescrVAO(cause->num_atribs, new DescrVBOAtribs(cause->ind_atrib_posiciones, GL_DOUBLE, 2, vertices.size(), vertices.data()));  
        assert( glgetError() == GL_NO_ERROR );  
    }  
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );  
    vao_glm->draw(GL_LINE_LOOP);  
    assert( glgetError() == GL_NO_ERROR );  
}
```

b) GL - LINES

```
void DibujarFigura2(unsigned int n){  
    assert(n>2);  
    assert( glgetError() == GL_NO_ERROR );  
    std::vector<glm::dvec2> vertices;  
    for(int i=0; i<n; i++){  
        vertices.push_back(glm::dvec2(cos(2*M_PI*i/n),sin(2*M_PI*i/n)));  
        vertices.push_back(glm::dvec2(cos(2*M_PI*(i+1)/n),sin(2*M_PI*(i+1)/n)));  
    }  
    if(vao_glm == nullptr){  
        vao_glm = new DescrVAO(cause->num_atribs, new DescrVBOAtribs(cause->ind_atrib_posiciones, GL_DOUBLE, 2, vertices.size(), vertices.data()));  
        assert( glgetError() == GL_NO_ERROR );  
    }  
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );  
    vao_glm->draw(GL_LINES);  
    assert( glgetError() == GL_NO_ERROR );  
}
```

Esto se debe a que **GL - LINE - LOOP** une toda la secuencia de vértices, mientras que **GL - LINE** une pares de vértices, luego necesitamos i y $i+1$ siguiente siempre (La secuencia no está indexada)



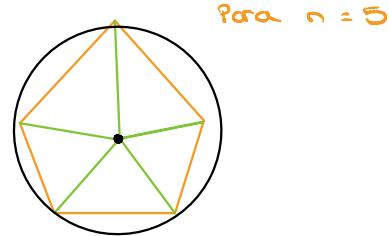
No tienen un número / índice que diga donde está almacenado el elemento. Por eso, almacenamos las tablas de esta forma en el ejercicio

Problema 1.2.

Escribe otra función para generar una tabla de vértices y una tabla de índices (y el correspondiente descriptor de VAO en una variable global), que permitiría visualizar el mismo polígono regular del problema anterior pero ahora como un conjunto de n triángulos iguales 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.



Esta primitiva forma triángulos cada 3 vértices (I = {a,b,c}, d,e,f ...)

a) Secuencia NO indexada

```
void DibujarFigura1(unsigned int n){  
    assert(n>2);  
    assert( glGetError() == GL_NO_ERROR );  
    std::vector<glm::vec3> vertices;  
    for(int i = 0; i < n; i++){  
        vertices.push_back(glm::vec3(0.0,0.0,0.0));  
        vertices.push_back(glm::vec3(cos(2*M_PI*i/n),sin(2*M_PI*i/n),0.0));  
        vertices.push_back(glm::vec3(cos(2*M_PI*(i+1)/n),sin(2*M_PI*(i+1)/n),0.0));  
    }  
    if(vao_glm == nullptr){  
        vao_glm = new DescrVAO(cauce->num_atribos, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));  
        assert( glGetError() == GL_NO_ERROR );  
    }  
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );  
    vao_glm->draw( GL_TRIANGLES );  
    assert( glGetError() == GL_NO_ERROR );  
}
```

b) Secuencia indexada

```
void DibujarFigura1(unsigned int n){  
    assert(n>2);  
    assert( glGetError() == GL_NO_ERROR );  
    std::vector<glm::vec3> vertices;  
    std::vector<glm::uvec3> indices;  
    vertices.push_back(glm::vec3(0.0,0.0,0.0));  
    for(int i = 0; i < n; i++){  
        vertices.push_back(glm::vec3(cos(2*M_PI*i/n),sin(2*M_PI*i/n),0.0));  
    }  
    for(int i = 0; i < n; i++){  
        indices.push_back(glm::uvec3(0,i+1,(i+1)%n+1));  
    }  
    if(vao_glm == nullptr){  
        vao_glm = new DescrVAO(cauce->num_atribos, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));  
        vao_glm->agregar(new DescrVBOInds(indices));  
        assert( glGetError() == GL_NO_ERROR );  
    }  
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );  
    vao_glm->draw( GL_TRIANGLES );  
    assert( glGetError() == GL_NO_ERROR );  
}
```

Problema 1.3.

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

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

(el enunciado continua en la siguiente transparencia)

Problema 1.3. (continuación)

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

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

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

```
void DibujarFigura3(unsigned int n){  
    assert(n>2);  
    assert( glgetError() == GL_NO_ERROR );  
    cauce->fijarUsarColorPlano( true );  
    cauce->fijarColor( { 1.0, 0.0, 1.0 } );  
    DibujarFigura1(n);  
    cauce->fijarColor( { 0.0, 1.0, 1.0 } );  
    DibujarFigura2(n);  
    assert( glgetError() == GL_NO_ERROR );  
}
```

Problema 1.4.

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

```
void DibujarFigura4(unsigned int n){  
    assert(n>2);  
    assert( glgetError() == GL_NO_ERROR );  
    std::vector<glm::vec3> vertices;  
    std::vector<glm::uvec3> indices;  
    vertices.push_back(glm::vec3(0.0,0.0,0.0));  
    for(int i = 0; i < n; i++){  
        vertices.push_back(glm::vec3(cos(2*M_PI*i/n),sin(2*M_PI*i/n),0.0));  
    }  
    for(int i = 0; i < n; i++){  
        indices.push_back(glm::uvec3(0,i+1,(i+1)%n+1));  
    }  
    if(vao_glm == nullptr){  
        vao_glm = new DescrVAO(cauce->num_atribos, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));  
        vao_glm->agregar(new DescrVBOInds(indices));  
        assert( glgetError() == GL_NO_ERROR );  
    }  
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );  
    cauce->fijarUsarColorPlano( true );  
    cauce->fijarColor( { 1.0, 0.0, 1.0 } );  
    vao_glm->draw( GL_TRIANGLES );  
    assert( glgetError() == GL_NO_ERROR );  
    cauce->fijarColor( { 0.0, 1.0, 1.0 } );  
    vao_glm->draw(GL_LINE_LOOP);  
    assert( glgetError() == GL_NO_ERROR );  
}
```

Problema 1.5.

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

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

```
void DibujarFigura5(unsigned int n){
    assert(n>2);
    assert( glGetError() == GL_NO_ERROR );
    std::vector<glm::vec3> vertices;
    std::vector<glm::vec3> colores;
    std::vector<glm::uvec3> indices;
    vertices.push_back(glm::vec3(0.0,0.0,0.0));
    for(int i = 0; i < n; i++){
        vertices.push_back(glm::vec3(cos(2*M_PI*i/n),sin(2*M_PI*i/n),0.0));
    }
    for(int i = 0; i < n; i++){
        indices.push_back(glm::uvec3(0,i+1,(i+1)%n+1));
    }
    srand(time(NULL));
    for(int i = 0; i < n+1; i++){
        colores.push_back(glm::vec3((float)rand()/RAND_MAX,(float)rand()/RAND_MAX,(float)rand()/RAND_MAX));
    }
    if(vao_glm == nullptr){
        vao_glm = new DescrVAO(cauce->num_atribs, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));
        vao_glm->agregar(new DescrVBOAtribs(cauce->ind_atrib_colores, colores));
        vao_glm->agregar(new DescrVBOInds(indices));
        assert( glGetError() == GL_NO_ERROR );
    }
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    // Llama a glUniform1i
    cauce->fijarUsarColorPlano( false );
    vao_glm->habilitarAtrib( cauce->ind_atrib_colores, true );
    vao_glm->draw( GL_TRIANGLES );
    assert( glGetError() == GL_NO_ERROR );
    vao_glm->habilitarAtrib( cauce->ind_atrib_colores, false );
    cauce->fijarUsarColorPlano( true );
    cauce->fijarColor( { 1.0, 0.0, 1.0 } );
    vao_glm->draw(GL_LINE_LOOP);
    assert( glGetError() == GL_NO_ERROR );
}
```

Problema 1.6.

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

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

```
struct AtributosVertices {
    glm::vec3 posiciones;
    glm::vec3 color;
};
```

```
void DibujarFigura6(unsigned int n){
    assert(n>2);
    assert( glGetError() == GL_NO_ERROR );
    static GLuint array = 0;
    std::vector<AtributosVertices> vertices;
    std::vector<glm::uvec3> indices;
    vertices.push_back({glm::vec3(0.0,0.0,0.0),glm::vec3(0.0,0.0,0.0)});
    for(int i = 0; i < n; i++){
        vertices.push_back({glm::vec3(cos(2*M_PI*i/n),sin(2*M_PI*i/n),0.0),glm::vec3(0.0,0.0,0.0)});
    }
    for(int i = 0; i < n; i++){
        indices.push_back(glm::uvec3(0,i+1,(i+1)%n+1));
    }
    srand(time(NULL));
    for(int i = 0; i < n+1; i++){
        vertices[i].color = glm::vec3((float)rand()/RAND_MAX,(float)rand()/RAND_MAX,(float)rand()/RAND_MAX);
    }
    if(array == 0){
        // Creamos VAO
        glGenVertexArrays(1, &array);
        glBindVertexArray(array);
        // Creamos VBO
        GLuint buffer;
        glGenBuffers(1, &buffer);
        glBindBuffer(GL_ARRAY_BUFFER, buffer);
        glBufferData(GL_ARRAY_BUFFER, vertices.size()*sizeof(AtributosVertices), vertices.data(), GL_STATIC_DRAW);
        // Especificar formato de las posiciones
        glVertexAttribPointer(cauce->ind_atrib_posiciones, 3, GL_FLOAT, GL_FALSE, sizeof(AtributosVertices), (void*)0);
        glEnableVertexAttribArray(cauce->ind_atrib_posiciones);
        // Especificar formato de los colores
        glVertexAttribPointer(cauce->ind_atrib_colores, 3, GL_FLOAT, GL_FALSE, sizeof(AtributosVertices), (void*)(3*sizeof(float)));
        // Hacemos VBO indice
        DescrVBOInds *vbo_ind = new DescrVBOInds(indices);
        vbo_ind->crearVBO();
        assert( glGetError() == GL_NO_ERROR );
    }
    // Dibujamos relleno
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    cauce->fijarUsarColorPlano( false );
    glEnableVertexAttribArray(cauce->ind_atrib_colores);
    glDrawElements(GL_TRIANGLES, indices.size()*3, GL_UNSIGNED_INT, 0);
    // Dibujamos aristas
    glDisableVertexAttribArray(cauce->ind_atrib_colores);
    cauce->fijarUsarColorPlano( true );
    cauce->fijarColor( { 0.0, 0.0, 0.0 } );
    glDrawArrays(GL_LINE_LOOP, 1, n);
}
```

Problema 1.7.

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

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

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

OpenGL mantiene una matriz P de 4×4 valores reales (llamada matriz de proyección), que se aplica a las coordenadas de todos los vértices que envía la aplicación, antes de visualizar.

Para cambiar la zona visible hay que fijar la matriz P a estos valores:

$$P = \begin{pmatrix} s_x & 0 & 0 & -c_x \cdot s_x \\ 0 & s_y & 0 & -c_y \cdot s_y \\ 0 & 0 & s_z & -c_z \cdot s_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde:

$$\begin{aligned} s_x &= 2/(x_1 - x_0) & c_x &= (x_0 + x_1)/2 \\ s_y &= 2/(y_1 - y_0) & c_y &= (y_0 + y_1)/2 \\ s_z &= 2/(z_1 - z_0) & c_z &= (z_0 + z_1)/2 \end{aligned}$$

```
void fijarRegionVisible(const float x0, const float x1,
                       const float y0, const float y1,
                       const float z0, const float z1)
{
    assert( glGetError() == GL_NO_ERROR );
    float sx = 2.0/(x1-x0), sy = 2.0/(y1-y0), sz = 2.0/(z1-z0);
    float cx = (x0+x1)/2, cy = (y0+y1)/2, cz = (z0+z1)/2;
    glm::mat4 mat = glm::mat4(sx, 0.0, 0.0, -1*cx*sx,
                               0.0, sy, 0.0, -1*cy*sy,
                               0.0, 0.0, sz, -1*cz*sz,
                               0.0, 0.0, 0.0, 1.0);
    // Llama a glUniformMatrix4fv
    cauce->fijarMatrizProyeccion(mat);
}
```

En `VisualizarFrame()`

```
// fija la matriz de proyección (la hace igual a la matriz identidad)
cauce->fijarMatrizProyeccion( glm::mat4(1.0) );

float r = (float)(ancho_actual)/alto_actual;
float x0, x1, y0, y1, z0, z1;
z0 = -1.0;
z1 = 1.0;
if(r > 1.0){
    x0 = -1.0*r;
    x1 = 1.0*r;
    y0 = -1.0;
    y1 = 1.0;
} else{
    x0 = -1.0;
    x1 = 1.0;
    y0 = -1.0/r;
    y1 = 1.0/r;
}
fijarRegionVisible(x0,x1,y0,y1,z0,z1);

// limpiar la ventana
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

Problema 1.8.

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

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

Problema 1.9.

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

Sea $C = [\hat{x}, \hat{y}, \hat{z}, \vec{0}]$ un marco de referencia cartesiano.

Sea $\vec{a} = C(a_x, a_y, a_z, 0)^t$ y $\vec{b} = C(b_x, b_y, b_z, 0)^t \Rightarrow$

$$\Rightarrow \vec{a} \cdot \vec{b} = (a_x \cdot \hat{x} + a_y \cdot \hat{y} + a_z \cdot \hat{z}) (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z}) \rightarrow \text{linealidad}$$

$$= a_x (\hat{x} \cdot (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z})) + a_y (\hat{y} \cdot (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z})) + a_z (\hat{z} \cdot (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z})) =$$

$$= a_x (b_x \cdot \hat{x} \cdot \hat{x} + b_y \cdot \hat{y} \cdot \hat{x} + b_z \cdot \hat{z} \cdot \hat{x}) + a_y (b_x \cdot \hat{x} \cdot \hat{y} + b_y \cdot \hat{y} \cdot \hat{y} + b_z \cdot \hat{z} \cdot \hat{y}) +$$

$$+ a_z (b_x \cdot \hat{x} \cdot \hat{z} + b_y \cdot \hat{y} \cdot \hat{z} + b_z \cdot \hat{z} \cdot \hat{z}) = \rightarrow \hat{x} \cdot \hat{x} = \hat{y} \cdot \hat{y} = \hat{z} \cdot \hat{z} = 1$$

$$= a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z \quad \text{perpendiculares}$$

Problema 1.10.

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

Sea $C = [\hat{x}, \hat{y}, \hat{z}, \vec{0}]$ un marco de referencia cartesiano.

Sea $\vec{a} = C(a_x, a_y, a_z, 0)^t$ y $\vec{b} = C(b_x, b_y, b_z, 0)^t \Rightarrow$

$$\Rightarrow \vec{a} \times \vec{b} = (a_x \cdot \hat{x} + a_y \cdot \hat{y} + a_z \cdot \hat{z}) \times (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z}) =$$

$$= a_x (\hat{x} \times (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z})) + a_y (\hat{y} \times (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z})) + a_z (\hat{z} \times (b_x \cdot \hat{x} + b_y \cdot \hat{y} + b_z \cdot \hat{z})) =$$

$$= a_x (b_x (\hat{x} \times \hat{x}) + b_y (\hat{x} \times \hat{y}) + b_z (\hat{x} \times \hat{z})) + a_y (b_x (\hat{y} \times \hat{x}) + b_y (\hat{y} \times \hat{y}) + b_z (\hat{y} \times \hat{z})) +$$

$$+ a_z (b_x (\hat{z} \times \hat{x}) + b_y (\hat{z} \times \hat{y}) + b_z (\hat{z} \times \hat{z})) = \rightarrow \hat{x} \times \hat{x} = \hat{y} \times \hat{y} = \hat{z} \times \hat{z} = 0 \quad \hat{x} \times \hat{y} = \hat{z} \quad \hat{y} \times \hat{x} = \hat{z} \quad \hat{z} \times \hat{y} = \hat{x}$$

$$= a_x b_y \hat{z} - a_x b_y \hat{y} - a_y b_x \hat{z} + a_y b_x \hat{y} + a_z b_y \hat{x} - a_z b_y \hat{x} =$$

$$= \hat{x} (a_y b_z - a_z b_y) + \hat{y} (a_z b_x - a_x b_z) + \hat{z} (a_x b_y - a_y b_x) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{pmatrix} \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

Problema 1.11.

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

Sea $C = [\hat{x}, \hat{y}, \hat{z}, \vec{0}]$ un marco de referencia cartesiano.

Sea $\vec{a} = C(a_x, a_y, a_z, 0)^t$ y $\vec{b} = C(b_x, b_y, b_z, 0)^t$

Hemos visto que $\vec{a} \times \vec{b} = \hat{x} (a_y b_z - a_z b_y) + \hat{y} (a_z b_x - a_x b_z) + \hat{z} (a_x b_y - a_y b_x)$

Luego:

$$\vec{a} \cdot (\vec{a} \times \vec{b}) = (a_x \cdot \hat{x} + a_y \cdot \hat{y} + a_z \cdot \hat{z}) \cdot [\hat{x} (a_y b_z - a_z b_y) + \hat{y} (a_z b_x - a_x b_z) + \hat{z} (a_x b_y - a_y b_x)]$$

$$= a_x (a_y b_z - a_z b_y) + a_y (a_z b_x - a_x b_z) + a_z (a_x b_y - a_y b_x) =$$

Distributiva y que $\hat{x} \cdot \hat{x} = 1 \quad \hat{y} \cdot \hat{y} = 0$

$$= a_x a_y b_z - a_x a_z b_y + a_y a_z b_x - a_y a_x b_z + a_z a_x b_y - a_z a_y b_x = 0$$

Análogo para \vec{b} .

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).

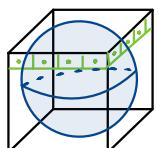
(continua en la siguiente transparencia)

a) celdas de Cubos

La esfera tiene diámetro 1, por lo que el cubo que la envuelve tiene $\ell = 1$. Consideraremos que cada cara tiene K celdas \Rightarrow cada celda es de lado $1/K$ y hay un total de K^3 celdas

④ se refiere a que no estamos dividiendo cada cara del cubo, sino el espacio del cubo en todas las direcciones. o sea K celdas en cada dimensión de $(x, y, z) \in R^3$

Enumeración espacial



Cada celda ocupa 1 bit, por lo que esta representación ocuparía K^3 bits = $\frac{K^3}{8}$ bytes = $(\frac{K}{2})^3$ bytes

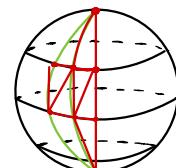
- $K = 16 \Rightarrow (\frac{16}{2})^3 = 8^3$ bytes = 512 bytes = 0,5 KB
- $K = 1024 \Rightarrow 512^3 = 128$ MB

Malla indexada de triángulos

En cada meridiano y paralelo hay K vértices, luego, tenemos un total de K^2 vértices. A su vez, cada vértice son 3 componentes, que ocupan 4 bytes $\Rightarrow 3 \cdot 4 \cdot K^2 = 12K^2$ bytes ocupan los vértices.

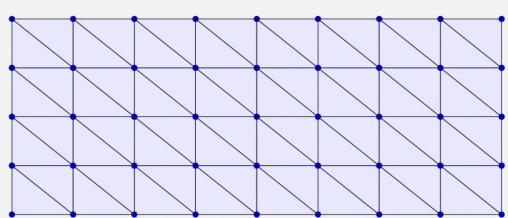
Cada cuadrado está formado por dos paralelos y dos meridianos, por lo que hay K^2 cuadrados $\Rightarrow 2K^2$ triángulos $\Rightarrow 3 \cdot 4 \cdot 2 \cdot K^2 = 24K^2$ bytes

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



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 $m = 4$).



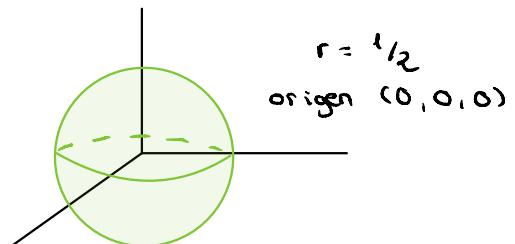
Almacenamos $(n+1)(m+1)$ vértices. Cada vértice contiene 3 valores, donde cada valor ocupa 4 bytes $\Rightarrow 4 \cdot 3 \cdot (n+1)(m+1)$ bytes para los vértices. Además, hay n filas de pares de triángulos y m columnas de triángulos $\Rightarrow 2nm$ triángulos.

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 $k = 1024$).



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.

Cada vértice contiene 3 valores, donde

cada valor ocupa 4 bytes $\Rightarrow 4 \cdot 3 \cdot (n+1)(m+1)$ bytes para los vértices. Además, hay n filas de pares de triángulos y m columnas de triángulos $\Rightarrow 2nm$ triángulos.

$$12(n+1)(m+1) + 2nm \cdot 3 \cdot 4 = 12(n+1)(m+1) + 24nm \text{ bytes.}$$

Supongamos n, m grandes.

$$\frac{n_C}{n_V} = \frac{2nm \cdot 12}{12(n+1)(m+1)} = \frac{2nm}{nm+n+nm+1} \xrightarrow{n,m \rightarrow \infty} \frac{2}{1} \Rightarrow n_C = 2n_V$$

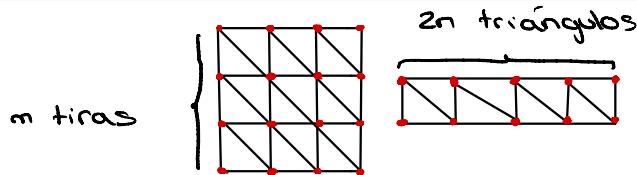
Problema 2.3.

Imagina de nuevo una malla como la del problema anterior, supongamos que usamos una representación como tiras de triángulos, de forma que cada fila de triángulos (con $2n$ triángulos) se almacena en una tira, habiendo un total de m tiras.

La tabla de punteros a tiras tiene un entero (el número de tiras) y m punteros, cada puntero suponemos que tiene 8 bytes de tamaño. De nuevo, asume que las coordenadas son de tipo float (4 bytes).

Responde a estas cuestiones:

(continua en la siguiente transparencia)



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.

Tenemos m tiras de $2n$ triángulos. Al tratarse de tiras, cada triángulo está identificado por un único vértice. Además, para garantizar la continuidad de la malla, habrá almacenado uno adicional al principio y otro adicional al final $\Rightarrow 2n+2$ vértices por cada tira $\Rightarrow 12(2n+2)$ bytes por cada tira, luego, en total, será $24mn + 24m$ bytes. Multiplicar por el nº tiras. Por otro lado, los punteros ocupan $4 + m8$ bytes, luego, en total: $24nm + 24m + 8m + 4$ bytes. Entero con número de tiras.

$$\frac{\text{Tam - ind}}{\text{Tam - tiras}} = \frac{36mn + 12n + 12m + 12}{24nm + 82m + 4} \xrightarrow{n,m \rightarrow \infty} \frac{36}{24} = 1.5$$

$$\frac{\text{Procesamiento - ind}}{\text{Procesamiento - tiras}} = \frac{n^{\circ} \text{ vértices - ind} \cdot t^{-1}}{n^{\circ} \text{ vértices - tiras} \cdot t^{-1}} = \frac{(n+1)(m+1)}{m(2n+2)} = \frac{mn+n+m+1}{2nm+2m} \xrightarrow{n,m \rightarrow \infty} \frac{1}{2}$$

Problema 2.4.

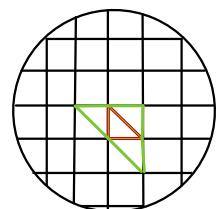
Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un poliedro simplemente conexo de caras triangulares). Considera el número de vértices n_V , el número de aristas n_A y el número de caras n_C en este tipo de mallas.

Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

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

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

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



cada triángulo es adyacente a 3 triángulos, luego, es adyacente a 3 aristas y cada arista es adyacente a exactamente dos triángulos, por lo que:
 $3n_C = 2n_A \Rightarrow n_A = \frac{3}{2}n_C$

usando la fórmula de Euler: $n_V - n_A + n_C = 2$

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

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

Problema 2.5.

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector `ari`, que en cada entrada tendrá una tupla de tipo `uvec2` (contiene dos `unsigned`) con los índices en la tabla de vértices de los dos vértices en los extremos de la arista. El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

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

(continua en la transparencia siguiente)

Problema 2.5. (continuación)

Considerar dos casos:

- 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....)
- 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.

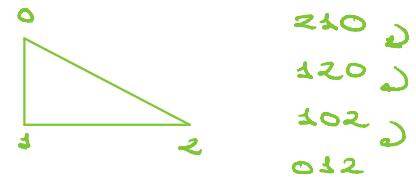
```
std::vector<glm::uvec2> CalcularAristas(std::vector<glm::uvec3> triangulos,
                                         int nvertices);
```

```
std::vector<uvec2> aristas;
for (const auto &triangulo : triangulos) {
    uvec3 vertices = triangulo;
    // Ordenamos los índices
    if (vertices[0] > vertices[1]) swap(vertices[0], vertices[1]);
    if (vertices[1] > vertices[2]) swap(vertices[1], vertices[2]);
    if (vertices[0] > vertices[2]) swap(vertices[0], vertices[2]);
    aristas.push_back(uvec2(vertices[0], vertices[1]));
    aristas.push_back(uvec2(vertices[1], vertices[2]));
    aristas.push_back(uvec2(vertices[2], vertices[0]));
}
```

// Eliminamos aristas duplicadas (Si pudieramos usar funciones externas como `unique` o estructuras como `set`, mejorarían mucho la eficiencia)

```
for (int i=0; i < aristas.size(); ++i) {
    for (int j=i+1; j < aristas.size(); ++j)
        if (aristas[i] == aristas[j])
            aristas.erase(aristas.begin() + j); // Lo convierte a iterator
    }
}
```

Esta solución es válida para los dos casos.



```
return aristas;
```

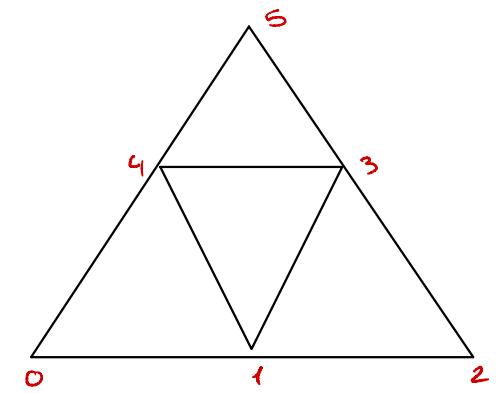
Caso 1 (orientación no coherente)

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

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

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

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



```
vector<Tupla2u> calculaAristas(vector<Tupla3u> triangulos, int nvertices){
    vector<Tupla2u> aristas;
    vector<vector<uint>> vertices_adyacentes;

    for (unsigned int i = 0; i < n; i++){
        vertices_adyacentes.push_back(vector<uint>());
    }

    for (unsigned int i = 0; i < triangulos.size(); i++){
        for (unsigned int j = 0; j < 3; j++){
            unsigned int a = triangulos[i][j];
            unsigned int b = triangulos[i][(j+1)%3];

            if (b > a){
                int aux = a;
                a = b;
                b = aux;
            }

            bool repetido = false;
            for (unsigned int k = 0; k < vertices_adyacentes[a].size() && !repetido; k++){
                if (vertices_adyacentes[a][k] == b)
                    repetido = true;
            }

            if (!repetido)
                vertices_adyacentes[a].push_back(b);
        }
    }

    for (unsigned int i = 0; i < vertices_adyacentes.size(); i++)
        for (unsigned int j = 0; j < vertices_adyacentes[i].size(); j++)
            aristas.push_back({i, vertices_adyacentes[i][j]});

    return aristas;
}
```

```
vector<Tupla2u> calculaAristas(vector<Tupla3u> triangulos, int nvertices){
    vector<Tupla2u> aristas;

    for (int i = 0; i < triangulos.size(); i++){
        for (int j = 0; j < 3; j++){
            unsigned int a = triangulos[i][j];
            unsigned int b = triangulos[i][(j+1)%3];

            if (a < b)
                aristas.push_back({a, b});
        }
    }

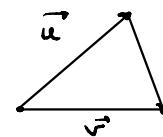
    return aristas;
}
```

Problema 2.6.

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

Área de un triángulo por vectores

$$A = \frac{1}{2} \| \vec{u} \times \vec{v} \|$$



```

float CalcularArea(MallaInd * malla) {
    float area = 0.0;
    for (const auto &triangulo : malla.triangulos) {
        vec3 a, b, c, ab, ac;
        a = vertices[triangulo[0]];
        b = vertices[triangulo[1]];
        c = vertices[triangulo[2]];
        ab = b - a;
        ac = c - a;
        area += 0.5 * length(cross(ab, ac));
    }
    return area;
}

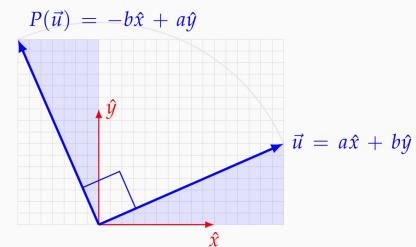
```

Problema 2.7.

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

Sea $C = [\hat{x}, \hat{y}, \hat{o}]$ un marco cartesiano.
Sean (a, b) las coordenadas de \vec{u}
en $C \Rightarrow \vec{u} = a\hat{x} + b\hat{y}$ y Sean
 $(-b, a)$ las coordenadas de
 $P(\vec{u})$ en $C \Rightarrow P(\vec{u}) = -b\hat{x} + a\hat{y}$. Luego, $\vec{u} \cdot P(\vec{u}) = a(-b) + ab = 0$.

En 2D definimos una transformación afín P tal que, aplicada a un vector $\vec{u} = C(a, b)$, produce otro vector $P(\vec{u}) = C(-b, a)$ perpendicular a \vec{u} (girado 90° a izquierdas).



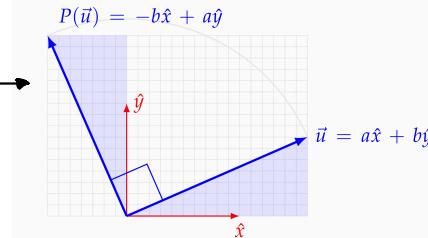
aquí $C = [\hat{x}, \hat{y}, \hat{o}]$ es un marco cartesiano cualquiera. Si se aplica P a un punto, se produce una rotación de 90° entorno al origen de C .

Problema 2.8.

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

Problema 2.9.

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



$P(\vec{u}) = b\hat{x} - a\hat{y}$
cambiando el signo
de la 2º componente
en vez de la 1º

Dicha transformación afín la podemos representar con la matriz $R = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$. Dado un vector $\vec{v} = \begin{bmatrix} a \\ b \end{bmatrix}$ en un marco cartesiano C , la rotación $P(\vec{v}) = R \cdot \vec{v} = \begin{bmatrix} -b \\ a \end{bmatrix}$. Ahora cambiemos de marco cartesiano a C' . Tenemos que $\vec{v}' = C'(a', b')$. Aplicando R tenemos que $P(\vec{v}') = C'(-b', a')$ y vemos que obtenemos el mismo tipo de expresión. Esto se debe a que P es una aplicación lineal, luego, mantiene la estructura, independientemente del sistema usado.

Problema 2.10.

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

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

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

$$\begin{aligned} R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) &= (\cos(\theta)\vec{a} + \sin(\theta)P(\vec{a}^\circ)) \cdot (\cos(\theta)\vec{b} + \sin(\theta)P(\vec{b}^\circ)) = \\ &= \cos^2(\theta)(\vec{a} \cdot \vec{b}) + \sin(\theta)\cos(\theta)P(\vec{b})\vec{a} + \sin(\theta)\cos(\theta)P(\vec{a}^\circ)\vec{b} + \sin^2(\theta)P(\vec{a}^\circ)P(\vec{b}^\circ) = \\ &= \cos^2(\theta)(\vec{a} \cdot \vec{b}) + \sin^2(\theta)P(\vec{a}^\circ)P(\vec{b}^\circ) + \sin(\theta)\cos(\theta)(P(\vec{b}^\circ)\vec{a} + P(\vec{a}^\circ)\vec{b}) \end{aligned}$$

Considera un marco cartesiano $C = [x, y, O]$. Tenemos pues que:

-) $\vec{a}^\circ = C(a_1, a_2) \Rightarrow P(\vec{a}^\circ) = C(-a_2, a_1)$
-) $\vec{b}^\circ = C(b_1, b_2) \Rightarrow P(\vec{b}^\circ) = C(-b_2, b_1)$
-) $\vec{a}^\circ P(\vec{b}) = -a_2 b_2 + a_1 b_1$
-) $\vec{b}^\circ P(\vec{a}^\circ) = -a_2 b_1 + a_1 b_2$
-) $P(\vec{a}^\circ)P(\vec{b}^\circ) = a_2 b_2 + a_1 b_1 = \vec{a} \cdot \vec{b}$

Luego, tenemos que:

$$\dots = \cos^2(\theta)(\vec{a} \cdot \vec{b}) + \sin^2(\theta)(\vec{a} \cdot \vec{b}) + 0 = (\sin^2(\theta) + \cos^2(\theta))(\vec{a} \cdot \vec{b}) = \vec{a} \cdot \vec{b}$$

Problema 2.11.

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

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

$$\text{Luego, } R_\theta(\vec{v}) = C(v_1 \cos(\theta) - v_2 \sin(\theta), v_2 \cos(\theta) + v_1 \sin(\theta))$$

$$\|R_\theta(\vec{v})\|^2 = \sqrt{(v_1 \cos(\theta) - v_2 \sin(\theta))^2 + (v_2 \cos(\theta) + v_1 \sin(\theta))^2}$$

$$(v_1 \cos(\theta) - v_2 \sin(\theta))^2 + (v_2 \cos(\theta) + v_1 \sin(\theta))^2 =$$

$$= v_1^2 \cos^2(\theta) + v_2^2 \sin^2(\theta) - 2v_1 v_2 \sin(\theta) \cos(\theta) + v_2^2 \cos^2(\theta) + v_1^2 \sin^2(\theta) + 2v_1 v_2 \sin(\theta) \cos(\theta) =$$

$$= v_1^2 (\sin^2(\theta) + \cos^2(\theta)) + v_2^2 (\sin^2(\theta) + \cos^2(\theta)) = v_1^2 + v_2^2$$

$$\Rightarrow \|R_\theta(\vec{v})\| = \sqrt{v_1^2 + v_2^2} = \|\vec{v}\|$$

Considera un marco cartesiano arbitrario, C . Tenemos entonces que

$$\vec{v} = C(v_1, v_2) \Rightarrow P(\vec{v}) = C(-v_2, v_1)$$

$$R_\theta(\vec{v}) = \cos(\theta)\vec{v} + \sin(\theta)P(\vec{v})$$

Problema 2.12.

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

Probaremos el ejercicio para $\text{Rot}_x[0]$, siendo los otros resultados totalmente análogos. La rotación definida para un vector 3D se puede definir como una en 2D dejando invariante una de las coordenadas. En el caso de $\text{Rot}_x[0]$, la coordenada que queda invariante es la primera, la componente x .

Sea entonces $\vec{a}^\circ = C(a_1, a_2, a_3)$ y $\vec{b}^\circ = C(b_1, b_2, b_3)$ y queremos probar que $\text{Rot}_x[0](\vec{a}^\circ) \cdot \text{Rot}_x[0](\vec{b}^\circ) = \vec{a} \cdot \vec{b}$.

-) $\text{Rot}_x[0](\vec{a}^\circ) = (a_1, 0, 0) + R_\theta((a_2, a_3))$
-) $\text{Rot}_x[0](\vec{b}^\circ) = (b_1, 0, 0) + R_\theta((b_2, b_3))$

“Notación no muy correcta, pero se entiende la idea”

Por lo que, el producto queda como:

$$\begin{aligned} \text{Rot}_x[0](\vec{a}^\circ) \cdot \text{Rot}_x[0](\vec{b}^\circ) &= a_1 b_1 + R_\theta((a_2, a_3)) R_\theta((b_2, b_3)) = \\ &= a_1 b_1 + a_2 b_2 + a_3 b_3 = \vec{a} \cdot \vec{b} \end{aligned}$$

Problema 2.13.

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

Usando el razonamiento del ejercicio anterior, tenemos que:

$$\text{Rot}_x[\vec{\alpha}](\vec{a}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \cos\theta - a_3 \sin\theta \\ a_2 \sin\theta + a_3 \cos\theta \end{pmatrix}$$

$$\|\text{Rot}_x[\vec{\alpha}](\vec{a})\| = \sqrt{a_1^2 + (a_2 \cos\theta - a_3 \sin\theta)^2 + (a_2 \sin\theta + a_3 \cos\theta)^2} = \sqrt{a_1^2 + a_2^2 + a_3^2} = \|\vec{a}\|$$

$\epsilon_{1,2,3}$

Problema 2.14.

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

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

Considera $\vec{a} = C(a_1, a_2, a_3)$, $\vec{b} = C(b_1, b_2, b_3)$ y consideremos:

$$\Rightarrow \vec{a} \times \vec{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix} \Rightarrow \text{Rot}_0(\vec{a} \times \vec{b}) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

$$\cdot 1 \text{ Rot}_0(\vec{a}) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \quad \text{Rot}_0(\vec{b}) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

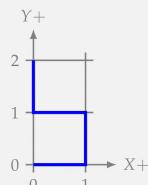
Calcular y hacer $\text{Rot}_0(\vec{a}) \times \text{Rot}_0(\vec{b})$ y ver que son iguales.

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).

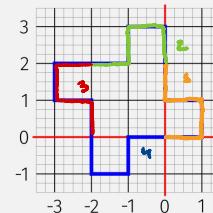
Usa la plantilla en el repositorio **opengl3-minimo** para esto.



```
void gancho(){
    std::vector<glm::vec3> vertices = {glm::vec3(0.0,0.0,0.0), glm::vec3(0.1,0.0,0.0),
                                         glm::vec3(0.1,0.1,0.0), glm::vec3(0.0,0.1,0.0), glm::vec3(0.0,0.2,0.0)};
    if(vao_glm == nullptr){
        vao_glm = new DescrVAO(cauce->num_atribs, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));
        assert(glGetError() == GL_NO_ERROR);
    }
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    cauce->fijarUsarColorPlano(true);
    vao_glm->draw(GL_LINE_STRIP);
    assert(glGetError() == GL_NO_ERROR);
}
```

Problema 2.16.

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



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

```

void gancho_x4(){
    cauce->resetMM();
    gancho();
    cauce->compMM(glm::translate(glm::vec3(0.0,0.2,0.0)));
    cauce->compMM(glm::rotate(glm::radians(90.0f),glm::vec3(0.0,0.0,1.0)));
    gancho();
    cauce->resetMM();
    cauce->compMM(glm::translate(glm::vec3(-0.2,0.2,0.0)));
    cauce->compMM(glm::rotate(glm::radians(180.0f),glm::vec3(0.0,0.0,1.0)));
    gancho();
    cauce->resetMM();
    cauce->compMM(glm::translate(glm::vec3(-0.2,0.0,0.0)));
    cauce->compMM(glm::rotate(glm::radians(270.0f),glm::vec3(0.0,0.0,1.0)));
    gancho();
    cauce->resetMM();
}

```

Problema 2.17.

Escribe el pseudocódigo OpenGL otra función (**gancho_2p**) para dibujar esa misma figura, pero escalada y rotada de forma que sus extremos coincidan con dos puntos arbitrarios distintos p_0 y p_1 , puntos cuyas coordenadas de mundo son $\mathbf{p}_0 = (x_0, y_0, 1)^t$ y $\mathbf{p}_1 = (x_1, y_1, 1)^t$. Estas coordenadas se pasan como parámetros a dicha función (como **vec3**)

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

```

d gancho_2p(glm::vec3 p0, glm::vec3 p1){
    glm::vec3 v = p1-p0;
    float l = glm::length(v);
    float angunlo = std::acos(v.y/l);
    if(v.y < 0.0){
        angunlo = -1*angunlo;
    }
    float escala = l/2.0f;
    cauce->resetMM();
    cauce->compMM(glm::translate(p0));
    angunlo = angunlo*180.0f/M_PI;
    cauce->compMM(glm::rotate(angunlo,glm::vec3(0.0,0.0,1.0)));
    cauce->compMM(glm::scale(glm::vec3(escala,escala,0.0)));
    gancho();
    cauce->resetMM();
}


$$\mathbf{r} = a(\mathbf{i}, \mathbf{j}) + b(\mathbf{j}, \mathbf{k}) \Rightarrow f(\mathbf{r}) =$$


$$f(\mathbf{r}) =$$


$$\text{Matricialmente:}$$


```

```

d gancho_2p_b(glm::vec3 p0, glm::vec3 p1){
    glm::vec3 y_prima = (p1-p0)/glm::vec3(2.0,2.0,2.0);
    glm::vec3 x_prima = glm::vec3(y_prima.y,-y_prima.x,y_prima.z);
    x_prima = glm::normalize(x_prima);
    glm::mat4 mat_trans = glm::mat4(1.0);
    mat_trans[0][0] = x_prima.x;
    mat_trans[0][1] = y_prima.x;
    mat_trans[1][0] = x_prima.y;
    mat_trans[1][1] = y_prima.y;
    mat_trans = glm::translate(p0)*mat_trans;
    cauce->resetMM();
    cauce->compMM(mat_trans);
    gancho();
    cauce->resetMM();
}

```

Explicación: Queremos que la figura se coloque y oriente entre los dos puntos, por lo que necesitamos:

- Distancia entre puntos: Para que se escale de forma correcta de \mathbf{p}_0 a \mathbf{p}_1 .
- Ángulo entre puntos: Para rotar la figura y que su orientación coincida con el vector $\overrightarrow{\mathbf{p}_0\mathbf{p}_1}$.

La aplicación $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ es lineal por ser composición de transformaciones lineales, por tanto, viene determinada por la imagen de los vectores de una base. Toma la base canónica:

$$\begin{aligned}\hat{\mathbf{i}} &= f((0,1)) = (\mathbf{p}_1 - \mathbf{p}_0)/2 = \hat{x}_1 \hat{\mathbf{i}} + \hat{y}_1 \hat{\mathbf{j}} \\ \hat{\mathbf{j}} &= f((1,0)) = \frac{R_{90^\circ}(f(0,1))}{\|R_{90^\circ}(f(0,1))\|} = \hat{x}_2 \hat{\mathbf{i}} + \hat{y}_2 \hat{\mathbf{j}}\end{aligned}$$

Por tanto, para cualquier vector $a\hat{\mathbf{i}} + b\hat{\mathbf{j}} = a \frac{R_{90^\circ}(f(0,1))}{\|R_{90^\circ}(f(0,1))\|} + b \frac{\mathbf{p}_1 - \mathbf{p}_0}{\|\mathbf{p}_1 - \mathbf{p}_0\|}$

Ampliándolo a \mathbb{R}^3 (2 invariantes)

$$H(f) = \begin{pmatrix} \hat{x}_2 & \hat{x}_1 & 0 \\ \hat{y}_2 & \hat{y}_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

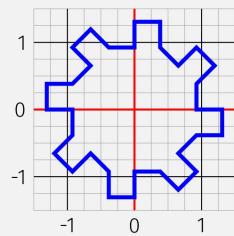
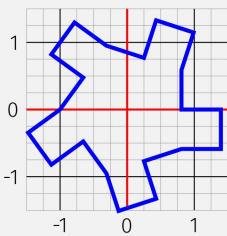
Finalmente, extendemos f como

$$f(q) = f(q_0, q_1, q_2, 1) = f(q_0, q_1, q_2) + \mathbf{p}_0$$

$$\begin{pmatrix} 1 & q_0 & q_1 & q_2 \\ 0 & 1 & 0 & p_1 \\ 0 & 0 & 1 & p_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{x}_2 & \hat{x}_1 & 0 & 0 \\ \hat{y}_2 & \hat{y}_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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).

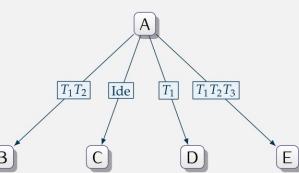
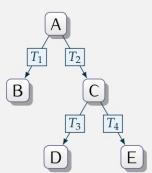


La idea es dividir el círculo en n lados ($\frac{360^\circ}{n}$) e ir tomando puntos consecutivos para aplicar la función anterior.

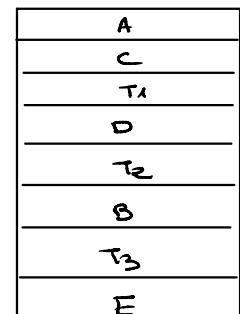
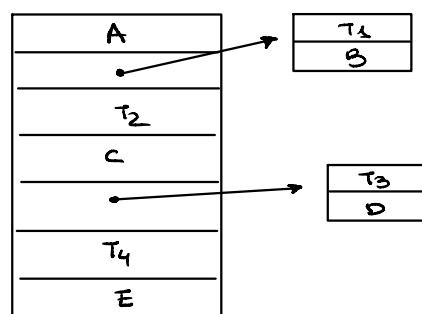
```
word concatenar( unsigned int n )
{
    float angulo= 2 * PI / n;
    for (int i=0; i<n, i++)
    {
        float angulo1 = angulo * i - 1           // Si quieras que aparezca
        float angulo2 = angulo * (i+1) - 2         orientado de alguna forma
        glm::vec3 p0 = glm::vec3( cos(angulo1), sen(angulo1), 0.0 );
        glm::vec3 p1 = glm::vec3( cos(angulo2), sen(angulo2), 0.0 );
        gandro - zb - p(p0, p1);
    }
}
```

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).



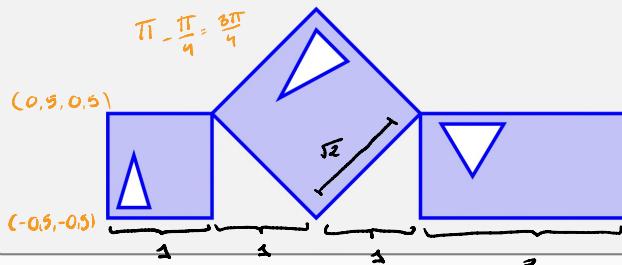
(usa el repositorio [opengl3-minimo](#))

```
void FiguraSimple(){
    assert( glGetError() == GL_NO_ERROR );
    std::vector<glm::vec3> vertices_cuadrado = {glm::vec3(-0.5,-0.5,0.0), glm::vec3(0.5,-0.5,0.0),
                                                glm::vec3(0.5,0.5,0.0), glm::vec3(-0.5,0.5,0.0)};
    std::vector<glm::vec3> vertices_triangulo = {glm::vec3(-0.4,-0.4,0.0), glm::vec3(-0.2,-0.4,0.0),
                                                glm::vec3(-0.3,0.1,0.0)};
    std::vector<glm::uvec3> indices_cuadrado = {glm::uvec3(0,1,2), glm::uvec3(0,2,3)};
    std::vector<glm::uvec3> indices_triangulo = {glm::uvec3(0,1,2)};
    DescrVAO *vao_cuadrados = nullptr;
    DescrVAO *vao_triangulos = nullptr;
    if(vao_cuadrados == nullptr){
        vao_cuadrados = new DescrVAO(cauce->num_atribs, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices_cuadrado));
        vao_cuadrados->agregar(new DescrVBOInds(indices_cuadrado));
        assert( glGetError() == GL_NO_ERROR );
    }
    if(vao_triangulos == nullptr){
        vao_triangulos = new DescrVAO(cauce->num_atribs, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices_triangulo));
        vao_triangulos->agregar(new DescrVBOInds(indices_triangulo));
        assert( glGetError() == GL_NO_ERROR );
    }
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    cauce->fijarUsarColorPlano( true );
    cauce->fijarColor( { 0.3, 0.7, 0.9 });
    vao_cuadrados->draw(GL_TRIANGLES);
    assert( glGetError() == GL_NO_ERROR );
    cauce->fijarColor( { 1.0, 1.0, 1.0 });
    vao_triangulos->draw(GL_TRIANGLES);
    assert( glGetError() == GL_NO_ERROR );
}
```

Para hacerlo con un solo VAO, lo mejor es hacerlo sin usar las clases del repositorio, pero, si las vas a usar, ten en cuenta que tienes que declarar los DescrVBOInds antes y no nulos, porque si son nulos, salta el assert de agregar.

Problema 2.21.

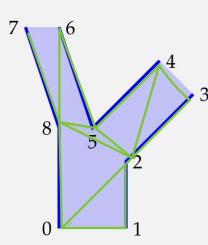
Usando exclusivamente llamadas a la función del problema 21, construye otra función llamada **FiguraCompleja** que dibuja la figura de aquí. Para lograrlo puedes usar manipulación de la pila de la matriz modelview (**pushMM** y **popMM**), junto con **MAT_Traslacion** y **MAT_Escalado**:



```
void figuraCompleja(){
    assert( glGetError() == GL_NO_ERROR );
    FiguraSimple();
    cauce->pushMM();
    cauce->compMM(glm::translate(glm::vec3(1.5,0.5,0.0)));
    cauce->compMM(glm::scale(glm::vec3(sqrt(2.0),sqrt(2.0),0.0)));
    float angulo = 3*M_PI/4.0f;
    cauce->compMM(glm::rotate(angulo,glm::vec3(0.0,0.0,1.0)));
    FiguraSimple();
    cauce->popMM();
    cauce->pushMM();
    cauce->compMM(glm::translate(glm::vec3(3.5,0.0,0.0)));
    cauce->compMM(glm::scale(glm::vec3(2.0,-1.0,0.0)));
    FiguraSimple();
    cauce->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)

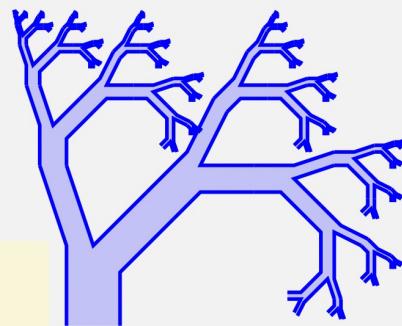
!! Recuerda que en el tamaño, si haces .size() tienes que multiplicar por el número de componentes que tiene !!

```
void Tronco(){
    assert( glGetError() == GL_NO_ERROR );
    std::vector<glm::vec3> vertices = {glm::vec3(0.0,0.0,0.0), glm::vec3(1.0,0.0,0.0), glm::vec3(1.0,1.0,0.0),
                                         glm::vec3(2.0,2.0,0.0), glm::vec3(1.5,2.5,0.0), glm::vec3(0.5,1.5,0.0),
                                         glm::vec3(0.0,3.0,0.0), glm::vec3(-0.5,3.0,0.0), glm::vec3(0.0,1.5,0.0)};
    std::vector<glm::uvec3> indices_relleno = {glm::uvec3(0,1,2),glm::uvec3(0,2,8),glm::uvec3(2,5,8),
                                                glm::uvec3(2,3,4),glm::uvec3(2,4,5),glm::uvec3(5,6,8),glm::uvec3(6,7,8)};
    std::vector<glm::uvec2> indices_aristas = {glm::uvec2(0,8),glm::uvec2(8,7),glm::uvec2(6,5),
                                                glm::uvec2(5,4),glm::uvec2(3,2),glm::uvec2(2,1)};
    DescrVAO *vao_relleno = nullptr;
    DescrVAO *vao_aristas = nullptr;
    if(vao_relleno == nullptr){
        vao_relleno = new DescrVAO(cauce->num_atribs, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));
        vao_relleno->agregar(new DescrVBOInds(indices_relleno));
        assert( glGetError() == GL_NO_ERROR );
    }
    if(vao_aristas == nullptr){
        vao_aristas = new DescrVAO(cauce->num_atribs, new DescrVBOAtribs(cauce->ind_atrib_posiciones, vertices));
        vao_aristas->agregar(new DescrVBOInds(GL_UNSIGNED_INT, indices_aristas.size()*2, indices_aristas.data()));
        assert( glGetError() == GL_NO_ERROR );
    }
}

glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
cauce->fijarUsarColorPlano( true );
cauce->fijarColor( { 0.5, 0.8, 1.0 } );
vao_relleno->draw(GL_TRIANGLES);
assert( glGetError() == GL_NO_ERROR );
cauce->fijarColor( { 0.0, 0.0, 1.0 } );
vao_aristas->draw(GL_LINES);
assert( glGetError() == GL_NO_ERROR );
}
```

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)



```
void Arbol(const unsigned int n{
    assert( glGetError() == GL_NO_ERROR );
    assert(n>0);
    Tronco();
    if(n>1){
        cauce->pushMM();
        cauce->compMM(glm::translate(glm::vec3(-0.5,3.0,0.0)));
        cauce->compMM(glm::scale(glm::vec3(0.5,0.5,1.0)));
        Arbol(n-1);
        cauce->popMM();
        cauce->pushMM();
        cauce->compMM(glm::translate(glm::vec3(1.5,2.5,0.0)));
        cauce->compMM(glm::rotate(-45.0f,glm::vec3(0.0,0.0,1.0)));
        cauce->compMM(glm::scale(glm::vec3(0.707,0.707,0.0)));
        Arbol(n-1);
        cauce->popMM();
    }
}
```

Problema 3.1.

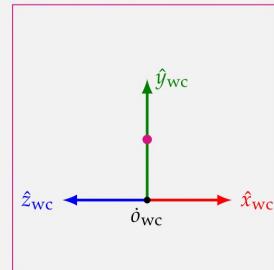
Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja, verde y azul), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

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



Sabemos que :

- \vec{a} es el punto de atención
- \vec{u} indica la dirección en la que el observador proyecta en vertical la imagen
- \vec{n} vector libre perpendicular al plano de visión

Como el punto morado debe aparecer en el centro del viewpoint, ese será nuestro punto de atención $\Rightarrow \vec{a} = (0.0, 0.5, 0.0)$.

Sabemos que \vec{n} es el vector que va del punto de atención \vec{a} en el mundo real al punto \vec{o}_{wc} , que es el punto del espacio donde estaría situado el observador que contempla la escena, en nuestro proyecto. Como el observador se encuentra a 3 unidades:

$$3 = \|\vec{n}\| = \|\vec{o}_{wc} - \vec{a}\| = \sqrt{(\vec{o}_x - 0)^2 + (\vec{o}_y - 0.5)^2 + (\vec{o}_z - 0)^2} = \sqrt{0x^2 + (0y - 0.5)^2 + 0z^2}$$

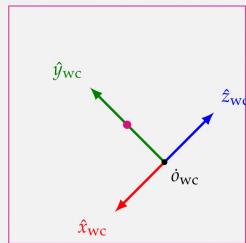
Si tomamos $0y = 0.5$ (como no nos ponen restricciones sobre 0_{wc} , tomamos el que queramos), tenemos que $3 = \sqrt{0x^2 + 0z^2}$. Por otro lado, el enunciado nos dice que los ejes X e Z se visualizan con la misma longitud, luego $x_{wc} = z_{wc}$, lo que nos lleva a que $0x = 0z$ (por como se define la matriz de vista V)

$$\Rightarrow 3 = \sqrt{20x^2} = \sqrt{2} \vec{o}_x \Rightarrow \vec{o}_x = \frac{3\sqrt{2}}{2} \Rightarrow \vec{n} = \left(\frac{3\sqrt{2}}{2}, 0, \frac{3\sqrt{2}}{2} \right)$$

A igual que en las prácticas, proyectarnos en vertical siguiendo el eje Y $\Rightarrow \vec{u} = (0, 1, 0)$.

Problema 3.2.

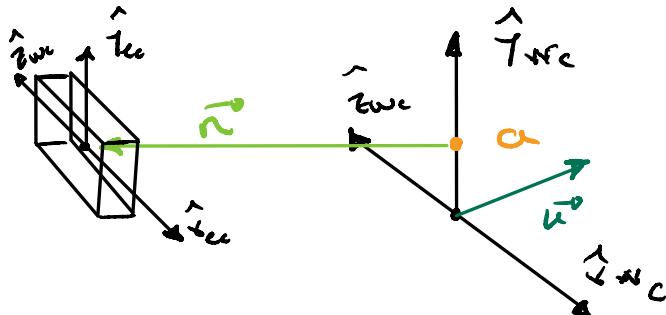
Repite el problema anterior, pero ahora para esta vista:



En este caso, el punto de atención sigue siendo el mismo ($\vec{a} = (0, 0.5, 0)$). Por otro lado, como se invierten los ejes X e Z , y , además, se mantienen las proporciones, tenemos que $\vec{n} = \left(-\frac{3}{\sqrt{2}}, 0, -\frac{3}{\sqrt{2}} \right)$

sigue siendo el mismo, ya que es el vector perpendicular al plano de visión y este no cambia $\vec{n} = \left(\frac{3}{\sqrt{2}}, 0, \frac{3}{\sqrt{2}} \right)$

Como podemos ver en la ilustración de la izquierda, $\vec{u} = (1, 1, 0)$



Problema 3.3.

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

Será pasar esas fórmulas a código →

Problema 3.4.

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

```
const GLfloat V[4][4] = {
    { a_x, a_y, a_z, d_x },
    { b_x, b_y, b_z, d_y },
    { c_x, c_y, c_z, d_z },
    { 0, 0, 0, 1 }
};
```

$$\begin{aligned} \mathbf{a} &= \mathbf{x}_{ec} & d_x &= -\mathbf{x}_{ec} \cdot \mathbf{o}_{ec} \\ \mathbf{b} &= \mathbf{y}_{ec} & d_y &= -\mathbf{y}_{ec} \cdot \mathbf{o}_{ec} \\ \mathbf{c} &= \mathbf{z}_{ec} & d_z &= -\mathbf{z}_{ec} \cdot \mathbf{o}_{ec} \end{aligned}$$

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)

$$\text{Como } \vec{n} = \vec{o}_{ec} - \vec{a} \Rightarrow \vec{o}_{ec} = \vec{n} + \vec{a}$$

$$\hat{x}_{ec} = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{eje Z paralelo a VPN, normalizado})$$

$$\hat{x}_{ec} = \frac{\vec{u} \times \vec{n}}{\|\vec{u} \times \vec{n}\|} \quad (\text{eje X perpendicular a VPN y VUP, normalizado})$$

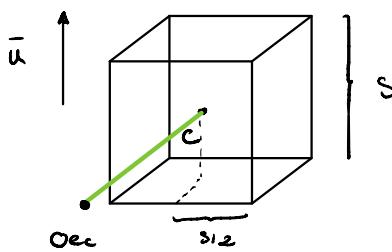
$$\hat{y}_{ec} = \hat{z}_{ec} \times \hat{x}_{ec} \quad (\text{eje Y perpendicular a los otros dos})$$

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) .



Como tenemos un cubo de lado s , se trata de proyección ortográfica. Por otro lado, tenemos que $a = c = (c_x, c_y, c_z)$ como el observador está fuera del cubo y el punto de atención en el centro del mismo, podemos usar los bordes para delimitar

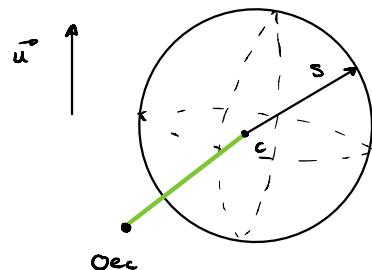
la distancia horizontal hasta los bordes $\Rightarrow l = c_x - \frac{s}{2}$, $r = c_x + \frac{s}{2}$.

Dado que $\vec{u} = (0, 1, 0)$, podemos repetir el mismo razonamiento que acabamos de aplicar y obtenemos que $\Rightarrow b = c_y - \frac{s}{2}$, $t = c_y + \frac{s}{2}$.

Para maximizar el tamaño de los objetos, sin que se recorte ningún triángulo, necesitamos que todos los vértices queden dentro del cubo, lo más cercano posible a las paredes, sin salirse del mismo. Nuevamente, usando el razonamiento de antes $\Rightarrow -n = c_z + \frac{s}{2}$, $-f = c_z - \frac{s}{2}$

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 c).

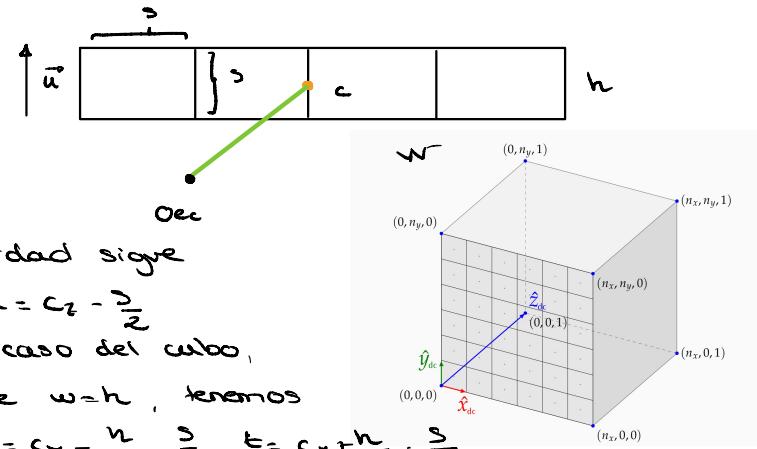


Para la distancia horizontal, proyectamos la esfera en el plano XY y obtenemos que (usando el mismo razonamiento que el ejercicio anterior) que $l = c_x - r$, $r = c_x + r$. Analógicamente al ejercicio anterior y el razonamiento que acabamos de hacer, obtenemos que $b = c_y - r$, $t = c_y + r$, $-n = c_z + r$, $-f = c_z - r$

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 $w = h$.

Como podemos observar, la profundidad sigue siendo la misma $\Rightarrow r = c_z + \frac{s}{2}$, $-l = c_z - \frac{s}{2}$. El resto del problema es análogo al caso del cubo, salvo que como no podemos suponer que $w=h$, tenemos que: $l = cx - \frac{w \cdot s}{h} \cdot \frac{3}{2}$, $r = cx + \frac{w \cdot s}{h} \cdot \frac{3}{2}$, $b = cy - \frac{w \cdot s}{h} \cdot \frac{3}{2}$, $t = cy + \frac{w \cdot s}{h} \cdot \frac{3}{2}$. Con esto, conseguimos mantener las proporciones. Para hallar esos valores, basta ver que $\frac{r-l}{t-b} = \frac{w}{h} \Leftrightarrow r-l = \frac{w}{h} (t-b) = \frac{w}{h} s$

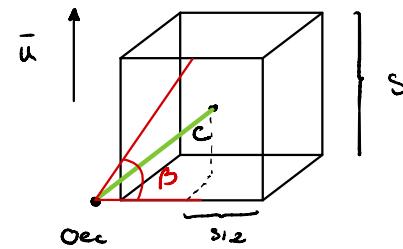


Problema 3.8.

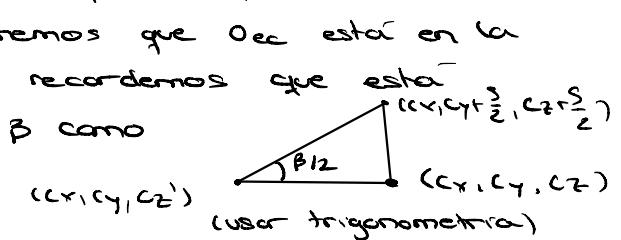
Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo β en grados que debe ser la apertura de campo vertical de la proyección perspectiva. Para ello, ahora tenemos libertad para situar al observador en la línea paralela al eje Z que pasa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

Indica como calcular la coordenada Z que debemos usar ahora para o_{dc} (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 $c = (c_x, c_y, c_z)$).

β está contenido entre 0 y 180° . Como tenemos que o_{dc} está en la recta paralela al eje Z que pasa por c (que recordemos que está centrado en el cubo), podemos calcular β como

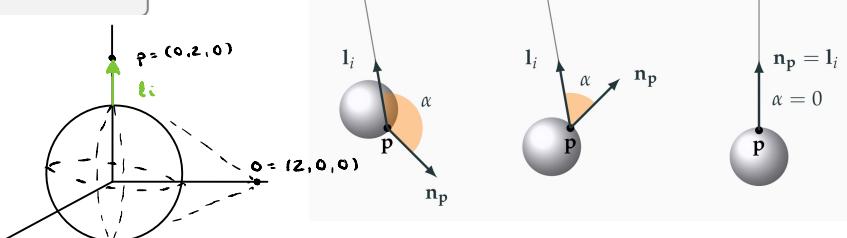


Sabemos que la apertura vertical de



- Si $\alpha > 90^\circ$, entonces:
 - $\cos(\alpha)$ es negativo.
 - la superficie, en p , está orientada de espaldas a la fuente de luz.
 - la contribución de esa fuente debe ser 0.
- Si $0^\circ \leq \alpha \leq 90^\circ$, entonces:
 - la superficie, en p , está orientada de cara a la fuente de luz.
 - $\cos(\alpha)$ estará entre 0 y 1 (entre $\cos(90^\circ)$ y $\cos(0^\circ)$).
 - se puede demostrar que el valor $\cos(\alpha)$ es proporcional a la densidad de fotones por unidad de área que inciden en el entorno de p , provenientes de la i -ésima fuente de luz.

$$\begin{array}{lll} 90^\circ < \alpha & 0^\circ < \alpha < 90^\circ & \alpha = 0^\circ \\ 0 > \cos(\alpha) & 1 > \cos(\alpha) > 0 & \cos(\alpha) = 1 \end{array}$$

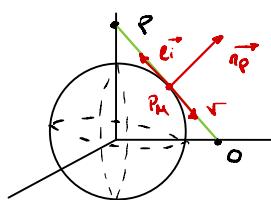


En el caso de los materiales puramente difusos, el brillo es máximo cuando la luz incide directamente sobre la superficie, ya que no reflejan la luz.

La normal en una esfera centrada es el vector que va del origen al punto normalizado (longitud = 1) $\Rightarrow \vec{n}_p = \frac{1}{\sqrt{x^2+y^2+z^2}} (x, y, z)$. Como nuestra esfera tiene $r = s$, $\vec{n}_p = (x, y, z)$. Para maximizarlo, queremos que la normal sea

paralela al vector que va del punto a la fuente de luz, es decir, queremos que la posición del punto en la esfera sea tal que (x_i, y_i, z) sea paralelo al vector que apunta a $(0, 2, 0)$ desde el origen. Buscamos el punto de coordenadas máximas que lo cumple (el más cercano a la fuente de luz), que en este caso es el $(0, 2, 0)$. Este punto no es visible desde la posición del espectador.

La componente especular se refiere a la parte de la reflexión de la luz que que causa reflejos o destellos intensos en una superficie



Que $MS = (1, 1, 1)$ y resto a 0 (K_d, K_s, K_a), significa que la componente especular domina completamente en la reflexión de la luz. El brillo especular será máximo cuando la luz se refleje directamente hacia el espectador.

Para un material pseudo - especular, esto ocurrirá cuando

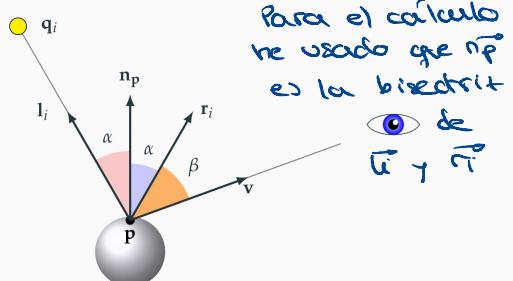
la dirección de la reflexión esté alineada desde el punto de la esfera al observador.

Como queremos maximizarlo, $r_i = \vec{v}$ y $\beta = 0^\circ$

$$\begin{aligned} \vec{r}_i &= (0, 2, 0) - \vec{p}_H = (0, 2, 0) - \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) = \\ &= \left(-\frac{\sqrt{2}}{2}, 2 - \frac{\sqrt{2}}{2}, 0\right) \\ \vec{r} &= (2, 0, 0) - \vec{p}_H = \left(2 - \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \\ \vec{n}_p &= \vec{p} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \\ \vec{r}_i &= 2(\vec{r}_i \cdot \vec{n}_p)\vec{n}_p - \vec{r}_i = \vec{v} = \left(2 - \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\right) \end{aligned}$$

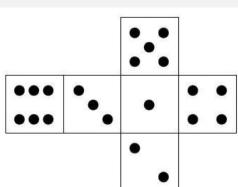
El punto sí es visible para el espectador.

El valor $\vec{r}_i \cdot \vec{v}$ es el coseno del ángulo β que hay entre la dirección de máxima brillo \vec{r}_i y la dirección \vec{v} hacia el observador. Cuando $\vec{r}_i = \vec{v}$ entonces $\beta = 0^\circ$, $\cos(\beta) = 1$, y el brillo es máximo:



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:

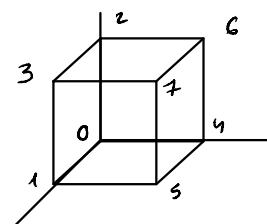


Problema 3.10. (continuación)

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 textura, y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en $(1/2, 1/2, 1/2)$. Dibuja un esquema de la texture en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de texture.

Como cada vértice es adyacente a tres caras, necesitaremos replicar 3 veces su valor para poder asignar correctamente la texture, luego, necesitaremos 24 vértices.



$$4 = \{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$$

$$\{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$$

$$\{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$$

Problema 3.12.

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

triangulos = { { 0, 1, 3 }, { 0, 3, 2 }, { 1, 4, 5 }, { 1, 4, 6 }, { 8, 13, 9 }, { 8, 12, 13 }, { 10, 11, 15 }, { 10, 15, 14 }, { 16, 22, 20 }, { 16, 18, 22 }, { 17, 21, 23 }, { 17, 23, 19 } }

Izq-dcha
Abajo-arriba
Atrás-delante

se puede usar la misma ya que necesitamos una normal en cada cara

cc-tt-ver = { { $\frac{1}{4}, \frac{2}{3}$ }, { $\frac{1}{4}, \frac{1}{3}$ }, { $\frac{1}{2}, \frac{2}{3}$ }, { $\frac{1}{2}, \frac{1}{3}$ }, { $1, \frac{2}{3}$ }, { $1, \frac{1}{3}$ }, { $\frac{3}{4}, \frac{2}{3}$ }, { $\frac{3}{4}, \frac{1}{3}$ }, { $\frac{1}{4}, \frac{2}{3}$ }, { $\frac{1}{4}, \frac{1}{3}$ }, { $\frac{1}{2}, \frac{2}{3}$ }, { $\frac{1}{2}, \frac{1}{3}$ }, { $0, \frac{2}{3}$ }, { $0, \frac{1}{3}$ }, { $\frac{3}{4}, \frac{2}{3}$ }, { $\frac{3}{4}, \frac{1}{3}$ }, { $\frac{1}{2}, 1$ }, { $\frac{1}{2}, 0$ }, { $1, \frac{2}{3}$ }, { $\frac{1}{2}, \frac{1}{3}$ }, { $\frac{3}{4}, 1$ }, { $\frac{3}{4}, 0$ }, { $\frac{3}{4}, \frac{2}{3}$ }, { $\frac{3}{4}, \frac{1}{3}$ } }

nor-ver = { { -1, 0, 0 }, { -1, 0, 0 }, { -1, 0, 0 }, { -1, 0, 0 }, { 1, 0, 0 }, { 1, 0, 0 }, { 1, 0, 0 }, { 1, 0, 0 }, { 0, -1, 0 }, { 0, -1, 0 }, { 0, -1, 0 }, { 0, -1, 0 }, { 0, 1, 0 }, { 0, 1, 0 }, { 0, 1, 0 }, { 0, 1, 0 }, { 0, 0, -1 }, { 0, 0, -1 }, { 0, 0, -1 }, { 0, 0, -1 }, { 0, 0, 1 }, { 0, 0, 1 }, { 0, 0, 1 }, { 0, 0, 1 } }

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 desplazar (float t, float s) {
     $\mathbf{o}_{ec} = (1 - \frac{t}{s})\mathbf{o}_0 + \frac{t}{s}\mathbf{o}_1$ 
     $\mathbf{a}_{ec} = (1 - \frac{t}{s})\mathbf{a}_0 + \frac{t}{s}\mathbf{a}_1$ 
     $\vec{n} = \mathbf{o}_{ec} - \mathbf{a}_{ec}$ 
     $\vec{u} = (0, 1, 0)$ 
     $\vec{z}_{ec} = \frac{\vec{n}}{\|\vec{n}\|}$ 
     $\vec{x}_{ec} = \frac{\vec{u} \times \vec{z}_{ec}}{\|\vec{u} \times \vec{z}_{ec}\|}$ 
     $\vec{y}_{ec} = \vec{z}_{ec} \times \vec{x}_{ec}$ 
}
```

En el modo de primera persona con traslaciones, la actualización de la cámara supone simplemente trasladar el origen del marco de cámara \mathbf{o}_{ec} y el punto de atención \mathbf{a}_t de forma solidaria:

► La operación **desplRotarXY** (Δ_a, Δ_b) supone:

1. $\mathbf{a}_t = \mathbf{a}_t + \Delta_x \mathbf{x}_{ec} + \Delta_y \mathbf{y}_{ec}$
2. $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$

► La operación **moverZ** (Δ_z) supone:

1. $\mathbf{a}_t = \mathbf{a}_t + \Delta_z \mathbf{z}_{ec}$
2. $\mathbf{o}_{ec} = \mathbf{a}_t + \mathbf{n}$

Las tuplas $\mathbf{s}, \mathbf{n}, \mathbf{x}_{ec}, \mathbf{y}_{ec}, \mathbf{z}_{ec}$ no cambian.

El marco de referencia de vista \mathcal{V} , se define a partir de los siguientes parámetros

\mathbf{o}_{ec} = es el punto del espacio foco de la proyección, donde estaría situado el observador ficticio que contempla la escena (*projection reference point, PRP*)

\vec{n} = vector libre perpendicular al *plano de visión* (plano ficticio donde se proyecta la imagen perpendicular al eje óptico de la cámara virtual). (*view plane normal, VPN*).

\mathbf{a} = punto en el eje óptico, también llamado *punto de atención o look-at point*.

\vec{u} = es un vector libre que indica una dirección que el observador ve proyectada en vertical en la imagen (apuntando hacia arriba) (*view-up vector, VUP*)

De los tres parámetros \mathbf{o}_{ec} , \vec{n} y \mathbf{a} solo hay que especificar dos, ya que no son independientes (se cumple $\mathbf{o}_{ec} = \mathbf{a} + \vec{n}$).

$$\hat{z}_{ec} = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{eje Z paralelo a VPN, normalizado})$$

$$\hat{x}_{ec} = \frac{\vec{u} \times \vec{n}}{\|\vec{u} \times \vec{n}\|} \quad (\text{eje X perpendicular a VPN y VUP, normalizado})$$

$$\hat{y}_{ec} = \hat{z}_{ec} \times \hat{x}_{ec} \quad (\text{eje Y perpendicular a los otros dos})$$

```
const GLfloat V[4][4] = // matriz V asociada al marco  $\mathcal{V}$  (por filas)
{{ a_x, a_y, a_z, d_x }, // coords. de mundo de  $\mathbf{x}_{ec}$ , y  $d_x = -\mathbf{o}_{ec} \cdot \mathbf{x}_{ec}$ 
 { b_x, b_y, b_z, d_y }, // coords. de mundo de  $\mathbf{y}_{ec}$ , y  $d_y = -\mathbf{o}_{ec} \cdot \mathbf{y}_{ec}$ 
 { c_x, c_y, c_z, d_z }, // coords. de mundo de  $\mathbf{z}_{ec}$ , y  $d_z = -\mathbf{o}_{ec} \cdot \mathbf{z}_{ec}$ 
 { 0, 0, 0, 1 } // origen de  $\mathcal{V}$ 
};

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glMultTransposeMatrixf( V );
```

Problema 4.2.

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

- ▶ El rayo tiene como origen o extremo el punto 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.

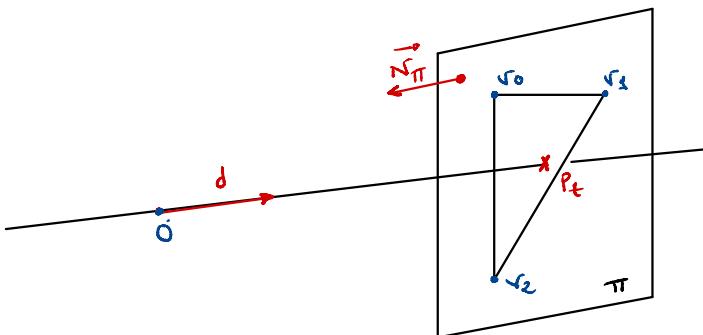
(ver la siguiente transparencia).

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).



La recta y el plano no intersecan y solo si $\vec{d} \perp \vec{N}_{\pi}$ (perpendiculares) (evidentemente, \vec{d} no está contenido en el plano), luego, si intersecan, tenemos que $\vec{d} \neq \vec{N}_{\pi} \Rightarrow \vec{d} \cdot \vec{N}_{\pi} \neq 0$

Podemos calcular el punto de intersección como $\mathbf{p}_t = \mathbf{o} + t\mathbf{d}$. Si $t < 0$, el rayo no interseca con el plano del triángulo. En caso contrario, calcularemos las coordenadas baricentricas de $\mathbf{p}_t - \mathbf{v}_0 = a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0)$ y si $0 \leq a+b \leq 1$, $a, b \geq 0$, devolvemos true.

Para calcular el punto intersección, el plano es el conjunto de puntos \mathbf{x} que satisfacen $(\mathbf{x} - \mathbf{v}_0) \cdot \vec{N}_{\pi} = 0$. cogiendo $\mathbf{p}_t = \mathbf{o} + t\mathbf{d}$ y sustituyendo:

$$((\mathbf{o} + t\mathbf{d}) - \mathbf{v}_0) \cdot \vec{N}_{\pi} = 0 \Rightarrow$$

$$\Rightarrow (\mathbf{o}^0 + t\mathbf{d}^0 - \mathbf{v}_0^0) \mathbf{N}_{\pi}^0 + (\mathbf{o}^1 + t\mathbf{d}^1 - \mathbf{v}_0^1) \mathbf{N}_{\pi}^1 + (\mathbf{o}^2 + t\mathbf{d}^2 - \mathbf{v}_0^2) \mathbf{N}_{\pi}^2 = 0$$

$$\Rightarrow t = \frac{-\mathbf{N}_{\pi}^0 \mathbf{o}^0 + \mathbf{N}_{\pi}^0 \mathbf{v}_0^0 - \mathbf{N}_{\pi}^1 \mathbf{o}^1 + \mathbf{N}_{\pi}^1 \mathbf{v}_0^1 - \mathbf{N}_{\pi}^2 \mathbf{o}^2 + \mathbf{N}_{\pi}^2 \mathbf{v}_0^2}{\mathbf{N}_{\pi}^0 \mathbf{d}^0 + \mathbf{N}_{\pi}^1 \mathbf{d}^1 + \mathbf{N}_{\pi}^2 \mathbf{d}^2}$$

Para calcular las coordenadas baricentricas:

$$(\mathbf{p}_t - \mathbf{v}_0) = a(\mathbf{v}_1 - \mathbf{v}_0) + b(\mathbf{v}_2 - \mathbf{v}_0), a, b \in \mathbb{R} \Rightarrow$$

$$\mathbf{p}_t^0 - \mathbf{v}_0^0 = a(\mathbf{v}_1^0 - \mathbf{v}_0^0) + b(\mathbf{v}_2^0 - \mathbf{v}_0^0) \quad \leftarrow a = \frac{\mathbf{p}_t^0 - \mathbf{v}_0^0 - b(\mathbf{v}_2^0 - \mathbf{v}_0^0)}{\mathbf{v}_1^0 - \mathbf{v}_0^0}$$

$$\Rightarrow \mathbf{p}_t^1 - \mathbf{v}_0^1 = a(\mathbf{v}_1^1 - \mathbf{v}_0^1) + b(\mathbf{v}_2^1 - \mathbf{v}_0^1)$$

$$\mathbf{p}_t^2 - \mathbf{v}_0^2 = a(\mathbf{v}_1^2 - \mathbf{v}_0^2) + b(\mathbf{v}_2^2 - \mathbf{v}_0^2) \quad \text{Redundante, ya que } \mathbf{p}_t - \mathbf{v}_0 \text{ en el plano y existe solución para } a, b$$

Sustituyendo: $\mathbf{p}_t^1 - \mathbf{v}_0^1 = \frac{\mathbf{p}_t^0 - \mathbf{v}_0^0 - b(\mathbf{v}_2^0 - \mathbf{v}_0^0)}{\mathbf{v}_1^0 - \mathbf{v}_0^0} (\mathbf{v}_1^1 - \mathbf{v}_0^1) + b(\mathbf{v}_2^1 - \mathbf{v}_0^1)$

$$b = \frac{(\mathbf{p}_t^1 - \mathbf{v}_0^1)(\mathbf{v}_1^0 - \mathbf{v}_0^0) - (\mathbf{v}_1^1 - \mathbf{v}_0^1)(\mathbf{p}_t^0 - \mathbf{v}_0^0 - b(\mathbf{v}_2^0 - \mathbf{v}_0^0))}{(\mathbf{v}_1^0 - \mathbf{v}_0^0)(\mathbf{v}_2^1 - \mathbf{v}_0^1)}$$

$$a = \frac{p_t^0 - r_0^0 - \left[\frac{(p_t^1 - r_0^1)(r_1^0 - r_0^0) - (r_1^1 - r_0^1)(p_t^0 - r_0^0) - b(r_2^0 - r_0^0)}{(r_1^0 - r_0^0)(r_2^1 - r_0^1)} \right] (r_2^0 - r_0^0)}{r_1^0 - r_0^0}$$

↑ cosas mías,
pasad de ello

bool intersect(O, d, r0, r1, r2) {

$$u = r_1 - r_0$$

$$v = r_2 - r_0$$

$$N_{\pi} = \| u \times v \|$$

if (d · N_{\pi} = 0)

return false

$$t = \frac{-N_{\pi}^0 r_0^0 + N_{\pi}^0 r_0^0 - N_{\pi}^0 r_1^0 + N_{\pi}^1 r_0^0 - N_{\pi}^2 r_0^2 + N_{\pi}^2 r_0^2}{N_{\pi}^0 d^0 + N_{\pi}^1 d^1 + N_{\pi}^2 d^2}$$

if ($t < 0$)

return false

$$p_t = o + t d$$

$$b = \frac{(p_t^1 - r_0^1)(r_1^0 - r_0^0) - (r_1^1 - r_0^1)(p_t^0 - r_0^0) - b(r_2^0 - r_0^0)}{(r_1^0 - r_0^0)(r_2^1 - r_0^1)}$$

$$a = \frac{p_t^0 - r_0^0 - b(r_2^0 - r_0^0)}{r_1^0 - r_0^0}$$

if ($a < 0 \text{ || } b < 0 \text{ || } a+b < 0 \text{ || } a+b \geq 2$)

return false

return true

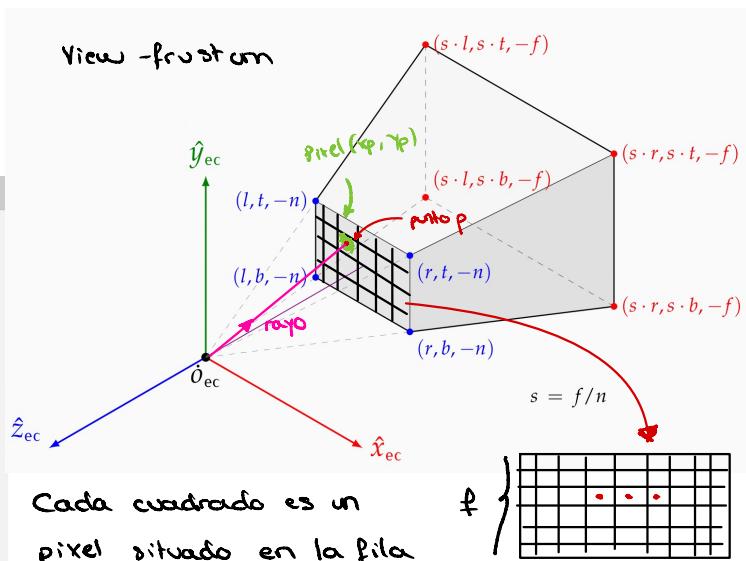
Problema 4.3.

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:

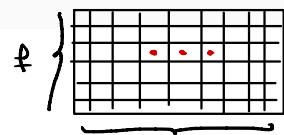
- ▶ Tenemos una vista perspectiva, y conocemos los 6 valores l, r, t, b, n, f usados para construir la matriz de proyección.
- ▶ También conocemos el marco de coordenadas de vista, es decir, las tuplas x_{ec}, y_{ec} y o_{ec} con los versores y la tupla \hat{o}_{ec} con el punto origen (todos en coordenadas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

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

Tiene de origen de coordenadas $\mathbf{o}_{ec} = (0, 0, 0)$ y $\mathbf{d} = \mathbf{p} - \mathbf{o}_{ec}$, siendo \mathbf{p} el centro del pixel (Hablando en coordenadas de vista). Como los pixeles mantienen siempre la misma proporción, podemos calcular cuánto ocupa cada pixel y, así, hallar fácilmente su punto central. Con eso bastaría para el ejercicio, pero, adicionalmente, vamos a pasar de coordenadas de vista a coordenadas de mundo.



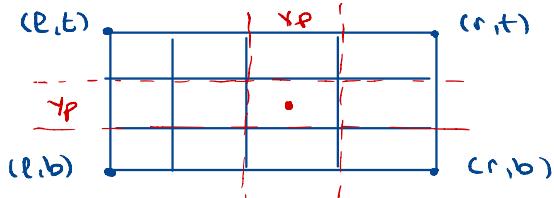
Cada cuadrado es un pixel situado en la fila $i \in [0, f-1]$ y columna $j \in [0, w-1]$ y tiene un punto central llamado centro del pixel. Si ha hecho click en el pixel $j=x_p, i=y_p$. El rayo



$$s = f/n$$

calcular Rayo (x_p, y_p) {

// Calculamos las coordenadas del punto central del pixel.



$$(t - b) / w = \Delta w$$

Incremento en cada eje

$$(t - b) / f = \Delta f$$

$$\Delta x = \frac{(x_{p+1} \Delta w) - (x_p \Delta w)}{2}$$

$$\Delta y = \frac{(y_{p+1} \Delta f) - (y_p \Delta f)}{2}$$

Incremento dentro de la celda hasta el punto

Para hallar el punto bastaría con

$$P_x = x_p \Delta w + \Delta x$$

$$P_y = y_p \Delta f + \Delta y$$

$P_z = -n$ (porque estamos en el plano near)

Ese ha sido el cálculo geométrico de P , también podemos hallarlo por interpolación:

$$P_x = l + (r - l) \cdot \frac{x_p + 0.5}{w} \quad P_y = b + (t - b) \cdot \frac{y_p + 0.5}{h}$$

vec3 p = vec3(P_x, P_y, P_z) // Usar fórmula que quieras

vec3 d = $p - \text{Oc}$ = P

d = d.normalize()

return (Oc, d)

t

Como bien he dicho antes, estas coordenadas son de vista, que es suficiente para el enunciado. Si quisieramos las coordenadas de mundo, tendríamos que hacer lo siguiente:

calcularRayo (x_p, y_p) {

vec3 p = vec3(P_x, P_y, P_z) // Usar fórmula que quieras

vec3 d = $p - \text{Oc}$ // = P

Como solo nos interesa la dirección del rayo en coordenadas de mundo, basta con multiplicar cada componente de d por el marco [Vec, Tex, Zec] sin considerar w .

vec3 d_wc = ($d_x \cdot \text{Vec}, d_y \cdot \text{Tex}, d_z \cdot \text{Zec}$)

return (Oc, d_wc - normalize())

t

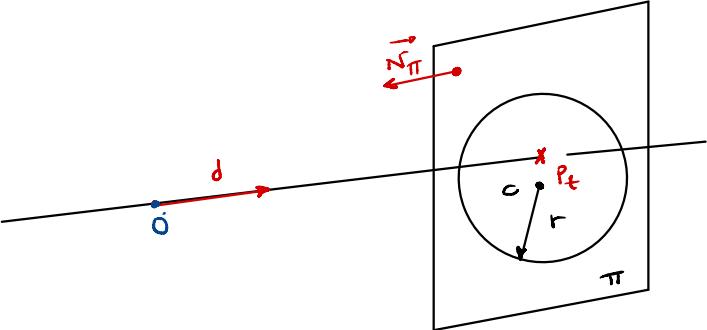
Problema 5.1.

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

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

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

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



2) Calcularemos el punto de intersección $P_t = \mathbf{o} + t\mathbf{d}$, $t \in \mathbb{R}$. Si $t < 0$, el rayo no interseca.

3) Si $t > 0$, veremos si la distancia de P_t a c es menor que r .

Para calcular el punto de intersección, el plano es el conjunto de puntos de \mathbb{A} que satisfacen $(\mathbf{c} - \mathbf{o}) \cdot \mathbf{N}_\pi = 0$. cogiendo $P_t = \mathbf{o} + t\mathbf{d}$ y sustituyendo:

$$((\mathbf{o} + t\mathbf{d}) - \mathbf{c}) \cdot \mathbf{N}_\pi = 0 \Rightarrow (\mathbf{o}^\circ + t\mathbf{d}^\circ - \mathbf{c}^\circ) \cdot \mathbf{N}_\pi^\circ + (\mathbf{o}^z + t\mathbf{d}^z - \mathbf{c}^z) \mathbf{N}_\pi^z + (\mathbf{o}^x + t\mathbf{d}^x - \mathbf{c}^x) \mathbf{N}_\pi^x = 0$$

$$\Rightarrow t = \frac{-\mathbf{N}_\pi^\circ \mathbf{o}^\circ + \mathbf{N}_\pi^z \mathbf{c}^z - \mathbf{N}_\pi^x \mathbf{o}^x + \mathbf{N}_\pi^y \mathbf{c}^y - \mathbf{N}_\pi^z \mathbf{o}^z + \mathbf{N}_\pi^x \mathbf{c}^x}{\mathbf{N}_\pi^\circ \mathbf{d}^\circ + \mathbf{N}_\pi^z \mathbf{d}^z + \mathbf{N}_\pi^x \mathbf{d}^x}$$

bool interseccionDisco($\mathbf{o}, \mathbf{d}, \mathbf{c}, r, \mathbf{n}$) {

if ($\mathbf{d} \cdot \mathbf{n} = 0$)

return false;

$$t = \frac{-\mathbf{N}_\pi^\circ \mathbf{o}^\circ + \mathbf{N}_\pi^z \mathbf{c}^z - \mathbf{N}_\pi^x \mathbf{o}^x + \mathbf{N}_\pi^y \mathbf{c}^y - \mathbf{N}_\pi^z \mathbf{o}^z + \mathbf{N}_\pi^x \mathbf{c}^x}{\mathbf{N}_\pi^\circ \mathbf{d}^\circ + \mathbf{N}_\pi^z \mathbf{d}^z + \mathbf{N}_\pi^x \mathbf{d}^x}$$

if ($t < 0$)

return false;

$$\mathbf{P}_t = \mathbf{o} + t\mathbf{d};$$

if ($\|\mathbf{P}_t - \mathbf{c}\| > r$)

return false;

return true

}

Problema 5.1. (continuación)

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

1. El rayo interseca con el plano que contiene al ~~disco~~, 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 $\mathbf{p}_t - \mathbf{c}$ es perpendicular a la normal al plano \mathbf{n} .
2. El punto \mathbf{p}_t citado arriba está dentro del disco, es decir, su distancia a \mathbf{c} es inferior al radio.

1) Siguiendo un poco la metodología de los anteriores ejercicios, la recta y el plano no intersecan sí, y solo sí, $\vec{d} \perp \vec{N}_\pi \Rightarrow$ intersecan sí, y solo si $\vec{d} \neq \vec{N}_\pi \Rightarrow \vec{d} \cdot \vec{N}_\pi = 0$

Problema 5.2.

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

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

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

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

Sea cualquier punto del rayo $\mathbf{x} = \mathbf{o} + \mathbf{d}t$. Calculamos los que tienen módulo igual a 1:

$$\mathbf{x} = (\mathbf{o}_0 + d_0 t, \mathbf{o}_1 + d_1 t, \mathbf{o}_2 + d_2 t)$$

$$\|\mathbf{x}\|^2 = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$

$$\begin{aligned} \|\mathbf{x}\|^2 = 1 &\Leftrightarrow (\mathbf{o}_0 + d_0 t)^2 + (\mathbf{o}_1 + d_1 t)^2 + (\mathbf{o}_2 + d_2 t)^2 = 1 \Leftrightarrow \\ &\Leftrightarrow \mathbf{o}_0^2 + d_0^2 t^2 + 2\mathbf{o}_0 d_0 t + \mathbf{o}_1^2 + d_1^2 t^2 + 2\mathbf{o}_1 d_1 t + \mathbf{o}_2^2 + d_2^2 t^2 + 2\mathbf{o}_2 d_2 t = 1 \Leftrightarrow \\ &\Leftrightarrow t^2 (\underbrace{\mathbf{d}^2}_{\|\mathbf{d}\|^2}) + t (\underbrace{2(\mathbf{o} \cdot \mathbf{d})}_{\mathbf{O} \cdot \mathbf{d}}) + \underbrace{(\mathbf{o}^2 + \mathbf{d}^2)}_{\|\mathbf{o}\|^2} - 1 = 0 \end{aligned}$$

Resolvemos:

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

glm:: vec3 interseccionEsferaUnidad (\mathbf{o}, \mathbf{d}) {

$$\text{discriminante} = \mathbf{O} \cdot \mathbf{d}^2 - 4\|\mathbf{d}\|^2(\|\mathbf{o}\|^2 - 1);$$

if (discriminante < 0)

return glm:: vec3 (0.0, 0.0, 0.0);

$$t_1 = \frac{-\mathbf{O} \cdot \mathbf{d} + \sqrt{\text{discriminante}}}{2\|\mathbf{d}\|^2};$$

$$t_2 = \frac{-\mathbf{O} \cdot \mathbf{d} - \sqrt{\text{discriminante}}}{2\|\mathbf{d}\|^2};$$

if ($t_1 < 0 \& t_2 < 0$)

return (0.0, 0.0, 0.0);

else if ($t_1 > 0 \& t_2 > 0$)

return $\mathbf{o} + \min(t_1, t_2) \mathbf{d};$

else if ($t_1 > 0$)

return $\mathbf{o} + t_1 \mathbf{d};$

else

return $\mathbf{o} + t_2 \mathbf{d};$

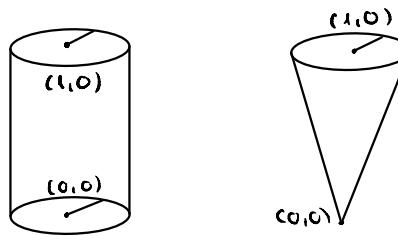
único punto donde aparece el radio. Para generalizar el algoritmo, basta con meter \rightarrow aquí el radio

Problema 5.3.

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

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

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



Cilindro

Un punto P_t está en el rayo si $P_t = \vec{O} + t\vec{d}$, $t > 0 \Rightarrow P_t = (O_0 + t d_0, O_1 + t d_1, O_2 + t d_2)$.

Por otro lado, está en el cilindro si:

$$\rightarrow (O_0 + t d_0)^2 + (O_1 + t d_1)^2 = O_0^2 + O_1^2 + 2(O_0 d_0 + O_1 d_1)t + t^2(d_0^2 + d_1^2) = 1 \quad (\text{para que esté en las aristas})$$

-) $0 \leq O_2 + t d_2 \leq 1$

Entonces, tenemos que:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \begin{cases} t_1 \\ t_2 \end{cases}$$

Si $a = 0 \Rightarrow d = (0, \pm 1, 0)$ y el rayo cortaría al cilindro si, y solo si, $O_2^2 + O_1^2 = 1$ y $O_1 < 0$. Cortaría por el punto P_t del cilindro (tomamos el primero) tal que $O_2 + t = 0 \Rightarrow t = -O_2 \Rightarrow P_t = (O_0, 0, O_2) = (O_0, O_1, O_2) - O_2(0, \pm 1, 0)$

Si $a \neq 0$, sea $\gamma_1 = O_2 + t_1 d_2$, $\gamma_2 = O_2 + t_2 d_2$

-) Si $0 \leq \gamma_1 \leq 1$, $\gamma_2 \notin [0, 1]$ $\Rightarrow P_t = \vec{O} + t_1 \vec{d}$
-) Si $0 \leq \gamma_2 \leq 1$, $\gamma_1 \notin [0, 1]$ $\Rightarrow P_t = \vec{O} + t_2 \vec{d}$
-) Si $\gamma_1, \gamma_2 \notin [0, 1]$ \Rightarrow No hay intersección
-) Si $0 \leq \gamma_1, \gamma_2 \leq 1 \Rightarrow P_t = \vec{O} + \min(t_1, t_2) \vec{d}$

Cono

Un punto P_t está en el rayo si: $P_t = \vec{O} + t\vec{d}$, $t > 0 \Rightarrow P_t = (O_0 + t d_0, O_1 + t d_1, O_2 + t d_2)$

Está en el cono si, y solo si,

$$\begin{aligned} \bullet) \quad & O_0^2 + O_1^2 + 2(O_0 d_0 + O_1 d_1)t + t^2(d_0^2 + d_1^2) = O_2^2 + t^2 d_2^2 + 2t O_2 d_2 \\ \bullet) \quad & 0 \leq O_2 + t d_2 \leq 1 \end{aligned} \quad x^2 + z^2 \leq y^2, \quad 0 \leq y \leq 1$$

Despejando la primera ecuación, tenemos que:

$$O_0^2 - O_2^2 + O_1^2 + 2(O_0 d_0 - O_2 d_2) t + t^2(d_0^2 - d_2^2 + d_1^2) = 0$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \begin{cases} t_1 \\ t_2 \end{cases}$$

Si $a = 0 \Rightarrow t_0 = \frac{O_2^2 - O_0^2 - O_1^2}{2(O_0 d_0 - O_2 d_2)}$. Si $0 \leq O_2 + t_0 d_2 \leq 1$, return $P_{t_0} = \vec{O} + t_0 \vec{d}$. En otro caso, no hay intersección.

Si $a \neq 0$, sea $\gamma_1 = O_2 + t_1 d_2$, $\gamma_2 = O_2 + t_2 d_2$.

-) Si $0 \leq \gamma_1 \leq 1$, $\gamma_2 > 1$, $P_t = \vec{O} + t_1 \vec{d}$
-) Si $0 \leq \gamma_2 \leq 1$, $\gamma_1 > 1$, $P_t = \vec{O} + t_2 \vec{d}$
-) Si $\gamma_1, \gamma_2 > 1$, no hay intersección
-) Si $0 \leq \gamma_1, \gamma_2 \leq 1$, return $P_t = \vec{O} + \min(t_1, t_2) \vec{d}$