

Problema 1.1.

Escribe el código que genera una tabla de coordenadas de posición de vértices con las coordenadas de los vértices de un polígono regular de n lados o aristas (es una constante del programa), con centro en el origen y radio unidad.

Los vértices se almacenan como flotantes de doble precisión (**double**), con 2 coordenadas por vértice. Escribe el código que crea el correspondiente VAO a esta secuencia de vértices.

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

Problema 1.1. (continuación)

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

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

- (a) tipo de primitiva **GL_LINE_LOOP**.
- (b) tipo de primitiva **GL_LINES**.

a) Tipos de primitiva **GL_LINES_LOOP**:

```
static GLenum nombre_vao=0; //variable estática o global inicializada a 0
const unsigned int n=5;
std::vector<Tupla2d> posiciones;
for(int i=0;i<n;i++){
    posiciones.push_back ({cos((2*M_PI*i)/n),sin((2*M_PI*i)/n)});
}
if (nombre_vao==0){
    nombre_vao = CrearVAO();
    CrearVBOAtrib(ind_atrib_posiciones,GL_DOUBLE,2, posiciones.size(),posiciones.data());
}
else{
    glBindVertexArray(nombre_vao);
}
```

b) Tipos de primitiva GL_LINES:

```
static GLenum nombre_vao=0; // variable estatica o global inicializada a 0
const unsigned int n=5;
std::vector<Tupla2d> posiciones;
for(unsigned int i=0;i<n;i++){
    posiciones.push_back ({cos((2*M_PI*i)/n), sin((2*M_PI*i)/n)});
    posiciones.push_back ({cos((2*M_PI*(i+1))/n), sin((2*M_PI*(i+1))/n)});
}
if (nombre_vao==0){
    nombre_vao = crearVAO();
    crearVBOAttrib(ind_atrib_posiciones, GL_DOUBLE, 2, posiciones.size(), posiciones.data());
}
else{
    glBindVertexArray(nombre_vao);
}
```

Problema 1.2.

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

a) Secuencia no indexada:

```
static GLenum nombre_vao=0; //variable estatica o global inicializada a 0
const unsigned int n=5;
std::vector<Tupla3f> posiciones;
for(unsigned int i=0;i<n;i++){
    posiciones.push_back({cos((2*M_PI*i)/n), sin((2*M_PI*i)/n), 0.0});
    posiciones.push_back({cos((2*M_PI*(i+1))/n), sin((2*M_PI*(i+1))/n), 0.0});
    posiciones.push_back({0.0,0.0,0.0});
}
if( nombre_vao==0){
    nombre_vao = crearVAO();
    crearVBOAttrib(ind_atrib_posiciones, posiciones);
}
else{
    glBindVertexArray(nombre_vao);
}
```

b) Secuencia indexada:

```
static GLenum nombre_vao=0; //variable estatica o global inicializada a 0
const unsigned int n=5;
std::vector<Tupla3f> posiciones;
std::vector<Tupla3u> indices;
for(unsigned int i=0;i<n;i++){
    posiciones.push_back({cos((2*M_PI*i)/n), sin((2*M_PI*i)/n), 0.0});
}
posiciones.push_back({0.0,0.0,0.0});

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

if(nombre_vao==0){
    nombre_vao = crearVAO();
    crearVBOAtrib(ind_atrib_posiciones, posiciones);
    crearVBONd(indices);
}
else{
    glBindVertexArray(nombre_vao);
}
```

Problema 1.3.

Crea una copia del repositorio `opengl3-minimo`, y modifica el código de ese repositorio para incluir una nueva función para visualizar las aristas y el relleno de polígono regular de n lados (donde n es una constante de tu programa), usando las tablas, VBOs y VAOs de coordenadas que codifican dicho polígono regular, según se describe en:

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

(el enunciado continua en la siguiente transparencia)

Problema 1.3. (continuación)

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

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

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

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).

Problema 1.5.

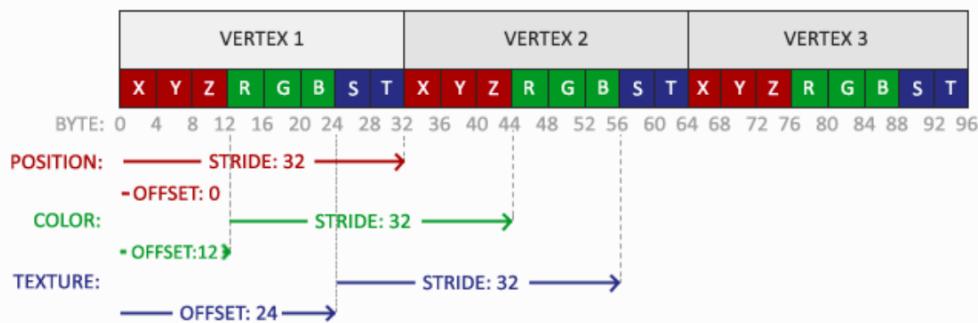
Repite el problema anterior (problema 1.4), pero ahora el relleno, en lugar de hacerse todo del mismo color plano, se hará mediante interpolación de colores (las aristas siguen estando con un único color). Para eso se debe generar una tabla de colores de vértices, inicializada con colores aleatorios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

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

Problema 1.6.

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

La estructura completa se debe alojar en un único VBO de atributos con todos los flotantes de las posiciones y colores, así que no uses las funciones dadas para creación de VAOs y VBOs (asumen una tabla por VBO con estructura SOA).



Problema 1.7.

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

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

Problema 1.8.

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

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

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

Problema 1.9.

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

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

(*aquí hay que tener más cuidado porque tenemos la propiedad anticommutativa, no commutativa*)

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

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

Problema 1.10.

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

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

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

por el anterior ejercicio: $\vec{a} \times \vec{b} = ((a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x))^t$.

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

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

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

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