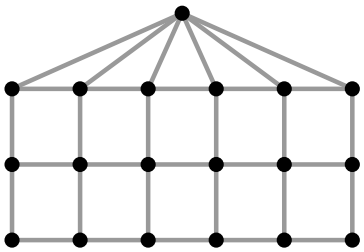
Informática Gráfica (2024-25) GIM+GIADE.

Relación de problemas (Febr. 2025) (con soluciones).

Problema 1



Queremos visualizar una figura como la de la derecha, formada por una rejilla de vértices y un vértice adicional sobre esa rejilla. Para ello vamos a usar una secuencia de vértices y atributos. Todos los vértices están en el plano con $z = 0$. En la rejilla hay n columnas de vértices y m filas (con $n, m > 1$, estos dos valores se suponen que ya están declarados como dos constantes enteras **`n`** y **`m`**). Cada dos columnas están separadas por a unidades de distancia en horizontal. Cada dos filas de pixels están separadas por b unidades de distancia en vertical (támbien hay una distancia en vertical de b unidades entre el vértice superior y la fila de vértices de arriba). Suponemos que los valores de a y b ya están declarados como dos constantes flotantes, llamadas **`a`** y **`b`**.

Dibujaremos las aristas con el tipo de primitiva *segmentos*, todas ellas con color gris (componentes R,G y B a $1/2$) y los puntos en los vértices con el tipo de primitiva *puntos*, cada vértice con un color aleatorio. En la figura, a modo de ejemplo, se usa $n = 6$ y $m = 3$. Los vértices se ha dibujado todos en negro para que se vean mejor.

Escribe el código con las declaraciones de las tablas necesarias (como variables globales de una aplicación), así como el de una función que inicializa dichas tablas (**no escribas ningún código OpenGL para generar VAOs, VBOs, ni para visualización**). Esta función no hará absolutamente nada si las tablas ya estaban inicializadas. Asume que la función **`rnd()`** devuelve un **`float`** aleatorio entre 0 y 1. Las tablas a generar son todas ellas de tipo **`std::vector`** y con vectores de GLM en las entradas, a saber:

- **`vertices`**: tabla con las coordenadas de la posición de los vértices (sin que se repita ninguno), cada entrada es de tipo **`dvec2`** (cada entrada son dos flotantes de **doble precisión**, es decir, de tipo **`double`** de C/C++),
- **`aristas`**: tabla de aristas, cada entrada representa una arista como una tupla **`uvec2`**, que tiene dos **enteros sin signo** (tipo **`unsigned int`** de C/C++), con los dos índices de los dos vértices en los extremos de la arista.
- **`colores`** : tabla de colores de vértices, donde cada entrada es una tupla **`vec3`** con un color RGB aleatorio.

Solución:

```
// declaraciones de las tablas
vector<dvec2> vertices; vector<vec3> colores; vector<uvec2> aristas;

// función que las inicializa
void InicializarTablas()
{
    // Generar todos los vértices (posiciones y colores) y las aristas de la rejilla
    for( unsigned ix = 0 ; ix < n ; ix++ )  // n columnas
    for( unsigned iy = 0 ; iy < m ; iy++ )  // m filas
    {
      // Insertar el vértice en fila iy y columna ix, y su color
      vertices.push_back( dvec2(  a*ix, b*iy ));
      colores.push_back( vec3( rnd(), rnd(), rnd() ));

      // Insertar aristas adyacentes a este vértice
      const unsigned iv = ix*m+iy ; // índice de este vértice
      if ( ix < n-1 ) aristas.push_back( uvec2( iv, iv+m )) ; // arista horiz. (derecha)
      if ( iy < m-1 ) aristas.push_back( uvec2( iv, iv+1 )) ; // arista vertical (arriba)
    }
    // Generar el vértice superior (con índice n*m)
    vertices.push_back( dvec2( a*(n-1)/2, b*m ));
    colores.push_back( vec3( rnd(), rnd(), rnd() ));

    // Generar las aristas adyacentes al vértice superior
    for( unsigned ix = 0 ; ix < n ; ix++ )
        aristas.push_back( uvec2( ix*m +(m-1), n*m ));
}
```

Problema 2

Escribe el código que sirva para visualizar la figura del ejercicio anterior. Dicho código puede usar las declaraciones de las tablas del ejercicio 1, y las constantes (ya definidas) **`n`** y **`m`**. Usa exclusivamente llamadas a funciones OpenGL (**no** uses las clases para descriptores de VBOs o de VAOs, ni la clase **`Cauce`**). El código estará compuesto de:

- La declaración de una variable global para el *nombre* o *identificador* del VAO.
- Una función para generar e inicializar un VAO nuevo adecuado para visualizar la figura. Si el VAO ya estaba creado en el momento de la llamada, no hace absolutamente nada. En otro caso, llama a la función que inicializa las tablas (del ejercicio anterior), y luego crea e inicializa el VAO.
- Una función para visualizar la figura, para ello llama en primer lugar a la función anterior, y luego visualiza las primitivas que componen la figura, cuya apariencia se explica en el enunciado del primer ejercicio.

Suponer que que el atributo de vértice *color* tiene asociado el índice de atributo 1, como hemos supuesto en las clases.

Solución:

Hay que tener en cuenta que los pasos y llamadas para crear un VAO y cada VBO (de atributos y de índices) están en las transparencias de la asignatura. Lo relevante de este tipo de problemas es saber adaptar esas llamadas a cada caso particular, en dos aspectos: por un lado saber que VBOs hacen falta o no hacen falta, si es una secuencia indexada o no, y para cada VBO, los tipos de datos de la tabla y la forma en la que están organizados.

Esquema de la función para crear e inicializar el VAO y sus VBOs:

```
GLuint nombre_vao = 0 ; // Identificador del VAO
void CrearInicializarVAO()
{ // Inicialización y declaraciones ....
  // Crea y activa VAO  ....
  // Crea e inicializar VBO de posiciones ...
  // Crea e inicializa VBO de colores ....
  // Crea VBO de índices (queda binded para DrawElements), unbind del VAO ...
}
```

Código de cada uno de los pasos:

```
// Inicialización y declaraciones:
if ( nombre_vao > 0 ) return ; // si el VAO ya está creado no hace nada
InicializarTablas() ;          // nos aseguramos de que las tablas están creadas.
const unsigned nv = n*m+1 ;    // número total de verts. (tmb puede ser vertices.size())
const unsigned na = aristas.size() ; // número total de aristas
GLuint buffer_pos = 0, buffer_col = 0, buffer_ind = 0 ; // idents de buffers
```

```
// Crea y activa VAO
glGenVertexArrays( 1, &nombre_vao );
glBindVertexArray( nombre_vao );
```

```
// Crea e inicializar VBO de posiciones (teniendo en cuenta que cada posición es un dvec2)
glGenBuffers( 1, &buffer_pos );
glBindBuffer( GL_ARRAY_BUFFER, buffer_pos );
glBufferData( GL_ARRAY_BUFFER, nv*2*sizeof(double), vertices.data(), GL_STATIC_DRAW );
glVertexAttribPointer( 0, 2, GL_DOUBLE, GL_FALSE, 0, 0 );
glEnableVertexAttribArray( 0 );
```

```
// Crea e inicializa VBO de colores (cada color es un vec3), unbind de VBOs de atribs.
glGenBuffers( 1, &buffer_col );
glBindBuffer( GL_ARRAY_BUFFER, buffer_col );
glBufferData( GL_ARRAY_BUFFER, nv*3*sizeof(float), colores.data(), GL_STATIC_DRAW );
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, 0, 0 );
glEnableVertexAttribArray( 1 ); // no estrictamente necesario, se hace después
glBindBuffer( GL_ARRAY_BUFFER, 0 ); // unbind de VBOs de atributos (no esencial en este problema)
```

```
  // Crea VBO de índices (cada arista en un uvec2), unbind del VAO
  glGenBuffers( 1, &buffer_ind );
  glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, buffer_ind );
  glBufferData( GL_ELEMENT_ARRAY_BUFFER, na*2*sizeof(unsigned), aristas.data(), GL_STATIC_DRAW );
  glBindVertexArray( 0 ); // no es esencial en este problema
```

Código de la función de visualización:

```
void VisualizarVAO()
{
  const unsigned nv = n*m+1 ;           // número total de verts. (=vertices.size())
  const unsigned na = aristas.size() ;  // número total de aristas

  // Crear el VAO (si no lo estaba ya), y activarlo.
  CrearInicializarVAO();                // asegurarse de que el VAO está creado
  glBindVertexArray( nombre_vao );      // activar el VAO

  // Visualizar (en 1er lugar) las aristas (2 ptos- junto con el glBind anterior)
  glDisableVertexAttribArray( 1 );      // deshabilitar uso de la tabla de colores
  glVertexAttrib3f( 1, 1.0, 1.0, 1.0 ); // fijar color gris
  glDrawElements( GL_LINES, na*2, GL_UNSIGNED_INT, 0 ); // visualizar segmentos (indexado)

  // Visualizar (en 2o lugar) los puntos (1 pto)
  glEnableVertexAttribArray( 1 );       // habilitar uso de la tabla de colores
  glDrawArrays( GL_POINTS, 0, nv );     // visualizar puntos (no indexado)

  // Desactivar VAO (no estrictamente necesario en el examen)
  glBindVertexArray( 0 );
}
```

Problema 3

Supón que tienes una malla indexada de triángulos almacenada, como es habitual, en un vector (**`std::vector`**) de **`vec3`** con las posiciones de los vértices, y otro vector de **`uvec3`** con los triángulos (ninguno vacío). **Escribe una función C/C++** que determine si las áreas de los triángulos son *desiguales*, y en ese caso divida un triángulo de área máxima en dos triángulos de igual área. La función tiene como parámetros ambos objetos **vector** (por referencia) y se encarga de actualizarlos. Suponer que no hay triángulos con área nula.

Consideramos las áreas de los triángulos *desiguales* si un triángulo de área máxima tiene el doble o más de área que un triángulo de área mínima. Para dividir un triángulo en dos, insertamos un vértice nuevo en mitad de una de sus aristas (una que tenga longitud máxima de las tres), y lo sustituimos por dos triángulos que comparten una arista adyacente al nuevo vértice y al opuesto en el triángulo original (si se hace la división, al final, la malla tendrá un vértice más y un triángulo más que la malla original).

Se valorará la eficiencia y simplicidad del código, para ello se deben usar las funciones y operadores de vectores de la librería GLM. Para dividir el triángulo original en dos, se puede actualizar ese triángulo en la tabla e insertar otro (no requiere eliminar triángulos de la tabla). Se valorará por separado: (a) calcular la ratio de áreas y el triángulo con área máxima, (b) encontrar la arista más larga de un triángulo, y (c) dividir un triángulo.

Solución:

El área A del triángulo con vértices $(\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2)$ se puede obtener como la mitad de la longitud del producto vectorial de dos vectores en sus aristas:

$$A \;=\; \frac{1}{2}\,\|(\boldsymbol{v}_1-\boldsymbol{v}_0)\times(\boldsymbol{v}_2-\boldsymbol{v}_0)\|,$$

esa es la fórmula que hay en las transparencias. También se puede usar la fórmula de Herón:

$$A \;=\; \sqrt{s(s-a)(s-b)(s-c)},$$

donde s es $(a+b+c)/2$, y a, b y c son las longitudes de los lados, es decir:

$$a \;=\; \|\boldsymbol{v}_1-\boldsymbol{v}_0\| \qquad b \;=\; \|\boldsymbol{v}_2-\boldsymbol{v}_1\| \qquad c \;=\; \|\boldsymbol{v}_0-\boldsymbol{v}_2\|$$

Declaración y esquema de la función:

```
void DividirTrianguloMayor( vector<vec3> & ver, vector<uvec3> & tri )
{
  // (a) Calcular áreas mínima y máxima, e índice de tri. con área máxima. Si es necesario, acabar.
  // (b) Encontrar los indices de los vérts en la arista con mayor longitud, y el del opuesto
  // (c) Actualiza la malla (actualizar un triángulo y añadir otro)
}
```

A continuación aparece el código de cada uno de esos pasos:

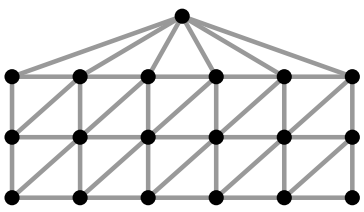
```
// (a) Calcular áreas mínima y máxima, e índice de tri. con área máxima
unsigned it_max = 0 ; // índice del triángulo con área máxima
float at_max = 0, at_min = 0 ; // áreas máxima y mínima de los triángulos

for( unsigned it = 0 ; it < tri.size() ; it++ ) // recorre todos los triángulos
{
  const uvec3 & t  = tri[it] ; // triángulo a procesar
  const float   at = 0.5*length( cross( ver[t[1]]-ver[t[0]], ver[t[2]]-ver[t[0]])); // área tri.
  if ( at > at_max || it == 0 ) { at_max = at ; it_max = it ; } // actualizar máximo
  if ( at < at_min || it == 0 ) at_min = at ;                   // actualizar mínimo
}
// si la ratio de áreas es menor que 2, no hace nada
if ( at_max/at_min < 2.0 ) return ;
```

```
// (b) Encontrar los indices de los vérts en la arista con mayor longitud, y el del opuesto
const uvec3 &   t    = tri[it_max] ; // t == triángulo con área mayor
const float     l0   = length( ver[t[0]]-ver[t[1]] ), // long. arista 0 -> 1
                l1   = length( ver[t[1]]-ver[t[2]] ), // long. arista 1 -> 2
                l2   = length( ver[t[2]]-ver[t[0]] ), // long. arista 2 -> 0
                lmax = glm::max( l0, glm::max( l1, l2 ) ) ;     // longitud de la arista mayor
const unsigned  nvam = (lmax == l0) ? 0 : (lmax == l1) ? 1 : 2, // nú. 1er vért. ady. arist. may.
                ivm0 = t[nvam],       // índice de 1er vértice en la arista mayor
                ivm1 = t[(nvam+1)%3], // índice de 2o vértice en la arista mayor
                ivop = t[(nvam+2)%3]; // índice de vértice opuesto
```

```
// (c) Actualiza la malla (actualizar un triángulo y añadir otro)
const unsigned ivnu = ver.size();             // calcula índice del nuevo vértice
ver.push_back( 0.5f*(ver[ivm0] + ver[ivm1])); // inserta nuevo vértice
tri[it_max]  = uvec3( ivm0, ivnu, ivop ) ;    // modifica el triángulo original
tri.push_back( uvec3( ivnu, ivm1, ivop ) ) ;  // añade nuevo triángulo
```

Problema 4

Describe razonadamente cuanta memoria (en bytes) es necesaria para almacenar las tablas para visualizar la figura 2D de aquí abajo, dando la solución como una expresión entera escrita en función de las constantes n y m (que son el número de columnas de vértices y el número de filas de vértices en la rejilla, similar al primer ejercicio). Puedes suponer que los flotantes son de tipo **`float`** (ocupan 4 bytes por valor) y los enteros **`unsigned int`** (también de 4 bytes). Diseñamos las tablas para que los vértices se vean de colores aleatorios. Las coordenadas son 2D (usan 2 **`float`** por tupla), y los colores RGB (3 **`float`** por color). Únicamente consideramos la memoria usada por los valores **`float`** y **`unsigned`** guardados en las tablas, no consideramos la memoria necesaria para punteros o para posibles metadatos de dichas tablas. Considera cada uno de estos supuestos:



(A) Usando las tablas de vértices (únicos), colores y aristas, tal y como se describe en el ejercicio 1, pero ahora también con las aristas en diagonal. Recuerda que las tablas están diseñadas para visualizar los vértices con primitivas de tipo *puntos* y las aristas como primitivas de tipo *segmentos*.

(B) Igual que en el caso anterior, pero sin usar la tabla de aristas, es decir, replicando vértices (posiciones y colores) cuando sea necesario. Los tipos de primitvas serán *puntos* y *segmentos*, al igual que antes.

(C) Usando una *malla indexada*, es decir, una tabla de vértices (únicos), otra de colores, y otra de triángulos. Las aristas se visualizarían con primitivas de tipo *triangulo* (en modo de visualización *no relleno*), y los vértices como *puntos*.

(D) Usando como tipo de primitva una *tira de triángulos* para cada fila de triángulos, y un *abanico de triángulos* para los triángulos adyacentes al vértice de arriba. Por tanto, tampoco hay ahora tabla de aristas. Para que se vean las aristas se visualizarían las tiras y el abanico en modo no relleno. Los vértices sí se visualizan usando como tipo de primitva *puntos*.

A la vista de los resultados, ordena las opciones de menor a mayor cantidad de memoria requerida, suponiendo que n y m son iguales y muy grandes.

Solución:

Resumen de la memoria usada, y comparativa

(solo se pedía la comparativa de la derecha, pero incluyo a la izquierda las cantidades de memoria, que se justifican por los razonamientos que hay más abajo)

Memoria en bytes:	**Comparativa**
(A) $44nm - 8n - 16m + 20$ (B) $120nm - 40n - 80m$ (C) $44nm - 12n - 24m + 32$ (D) $40nm + 20n + 20$ **Para el caso $n = m$** (A) $44n^2 - 24n + 20$ (B) $120n^2 - 120n$ (C) $44n^2 - 36n + 32$ (D) $40n^2 + 20n + 20$	Dividimos todos las cantidades, para compararlas, entre un mismo valor, el valor $4n^2 > 0$: (A) $11 - 8/n + 5/n^2$ (B) $30 - 30/n - 30/n^2$ (C) $11 - 9/n + 32/n^2$ (D) $10 + 5/n + 20/n^2$ Que n sea grande implica que se puede asumir que $1/n^2$ es nulo, también se puede asumir que $1/n$ es nulo (excepto para distinguir entre (A) y (C), ya que eso nos permite saber que (C) usará siempre menos memoria que (A)). Por tanto, el orden de menos memoria a más será: **(D) < (C) < (A) < (B)**.

En todos los casos:

- Cada vértice necesita 2 valores para la posición y 3 para el color, es decir, 5 **`float`** o 20 bytes, en los 4 casos se usa esto.
- Cada arista con dos índices enteros necesita 8 bytes (para el caso (A)).
- Cada triángulo necesita 3 enteros, es decir, 12 bytes (para el caso (C))

A continuación se deriva la cantidad en bytes para cada caso:

Caso (A) Total: $\boxed{44nm - 8n - 16m + 20 \quad \text{bytes.}}$ Justificación:

Hay vértices y aristas de dos enteros (es indexado).

- Tabla de **vértices** (y de colores): hay $nm + 1$ vértices, cada posición y color son 20 bytes, en total, ocupan $20nm + 20$ bytes.
- Tabla de **aristas** (con dos enteros cada una): en la rejilla hay $3(n-1)(m-1)$ aristas, sin contar con las de la fila de arriba ni la columna de la derecha. Además, hay $n + m - 1$ aristas adicionales en esas fila y columna, y finalmente n más adyacentes al vértice de arriba. En total son $3(n-1)(m-1) + (n + m - 1) + n$ aristas, es decir: $3nm - n - 2m$, que ocupan $24nm - 8n - 16m$ bytes.

Caso (B) Total: $\boxed{120nm - 40n - 80m \quad \text{bytes.}}$ Justificación:

Únicamente hay una tabla de vértices. Por cada arista del caso anterior, ahora hay dos vértices (no es indexado).

- Tabla de **vertices** (y de colores): hemos visto que en el caso anterior (A) son necesarias $3nm - n - 2m$ aristas, es decir que en este caso (B) necesitamos $6nm - 2n - 4m$ vértices.

Caso (C) Total: $\boxed{44nm - 12n - 24m + 32 \quad \text{bytes.}}$ Justificación:

Es una malla indexada de vértices y triángulos (tres enteros).

- Tabla de **vértices** (y de colores): hay $nm + 1$, ocupan $20nm + 20$ bytes.
- Tabla de **triángulos**: hay $2(n-1)(m-1)$ triángulos en la rejilla, es decir $2nm - 2n - 2m + 2$. Se le suman los $n - 1$ de arriba y tenemos $2nm - n - 2m + 1$ triángulos. Cada uno son 12 bytes, en total $24nm - 12n - 24m + 12$ bytes.

Caso (D) Total: $\boxed{40nm + 20n + 20 \quad \text{bytes.}}$ Justificación:

Hay tiras de triángulos y un abanico de triángulos. Cada tira y el abanico está formado de vértices (no indexado).

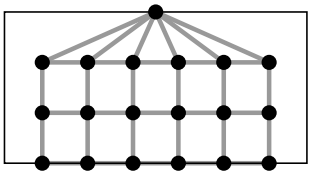
- **Tiras de triángulos**: hay m tiras, y cada una tiene $2n$ vértices, eso hace $2nm$ vértices en total.
- **Abanico de triángulos**: hay uno, con $n + 1$ vértices.

Por lo cual la memoria necesaria es:

- Tabla de **vértices** (y de colores): sumando ambas cantidades tenemos $2nm + n + 1$ vértices.

Problema 5

Queremos visualizar la figura del primer ejercicio en un viewport que tiene el doble de columnas de pixels que de filas (es el doble de ancho que de alto). Queremos que la figura aparezca centrada en el viewport, sin deformarse, y lo más grande posible, pero sin que se recorte ningún vértice. La proyección es ortográfica. A la derecha se ve la figura, tal y como quedaría inscrita en el viewport (los vértices de la fila de abajo y el de arriba siempre estarán en las aristas inferior y superior del viewport).



Describe razondamente cuales deben ser las coordenadas del punto de atención $\vec{a}$, el vector $\vec{u}$ (VUP), y el vector $\vec{n}$, (para la **matriz de vista**) así como los valores l, r, b, t, n y f para la **matriz de proyección**. Tu respuesta estará en términos de las constantes n, m, a y b.

Escribe el código C/C++ necesario para calcular cada una de esas dos matrices, suponiendo que puedes usar las constantes **`n`**, **`m`**, **`a`** y **`b`**, ya declaradas. Escribe dos versiones del código, en la la primera (a) usa las funciones de GLM para crear las matrices de vista y proyección, es decir, las funciones **`lookAt(...)`** y **`ortho(l,r,b,t,n,f)`**, y en la segunda (b) usa únicamente las funciones de GLM para crear matrices de escalado (**`scale`**), traslación (**`translate`**) y rotación (**`rotate`**), de forma que las matrices de vista y proyección se obtengan por composición de esas matrices.

Solución:

Suposiciones (todo esto no se pide, pero lo incluyo aquí para entender mejor el ejercicio)

Se puede hacer de diversas formas, lo importante es tener en cuenta que se cumplan los requisitos y que se tengan en cuenta como se generan los vértices en el ejercicio 1 del estudiante. Lo más frecuente (y simple) ha sido poner como coordenadas en el bucle (ai, bj) (con i desde 0 hasta $n-1$, y j desde 0 hasta $m-1$). Por tanto el vértice inferior izquierdo está en el origen $(0,0,0)$ de coordenadas del mundo, el vértice superior derecho de la rejilla está en $(a(n-1), b(m-1), 0)$, y que el vértice de arriba esté en $(a(n-1)/2, bm, 0)$.

Con esas suposiciones la caja englobante de todos los vértices va en X desde 0 hasta el valor w, con $w = a(n-1)$ y en Y desde 0 hasta h, con $h = bm$.

Como es una vista 2D, el *view frustum* es un *ortoedro*, que debe cumplir estos dos requisitos:

- Su centro (en coordenadas de mundo) debe estar en el centro de la figura, es decir, en el punto de atención (que es el punto que se va a proyectar en el centro del viewport). Ese centro, por tanto, tiene que tener coordenada Z igual a 0.
- Sus proporciones en X e Y deben coincidir con las del viewport, es decir, debe ser el doble de ancho que de alto.
- Su extensión en Z puede ser cualquiera, pero debe incluir el plano $z = 0$, ya que todos los vértices tienen coordenada Z 0.

Así que las matrices de vista y proyección deben cumplir estos requisitos:

- La matriz de vista debe ser una traslación que lleve el punto central de la figura (en coords. de mundo) al origen (en coords. de vista).
- La matriz de proyección debe ser un escalado que escale las dimensiones de ese ortoedro (en coordenadas de vista), para convertirlo en un cubo de lado 2 (en coordenadas de mundo). Se considera el caso del enunciado, es decir, se escala de forma que la altura de la caja englobante se proyecte exactamente sobre la altura del viewport, es decir, ajuste en vertical. De esta forma, la matriz de proyección debe ser un escalado uniforme en X y en Y.

Matriz de Vista

- El punto de atención $\boldsymbol{a}$ (**`a`**) debe estar en el centro de la figura, es decir, en $\boldsymbol{a} = (w/2, h/2, 0)$. Sustituyendo w y h, obtenemos:

$$\boldsymbol{a} = \left(\frac{a(n-1)}{2}, \frac{bm}{2}, 0 \right)$$

- El vector $\boldsymbol{u}$ (VUP) (**`u`**) debe ser el versor Y, es decir, $\boldsymbol{u} = (0,1,0)$.
- El vector $\boldsymbol{n}$ (VPN) **`n`** debe ser el versor Z (perpendicular al plano $z=0$) es decir, $\boldsymbol{n} = (0,0,1)$.
- La matriz de vista con **`lookAt`** se haría así: **`lookAt(a+n,a,u)`** (el primer parámetro es la posición del observador, $\boldsymbol{a} + \boldsymbol{n}$):

```
const vec3 a = vec3( a*(n-1)/2.f, b*m/2.f, 0.f );
const vec3 n = vec3( 0.0f, 0.0f, 1.0f );
const vec3 u = vec3( 0.0f, 1.0f, 0.0f );
glm::lookAt( vec3( a+n, a, u ));
```

- La matriz de vista con matrices elementales es una simple traslación por el vector $-\boldsymbol{a}$, es decir, se construye así (se puede hacer en una sola línea, sin declarar **`a`**):

```
const vec3 a = vec3( a*(n-1)/2.f, b*m/2.f, 0.f ); // igual que antes
glm::translate( -a );
```

Matriz de Proyección (ajuste en vertical)

- Los valores b y t deben ser las distancias con signo, en coordenadas de mundo, desde el centro de la figura hasta el borde inferior (b) y el superior (t). Como el alto del ortoedro es h, obtenemos $b = -h/2$ y $t = +h/2$, sustituyendo h:

$$b = -\frac{bm}{2} \qquad t = +\frac{bm}{2}$$

- Los valores l y r deben ser el doble de b y t, ya que deben preservar la proporción del viewport. Por tanto, $l = 2b$ y $r = 2t$. Sustituyendo b y t tenemos:

$$l = -bm \qquad r = +bm$$

- Los valores n y f deben ser -1 y $+1$ (o cualquier otro par de valores, el primero < 0 y el segundo > 0), pero esto es lo más simple.
- La matriz de proyección es literalmente **`ortho(l,r,b,t,n,f)`**, es correcto sustituir $l, r, \ldots$ por sus expresiones, en código es:

```
glm::ortho( -b*m, +b*m, -b*m/2.f, +b*m/2.f, -1.0f,1.0f );
```

- Respecto al uso de matrices elementales para la proyección, debemos de escalar el alto de la figura, que es h, hasta 2 unidades en vertical (entre -1 y $+1$ en Y en NDC). Por tanto, el factor de escala es $2/h$. La matriz debe escalar uniformemente en X e Y, para que no se deforme la figura. El factor de Z da igual pues esa coordenada es 0, así que dejamos un 1. La matriz de proyección sería: simplemente:

```
const float fac = 2.0f/(b*m);
glm::scale( vec3( fac, fac, 1.0f ) );
```