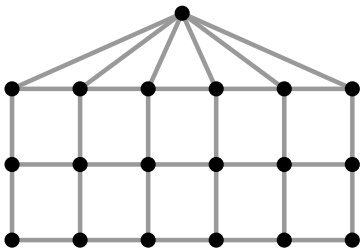
Informática Gráfica (2024-25) GIM+GIADE.

Relación de problemas (Febr. 2025) .

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.

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.

Problema 3

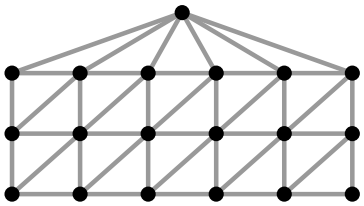
Supón que tienes una malla indexada de triángulos almacenada, como es habitual, en un vector (**`std::vector`**) de **`vec3`** con las posiciones de los vértices, y otro vector de **`uvec3`** con los triángulos (ninguno vacío). **Escribe una función C/C++** que determine si las áreas de los triángulos 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.

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.

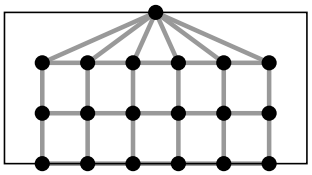
(C) Usando una *malla indexada*, es decir, una tabla de vértices (únicos), otra de colores, y otra de triángulos. Las aristas se visualizarían con primitivas de tipo *triangulo* (en modo de visualización *no relleno*), y los vértices como *puntos*.

(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.

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.