UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

INGENIERÍA EN COMPUTACIÓN

LABORATORIO DE COMPUTACIÓN GRÁFICA e INTERACCIÓN HUMANO COMPUTADORA

PREVIO Nº 01

NOMBRE COMPLETO: Hernández Vázquez Daniela

Nº de Cuenta: 318092867

GRUPO DE LABORATORIO: 01

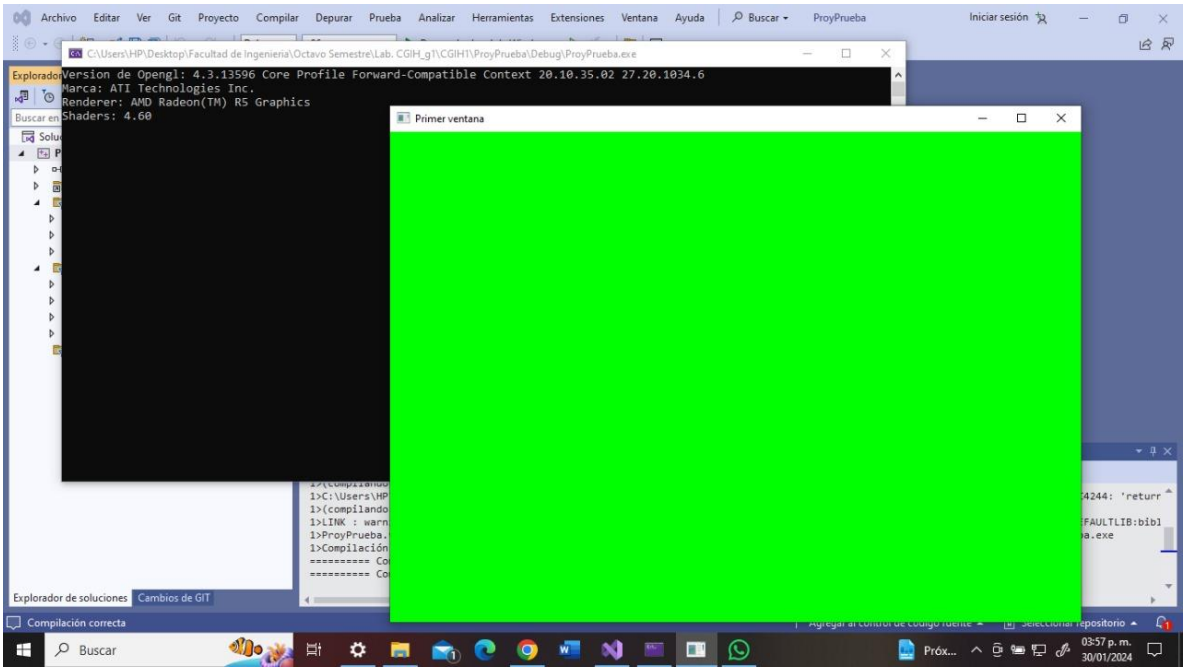
GRUPO DE TEORÍA: 02

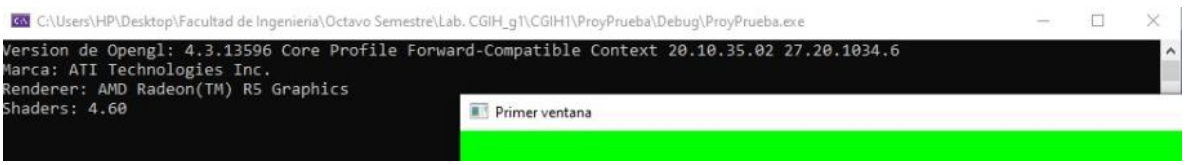
SEMESTRE 2024-2

FECHA DE ENTREGA LÍMITE: 10 de febrero de 2024

CALIFICACIÓN: _____

1:- Captura de pantalla como la del manual de configuración en la cual se muestra la ventana de fondo verde y la información de la consola con los datos de Hardware de su equipo de cómputo.





2:- ¿Qué es un VAO?

Un Vertex Array Object (VAO) es una estructura de datos que almacena configuraciones de estado relacionadas con la organización y los formatos de los datos de vértices durante la etapa de renderizado.

Una vez creados los Vertex Buffer Object (VBO's) y los datos de los vértices son almacenados se procede a crear los VAO; estos definen el mapeo entre los atributos (vinculados a los VBO's) y los nombres de las entradas del shader, en este caso los VAO proporcionan información de entrada al Vertex_shader.

Algunos comandos relacionados a los VAO son:

```
void glGenVertexArrays(GLsizei n, GLuint *vaoIDs);        // Crea n VAO's y almacena sus ID's en el array
void glDeleteVertexArrays(GLsizei n, const GLuint *vaoIDs); // Elimina VAO's referenciados del array
GLboolean glIsVertexArray(GLuint vaoID);                  // Verifica si un identificador corresponde a un VAO
void glBindVertexArray(GLuint vaoID);                     // Activa un determinado VAO
```

3.- ¿Qué es un VBO?

Un Vertex Buffer Object (VBO) es un objeto utilizado para almacenar información sobre los vértices para todas las primitivas que se quieran desplegar, desde la descripción de las coordenadas de los vértices hasta el color asociado con cada uno de ellos.

Los VBO's pueden almacenar la información de un vértice en diversas maneras como arreglo de estructuras (AoS) o estructuras de arreglos (SoA).

Los datos en un VBO tienen que ser contigua en memoria, pero no necesariamente "ajustada", es decir, los datos de un mismo tipo pueden estar separados por un número de bytes constante entre datos del mismo tipo.

Ej.- Un VBO puede almacenar en una posición de vértice (x, y, z), color (r, g, b) y vector normal (n_x, n_y, n_z) de todo un objeto 3D quedando como $xyz\ rgb\ n_x n_y n_z$ $xyz\ rgb\ n_x n_y n_z$ ... en memoria del servidor.

Dado que numerosos VBO pueden estar asociados con un objeto 3D, y una escena puede contener diversos objetos el organizar los datos resulta bastante complejo. Una estructura que permite simplificar el manejo de estos son los VAO's.

La forma de componer los vértices y sus atributos en objetos 3D es a través de las primitivas que OpenGL puede desplegar: puntos, líneas y triángulos.
Algunos comandos relacionados con los VBO son:

```
void glGenBuffers (GLsizei n, GLuint * vboIDs);        // Crea n VBOS y almacena sus identificadores en el array
void glDeleteBuffers (GLsizei n, const GLuint *vboIDs); // Elimina n VBOs relacionados en los elementos del array
GLboolean glIsBuffer (GLuint vboID);                   // Verifica si un ID corresponde a un VBO
void glBindBuffer (GLenum target, GLuint vboID);       // Activa un determinado VBO.
// Para almacenar atributos de vertices en "target" se utiliza GL_ARRAY_BUFFER.
// Para almacenar una lista de indices a vertices se utiliza GL_ELEMENT_ARRAY_BUFFER
```

4.- ¿Qué parámetros recibe el comando glVertexAttribPointer?

Para definir un atributo perteneciente a un VAO se usa el comando anterior

```
void glVertexAttribPointer (GLuint index,      // índice del atributo VAO
        GLint size,        // número de datos por atributo de vértice
        GLenum type,       // tipo de dato
        GLboolean normalized,  // indica si los valores deben normalizarse o no
        GLsizei stride,    // Desplazamiento en bytes entre atributos consecutivos
        const GLvoid * pointer); // puntero al primer componente del primer atributo del vértice en el VBO
```

Los datos se toman desde el buffer activo (glBindBuffer).
La función glVertexAttribPointer() asume que los datos se transformarán al tipo GLFloat. Si la opción de normalizar está activa, los datos enteros se transformarán en reales en el rango [-1,1] para valores con signo (GLint, GLshort, ...) o el rango [0,1] para valores sin signo (GLuint, GLushort, ...).
// Para configurar atributos cuyos valores sean enteros (I) o double (L) se utiliza
`Void glVertexAttrib XPointer (index, size, type, stride, pointer);`

5.- ¿Qué información maneja Vertex Shader?

Un shader es un pequeño programa responsable de la implementación de características inherente a la aplicación que se encuentra dentro del pipeline gráfico y es ejecutado directamente por la GPU.
Para crear un shader se utiliza el comando

```
GLuint shader = glCreateShader (type);
```

// donde type puede ser GL_VERTEX_SHADER, GL_FRAGMENT_SHADER, GL_GEOMETRY_SHADER, GL_TESS_CONTROL_SHADER o GL_TESS_EVALUATION_SHADER

Cada shader tiene una serie de entradas y salidas predefinidas que no hay que declarar.

El Vertex Shader permite manipular toda la geometría 3D provista por un programa OpenGL encargado de trabajar sobre cada uno de los vértices. Los vértices que se procesan son aquellos que son invocados por alguna función de despliegue en la aplicación. La complejidad de un Vertex Shader dependerá de las etapas previas a la rasterización, pudiendo calcular operaciones complejas basadas en la posición de los vértices aplicando diversas transformaciones.
Utiliza variables uniformes para compartir valores de entrada entre vértices. Las matrices de transformación de sistemas de coordenadas se tratan de esta forma.
Utiliza dos entradas predefinidas `in int gl_VertexID;` para el índice del vértice en los array de atributos y `in int gl_InstanceID;` para el número de instancia del vértice.
Genera como salida la posición del vértice (gl_Position) en coordenadas normalizadas (clipping volume) y otros valores que corresponden a entradas usadas posteriormente en el Fragment Shader

6.- ¿Qué información maneja Fragment Shader?

Un Fragment Shader es similar a un Vertex Shader, pero se utiliza para calcular los colores de los fragmentos individuales, es decir, por cada pixel de cada primitiva. Es una etapa programable y no opcional.
Recibe como entrada los valores interpolados para cada pixel.
Genera como salida el color del pixel asociado, en esta etapa se realizan los efectos de iluminación y mapeo de relieve.

7.- ¿Qué parámetros recibe el comando glDrawArrays?

Para lanzar un proceso de renderizado, asumiendo que el programa con los shaders está en uso y que los datos de los vértices y de las variables uniformes han sido cargados, se utilizan los comandos de dibujo glDraw...(). Entre ellos el comando glDrawArrays asigna los vértices a las primitivas de manera consecutiva

```
void glDrawArrays(GLenum mode,  // Especifica el tipo de primitiva a dibujar
                  GLint first,  // Indice del primer vertice a dibujar
                  GLsizei count); // Numero de vértices a dibujar
```

La versión instanciada de este comando ejecuta las primitivas un cierto numero de veces (primcount).

Conclusión

Existen un gran numero de directivas con las cuales nos es posible renderizar o generar a través de las tecnologías de la información imagenes 2D o 3D en una pantalla utilizando datos de vértices y fragmentos procesados. Para realizar esto se utiliza un proceso de pipeline gráfico mediante el cual se transforman diversos datos de entrada para poder generar una imagen final que pueda interactuar con el usuario.

Tanto los VAO's como los VBO's son objetos esenciales en OpenGL para realizar las acciones del renderizado ya que se encargan de almacenar y organizar los datos de los vértices en la memoria de la GPU, cada uno con comandos para crear y modificar sus atributos. Tanto las entradas como salidas de estos elementos están estrechamente relacionados con otras etapas tales como el Vertex Shader y el Fragment Shader pues sin ellas ninguna de las etapas y procesos podrían ser realizados.

Referencias:

Kessenich, J. M., Sellers, G., & Shreiner, D. (2016). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. OpenGL.

Méndez Servín, M. (2022). *NOTAS PARA EL CURSO DE GRAFICACIÓN POR COMPUTADORA* [Libro electrónico]. https://prometeo.matem.unam.mx/recursos/VariosNiveles/iCartesiLibri/recursos/Notas_Graficacion_por_Computadora/index.html

Moreno V., J. (2023). *Tema 3 y Tema 4* [Diapositivas; Presentación electrónica]. www.uhu.es. https://www.uhu.es/francisco.moreno/gii_rv/

Ramírez, E. (2014). Despliegue básico en OpenGL moderno. *Lecturas En Ciencias de la Computación*, ISSN 1316-6239.

Sellers, G., Wright, R. S., & Haemel, N. (2015). *OpenGL superbible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional.