OCEANIA

MANUAL TÉCNICO

1. Plataformas de Trabajo.

Para el desarrollo de nuestro ambiente del acuario fue necesario trabajar bajo el entorno de desarrollo integrado de Microsoft Visual Studio (IDE) el cual es utilizado para desarrollar programas informáticos, aplicaciones, y servicios, para nuestro caso, siguiendo de la mano las operaciones de OpenGL que se considera principalmente una API (una interfaz de programación de aplicaciones) el cual nos brinda un gran conjunto de funciones que podemos usar para manipular gráficos e imágenes. Sin embargo, OpenGL por sí mismo no es una API, sino simplemente una especificación, desarrollada y mantenida por el Grupo Khronos.

La especificación OpenGL define exactamente cuál debe ser el resultado/salida de cada función y cómo debe funcionar. Luego, depende de los desarrolladores como nosotros que implementemos estas especificaciones encontrar una solución de cómo debería operar esta función. Dado que la especificación de OpenGL no nos brinda detalles de implementación, las versiones desarrolladas reales de OpenGL pueden tener diferentes implementaciones, siempre que sus resultados cumplan con la especificación (y, por lo tanto, sean los mismos para los usuarios).

Es importante denotar que para el desarrollo de este proyecto jurásico se trabajó

con las bibliotecas OpenGL que están escritas en C y permiten muchas derivaciones en otros lenguajes, pero en esencia sigue siendo una biblioteca C.

Dado que muchas de las construcciones de lenguaje de C no se traducen tan bien a otros lenguajes de nivel superior, OpenGL se desarrolló con varias abstracciones en mente. Una de esas abstracciones son los objetos en OpenGL.

Se definieron múltiples objetos para este proyecto de recreación de acuario, siguiendo las especificaciones de requerimientos funcionales del apartado anterior. Un objeto en OpenGL es una colección de opciones que representa un subconjunto del estado de OpenGL. Por ejemplo, podríamos tener un objeto que represente la configuración de la ventana de dibujo; luego podríamos establecer su tamaño, cuántos colores admite, etc. Uno podría visualizar un objeto como una estructura tipo C.

Lo bueno de usar estos objetos es que podemos definir más de un objeto en nuestra aplicación, configurar sus opciones y cada vez que comenzamos una operación que usa el estado de OpenGL, vinculamos el objeto con nuestra configuración preferida. Hay objetos, por ejemplo, que actúan como objetos contenedores para los datos del modelo 3D (una casa o un personaje) y cada vez que queremos dibujar uno de ellos, vinculamos el objeto que contiene los datos del modelo que queremos dibujar (primero creamos y configuramos opciones para estos objetos). Tener varios objetos nos permite especificar muchos modelos y siempre que queramos dibujar un modelo específico, simplemente vinculamos el objeto correspondiente antes de dibujar sin volver a configurar todas sus opciones.

2. Parámetros Básicos del Entorno

2.1. Uso de la Ventana

Lo primero que debimos configurar antes de comenzar a crear el entorno gráfico submarino o de acuario fue crear un contexto OpenGL y una ventana de aplicación para dibujar. Sin embargo, esas operaciones

son específicas por sistema operativo y OpenGL intenta abstraerse de estas operaciones a propósito. Esto significa que tenemos que crear una ventana, definir un contexto y manejar la entrada del usuario por nosotros mismos. Afortunadamente, existen bastantes bibliotecas que brindan la funcionalidad que buscábamos, algunas dirigidas específicamente a OpenGL. Utilizamos GLFW. GLFW es una biblioteca, escrita en C, dirigida específicamente a
OpenGL. GLFW nos brinda las necesidades básicas requeridas para mostrar cosas en la pantalla. Nos permite crear un contexto
OpenGL, definir parámetros de ventana y manejar la entrada del usuario, que es suficiente para nuestros propósitos.

2.2. Vinculación

Para que el proyecto use GLFW, necesitábamos vincular la biblioteca con nuestro proyecto. Esto se pudo lograr especificando que queremos usar glfw3.lib en la configuración del enlazador, pero nuestro proyecto aún necesitaba saber dónde encontrar glfw3.lib ya que almacenamos nuestras bibliotecas de terceros en un directorio diferente. Por lo tanto, primero debemos agregar este directorio al proyecto.
Podemos, además, decirle al IDE que tenga en cuenta este directorio cuando necesite buscar una biblioteca e incluir archivos.
Nuestra lista de vinculaciones contempla:
$(SolutionDir)/External
Libraries/GLEW/lib/Release/Win32
$(SolutionDir)/External Libraries/GLFW/lib-vc2015
$(SolutionDir)/External Libraries/SOIL2/lib
$(SolutionDir)/External Libraries/assimp/lib

2.2.1 Bibliotecas Externas Incluidas.

Para que el proyecto pudiera saber en que parte buscar las herramientas necesarias, se inserto manualmente la cadena de ubicación adecuada, el IDE también buscará en esos directorios cuando busque bibliotecas y archivos de encabezado. Tan pronto como se incluyeron las carpetas como la de GLFW, se pudo

encontrar todos los archivos de encabezado para GLFW al incluir
<GLFW/..>., etc. Nuestra lista de Biblioteca incluye:
$(SolutionDir)/External Libraries/GLEW/include
$(SolutionDir)/External Libraries/GLFW/include
$(SolutionDir)/External Libraries/glm
$(SolutionDir)/External Libraries/assimp/include

2.2.2 Parámetros de Entrada del Vinculador.

Una vez que establecimos las cabeceras externas, nuestra IDE VisualStudio puede encontrar todos los archivos necesarios, y finalmente podemos vincular las bibliotecas al proyecto yendo a la pestaña Vinculador y Entrada para terminar nuestra configuración.
soil2-debug.lib;assimp-vc140-mt.lib;opengl32.lib;glew32.lib;glfw3.lib;

3 Shaders

Se utilizaron shaders para ayudar a construir el entorno de acuario de este proyecto, Los shaders son pequeños programas que descansan en la GPU.
Estos programas se ejecutan para cada sección específica de la canalización de gráficos. En un sentido básico, los shaders no son más que programas que transforman entradas en salidas. Los shaders también son programas muy aislados en el sentido de que no se les permite comunicarse entre sí; la única comunicación que tienen es a través de sus

entradas y salidas.
Los shaders utilizados están escritos en el lenguaje GLSL similar a C. GLSL está diseñado para su uso con gráficos y contiene características útiles específicamente dirigidas a la manipulación de vectores y matrices.
Los shaders siempre comienzan con una declaración de versión, seguida de una lista de variables de entrada y salida, uniformes y su función principal.
El punto de entrada de cada shader está en su función principal donde procesamos cualquier variable de entrada y mostramos los resultados en sus variables de salida.
Se utilizaron shaders para la carga de modelos, para la iluminación, para la ambientación (cubemaps), y para las animaciones del proyecto acuario.
Aparecen listados en una carpeta especifica llamada Shaders y su programación es autóctona de su función.

4. Carga de Modelos

4.1 ASSIMP

Para nuestro entorno acuario no podemos definir manualmente todos los vértices, las normales y las coordenadas de textura de formas complicadas.
En cambio, lo que queremos es importar estos modelos a la aplicación; modelos cuidadosamente diseñados por nosotros en herramientas graficas
como 3DS Max o Maya. Estas herramientas de modelado 3D nos permiten crear formas complicadas y aplicarles texturas a través de un proceso
llamado mapeo uv. Luego, las herramientas generan automáticamente todas las coordenadas de vértice, las normales de vértice y las coordenadas de textura mientras las exportan a un formato de archivo de modelo que podemos usar. De esta forma, contamos con un extenso conjunto de herramientas para crear modelos de alta calidad sin tener que preocuparnos demasiado por los detalles técnicos. Todos los aspectos
técnicos están ocultos en el archivo del modelo exportado. Sin embargo, nosotros, como programadores de gráficos, tenemos que preocuparnos por estos detalles técnicos.
Una biblioteca de importación de modelos muy popular que fue implementada para este proyecto, es Assimp, que significa Biblioteca abierta de importación de activos. Assimp puede importar docenas de formatos de archivo de modelo diferentes (y exportar a algunos también) cargando todos los datos del modelo en las estructuras de datos generalizadas de Assimp. Tan pronto como Assimp haya cargado el modelo, podemos recuperar todos los datos que necesitamos de las estructuras de datos de Assimp. Debido a que la estructura de datos de Assimp permanece igual, independientemente del tipo de formato de archivo que importamos,
nos abstrae de todos los diferentes formatos de archivo que existen.
Al importar un modelo a través de Assimp, carga todo el modelo en un objeto de escena que contiene todos los datos del modelo/escena importados. Assimp luego tiene una colección de nodos donde cada nodo contiene índices de datos almacenados en el objeto de escena donde cada nodo puede tener cualquier número de hijos.

4.2 Mesh

Con Assimp pudimos cargar muchos modelos diferentes en la aplicación, pero una vez cargados, todos se almacenan en las estructuras

de datos de Assimp. Lo que eventualmente necesitamos es transformar esos datos a un formato que OpenGL entienda para que podamos representar los objetos. Una malla debería necesitar al menos un conjunto de vértices, donde cada vértice contiene un vector de posición, un vector normal y un vector de coordenadas de textura. Una malla también debe contener índices para el dibujo indexado y datos de materiales en forma de texturas (mapas difusos/especulares)

Gracias al constructor, obtuvimos grandes listas de datos de malla que podemos usar para renderizar. Necesitamos configurar los búferes apropiados y especificar el diseño del shader de vértices a través de punteros de atributo de vértice para que finalmente con la última función que necesitamos definir para que la clase Mesh esté completa es su función 'Draw'. Antes de renderizar la malla, primero queremos enlazar las texturas apropiadas antes de llamar a glDrawElements. Sin embargo, esto es algo difícil ya que no sabemos desde el principio cuántas texturas (si las hay) tiene la malla y qué tipo pueden tener.

4.3 Model

A partir de este punto en el proyecto, comenzamos a crear el código de traducción y carga real con Assimp. El objetivo es crear otra clase que represente un modelo en su totalidad, es decir, un modelo que contenga múltiples mallas, posiblemente con múltiples texturas. Un dinosaurio, por ejemplo, que mantiene garras, dientes, ojos y piel aún podría cargarse como un solo modelo. Cargaremos el modelo a través de Assimp y lo traduciremos a varios objetos de malla que hemos creado.

Teniendo en cuenta que asumimos que las rutas de los archivos de textura en los archivos del modelo son locales para el objeto del modelo real, Ejemplo. en el mismo directorio que la ubicación del propio modelo.

Entonces podemos simplemente concatenar la cadena de ubicación de textura y la cadena de directorio que recuperamos anteriormente (en la función loadModel) para obtener la ruta de textura completa (es por eso que la función GetTexture también necesita la cadena de directorio).

Algunos modelos que se obtuvieron en Internet para este proyecto usaron rutas absolutas para sus ubicaciones de textura, lo que no funcionó al momento de llegar a este punto. En ese caso, editamos manualmente el archivo para usar rutas locales para las texturas (si es posible). O se reemplazaron las texturas y se inició un proceso de adaptación al modelo que se describe a continuación.