DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Renderizado 3D por software

Organización del computador 2

Integrante	LU	Correo electrónico
Germán Pinzón	475/13	pinzon.german.94@gmail.com
Angel More	931/12	angel_21_fer@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

El propósito de este trabajo es desarrollar un renderizador 3D sin la ayuda de las librerías gráficas (Direct3D y OpenGL) que, junto con la placa de video, facilitan la tarea de dibujar primitivas tridimensionales. De esta manera, podemos ver que tan performante puede ser realizar este tipo de cómputos pura y exclusivamente con la CPU. El desarrollo será principalmente en lenguaje C y para las partes críticas de cálculo utilizaremos Assembler junto con SIMD para poder equilibrar un poco el hecho de no apoyarnos en la placa de video. Además, debido a que necesitamos de alguna manera dibujar píxeles en la pantalla, haremos uso de una librería gráfica de bajo nivel llamada SDL que nos permite realizar este tipo de tareas usando exclusivamente la CPU.

Índice

1. Introducción

1.1. ¿Qué es un renderizador 3D?

Para dibujar los gráficos 3D que vemos en muchos de los juegos o software de simulación que podemos encontrar hoy en día, se hacen uso de librerías que facilitan este trabajo. En general siempre existe una abstracción de alto nivel que nos permite dibujar en pantalla un modelo o primitiva 3D de manera directa, indicando su posición en el espacio, rotación o demás características. Sin embargo, si empezamos a bajar el nivel de abstracción, siempre vamos a converger en el uso de dos librerías de bajo nivel: Direct3D y OpenGL.

Direct3D y OpenGL son librerías gráficas que trabajan con la placa de video independientemente del fabricante, permitiendo de esta manera que el desarrollador pueda concentrarse en el código de más alto nivel y no tenga que preocuparse por los detalles de hardware. Además, brindan el soporte matemático necesario para que el programador no tenga que realizar ciertos cálculos o construir matrices desde 0, si no que con algunos datos (por ejemplo para construir una matriz de rotación basta con pasarle el ángulo de rotación) ya se encarga de realizar la tarea por completo. De esta manera, si se desea dibujar una primitiva 3D, basta con pasar sus vértices a alguna función específica que provean estas librerías.

Podemos pensar entonces a un renderizador 3D como un software que se encarga de hacer ésto último, es decir, tomar la información necesaria sobre un modelo 3D (en un formato tridimensional) y convertirla a píxeles para posteriormente dibujarlos en la pantalla. Cuando decimos "información necesaria" estamos resumiendo muchísimas cosas ya que para dibujar un modelo 3D necesitamos bastantes más datos que solo sus vértices. El objetivo de este trabajo va a ser programar una suerte de DirectX u OpenGL propio pero que interaccione solamente con el CPU.

1.2. Librería SDL

SDL es una librería gráfica open source de bajo nivel, multiplataforma, que trabaja con C y que brinda soporte para el desarrollo de aplicaciones gráficas 2D (aunque puede ser usada también con OpenGL). Esta librería nos permite realizar tareas como crear la ventana, imprimir texto en pantalla, detectar eventos de mouse y teclado y pintar píxeles en la ventana de manera sencilla.

Elegimos SDL básicamente por dos motivos, el primero porque es una librería "madura" en el sentido de que lleva varios años en desarrollo y tiene una comunidad importante detras. El segundo motivo fue que también nos permite trabajar con la CPU en el sentido de que podemos evitar la aceleración por hardware de la placa de video, pudiendo de esta manera medir de manera más exacta la performance a la hora de realizar experimentos y poder sacar conclusiones más objetivamente.

1.3. Otras features

Si bien nuestro principal objetivo es el de realizar los cálculos algebraicos necesarios para convertir los vértices de un modelo 3D en píxeles y así poder dibujarlos en la pantalla, existen otras cosas como por ejemplo computar transformaciones básicas sobre el objeto como rotarlo y estirarlo en distintas direcciones, aplicarle texturas y también un tipo específico de iluminación. Además, dado que en general los modelos 3D son generados por programas de diseño como 3D Studio Max o Blender y existen numerosos formatos de archivos de modelos 3D, resulta conveniente elegir uno e implementar un lector para poder cargar la información fácilmente. Nosotros elegimos el formato *.obj e implementamos dicho lector.

2. Código SDL

Como dijimos anteriormente, para realizar este proyecto nos apoyamos en la librería SDL y por lo tanto una parte importante del código hace uso de la misma. En esta sección vamos a detallar y explicar

cuándo y para qué llamamos a la SDL.

2.1. Inicialización, terminación y manejo de errores

Antes de empezar a utilizar SDL hay que hacer una llamada a SDL_Init y pasarle como parámetro SDL_INIT_VIDEO para indicarle que queremos hacer uso del subsistema de video. Esto se debe a que existen otros subsistemas de SDL como por ejemplo el que brinda soporte para el sonido. Así como debemos inicializar la librería, también debemos llamar a SDL_Quit una vez que terminamos de hacer uso de la misma (en nuestro caso antes de que el programa termine).

Existen numerosas funciones de SDL, algunas devuelven punteros a instancias de estructuras propias de la librería (como veremos en las siguientes secciones) y otras simplemente algún número. Es importante saber que estas funciones pueden fallar, en general las que devuelven un número, si este número es negativo significa que falló. En los casos que devuelven un puntero, fallan cuando el puntero es nulo. Para poder detectar correctamente e informar cuando una de estas funciones falla, tenemos la función $SDL\_GetError$ que nos devuelve un mensaje informándonos sobre la falla.

2.2. Manejo de ventana

La función SDL_CreateWindow nos permite crear la ventana donde se van a dibujar las primitivas. Dicha función devuelve un puntero a la estructura SDL_Window y nos permite definir las dimensiones y el título que queramos. Antes de terminar, el programa debe llamar a $SDL\_DestroyWindow$ y pasarle como parámetro el puntero obtenido por $SDL\_CreateWindow$.

En SDL para poder dibujar sobre la ventana, es necesario trabajar sobre superficies, definidas con la estructura $SDL\_Surface$. Así como cuando uno carga una imagen trabaja con la superficie asociada a la imagen, cuando queremos dibujar sobre la ventana debemos obtener la superficie asociada a la ventana utilizando la función $SDL\_GetWindowSurface$. Esta función toma como parámetro un puntero a una instancia de SDL_Window (el que obtenemos al crear la ventana) y nos devuelve el puntero a la instancia de $SDL\_Surface$ que necesitamos.

2.3. Ciclo principal y detección de eventos

Dado que esta es una aplicación interactiva, en el sentido de que no se trata de un software al cual ejecutamos, devuelve una entrada y termina, es necesario que todo el tiempo se esté ejecutando cierto código y ciertas llamadas a funciones. Para esto definimos un ciclo principal que termine únicamente cuando el usuario cierra la ventana.

El ciclo principal es básicamente el nucleo de la aplicación y su condición de corte es que el usuario quiera cerrar la ventana (más adelante veremos como detectar esto). Ahí es donde se ejecuta desde la detección de eventos hasta las llamadas a las funciones encargadas de renderizar los modelos 3D, el cálculo de los FPS y otras cosas más. Cuando dibujamos un modelo 3D y decidimos rotarlo, estirarlo o cambiar el modo del renderizado debemos volver a dibujarlo todo desde cero. Para lograr esto sin que se pise con el dibujo que estaba antes lo que hay que hacer es limpiar la ventana, es decir borrar todo lo que habíamos hecho antes. De esto se encarga la función $SDL\_FillRect$, a la cuál le pasamos como parámetros el puntero a la superficie asociada a la ventana y el color que queremos que tengan todos los píxeles (para esto usamos SDL_MapRGB). Luego de dibujar sobre la superficie asociada a la ventana debemos actualizarla, esto lo hacemos con SDL_UpdateWindowSurface pasándole como parámetro dicha superficie (el puntero).

Para el manejo de eventos primero necesitamos crear una instancia e de tipo SDL_Event. Luego, para detectar eventos del usuario con SDL hay que definir un subciclo del ciclo principal, cuya condición de corte es $SDL\_PollEvent(\&e) == NULL$. Si entramos al ciclo significa que detectamos un evento, para identificar que tipo de evento es usamos el campo $type$ de SDL_Event (el valor puede ser SDL_QUIT

si el usuario quiere cerrar la ventana, SDL_KEYUP si el usuario soltó una tecla, $SDL\_KEYDOWN$ si el usuario está apretando una tecla, entre otros). Es cuestión de anidar distintos switches para poder identificar los eventos que nos importan y actuar en consecuencia. Dado que la detección de eventos es una parte importante del código, está toda dentro de la función $EventDetection$, la cual se ejecuta una vez por cada iteración del ciclo principal.

2.4. Renderizado de texto (HAY QUE LLAMAR A TTF_QUIT)

Tanto para mostrar los FPS como las indicaciones de uso, nuestro programa muestra un texto a la izquierda. Para hacer esto utilizamos una librería llamada SDL_TTF ya que SDL no brinda soporte de manera directa para renderizado de texto.

Esta librería es muy simple de usar y si bien es externa a SDL, está hecha para interactuar con ésta última. De manera similar a SDL, para utilizarla debemos inicializarla llamando a la función TTF_Init y antes de terminar el programa hay que hacer una llamada a TTF_Quit.

Para empezar a hacer uso de la librería y dibujar texto en la ventana primero hay que cargar una fuente, esto lo hacemos utilizando la función $TTF\_OpenFont$ a la cual le pasamos como parámetros la ruta de la fuente que queremos utilizar y el tamaño de letra y nos devuelve un puntero a una instancia de tipo TTF_Font. Una vez hecho esto ya estamos en condición de dibujar texto en la ventana, para lo cual hacemos uso de las funciones $TTF\_RenderText\_Solid$, $SDL\_BlitSurface$ y $SDL\_FreeSurface$. La primera de las tres es la que nos devuelve la superficie asociada al texto que queremos dibujar, toma la fuente que cargamos anteriormente, la cadena de caracteres correspondiente al texto y el color. La segunda función es la que efectivamente, haciendo uso de los conceptos de superficies de SDL, dibuja el texto que se encuentra en la superficie que obtuvimos con la función anterior dentro de la superficie correspondiente a la ventana. Con lo cual sus parámetros son la superficie del texto, la de la ventana y la posición (con el formato de la estructura SDL_Rect). Finalmente debemos llamar a $SDL\_FreeSurface$ pasándole como parámetro la superficie asociada al texto para liberar la misma, ya que no la vamos a necesitar.

Como todo el proceso descripto en el párrafo anterior sobre como se dibuja texto con estas librerías puede ser un poco tedioso si lo que queremos hacer es escribir varias lineas, lo que decidimos hacer fue definir dos funciones que nos abstraigan de todos estos detalles. Estas funciones se llaman $RenderText$ y $RenderTextR$ y se encuentran definidas en sdlHelper.c. La diferencia entre ambas es que la primera no pide la posición del texto a renderizar (siempre lo dibuja arriba a la izquierda, es útil cuando solo se quiere renderizar pocos datos como por ejemplo los FPS) y la segunda sí. Ambas piden el color, la superficie asociada a la ventana, el texto y la fuente a utilizar.

2.5. Timers

Para medir la performance de la aplicación una de las técnicas que usamos fue el uso de timers. Si bien SDL provee soporte para esto, decidimos utilizar los clásicos Clocks de C pero hicimos uso de la función $SDL\_GetTicks$ que nos devuelve la cantidad de milisegundos que pasaron desde que se inicializó la librería.

3. Función $putpixel$

Si bien esta función no se encuentra dentro de la SDL, tiene una relación muy cercana a la misma ya que trabaja a un nivel bastante bajo con sus estructuras y formatos (efectuando operaciones como shifteos y and's) y es por eso que se encuentra en sdlHelper.c. Toma como parámetros la superficie ($SDL\_Surface*$) de la ventana, coordenadas enteras x, y, z, un píxel en formato entero de 32 bits, tamaños enteros ancho y alto de la ventana y un puntero al depthBuffer ($float*$). Lo que hace es manipular

la superficie de la ventana de manera tal de modificar el píxel que se encuentra en la posición (x, y) para que ahora tenga el valor del píxel que toma como parámetro.

Sin embargo, vemos que también hay otros parámetros como una coordenada z y un depthBuffer. Esto es porque, debido a que estamos renderizando en una ventana 2D que representa a un mundo 3D, cada píxel tiene una "profundidad" (que guardamos en el depthBuffer) y que utilizamos para testear si el píxel original que se encuentra en dichas coordenadas está delante o detras (en términos de la coordenada z) del que toma esta función por parámetro. En el caso de que el nuevo píxel esté delante del viejo, entonces efectivamente lo modificamos, caso contrario dejamos todo como está.