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.

2.6. 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.

3. Implementación de estructuras

Dado que este proyecto está realizado en C y algunas partes específicas en Assembler, no contamos con estructuras dinámicas implementadas que podamos usar de manera práctica. Estas estructuras en nuestro caso son arreglos dinámicos, las definimos nosotros en structHelper.h y sus implementaciones están en structHelper.c. Fueron construidas para el almacenamiento dinámico de coordenadas 1D, 2D, 3D y 4D a la hora de cargar un modelo 3D desde un archivo.

Existen varias implementaciones de estas estructuras y todas son similares, lo que cambia son las dimensiones de los puntos que almacena y el tipo de dato utilizado. Esto se debe a que a la hora de cargar un modelo 3D se necesitan cargar muchos datos de distinta naturaleza como por ejemplo los vértices del mismo (tres coordenadas de tipo float), las coordenadas de la textura en el caso de que use alguna (dos coordenadas de tipo float), los índices de los vértices (enteros) y las caras de los triángulos (tres coordenadas enteras).

Si observamos el código en structHelper.h, vemos que cada estructura posee una declaración similar y existen exactamente tres funciones para su manejo: la inicialización, la inserción y la liberación de memoria. Estas tres funciones nos dan el suporte suficiente para la carga de datos.

4. Conceptos matemáticos

En esta sección vamos a exponer un poco los conceptos matemáticos subyacentes de la aplicación. Las librerías gráficas que brindan soporte 3D de alto nivel como también las de bajo nivel como Direct3D y OpenGL, tratan de abstraer un poco al programador de estos conceptos. Esto significa que muchas veces el programador no necesita saber que formato tiene una matriz específica si no que simplemente llama a la función encargada de construir dicha matriz que la librería provee y listo. En este trabajo, dado que no hacemos uso de ninguna librería que brinde el soporte matemático para el manejo 3D, tuvimos que ponernos un poco más en contacto con varios de los conceptos básicos necesarios para tratar con modelos 3D y vamos a introducirlos un poco en esta sección.

Cuando un modelo 3D es creado, éste vive en un espacio de coordenadas por defecto (proporcionado generalmente por el programa que se utilizó para diseñarlo), conocido como espacio del modelo o $ModelSpace$. Luego, para poder ser renderizado pasa por múltiples transformaciones que lo llevan a otros espacios de coordenadas hasta que finalmente es visible en la pantalla.

La idea se basa en utilizar tres matrices fundamentales conocidas como World, View y Projection y combinarlas utilizando el producto de matrices en una sola: PVW = Projection*View*World De esta manera multiplicando cada coordenada del objeto (en $ModelSpace$) por la PVW, el objeto se mueve a través de varios espacios para finalmente proyectarse en pantalla.

4.1. Matriz del mundo (World Matrix)

La primera matriz que describiremos sera la World Matrix o Matriz del mundo. Como mencionamos anteriormente, cada modelo vive inicialmente en un espacio local, por lo que transformaremos sus vértices a un sistema de coordenadas común conocido como espacio de mundo o World Space. Esta matriz

nos va a permitir determinar una posición, orientación e incluso escalar el modelo en el nuevo espacio 3D.

La World Matrix la conseguiremos por medio del producto de otras 3 matrices, cada una de tamaño 4x4 y asumiendo una notación donde las filas representan cada uno de los ejes. Es decir la primera, segunda y tercer fila corresponden a los ejes x, y, z respectivamente y la cuarta fila se utilizará por cuestiones más matemáticas y para diferenciar entre vectores (direcciones) y puntos. Dado que adoptamos esta convención, al efectuar el producto entre un vector y una matriz, el vector deberá ubicarse a la izquierda de la matriz.

La $WorldMatrix$ se compone a su vez por distintos tipos de matrices combinadas, expresando esta combinación como el producto de estas matrices al igual que hicimos con la matriz PVW. A continuación vemos los distintos tipos de matrices que pueden utilizarse para conformar la $WorldMatrix$.

Matriz de traslación (Translation Matrix): nos permite trasladar las coordenadas del modelo en el espacio

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Traslacion.x & Traslacion.y & traslacion.z & 1 \end{bmatrix}$$

Si bien para representar coordenadas en un espacio 3D son necesarios tres valores (ubicación en el eje x, y y z) utilizaremos un cuarto, w, con valor 1, necesario para la aritmética matricial. En este caso Traslación indica en cuanto queremos mover cada una de las coordenadas. Traslacion.x indicara cuanto moveremos el valor x; Traslacion.y el valor de la coordenada en y y finalmente Traslacion.z a z.

Si quisieramos trasladar las coordenadas (5,5,5,1) 12 unidades en dirección del eje x, al multiplicar dicho vector por la matriz de traslación obtendremos como resultado (5 + 12, 5 5, 1) efectivamente obteniendo el resultado deseado.

Matriz de escala (Scale Matrix): nos permite estirar en distintas direcciones al modelo

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

sx, sy y sz representan el tamaño a escalar en la dirección correspondiente a la fila de la matriz. Si tomamos por ejemplo la primera fila, que representa al eje x, se puede observar que el eje conserva la dirección pero es escalado por un valor sx.

Matriz de rotación (Rotation Matrix):

Aquí nos encontramos con distintas variantes

Matriz de rotación alrededor del eje x: nos permite rotar en cierto ángulo al modelo en una dirección particular

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(ang) & sin(ang) & 0 \\ 0 & -sin(ang) & cos(ang) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matriz de rotación alrededor del eje y:

$$\begin{bmatrix} cos(ang) & 0 & -sin(ang) & 0 \\ 0 & 1 & 0 & 0 \\ sin(ang) & 0 & cos(ang) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matriz de rotación alrededor del eje z:

$$\begin{bmatrix} cos(ang) & -sin(ang) & 0 & 0 \\ sin(ang) & cos(ang) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Las matrices representan la rotación de ang grados del plano. Notemos el valor de la coordenada en el eje sobre el cual queremos rotar nunca cambia. Esto es lo que se espera cuando nosotros queremos rotar alrededor del eje en cuestión, ya que la rotación es exclusivamente alrededor del eje y por lo tanto las coordenadas se mantienen constantes.

Ya vimos como es posible trasladar, estirar y rotar el modelo y como dijimos antes, es posible combinar cada una de estas transformaciones en una sola matriz ($WorldMatrix$):
World Matrix = Rotation Matrix * Tanslation Matrix * Scale Matrix

Dado que el producto de matrices no es conmutativo entonces el orden el que se realice puede dar como resultado distintas transformaciones y posicionamiento del modelo final. Nosotros, por convención, adoptaremos el orden anterior. Por ultimo la transformación de cada una de las coordenadas estará dada por:

coordenadas_transformadas = coordenadas_originales * WorldMatrix

4.2. Matriz de vista (View Matrix)

Como explicamos en la sección anterior, el primer paso para renderizar un objeto 3D es poner a todos los modelos en el mismo espacio, $WorldSpace$. Para continuar con la proyección en pantalla de nuestro modelo necesitamos efectuar un nuevo cambio de coordenadas hacia otro espacio llamado $ViewSpace$.

¿Por que es necesario un nuevo espacio? Básicamente porque podríamos estar mirándolo desde cualquier dirección. La transformación consiste en cambiar el sistema de coordenadas de toda la escena a uno nuevo que represente la cámara que está visualizando el modelo (por eso también se lo llama $CameraSpace$. Así, la matriz asociada a esta transformación ($ViewMatrix$) definirá la posición y orientación de la cámara, modificando sus valores alteramos el centro y el plano de proyección respecto al espacio del mundo.

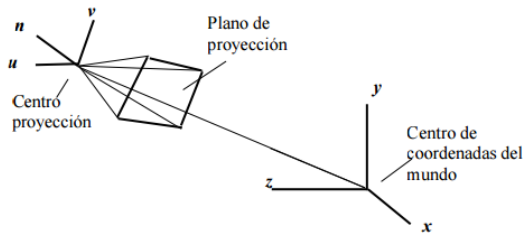
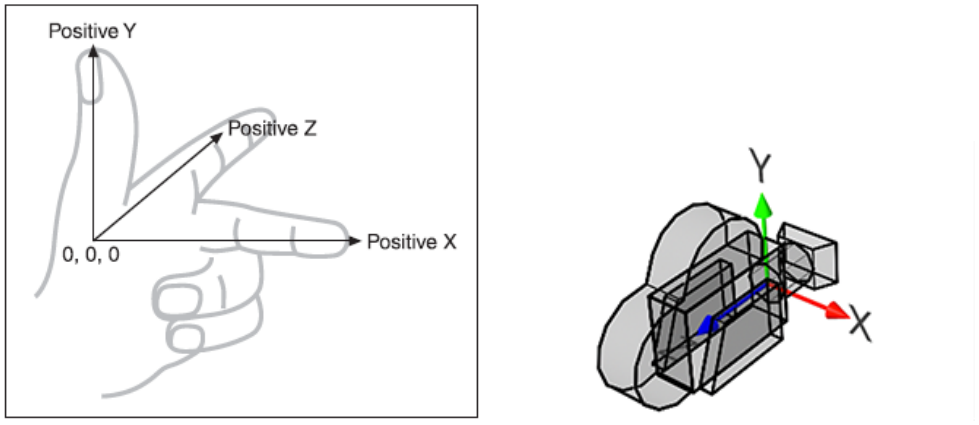


Figura 1: Los vectores n, u, v definen los ejes principales del nuevo sistema de referencia en el $View\ Space$.

Para la creación de la matriz haremos uso de tres vectores eye, $target$ y up. El vector eye determina la posición de la cámara con respecto al $WorldSpace$. $target$ determina la dirección hacia donde está enfocando la cámara. up que define la dirección “hacia arriba” desde el punto de vista de la cámara. Una implementación típica, la cual adoptamos, es asumir que que la cámara esta colocada sobre el eje -z (left-handed coordinate system o regla de la mano izquierda por convención).

Para construir esta matriz debemos primero calcular tres nuevos vectores: x’, y’ y z’. Para calcularlos vamos a empezar por z’ e ir utilizando los datos que ya sabemos (como $target$, eye y up). cross es la



(a) (b)

Figura 2: A la izquierda se muestra la ubicación de los ejes utilizando la regla de la mano izquierda y a la derecha como estaría posicionada nuestra cámara según dicha regla.

función que computa el producto cruzado (así aseguramos perpendicularidad) y normal la función de normalización de vectores. Así, los cálculos se efectúan de la siguiente manera:

z' = normal(eye - target);
x' = normal(cross(up, z'));
y' = cross(z', x');

Con estos datos podemos construir una nueva matriz que represente a estos nuevos ejes:

$$orientation = \begin{bmatrix} x'.x & y'.x & z'.x & 0 \\ x'.y & y'.y & z'.y & 0 \\ x'.z & y'.z & z'.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Como la cámara todavía no está en la posición deseada (eye), para llegar allí se crea la siguiente matriz de traslación:

$$translation = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye.x & -eye.y & -eye.z & 1 \end{bmatrix}$$

Combinando la matriz de orientación con la de traslación obtendremos la matriz deseada ($View\ Matrix$): $View\ Matrix = orientation * translation$;

Equivalentemente se puede expresar directamente como

$$\begin{bmatrix} x'.x & y'.x & z'.x & 0 \\ x'.y & y'.y & z'.y & 0 \\ x'.z & y'.z & z'.z & 0 \\ -dot(x', eye) & -dot(y', eye) & -dot(z', eye) & 1 \end{bmatrix}$$

Donde dot es la función que calcula el producto escalar. Este método es mas optimizado ya que no requiere realizar un producto matricial.

4.3. Matriz de proyección (Projection Matrix)

Todos los vértices transformados dentro del $View\ Space$ siguen teniendo tres dimensiones cuando en realidad para poder dibujarlos en pantalla se necesitan exactamente dos dimensiones. Además, aún no

tenemos perspectiva, es decir, que tan lejos o cerca estamos del objeto. Dado que adoptamos la idea de posicionar la cámara sobre el eje -z, esto determinará la cercanía o lejanía con el modelo, modificando así el tamaño del mismo. La matriz de proyección será la encarga de solucionar estas cuestiones.

Existen distintos tipos de proyecciones, por ejemplo en una proyección en perspectiva los objetos que están más lejos de la cámara se hacen más pequeños, mientras que en una proyección ortográfica la distancia a la cámara no tiene un efecto alguno sobre el tamaño del objeto. En nuestro caso utilizaremos proyección en perspectiva.

La proyección en perspectiva se basa en la idea que la parte visible se encuentra dentro de una determinada figura (llamada $View\ Frustum$). El volumen de esta figura está definido por seis planos: un plano cerca de la cámara ($near\ plane$), un plano más lejos ($far\ plane$) y cuatro planos laterales que definen los bordes. El $near\ plane$ define a partir de donde se puede visualizar y el $far\ plane$ define hasta que punto podemos ver. De este modo, solo los objetos que se encuentren entre estos planos serán representados en pantalla.

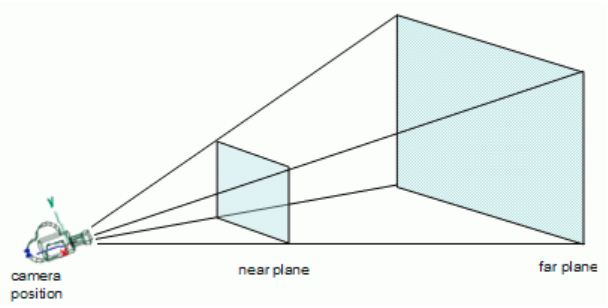


Figura 3: Visualización basada en $View\ Frustum$

La matriz calculará las coordenadas dentro del intervalo [-1,1], de esta forma un punto no será visible sii luego de multiplicar la $projectionmatrix$ es menor que -1 o mayor que 1. Se adopta utilizar las coordenadas dentro de ese intervalo para que sea independiente de las dimensiones de la pantalla. Una vez que tengamos las coordenadas de la matriz solo deberemos multiplicarlas por el ancho y alto de la pantalla en la que se este trabajando para dimensionarlos.

Dado que adoptamos la ubicación de los ejes con la regla $left - handed$ estaremos posicionados sobre el eje -z. Por lo que debemos convertir cada punto de la coordenada z en -z, esto se puede lograr multiplicando las coordenadas por:

$$M1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

De esta forma si tenemos el vector de coordenadas v = (x,y,z,1) quedaría como:

v'= (x, y, -z, 1); Luego de efectuar v*M1

El segundo paso requiere dividir las coordenadas x e y por -z. De esta manera cuando el objeto esté mas lejos de la cámara el valor de z sera mayor por lo que los cocientes serán más chicos (en módulo) y estaremos dándole perspectiva a nuestro objeto.

Recordemos que estamos trabajando con vectores de coordenadas de 4 dimensiones (conocidas como coordenadas homogéneas) donde el ultimo valor es igual a 1. Si multiplicamos cualquiera de estas coordenadas por la World y/o Translation matrix observaremos que el ultimo valor se sigue manteniendo en un 1 (que es lo que queremos). Sin embargo, no es el caso cuando cuando multiplicamos por esta matriz. Esto significa que para volver a las coordenadas homogéneas tenemos que dividir a x, y y z

por w (????). Entonces, si w fuese igual a -z, conseguiríamos lo que estamos buscando: dividir x e y por -z.

Consideremos cambiar la cuarta columna de la matriz M1 por el siguiente vector (0,0,-1,0) tenemos una M2 tal que:

$$M2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

De esta forma un vector v = (x, y, z, w) quedaría como (x, y, z, -z) luego de efectuar v*M2 ya que:

w'=$x * 0 + y * 0 + z * (-1) + 1 * 0 = -z$

Esto tiene por efecto establecer w como -z pero, si -z es diferente a 1, entonces tendrá que ser normalizado es decir, vamos asignar z al rango [0,1]. Para ello, vamos a utilizar los valores $near\ plane$ y $far\ plane$.
Estableceremos los coeficientes de la matriz utilizada M2 cambiando los valores de la tercera columna por (0, 0, $\frac{-f}{f-n}$, $\frac{-f*n}{f-n}$), siendo f el valor de $far\ plane$ y n el valor de $near\ plane$, y veamos que sirve:

Ahora tendremos la matriz:

$$M3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-f}{f-n} & -1 \\ 0 & 0 & \frac{-f*n}{f-n} & 0 \end{bmatrix}$$

Sea v = (x, y, z, w) luego de efectuar v*M3 tendremos un nuevo vector v' con la tercer coordenada igual a:

z' =$x * 0 + y * 0 + \frac{-f}{f-n} * z + \frac{-f*n}{f-n} * 1$

Se deberían cumplir dos condiciones: cuando v este mas cerca del $near\ plane$ z' debería ser 0 y mientras mas cerca de $far\ plane$ debería tender al valor 1. Veamos cuando z es igual a f

z'= $\frac{-f}{f-n} * (-f) + \frac{-f*n}{f-n} * 1 = \frac{f*f+-f*n}{f-n} = \frac{f*(f-n)}{f-n} = f$

Y cuando z es igual a n z'= $\frac{-f}{f-n} * (-n) + \frac{-f*n}{f-n} * 1 = \frac{f*n-f*n}{f-n} = 0$

Recordemos que aun es necesario dividir por w' que es -z por lo que en el primer caso -z = f entonces, tendríamos como resultado final 1; en el segundo caso 0. Efectivamente obtuvimos los valores deseados.
Por último, hay que tener en cuenta el ángulo de visión (AOV) o campo de visión (FOV) de la cámara. Este parámetro controla qué parte de la escena es visible a la cámara.
Generalmente se utiliza es suponer que la ventana de proyección es un cuadrado de [-1: 1] en cada dimensión. y que la distancia entre la misma y la cámara es igual a 1.

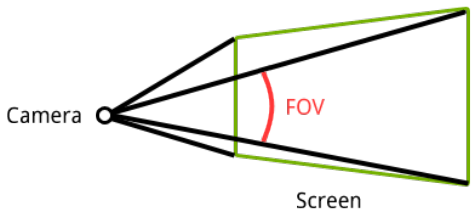


Figura 4: Con verde la porción de la pantalla que queda delimitada al utilizar FOV.

Dado que la parte observable de nuestro mundo 3D puede variar de acuerdo al valor de FOV, debemos

incluirlo de alguna manera en nuestro cambio de coordenadas. Sin entrar en detalle en la trigonometría que hay detras de esto, se utiliza el valor $P = 1 \ / \ tan(FOV * 0{,}5)$. Esto nos va a permitir escalar las coordenadas para poder definir la visibilidad de los vértices de acuerdo a este ángulo y no solo al posicionamiento de los mismos. La matriz de proyección entonces queda de la siguiente forma:

$$\begin{bmatrix} P & 0 & 0 & 0 \\ 0 & P & 0 & 0 \\ 0 & 0 & \frac{-f}{f-n} & -1 \\ 0 & 0 & \frac{-f*n}{f-n} & 0 \end{bmatrix}$$

Ya tenemos todas las matrices necesarias, con lo cual si multiplicamos cada coordenada dentro del $Model\ Space$ por la matriz PVW estaremos casi en condiciones de tener representado nuestro modelo en la pantalla. Lo único que falta es dividir estas coordenadas por w' y mapear estos valores a las dimensiones de nuestra pantalla.

Llamemos x'' e y'' a las coordenadas finales que permitirán graficar en nuestra pantalla al modelo 3D y (x', y', z', w') al valor obtenido luego de multiplicar una coordenada por PVW tendremos que:

$x'' = \frac{x'}{w'} * width + width * 0{,}5$

$y'' = \frac{y'}{w'} * height + height * 0{,}5$

Donde $width$ y $height$ representa el ancho y alto de nuestra pantall. Notar que luego sumamos $width * 0{,}5$ a x'' y $height * 0{,}5$ a y'' para tener las coordenadas finales centradas en la pantalla.

4.4. mathHelper

En el archivo mathHelper.h declaramos numerosas funciones matemáticas que luego definimos en mathHelper.c. Estas funciones van desde cálculos básicos como el producto cruzado, el producto escalar, normalizaciones y productos de matrices hasta la construcción de matrices específicas como de rotación, escala, vista y proyección.

Existen cálculos que se repiten mucho a lo largo del código y que, en algunos casos, es necesario realizarlos por cada ejecución del ciclo principal por ejemplo. En estos casos decidimos implementar también una versión de los mismos en Assembler con el objetivo de optimizar la performance y poder realizar mediciones y comparaciones con las versiones de estas funciones en C. Las funciones matemáticas que decidimos implementar en Assembler son las siguientes: producto escalar (ScalarProdASM), interpolación (InterpolateASM, hablaremos del uso de esta función más adelante) y producto de vector por matriz (Vec4Mat4ProductASM). También existen otras funciones que decidimos implementar en Assembler, que no realizan cálculos matemáticos pero que se ve reflejado en la experimentación que haberlas programado en Assembler mejoró notablemente la performance, hablaremos de ellas más adelante.