Treball de Recerca:
Teoría e implementación de un motor gráfico

Alumno: Joel Pérez
Profesor: José Ramírez
Departamento de Matemáticas

Curso 2017/2018

Abstract

En este Treball de Recerca he intentado demostrar cómo funcionan las principales librerías gráficas comerciales y poder entender los diversos cálculos que se efectúan en la ejecución de una aplicación gráfica 3D. Para esto he programado una librería gráfica relativamente simple y la he usado en un motor gráfico que también he programado.

Agradecimientos

Me gustaría dar las gracias a todos los profesores que he tenido a lo largo de mi vida, en especial a José Ramírez por ser un brillante tutor que además ha invertido mucho de su tiempo libre en ayudarme.

También me gustaría agradecer a mis familiares y mis amigos por soportarme, en especial a mi gran amigo Ori, por introducirme a la programación orientada a gráficos y darme la idea para este TdR.

Contenidos

Capítulo 1

Introducción

1.1 ¿Qué es un motor gráfico?

Un **motor gráfico** es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Del mismo modo existen motores de juegos que operan tanto en consolas de videojuegos como en sistemas operativos. A continuación se mostrarán dos ejemplos de videojuegos con distintos motores gráficos.

(a) *Super Mario Bros.* (1985)

(b) *Super Mario 64* (1996)

Figura 1.1: (a) utiliza un motor gráfico 2D (bidimensional) y (b) utiliza uno 3D (tridimensional)

El primer juego con un motor gráfico 3D en alcanzar un gran nivel de popularidad fue *Doom* en 1993 y fue desarrollado mayoritariamente por John Carmack, jefe de programación en *id Software*.

1.2 Introducción al Treball de Recerca

Una revolución industrial consiste en la llegada de nuevas máquinas y métodos que permiten optimizar los procesos de producción. La industria del software está pasando actualmente por una de estas revoluciones, dónde están surgiendo multitud de herramientas para agilizar la producción. Estas herramientas oscilan desde librerías[1] para funciones matemáticas simples, pasando por editores de texto inteligentes que van acompañados por un compilador y depurador (*IDE*s) e incluso llegan a motores gráficos comerciales disponibles a cualquier programador con una habilidad intermedia, permitiendo saltar uno de los pasos más laboriosos del proceso de creación de videojuegos.

La llegada de esta ola ha sido bastante polémica y ha dividido a las comunidades. Por una parte no se puede negar la importancia de tener herramientas más potentes. Pero por la otra parte hay muchos que piensan que esto ha llevado a la gente a no preocuparse por algunas cuestiones sólo porque lo ven como una cosa del pasado, un problema menor que ya está solucionado por la herramienta que usan. Estos problemas terminan por acumularse y pueden desencadenar en problemas mayores si no se intentan solventar desde la raíz.

Mi opinión siempre ha sido la de permanecer escéptico a los programas que pretenden tener un uso generalizado, y siempre he creído que cada herramienta es para una situación concreta, y no se puede intentar usar el mismo método y herramientas para todo.

1.3 Objetivos

Según la prioridad asignada he separado los objetivos en dos categorías. Es importante recordar que **no es un objetivo** obtener un producto de una complejidad y rendimiento similar a librerías y motores gráficos comerciales.

1.3.1 Objetivos principales

- Investigar la factibilidad de desarrollar una librería gráfica.
- Estudiar la factibilidad de desarrollar un motor gráfico.
- Comprender el funcionamiento de un motor gráfico a nivel interno.
- Crear un motor gráfico que funcione en cualquier sistema operativo.
- Familiarizarse con herramientas de expresión científica.

[1] Colección de recursos no volátiles utilizados por programas informáticos.

1.3.2 Objetivos secundarios

- Permitir la compilación[2] tanto de la librería gráfica como del motor en cualquier entorno de desarrollo.
- Comparar la librería gráfica propia con ejemplos comerciales.
- Comparar el motor gráfico creado con ejemplos comerciales.

1.4 Motivación personal

Existen una gran cantidad de motivaciones personales alrededor del proyecto. Por una parte soy un gran fanático de la computación y con el Treball de Recerca tengo una buena oportunidad de hacer algo que siempre he deseado hacer. También me he inspirado por un videojuego llamado *Handmade Hero*, el cuál está siendo desarrollado, como hace referencia el nombre, *a mano*, sin hacer uso de librerías modernas. También tengo interés en hacer un programa de un tamaño mayor a lo que estoy acostumbrado y aprender a organizar y a documentar el proceso. Además pretendo aprender a usar diversas herramientas para redactar textos científicos, como el propio lenguaje en el que está escrito este documento, LaTeX[3], y el editor de texto Emacs.

1.5 Relevancia del estudio en el campo

Como he comentado previamente, en la actualidad existe una gran polémica con respecto a este tema. Muchas empresas de la industria de los videojuegos han decidido abandonar la creación de motores para uso específico de sus videojuegos en pro de adoptar motores comerciales generales. Esta decisión ha traído muchos escándalos en los que estos motores de uso generalizado funcionan terriblemente mal en algunas configuraciones concretas y debido a la naturaleza cerrada del motor la desarrolladora del videojuego no ha podido solucionar el problema.

Independientemente de la rentabilidad de desarrollar un motor gráfico en la época actual, algo que es indudable es que es completamente necesario entender el funcionamiento del mismo, para poder extraer el máximo rendimiento posible, sea tanto un motor *hecho a mano* como uno comercial.

[2] Traducción de un programa escrito en lenguaje de programación a un lenguaje que el ordenador puede entender.

[3] Sistema de preparación de documentos ampliamente utilizado en la comunidad cientificoténcnica.

1.6 Límites del trabajo

Está clara la inviabilidad de desarrollar un motor gráfico a la altura de los ofrecidos actualmente en el mercado, pues estos han estado desarrollados por un equipo de expertos durante años y tienden a ofrecer una cantidad de rendimiento y funcionalidad que duramente se puede replicar en un proyecto hecho en unos meses por alguien que no tiene ni las nociones académicas ni la experiencia ni los recursos necesarios. Sin embargo esto sólo significa que mi producto no será de uso a la industria, pero puede tener valor académico.

1.7 Metodología empleada para realizar la investigación

El método del trabajo consiste en la creación de un motor gráfico con una funcionalidad limitada pero suficiente para elaborar una conclusión y reconsiderar la hipótesis. No sólo se intentará hacer un motor gráfico sino que además se busca replicar la funcionalidad de OpenGL[4] en una librería gráfica desarrollada paralelamente al motor gráfico para poder ir un paso más allá. La hipótesis será revisada tanto por el resultado final del proceso de elaboración del motor como por la investigación en el funcionamiento de la industria de los motores gráficos.

1.8 Hipótesis

Los equipos de desarrollo de software deberían de aspirar a ser ellos mismos quienes crean las librerías que utilizará el programa en sí. En caso de que esta sea una decisión no rentable, su máxima prioridad debería de ser entender cómo funcionan las herramientas y librerías que utilizarán.

Considero que cuando uno forma parte del proceso de creación de software, tiene que tener el mayor conocimiento posible del funcionamiento de aquello que está haciendo. Eso significa pasar por pasos que muchas veces se saltan con la finalidad de agilizar el proceso, no obstante, no creo que esto sea positivo ya que ha llevado a serios problemas a la larga.

[4]Librería gráfica muy popular por su adaptabilidad, flexibilidad y rendimiento.

Capítulo 2

Conceptos previos

2.1 Entorno de desarrollo

En esta sección se comentarán los diversos entornos en los que he trabajado en el Treball de Recerca, tanto desde un punto de vista de desarrollo de programación como del desarrollo del documento escrito y la presentación oral.

2.1.1 Programación

Dependiendo del sistema operativo en el que se quiera trabajar es necesario tener una preparación concreta. Independientemente del sistema operativo es necesario descargar la carpeta con el código fuente del proyecto. Independientemente del Sistema Operativo utilizado será necesario descargar la última versión del proyecto desde el siguiente enlace:
`https:\github.com/gorostuck/treball-recerca`

Compilación

Para poder compilar[1] y ejecutar la librería y el motor gráfico son necesarias las herramientas básicas de compilación.

- **make**: necesario para automatizar la compilación
- **gcc**: programa utilizado como compilador y/o linker
- **sdl2**: librería utilizada para desarrollo multiplataforma fácilmente, sirve como abstracción para la ventana y comunicación entre el Sistema Operativo y el programa

[1] Consiste en traducir el código fuente a código máquina.

Posteriormente se puede compilar yendo a la raíz de la carpeta y introduciendo algunos de los siguientes comandos en la terminal:

- **compilar** la librería gráfica propia y posteriormente compilar el programa

  ```
  make trgl
  ```

- **compilar** el programa utilizando OpenGL

  ```
  make opengl
  ```

- **ejecutar** el programa una vez la compilación ha terminado satisfactoriamente

  ```
  ./bin/program
  ```

- **limpiar** todos los archivos creados durante procesos anteriores de compilación

  ```
  make clean
  ```

GNU Linux

GNU Linux es el entorno en el que más cómodo trabajo por diversos motivos. Uno de ellos es lo simple que es encontrar, descargar, instalar y configurar nuevos programas en el sistema, gracias al gestor de paquetes que normalmente viene incluido en la mayoría de distribuciones.

Ubuntu[2] es una de las distribuciones más comunes así que explicaré los requisitos previos para la compilación y ejecución del programa en Ubuntu.
Todos los paquetes necesarios se pueden obtener introduciendo este comando en la terminal.

```
sudo apt-get install git libsdl2-dev gcc
```

Se puede descargar el Treball de Recerca utilizando el siguiente comando en la carpeta deseada.

```
git clone https://github.com/gorostuck/treball-recerca
```

[2] `https://www.ubuntu.com`

macOS

Para compilar macOS es un poco más laborioso debido a que no tiene un gestor de paquetes por defecto.
Lo primero que hay que hacer es ir la página web de SDL [3] y descargar las librerías de desarrollo.
Una vez descargado el fichero .dmg, abrirlo y copiar "SDL2.framework" a */Library/Frameworks*.
Después quizás es necesario volver a firmar el Framework, para hacer esto, hay que abrir una terminal en */Library/Frameworks/SDL2.framework/* y utilizar el siguiente comando:

```
codesign -f -s - SDL2
```

Si las *Command Line Tools* no han sido instaladas previamente, la primera vez que intentemos utilizar el comando *make* nos saltará una ventana preguntando si deseamos instalar las herramientas, para poder compilar el programa será necesario utilizar esas herramientas.

Microsoft Windows

Se recomienda fuertemente utilizar Code::Blocks para compilar en MS Windows, aunque es posible simular el compilador usado en los demás entornos (y de hecho es lo que ocurre en Code::Blocks).

Code::Blocks

Aunque no es mi entorno favorito, la verdad es que utilizar Code::Blocks es bastante conveniente y ha sido de utilidad para la depuración[4]. Es tan sencillo como ir a la página y descargar Code::Blocks[5]. Una vez descargado codeblocks, abrir el archivo .cbp dentro de la carpeta del proyecto y todo está listo para compilar y ejecutar. Otra ventaja que tiene es que es multiplataforma con lo que se puede usar en GNU Linux, macOS y en Windows. De hecho se recomienda usar Code::Blocks cuando se trabaje en Windows.

El proceso de configuración es relativamente sencillo ya que las librerías necesarias para lo compilación del proyecto ya están incluidas en este y las instrucciones para compilar ya vienen dadas en el archivo *makefile*.

[3] `https://www.libsdl.org`

[4] El proceso de identificar y corregir errores de programación, en inglés conocido como *debugging*

[5] `www.codeblocks.org`

Control de versión con git y GitHub

Git es un programa utilizado para controlar un repositorio de código[6] En concreto, he utilizado el servicio web *GitHub*. Aunque uno de sus principales objetivos es la colaboración entre distintos desarrolladores yo lo he empleado en las siguientes situaciones:

- Ser capaz de tener todo el código archivado en un repositorio en internet para poder acceder remotamente y descargarlo.
- Mantener un historial con todos los cambios que se han hecho.
- Tener diversas ramas donde cambian elementos clave y desarrollarlas de forma paralela.
- Visualizar rápidamente los cambios producidos entre diferentes versiones y ramas.

[6]Localización central de los archivos, utilizado para guardar múltiples versiones del código fuente de un programa.

2.1.2 Documentación

Documento escrito, LaTeX

Para redactar el documento escrito he utilizado LaTeX[7], un lenguaje de programación orientado al procesamiento de texto. Esta herramienta me ha permitido organizar el Treball de Recerca de forma simple y efectiva, además de incluir trozos de código y comandos e incluso expresión matemática. LaTeXutiliza un sistema de tags para formatear y procesar textos de la forma que desee el usuario. A continuación se muestra un ejemplo:

```
\begin{equation*}
  (g \circ f)^\prime (x) =
  g^\prime (f(x))\cdot f^\prime(x)
\end{equation*}
```

$$(g \circ f)'(x) = g'(f(x)) \cdot f'(x)$$

Además se han utilizado librerías para permitir adjuntar imágenes, dividir las páginas, modificar los márgenes y otros.

También se ha utilizado la aplicación web GeoGebra[8] para representar gráficos a lo largo de este trabajo.

2.1.3 GNU Emacs

He decidido dar a GNU Emacs[9] su propia subsección debido a que lo he usado constantemente a lo largo de la elaboración de este Treball de Recerca. Lo he usado tanto para programar la librería gráfica y el motor como para documentar todo el proceso. Además también me ha permitido hacer *version control*, compilar e incluso ejecutar el motor gráfico desde el mismo programa. Decidí utilizarlo porque así me permitiría aumentar en gran medida mi rendimiento. Puesto a que es un programa extremadamente complejo, no entraré en gran detalle, pero las principales ventajas que ofrece es la gran modularidad y personalización, por lo que lo he ido adaptando a mis necesidades según he ido avanzando en el proyecto.

[7] https://wwww.latex-project.org

[8] https://www.geogebra.org

[9] https://www.gnu.org/software/emacs/

2.2 Gestión de matrices

La librería hace uso de las siguientes matrices:

- **GL_MODELVIEW**: Se utiliza para hacer transformaciones en los objetos. Más en el capítulo 5.
- **GL_PROJECTION**: Se utiliza para crear el volumen de visión, más en el capítulo 6.

2.2.1 Selección de la matriz

Para seleccionar la matriz en la que se quiere trabajar se usa la función

```
glMatrixMode(GLEnum matriz_deseada);
```

Y como único argumento pasamos el nombre de la matriz que queremos seleccionar, **GL_MODELVIEW** o **GL_PROJECTION**.

2.2.2 Reseteo de la matriz

Al llamar a la función

```
glLoadIdentity();
```

la matriz que esté seleccionada actualmente verá sus valores cambiados por los siguientes.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.2.3 Multiplicación de la matriz

Multiplica la matriz seleccionada por la matriz especificada.

```
glMultMatrix(GLfloat *m);
```

m es un puntero hacia 16 floats consecutivos que representan una matriz 4x4.

2.2.4 Carga de la matriz

Se pueden sustituir los valores de la matriz seleccionada con la función

```
glLoadMatrix(GLfloat *m)
```

m es un puntero hacia 16 floats consecutivos que representan una matriz 4x4.

Capítulo 3

Introducción a la librería gráfica

Una librería gráfica es una librería diseñada para ayudar a renderizar gráficos computados a un monitor. Esto típicamente involucra procurar de versiones optimizadas de funciones que llevan tareas de renderización comunes. Esto se puede hacer puramente con el software funcionando en la CPU, como es común en los sistemas enlazados o con hardware acelerado por una GPU, más común en los PCs. Utilizando estas funciones un programa puede ensamblar una imagen para ser mostrada en un monitor. Eso le permite al programador saltarse el paso de crear y optimizar estas funciones, y le permite centrarse en construir el programa gráfico. Las librerías gráficas se usan principalmente en videojuegos y simulaciones. Algunos ejemplos de librerías gráficas son los siguientes:

- Direct3D
- Mantle
- Metal
- OpenGL
- Vulkan

Para este proyecto se ha tomado como referencia OpenGL.

3.1 Pasos que sigue el motor gráfico

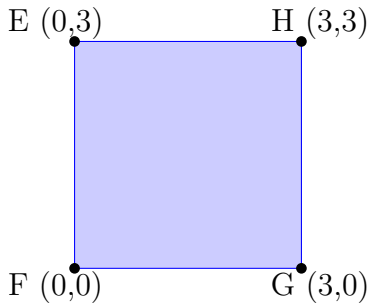
Para mostrar en una pantalla 2D una escena 3D se siguen unos pasos conceptuales que se describirán a continuación:

3.1.1 Espacio local del objeto

Es el espacio en el que se definen las coordenadas de los vértices, las normales y otros. Esto ocurre antes de que tome lugar cualquier transformación. Las coordenadas son relativas al centro del objeto en cuestión.

Tomemos como ejemplo el siguiente cuadrado, éste tiene las siguientes coordenadas:

$$E = (0, 3) \quad F = (0, 0) \quad G = (3, 0) \quad H = (3, 3)$$



Ahora, los valores de estas coordenadas son trasladadas al espacio del mundo, y por lo tanto, los valores de las coordenadas cambiarán.

3.1.2 Espacio global

Es el espacio en el que todos los objetos que hay en la escena están colocados en valores *absolutos* relativos a un punto arbitrario.

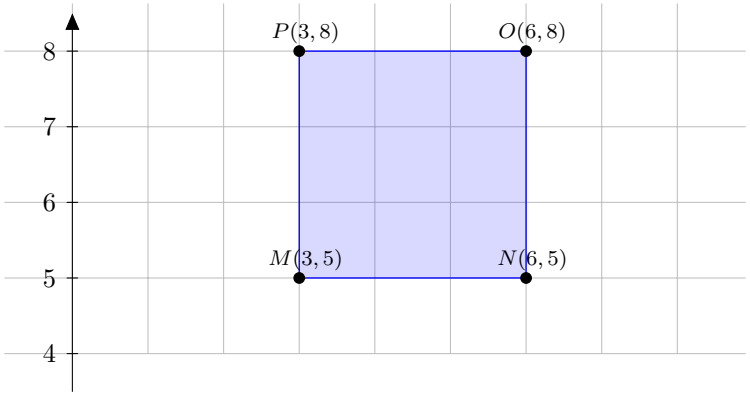
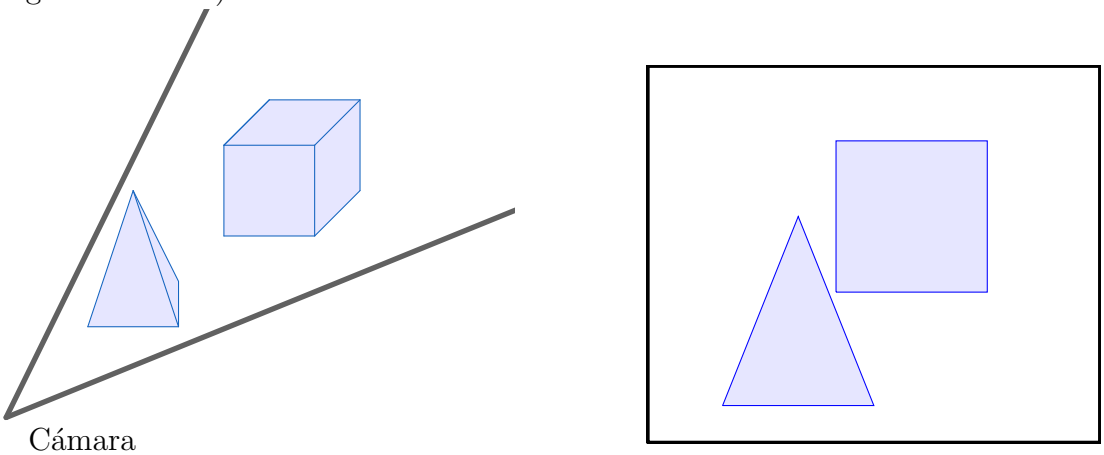


Figura 3.1: Todos los puntos del cubo se han desplazado 3 unidades en el eje X y 5 unidades en el eje Y con respecto a la figura anterior, una transformación ha ocurrido.

3.1.3 Espacio de vista

En el espacio de vista se introduce una *cámara* (coordenada desde la que se proyectará la imagen a mostrar)



3.1.4 Espacio de clipping y división de perspectiva

En este espacio, la escena se inserta en un cubo de coordenadas opuestas (-1,-1,-1) a (1,1,1) y se recortan los objetos que sobresalen.

3.1.5 Espacio de la pantalla

Consiste en mapear las coordenadas NDC a la ventana. X y Y son íntegros, relativos a la esquina izquierda inferior de la ventana. Las Z son escaladas y sesgadas a [0,1]. La rasterización se produce en este espacio.

A estos pasos se le refiere comúnmente como *pipeline*.

Capítulo 4

Espacio local del objeto

4.1 Diferencias entre espacio local y espacio global

Las posiciones 3D y las transformaciones existen dentro de un sistema de coordenadas llamados *espacios*.

El **espacio global** es el sistema de coordenadas para la escena entera. Su origen está en el centro de la escena.

El **espacio local del objeto** es el sistema de coordenadas desde el punto de vista del objeto. El origen del espacio local del objeto está en el centro del mismo, y sus ejes son rotados con el objeto. Tanto en OpenGL como en la librería gráfica propia el concepto de *objeto* es simplemente una lista ordenada de vértices. Esto es llamado *Primitive*. El proceso de creación de estas listas ordenadas de vértices se llama *Vertex Specification*. Los vértices se relacionan de una forma u otra dependiendo del tipo de *primitive* que componen.

4.2 Tipos de primitivos

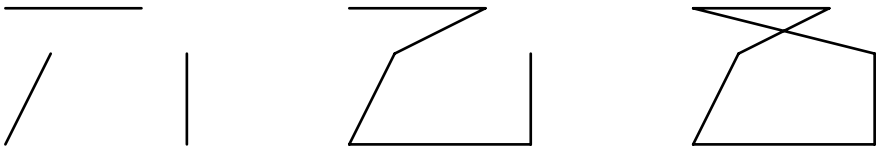
4.2.1 Primitivos de puntos

- GL_POINTS: se interpreta cada vértice individual como un punto.

Figura 4.1: GL_POINTS

4.2.2 Primitivos de líneas

- GL_LINES: Los vértices 0 y 1 se consideran una línea. Los vértices 2 y 3 se consideran una línea, y así progresivamente. Si el usuario especifica un número impar de vértices, el vértice extra es ignorado.
- GL_LINE_STRIP: Los vértices adyacentes se consideran líneas. Entonces, si se pasan n vértices, se producen n-1 líneas. Si el usuario sólo pasa una línea se ignora el comando de dibujo.
- GL_LINE_LOOP: Como el anterior excepto que el primer y último vértice también son usados como línea. Entonces, se producen n líneas para n vértices de entrada.



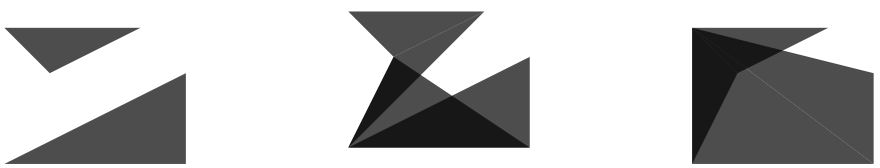
(a) GL_LINES (b) GL_LINE_STRIP (c) GL_LINE_LOOP

Figura 4.2: Primitivos de líneas

4.2.3 Primitivos triangulares

Un triangulo es un primitivo formado por 3 vértices. Es la forma bidimensional con el menor número de vértices, así que los renderizadores normalmente están diseñados para renderizar con ellos.

- GL_TRIANGLES: Los vértices 0, 1 y 2 forman un triángulo, Los vértices 3, 4 y 5 forman un triángulo, y así progresivamente.
- GL_TRIANGLE_STRIP: Cada grupo de 3 vértices adyacentes forma un triángulo. Una lista de n vértices forma n-2 triángulos.
- GL_TRIANGLE_FAN: El primer vértice siempre se mantiene fijo. De ahí en adelante, por cada grupo de dos vértices adyacentes se hará un triángulo con el primero. Una lista de n vértices generará n-2 triángulos.



(a) GL_TRIANGLES (b) GL_TRIANGLE_STRIP (c) GL_TRIANGLE_FAN

Figura 4.3: Primitivos triangulares

4.2.4 Primitivos *quad*

Un *quad* es un primitivo cuadriliteral de cuatro vértices. Frecuentemente se tratan como un par de triángulos.

- GL_QUADS: Vértices 0-3 forman un quad, vértices 4-7 forman otro, y así progresivamente. El número de vértices en la lista debe de ser divisible por 4 para funcionar.
- GL_QUAD_STRIP: De forma similar a GL_TRIANGLE_STRIP, un *quad strip* utiliza los bordes adyacentes para formar el siguiente quad. En el caso de los quads, el tercer y cuarto vértice de un quad se utilizan para el siguiente *quad*. De esta forma, los vértices 0-3 son un quad, 2-6 son otro quad y así progresivamente. Una lista de n vértices generará n-2 / 2 quads.

(a) GL_QUADS (b) GL_QUAD_STRIP

Figura 4.4: Primitivos *quad*

4.3 Especificación de vértices usando la librería gráfica

Es importante destacar que desde que se inicia la creación de un *primitive* hasta que se termina **no** se puede utilizar una función de transformación.

4.3.1 Inicio del *primitve*

Para iniciar la especificación de vértices hace falta llamar a la siguiente función

```
glBegin(GLEnum mode)
```

Y como único argumento introducimos el tipo de primitivo (ver sección anterior).

4.3.2 Pasar coordenadas de los vértices

Existen diversas funciones para crear vértices, dependiendo de si quieren utilizar números flotantes o enteros, y si se quieren utilizar tres dimensiones o solo dos. Esas funciones son las siguientes.

```
glVertex2f(GLfloat x, GLfloat y);
glVertex3i(GLint x, GLint y, GLint z);
```

4.3.3 Pasar color de los vértices

Cada vértice tiene un color con unos valores RGBA concretos, para asignarlos se usa una de las siguientes funciones.

```
glColor3f(GLfloat red, GLfloat green, GLfloat blue);
glColor4i(GLint red, GLint green, GLint blue, GLfloat alpha);
```

De forma similar a pasar coordenadas a los vértices, existen diferentes funciones dependiendo de si se quiere pasar el valor *alpha* o no, y dependiendo de el tipo de número.

Cabe destacar que con llamar a esta función una vez se aplica a todos los vértices especificados posteriormente (incluso si forman parte de otro primitivo) hasta que se vuelve a llamar la función con otro color.

4.3.4 Finalización del *primitive*

Una vez que se han terminado de introducir los vértices, hace falta llamar a la siguiente función sin ningún argumento.

```
glEnd();
```

Capítulo 5

Espacio de vista y transformación de la vista del modelo

En este paso los objetos son transformados del espacio local al espacio de vista utilizando la matriz **GL_MODELVIEW**. Esta matriz es una combinación de las matrices de Modelo y Vista. ($M_{view} \cdot M_{model}$). La transformación de Modelo es para convertir del espacio local del objeto al espacio global. Y, la transformación de Vista es para convertir del espacio global al espacio de vista.

$$\begin{pmatrix} x_{vista} \\ y_{vista} \\ z_{vista} \\ w_{vista} \end{pmatrix} = M_{modeloVista} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{vista} \cdot M_{modelo} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

Nótese que no hay una matriz de cámara (vista) separada en OpenGL. Entonces, para simular la transformación de la cámara o de la vista, la escena (objetos 3D y luces) deben ser transformados con lo inverso de la transformación de vista. En otras palabras, OpenGL define que la cámara siempre se encuentra en (0, 0, 0) mirando al eje -Z en las coordenadas de espacio de vista, y no puede ser transformada.

5.1 Transformación de vectores normales

Los vectores normales también son transformados desde coordenadas locales de objeto a coordenadas de vista para hacer cálculos de iluminación. Es importante destacar que los vectores normales no son transformados de la misma forma.
Para un punto (P_x, P_y, P_z) en un plano; Con un vector normal (n_x, n_y, n_z) en el mismo plano;
La ecuación del plano es la siguiente:

$$n_x x + n_y y + n_z z = 0$$

Ahora podemos sustituir con las coordenadas de ese punto y resulta en:

$$n_x P_x + n_y P_y + n_z P_z = 0$$

En forma de producto de matrices:

$$\begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \cdot \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = 0$$

Nótese que para poder resolver la operación correctamente, la matriz del vector normal está transpuesta. De forma más simple, la operación es la siguiente:

$$n^t \cdot P = 0$$

La inversa de una matriz por la matriz original es igual a la matriz identidad. Por ello, la matriz inversa de GL_MODELVIEW por GL_MODELVIEW es igual a la matriz identidad. Esto se puede introducir en la ecuación y seguiría siendo equivalente.

$$\underbrace{n^t \cdot M^{-1}}_{normal} \underbrace{M \cdot P}_{Vértice} = 0$$

Con esta ecuación hemos llegado a la forma de calcular la transformación de las normales.

La forma final de la ecuación se obtiene aplicando la regla $A^t \cdot B^t = (B \cdot A)^t$.

$$n^t \cdot M^{-1} = n^t \cdot \left((M^{-1})^t\right)^t = \left((M^{-1})^t \cdot n\right)^t$$

La forma en matriz de la operación es la siguiente.

$$\begin{pmatrix} nx_{vista} \\ ny_{vista} \\ nz_{vista} \\ nw_{vista} \end{pmatrix} = ((M_{modeloVista})^{-1})^t \cdot \begin{pmatrix} nx_{obj} \\ ny_{obj} \\ nz_{obj} \\ nw_{obj} \end{pmatrix}$$

5.2 Matriz de modelo-vista (GL_MODELVIEW)

La matriz GL_MODELVIEW combina las matrices de vista y de modelado en una sola matriz. Para transformar la vista (cámara), es necesario mover la escena con la transformación inversa.

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

Los 3 elementos más a la derecha de la matriz (m_{12}, m_{13}, m_{14}) son para la transformación de translación, **glTranslatef()**. El elemento m_{15} es la coordenada homogénia. Se usa específicamente para la transformación proyectiva.

3 conjuntos de elementos, (m_0, m_1, m_2), (m_4, m_5, m_6) y (m_8, m_9, m_{10}) son para transformaciones euclidianas y afines, como la rotación **glRotatef()** o el escalado **glScalef()**. Nótese que estos 3 sets están representando 3 ejes ortogonales:

- (m_0, m_1, m_2) : eje +X, vector *izquierda*, (1, 0, 0) por defecto
- (m_4, m_5, m_6) : eje +Y, vector *arriba*, (0, 1, 0) por defecto
- (m_8, m_9, m_{10}) : eje +Z, vector *adelante*, (0, 0, 1) por defecto

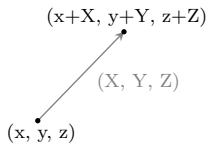
5.3 Reseteo

Para restaurar la matriz a su estado por defecto se hace uso de las siguientes funciones. La primera sólo es necesaria en caso de que la matriz seleccionada actualmente no sea GL_MODELVIEW.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

5.4 Translación

Una operación de translación consiste en desplazar un punto por un vector de coordenadas (x, y, z). La operación tiene el siguiente aspecto:



$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + X \cdot 1 \\ y + Y \cdot 1 \\ z + Z \cdot 1 \\ 1 \end{pmatrix}$$

Con la operación de translación la matriz actual se traslada por (x, y, z). Primero, se crea una matriz de translación, M_T, después se multiplica con la matriz seleccionada actualmente para producir la matriz de transformación final: $M = M \cdot M_T$

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

Figura 5.1: Matriz de translación, M_T

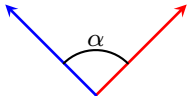
La función utilizada en OpenGL para efectuar una transformación es la siguiente.

```
glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

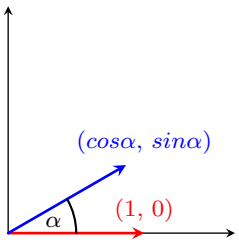
En los argumentos pasamos los valores de las coordenadas del vector deseado.

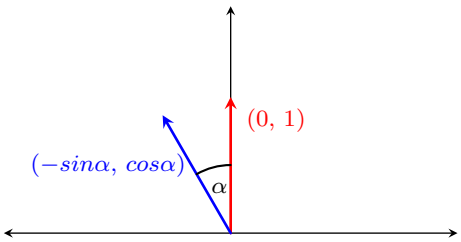
5.5 Rotación

Una transformación de rotación rota un vector alrededor del origen (0, 0, 0) utilizando un eje y un ángulo.



Los objetos pueden ser rotados alrededor de cualquier eje, pero por ahora, sólo son importantes los ejes X, Y y Z. Posteriormente se explicará que se puede establecer cualquier eje de rotación rotando por el eje X, Y y Z simultáneamente. Para comprender de donde salen las matrices de rotación se puede considerar el siguiente ejemplo. Imaginemos que vamos a rotar alrededor del eje Z, esto lleva a que los únicos valores que serán alterados serán los de las coordenadas X y Y. Vamos a ver cómo se rotan por el ángulo α los siguientes vectores.





Por lo tanto, la operación con matrices para rotar alrededor del eje Z con un ángulo α es la siguiente:

$$\begin{bmatrix} cos\alpha & -sin\alpha & 0 & 0 \\ sin\alpha & cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\alpha \cdot x - sin\alpha \cdot y \\ sin\alpha \cdot x + cos\alpha \cdot y \\ z \\ 1 \end{pmatrix}$$

Una operación similar ocurre en la rotación alrededor del eje X:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha & 0 \\ 0 & sin\alpha & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ cos\alpha \cdot y - sin\alpha \cdot z \\ sin\alpha \cdot y + cos\alpha \cdot z \\ 1 \end{pmatrix}$$

Y por último, la rotación alrededor del eje Y:

$$\begin{bmatrix} cos\alpha & 0 & sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\alpha & 0 & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\alpha \cdot x + sin\alpha \cdot z \\ y \\ -sin\alpha \cdot x + cos\alpha \cdot z \\ 1 \end{pmatrix}$$

Estas tres matrices se pueden combinar en una misma con parámetros (x, y, z) para definir el eje de rotación y un ángulo α por el cuál rotar. Esta matriz de rotación M_R luego es multiplicada por la matriz seleccionada actualmente y sustituye esa por el resultado: $M = M \cdot M_R$.

$$\begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ xy(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde $c = cos\alpha$, $s = sin\alpha$

Figura 5.2: Matriz de rotación, M_R

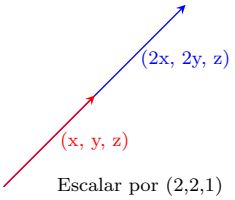
La función utilizada en OpenGL para efectuar una rotación es la siguiente.

```
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

En los argumentos pasamos el ángulo (en grados) que se usará cómo α en la matriz de rotación y los valores de los ejes.

5.6 Escalado

Una transformación de escalado consiste en escalar cada componente de un vector por un número (que puede ser diferente). Se puede utilizar para alargar o acortar un vector. La operación es así:



$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} SX \cdot x \\ SY \cdot 1 \\ SZ \cdot 1 \\ 1 \end{pmatrix}$$

El escalado produce una matriz de transformación de escalado no uniforme en cada eje (x, y, z) multiplicando la matriz de escalado por la matriz seleccionada actualmente. $M = M \cdot M_S$

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 5.3: Matriz de escalado, M_S

La función utilizada en OpenGL para efectuar un escalado es la siguiente.

```
glScalef(GLfloat x, GLfloat y, GLfloat z);
```

En los tres parámetros se indica el factor de escalado deseado en cada uno de los ejes.

Capítulo 6

Espacio de clip y transformación de proyección

En este paso los objetos son transformados del espacio de vista al espacio de clip utilizando la matriz **GL_PROJECTION**. Esta matriz se utiliza para definir el frustum o tronco. Este frustum determina qué objetos o porciones de objetos serán recortados (*clipped out*). También determina cómo se proyecta a la pantalla la escena 3D.

OpenGL proporciona dos funciones para la transformación de GL_PROJECTION.

- **glFrustum()**, para producir una proyección con perspectiva
- **glOrtho()**, para producir una proyección ortográfica (paralela)

Ambas funciones requieren 6 parámetros para especificar los 6 planos de corte; *left*, *right*, *bottom*, *top*, *near* y *far*.

También se puede utilizar **gluPerspective()**, que requiere de sólo 4 parámetros;p el campo de vista vertical (FOV), la relación de aspecto de ancho a alto y las distancias a los planos de corte *near* y *far*. Es posible convertir la función gluPerspective() a glFrustum() con un código que será explicado en una sección posterior.

6.1 Matriz de proyección (GL_PROJECTION)

Un monitor de ordenador es una superficie 2D. Una escena 3D renderizada por OpenGL tiene que ser proyectada a la pantalla del ordenador como una imagen 2D. La matriz GL_PROJECTION es usada para esta transformación de proyección. Primero, la información de las coordenadas de vista es transformada a coordenadas de clip. Luego, estas coordenadas de clip son también transformadas a

coordenadas normalizadas de dispositivo (NDC) dividiendo por el componente w de las coordenadas de clip.

Por ello, tenemos que tener en cuenta que tanto el clipping (corte de frustum) como las transformaciones NDC están integradas en la matriz **GL_PROJECTION**. Las siguientes secciones describen cómo construir la matriz de proyección utilizando 6 parámetros; *left*, *right*, *bottom*, *top*, *near* y *far*.

Nótese que el corte de frustum (clipping) se efectúa en las coordenadas de clip, justo antes de dividir por w_c. Las coordenadas de clip, x_c, y_c y z_c se comprueban siendo comparadas con w_c. Si cualquiera de las coordenadas de clip es menor que $-w_c$, o más grande que w_c, entonces el vértice será descartado. $-w_c < x_c, y_c, z_c < wc$

Entonces, OpenGL reconstruirá los filos del polígono donde ocurre el clipping.

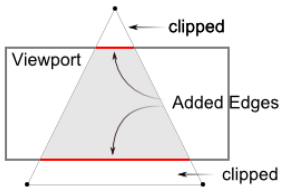


Figura 6.1: Un triangulo clipeado por el frustum

6.2 Proyección de perspectiva

En una proyección de perspectiva, un punto 3D en un en un frustum piramidal truncado (coordenadas de vista) es mapeado a un cubo (NDC); el rango de la coordenada X desde [l, r] hasta [-1, 1], la coordenada Y desde [b, t] hasta [-1, 1] y la coordenada Z desde [n, f] hasta [-1, 1].

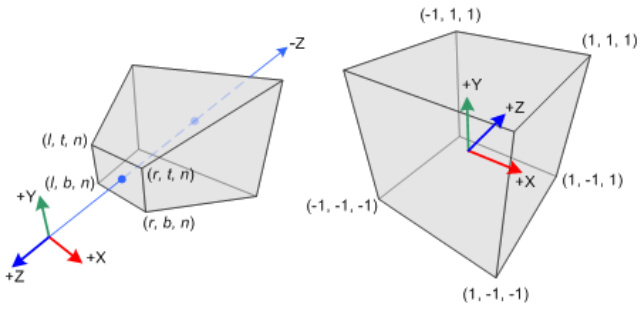
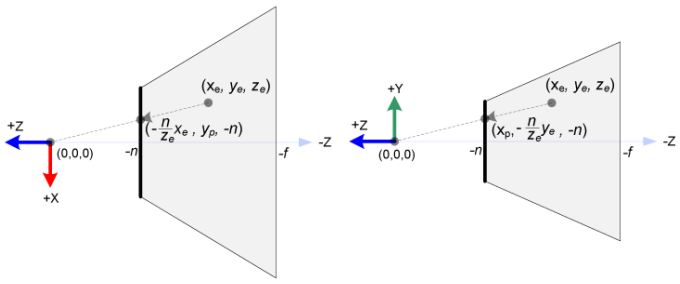


Figura 6.2: Frustum de perspectiva y *Normalized Device Coordinates* (NDC)

Nótese que las coordenadas de vista están definidas por el sistema de coordenadas de la mano derecha, pero NDC utiliza el sistema de coordenadas de la mano izquierda. Por ello, la cámara en el origen está mirando a través del eje -Z en el espacio de vista, pero está mirando en el eje +Z en NDC. Puesto que **glFrustum()** sólo acepta valores positivos cómo distancias *near* y *far*, necesitamos negarlos durante la construcción de la matriz GL_PROJECTION.

En OpenGL, un punto 3D en el espacio de vista es proyectado a el plano *near* (plano de proyección). Los siguientes diagramas muestran como un punto (x_v, y_v, z_v) en el espacio de vista está proyectado a (x_p, y_p, z_p) al plano *near*.



(a) Vista superior del Frustum

(b) Vista lateral del Frustum

Desde la vista superior del frustum, la coordenada X del espacio de vista, x_v está mapeada a x_p, que es calculada utilizando la proporción de triángulos similares;

$$\frac{x_p}{x_v} = \frac{-n}{z_v} x_p = \frac{-n \cdot x_v}{z_v} = \frac{n \cdot x_v}{-z_v}$$

Desde la vista lateral del frustum, y_p puede ser calculada con un método similar;

$$\frac{y_p}{y_v} = \frac{-n}{z_v} y_p = \frac{-n \cdot y_v}{z_v} = \frac{n \cdot y_v}{-z_v}$$

Nótese que tanto x_p como y_p dependen de z_v; son inversamente proporcionales a $-z_v$. En otras palabras, ambas están divididas por $-z_v$. Esta es la primera pista para la construcción de la matriz GL_PROJECTION. Después de que las coordenadas sean transformadas multiplicando la matriz GL_PROJECTION, las coordenadas siguen siendo coordenadas homogéneas. Se vuelven finalmente *Normalized Device Coordinates* (NDC) cuando se dividen por el componente w de las coordenadas de clip.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{vista} \\ y_{vista} \\ z_{vista} \\ w_{vista} \end{pmatrix}, \begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{ndc}/w_{clip} \\ y_{ndc}/w_{clip} \\ z_{ndc}/w_{clip} \end{pmatrix}$$

Entonces, podemos definir el componente w de las coordenadas de clip como $-z_v$. Y la cuarta fila de la matriz GL_PROJECTION se vuelve (0, 0, -1, 0).

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_{vista} \\ y_{vista} \\ z_{vista} \\ w_{vista} \end{pmatrix}, \quad \therefore w_c = -z_v$$

Después, mapeamos x_p y y_p a x_n y y_n de NDC con una relación linear; $[l, r] \implies [-1, 1]$ y $[b, t] \implies [-1, 1]$.

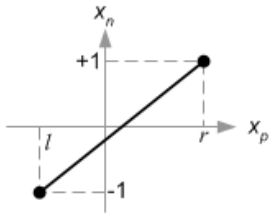


Figura 6.4: Mapeado de x_p a x_n

$$
\begin{aligned}
x_n &= \frac{1-(-1)}{r-l} \cdot x_p + \beta \\
1 &= \frac{2r}{r-l} + \beta \quad (sustituir(r,1)en(x_p, x_n)) \\
\beta &= 1 - \frac{2r}{r-l} = \frac{r-l}{r-l} - \frac{2r}{r-l} \\
&= \frac{r-l-2r}{r-l} = \frac{-r-l}{r-l} = -\frac{r+l}{r-l} \\
\therefore x_n &= \frac{2x_p}{r-l} - \frac{r+l}{r-l}
\end{aligned}
$$

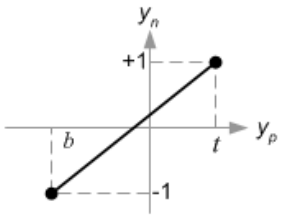


Figura 6.5: Mapeado de y_p a y_n

$$
\begin{aligned}
y_n &= \frac{1-(-1)}{t-b} \cdot y_p + \beta \\
1 &= \frac{2t}{t-b} + \beta \quad (sustituir(t,1)en(y_p, y_n)) \\
\beta &= 1 - \frac{2t}{t-b} = \frac{t-b}{t-b} - \frac{2t}{t-b} \\
&= \frac{t-b-2t}{t-b} = \frac{-t-b}{t-b} = -\frac{t+b}{t-b} \\
\therefore y_n &= \frac{2y_p}{t-b} - \frac{t+b}{t-b}
\end{aligned}
$$

Ahora, sustituimos x_p y y_p a esas ecuaciones.

$$
\begin{aligned}
x_n &= \frac{2x_p}{r-l} - \frac{r+l}{r-l} \quad (x_p = \frac{nx_e}{-z_e}) \\
&= \frac{2 \cdot \frac{n \cdot x_e}{-z_e}}{r-l} - \frac{r+l}{r-l} \\
&= \frac{2n \cdot x_e}{(r-l)(-z_e)} - \frac{r+l}{r-l} \\
&= \frac{\frac{2n}{r-l} \cdot x_e}{-z_e} - \frac{r+l}{r-l} \\
&= \frac{\frac{2n}{r-l} \cdot x_e}{-z_e} + \frac{\frac{r+l}{r-l} \cdot z_e}{-z_e} \\
&= \Big(\underbrace{\frac{2n}{r-l} \cdot x_e + \frac{r+l}{r-l} \cdot z_e}_{x_c} \Big) \Big/ -z_e
\end{aligned}
\qquad
\begin{aligned}
y_n &= \frac{2y_p}{t-b} - \frac{t+b}{t-b} \quad (y_p = \frac{ny_e}{-z_e}) \\
&= \frac{2 \cdot \frac{n \cdot y_e}{-z_e}}{t-b} - \frac{t+b}{t-b} \\
&= \frac{2n \cdot y_e}{(t-b)(-z_e)} - \frac{t+b}{t-b} \\
&= \frac{\frac{2n}{t-b} \cdot y_e}{-z_e} - \frac{t+b}{t-b} \\
&= \frac{\frac{2n}{t-b} \cdot y_e}{-z_e} + \frac{\frac{t+b}{t-b} \cdot z_e}{-z_e} \\
&= \Big(\underbrace{\frac{2n}{t-b} \cdot y_e + \frac{t+b}{t-b} \cdot z_e}_{y_c} \Big) \Big/ -z_e
\end{aligned}
$$

Nótese que hacemos que ambos términos sean divisibles por $-z_e$ para la división de perspectiva (x_c/w_c, y_c/w_c). Y ya establecimos que $w_c = -z_e$ anteriormente, y que los términos dentro del paréntesis se vuelven las coordenadas x_c y y_c de las coordenadas de clip.

De estas dos ecuaciones, podemos encontrar la primera y segunda fila de la matriz GL_PROJECTION.

$$
\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_{vista} \\ y_{vista} \\ z_{vista} \\ w_{vista} \end{pmatrix}
$$

Ahora sólo queda la tercera fila de la matriz GL_PROJECTION por resolver. Encontrar z_n es un poco más diferente de los demás porque en z_{vista} en el espacio de vista está siempre proyectado hacia $-n$ en el plano *near*. Pero ahora necesitamos un valor z único para el clipping y la prueba de profundidad. Además, deberíamos de ser capaces de desproyectar (transformación inversa). Puesto que sabemos que z no depende del valor x o y, tomamos prestado el componente w para encontrar la relación entre z_{ndc} y z_{vista}. Entonces, podemos especificar la tercera fila de la matriz GL_PROJECTION de la siguiente forma.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_{vista} \\ y_{vista} \\ z_{vista} \\ w_{vista} \end{pmatrix}, \qquad z_{ndc} = \frac{z_{clip}}{w_{clip}} = \frac{Az_{vista} + Bw_{vista}}{-z_e}$$

En el espacio de vista, $w_{vista} = 1$. Por lo tanto, la ecuación se vuelve;

$$z_{ndc} = \frac{Az_{vista} + B}{-z_e}$$

Para encontrar los coeficientes A y B usamos la relación (Z_{vista}, Z_{ndc}); $(-n, -1)$ y $(-f, 1)$, y las ponemos en la ecuación de arriba.

$$\begin{cases} \frac{-An+B}{n} = -1 \\ \frac{-Af+B}{f} = 1 \end{cases} \rightarrow \begin{cases} -An + B = -n & (1) \\ -Af + B = f & (2) \end{cases}$$

Para resolver las ecuaciones para A y B, se reescribe la ecuación (1) para B;

$$B = An - n$$

Se sustituye B en la ecuación (2) por la ecuación anterior y se aísla A;

$$\begin{gathered} -Af + (An - n) = f \\ -(f - n)A = f + n \\ A = -\frac{f+n}{f-n} \end{gathered}$$

Se sustituye A en la ecuación (1) para encontrar B;

$$\Big(\frac{f+n}{f-n}\Big)n + B = -n$$

$$\begin{aligned} B &= -n - \Big(\frac{f+n}{f-n}\Big)n = -\Big(1 + \frac{f+n}{f-n}\Big)n = -\Big(\frac{f-n+f+n}{f-n}\Big)n \\ &= -\frac{2fn}{f-n} \end{aligned}$$

Se han hallado A y B. Por lo tanto, la relación entre z_{vista} y z_{ndc} se vuelve;

$$z_n = \frac{-\frac{f+n}{f-n}z_v - \frac{2fn}{f-n}}{-z_v} \qquad (3)$$

Finalmente, se han encontrado todas las entradas de la matriz GL_PROJECTION. La matriz de proyección completa es;

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Figura 6.6: Matriz de proyección de perspectiva de OpenGL

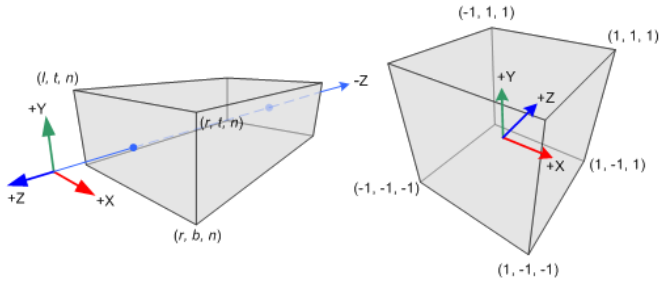
Esta matriz de proyección es para un frustum general. Si el volumen de vista es simétrico, es decir, $r = -l$ y $t = -b$, entonces puede ser simplificada como;

$$\begin{cases} r + l = 0 \\ r - l = 2r \ \ (anchura) \end{cases} , \quad \begin{cases} t + b = 0 \\ t - b = 2t \ \ (altura) \end{cases}$$

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

6.3 Proyección ortográfica

Construir la matriz GL_PROJECTION para la proyección ortográfica es mucho más simple que para la proyección de perspectiva.



Los componentes x_{vista}, y_{vista} y z_{vista} en el espacio de vista son mapeados linealmente a NDC. Sólo es necesario escalar el volumen rectangular al cubo, y luego moverlo al origen. Ahora se encontrarán los elementos de GL_PROJECTION utilizando relaciones lineares.

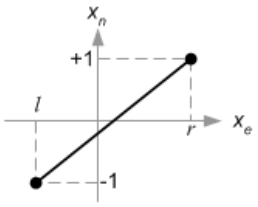


Figura 6.7: Mapeado de x_v a x_n

$$
\begin{aligned}
x_n &= \frac{1-(-1)}{r-l} \cdot x_v + \beta \\
1 &= \frac{2r}{r-l} + \beta \quad (sustituir(r,1)en(x_v, x_n)) \\
\beta &= 1 - \frac{2r}{r-l} = -\frac{r+l}{r-l} \\
\therefore x_n &= \frac{2}{r-l} \cdot x_v - \frac{r+l}{r-l}
\end{aligned}
$$

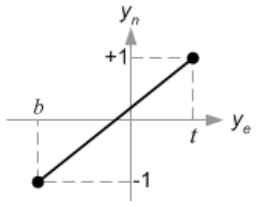


Figura 6.8: Mapeado de y_v a y_n

$$
\begin{aligned}
x_n &= \frac{1-(-1)}{t-b} \cdot x_v + \beta \\
1 &= \frac{2t}{t-b} + \beta \quad (sustituir(t,1)en(y_v, y_n)) \\
\beta &= 1 - \frac{2t}{t-b} = -\frac{t+b}{t-b} \\
\therefore y_n &= \frac{2}{t-b} \cdot y_v - \frac{t+b}{t-b}
\end{aligned}
$$

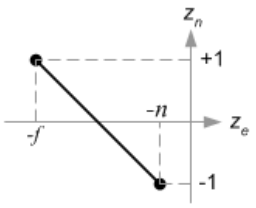


Figura 6.9: Mapeado de z_v a z_n

$$z_n = \frac{1-(-1)}{-f-(-n)} \cdot z_v + \beta$$

$$1 = \frac{2f}{f-n} + \beta \quad (sustituir(-f,1)en(z_v,z_n))$$

$$\beta = 1 - \frac{2f}{f-n} = -\frac{f+n}{f-n}$$

$$\therefore z_n = \frac{-2}{f-n} \cdot z_v - \frac{f+n}{f-n}$$

Puesto que el componente w no es necesario para la proyección ortográfica, la cuarta fila de la matriz GL_PROJECTION permanece como $(0,0,0,1)$. Por lo tanto, la matriz GL_PROJECTION completa para la proyección ortográfica es;

$$\begin{pmatrix} \frac{2}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 6.10: Matriz de proyección ortográfica de OpenGL

Puede ser más simplificada si el volumen de vista es simétrico, $r = -l$ y $t = -b$.

$$\begin{cases} r+l=0 \\ r-l=2r \ \ (anchura) \end{cases}, \quad \begin{cases} t+b=0 \\ t-b=2t \ \ (altura) \end{cases}$$

$$\begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Capítulo 7

Espacio de la pantalla

7.1 Conversión a coordenadas de la pantalla

Una vez se tienen las coordenadas de los puntos entre las coordenadas (-1, -1) y (1, 1) se busca encontrar la coordenada correspondiente en la pantalla.

- x_d es la coordenada en la pantalla
- x_s es la coordenada real normalizada
- x_m es la coordenada donde comienza la superficie en la que queremos dibujar
- x_M es la coordenada donde termina la superficie en la que queremos dibujar
- $\Delta x = x_M - x_m$

$$\frac{x_s - (-1)}{1 - (-1)} = \frac{x_d - x_m}{\Delta x}$$

Si aislamos x_d obtenemos la siguiente ecuación:

$$x_d = \frac{\Delta x \cdot x_s}{2} + \frac{\Delta x}{2} + x_m$$

En el caso de la coordenada y funciona igual excepto que cambiamos el sentido así que tiene una pequeña variación.

$$y_d = \frac{-(\Delta y) \cdot y_s}{2} + \frac{\Delta y}{2} + y_m$$

Y en forma de matriz, la operación quedaría así.

$$\begin{bmatrix} \frac{\Delta x}{2} & 0 & \frac{\Delta x}{2} + x_m \\ 0 & -\frac{\Delta y}{2} & \frac{\Delta y}{2} + y_m \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$$

7.2 Funcionamiento de SDL2

Uno de los objetivos de este Treball de Recerca era utilizar una librería que permitiera compilar y ejecutar el programa en los principales Sistemas Operativos y entornos de desarrollo.

SDL2 es una librería de desarrollo multi-plataforma, que proporciona acceso de nivel bajo al audio, teclado, ratón y hardware gráfico utilizando OpenGL.

Es por eso que ha sido de vital importancia, ya que nos ha permitido simular el funcionamiento de OpenGL sin necesidad de tener que recrear partes más técnicas y complejas que no tenían relevancia en este proyecto.

En general, ha permitido crear, de forma sencilla, un entorno sobre el que aplicar la librería gráfica (tanto la de OpenGL como la nuestra propia).

7.2.1 Cómo lo he usado

SDL2 ofrece dos métodos para la gestión de gráficos, en uno se utiliza OpenGL directamente, se llama a funciones de OpenGL y se trabaja como lo haría OpenGL.

Sin embargo, también existe un método en el que OpenGL se limita a funcionar como comunicador a la tarjeta gráfica.

Este doble funcionamiento ha sido vital, pues ha permitido crear dos versiones del programa. En la primera, OpenGL se usa plenamente. En la segunda, todas las funciones de OpenGL se han mimetizado para intentar comprender cuál es su funcionamiento. Esto ha permitido crear comparaciones entre las dos versiones.

Ahora se explicarán algunas de las funciones de SDL2 que se han ido usando a lo largo del código y que hace falta definir. Para explorar estas funciones y SDL2 en general, existe un sitio oficial[1] que funciona a modo de documentación.

7.2.2 Funciones importantes de SDL2 - OpenGL

Estas son las funciones relevantes para la versión del programa que utiliza plenamente OpenGL.

Inicialización de SDL2

Para iniciar un entorno de SDL2 se llama a la siguiente función:

```
int SDL_Init(Uint32 flags)
```

Como parámetro, se pueden añadir flags de inicialización como por ejemplo *SDL_INIT_VIDEO*, para inicializar el vídeo.

[1] https://wiki.libsdl.org

Creación de ventana

Para crear una ventana se llama a esta función:

```
SDL_Window* SDL_CreateWindow(const char* title,
                             int x, int y, int w, int h,
                             Uint32 flags)
```

Donde los parámetros son el título, las coordenadas x y y iniciales, w y h son la anchura y la altura, respectivamente y además se puede añadir alguna opción en flag, en este caso, es necesario añadir SDL_WINDOW_OPENGL para indicar que esta ventana se destinará a OpenGL.

Creación de contexto

En la anterior función se creó la ventana, pero ahora hace falta crear el entorno (contexto) de renderización de OpenGL.

```
SDL_GLContext SDL_GL_CreateContext(SDL_Window* window)
```

El único parámetro es la ventana previamente creada en la cuál queremos crear el contexto de renderización.

Actualización de la ventana

Puesto a que se está utilizando OpenGL será necesario llamar a

```
void SDL_GL_SwapWindow(SDL_Window* window)
```

Con esta función se actualizará la ventana actual con el búfer[2] de OpenGL.

Finalización de SDL2

Una vez que se desea terminar con la ejecución del programa, hay que terminar antes el entorno de SDL2 con las siguientes funciones.

```
void SDL_DestroyWindow(SDL_Window* window)
```

El único parámetro es la ventana que se desea finalizar.

```
void SDL_Quit()
```

Para la demás funcionalidad, hay que referirse a OpenGL.

[2]Espacio de memoria en el que se almacenan los datos de forma temporal

7.2.3 Funciones importantes de SDL2 - sin OpenGL

Inicialización de SDL2

Para iniciar un entorno de SDL2 se llama a la siguiente función:

```
int SDL_Init(Uint32 flags)
```

Como parámetro, se pueden aladir flags de inicialización como por ejemplo *SDL_INIT_VIDEO*, para inicializar el vídeo.

Creación de ventana

Para crear una ventana se llama a esta función:

```
SDL_Window* SDL_CreateWindow(const char* title,
                             int x, int y, int w, int h,
                             Uint32 flags)
```

Donde los parámetros son el título, las coordenadas x y y iniciales, w y h son la anchura y la altura, respectivamente y además se puede añadir alguna opción en flag.

Creación de renderizador

Puesto a que no estamos utilizando OpenGL de forma directa, es necesario crear un Renderer que gestione lo que SDL tiene que dibujar.

```
SDL_Renderer* SDL_CreateRenderer(SDL_Window* window,
              int index, Uint32 flags)
```

El primer parámetro es la ventana en la que se desea dibujar, la segunda es el índice del dispositivo de renderización (medio que se quiere utilizar para los gráficos) y finalmente una flag que indique el tipo de renderización a usar.

Cambio de color

En lugar lugar de escoger el color cada vez que se dibuja algo, SDL2 guarda el último color seleccionado y es el que utiliza hasta que se llama a la siguiente función.

```
void SDL_SetRenderDrawColor(SDL_Renderer* renderer,
     int8 r, Uint8 g, Uint8 b, Uint a)
```

El primer parámetro es el contexto de renderización que se está utilizando, y los siguientes son los valores RGBA, que definen el color y oscilan del 0 al 255 cada uno.

Dibujo de puntos

Para dibujar un punto, simplemente hay que usar la siguiente función.

```
int SDL_RenderDrawPoint(SDL_Renderer* renderer,
                        int x, int y)
```

El primer parámetro es el contexto usado actualmente. Los valores x y y corresponden a la coordenada de la pantalla en la que se desea dibujar un punto.

Dibujo de líneas

Para dibujar una línea, funciona de forma similar a un punto, pero con una coordenada adicional (la final).

```
int SDL_RenderDrawLine(SDL_Renderer* renderer,
                       int x1, int y1,
                       int x2, int y2)
```

El primer parámetro es el contexto de renderización que se está utilizando, *x1* y *y1* corresponden a la coordenada origen de la línea y *x2* y *y2* corresponden a la coordenada final de la línea.

En caso de de no estar utilizando OpenGL directamente, se puede llamar a

```
void SDL_RenderPresent(SDL_Renderer* renderer)
```

El único parámetro es el renderer utilizado actualmente.

Actualización de la ventana

Una vez se hayan aplicado todas las operaciones que se deseen, se ha de llamar a la siguiente función para mostrarlo.

```
void SDL_RenderPresent(SDL_Renderer* renderer)
```

El único parámetro es el renderizador que se quiere actualizar.

Finalización de SDL2

La forma de finalizar SDL2 cuando no se ha usado directamente OpenGL es un poco más compleja. Primero hace falta destruir el renderer que se ha usado.

```
  void SDL_DestroyRenderer(SDL_Renderer* renderer)
```

Posteriormente, se destruye la ventana.

```
  void SDL_DestroyWindow(SDL_Window* window)
\end{lstlsiting}

Finalmente, se cierra el entorno de SDL.
\begin{lstlisting}[language=C]
  void SDL_Quit()
```

7.3 Estructura del código

El código se estructura en dos tipos de archivos, unos compatibles con OpenGL y con nuestra adaptación de la librería y otros que son los que sustuyen parcialmente la funcionalidad de OpenGL.

Archivos genéricos

Examples Contiene diferentes ejemplos de escenas que incluyen objetos definidos en el archivo *Objects* con la finalidad de comprobar el funcionamiento correcto de la implementación.

Init Inicializa la comunicación gráfica entre software y hardware, es decir, entre el programa y la tarjeta gráfica, mediante la librería SDL2.

main Es donde comienza y acaba el programa, haciendo las llamadas necesarias a las diferentes partes del programa para su continua ejecución.

Objects Instucciones para crear objetos gráficos individuales, que agrupados formarán los ejemplos.

Renderer Es donde se encuentra el bucle de renderización de la escena, además de contener dos funciones de utilidad gráfica para simplificar la visualización en perspectiva, deprecadas en las últimas versiones de OpenGL.

Archivos específicos

Execute Es el archivo que contiene una función llamada desde la función inicial main() en cada bucle de renderización. Se encarga de dos cosas principalmente, la primera es inicializar las variables que se necesitan para los cálculo gráficos y la segunda es la manipulación y gestión de las estructuras de memoria gráfica en cada ciclo.

List Es una lista que contiene todas las intrucciones que se tienen que ejecutar en cada ciclo del programa.

Matrix Se realizan algunas operaciones con matrices necesarias, como es: multiplicar dos matrices, inversa de una matriz, matriz por un vector, normalizar un vector, ...

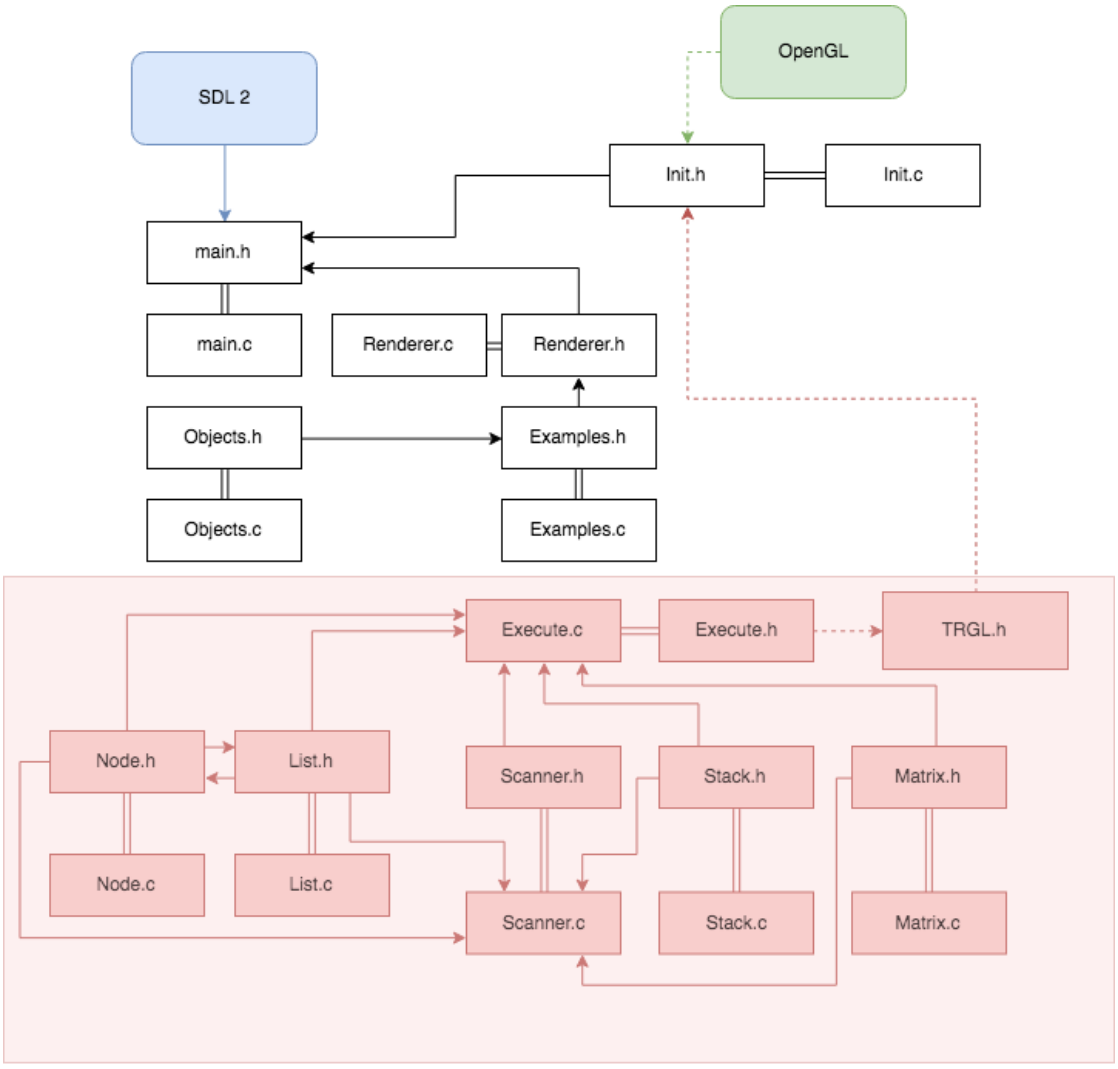
Model Contiene las definiciones de las variables del modelo. Matrices de rotación, translación y escalado, matriz que guarda el último movimiento realizado y los elementos mínimos de funcionamiento del programa, los nodos.

Node Manipula los elementos mínimos del programa, los nodos. Aquí se crean y se destruyen. También es donde se enlazan con el siguiente.

Scanner En este archivo se interpretan las instrucciones y se va creando una lista de nodos a ejecutar en el ciclo actual, guardándose todas en un pila.

Stack Manipula las pilas, es decir, las crea, las destruye y las enlaza con la siguiente. Estas pilas serán recorridas y ejecutadas para crear la escena.

TRGL Encapsula funciones análogas a las OpenGL, de manera que sea posible utilizar la librería OpenGL o el propio motor gráfico con sólo cambiar un parámetro del make (la compilación).


OpenGL
SDL 2
Init.h
Init.c
main.h
main.c
Renderer.c
Renderer.h
Objects.h
Examples.h
Objects.c
Examples.c
Execute.c
Execute.h
TRGL.h
Node.h
List.h
Scanner.h
Stack.h
Matrix.h
Node.c
List.c
Scanner.c
Stack.c
Matrix.c

Capítulo 8

Valoración general y conclusión

8.1 Procesos por los que pasó el Treball de Recerca

Inicialmente el objetivo del Treball de Recerca era el de crear una escena tridimiensional interactiva en la que se mostrase alguna figura. Tras unas pocas semanas, nos dimos cuenta de que hoy en día existen muchas herramientas para realizar esto, pero nos daba la sensación de que no se comprendía muy bien cómo funcionaba, y por lo tanto no era honesto simplemente utilizar esas herramientas, duramente se podría extraer una conclusión.

Por ello decidimos ir un paso más allá e intentar crear nosotros mismos esas herramientas, es decir, crear una librería gráfica con nuestras propias funciones que asimilaran lo que hacían las librerías gráficas populares. Sin embargo hicimos un poco de trampa, pues es increíblemente difícil crear un software que se comunique propiamente con una tarjeta gráfica moderna (hoy en día vienen bloqueadas y se requiere contactar con el fabricante para tener el permiso). Además de la dificultad del proceso, es importante recordar que el enfoque que pretendíamos dar a este trabajo era matemático y no técnico o tecnológico, por lo que entrar en detalle sobre el funcionamiento de una tarjeta gráfica moderna se escapaba de las intenciones originales y no es representativo de una situación moderna en un equipo de desarrollo de Software.

Por lo tanto, nuestra *librería gráfica* utilizaba OpenGL en el fondo, pero éste sólo se dedicaba a comunicarse con la tarjeta, y la manipulación los vértices, el cálculo operaciones y la decisión de qué se muestra en la pantalla quedaron relegados a nuestra librería gráfica.

Aunque parezca que el trabajo se ha simplificado mucho, aún seguía siendo una tarea mucho más laboriosa que el acercamiento original. Esta nueva dirección nos permite atajar algo tan complejo como el mundo de los motores gráficos con preguntas relativamente básicas cómo:

- ¿Cómo se hace una línea?
- ¿Qué operaciones se tienen que aplicar para pasar de coordenadas imaginarias las coordenadas concretas de una ventana?
- ¿Cómo funciona la ocultación de caras?
- ¿Cómo se calcula la rotación de un punto en tres dimensiones?

Y estas son sólo unas pocas de las docenas que han surgido a lo largo del Treball de Recerca y se han ido respondido de forma bastante satisfactoria. Claro está, los recursos son los que hay y han habido cuestiones que han sido más difíciles y prácticamente imposibles de resolver, pero que al menos han surgido en primer momento.

Finalmente, coincidiendo con la aproximación de la fecha de entrega del documento redactado, tocó el momento de trazar la línea y intentar pasar a centarnos en cómo enfocar la defensa del Treball de Recerca. Para esta, al no ser importante el conocimiento adquirido, usaremos herramientas que nos permitan comunicar y explicar el trabajo de una forma más directa y comprensible.

8.2 Revisión de la hipótesis

La hipótesis inicial era la siguiente:

Los equipos de desarrollo de software deberían de aspirar a ser ellos mismos quienes crean las librerías que utilizará el programa en sí. En caso de que esta sea una decisión no rentable, su máxima prioridad debería de ser entender cómo funcionan las herramientas y librerías que utilizarán.

Aunque quizás puede parecer que este trabajo sea más una compilación de cómo funciona una librería gráfica desde un punto matemático más que un intento de comprobar esta hipótesis, yo era plenamente consciente de esto a la hora de formularla. Pensé que ser capaz de crear algo similar a una librería gráfica debería de proporcionarme el conocimiento para determinar si alguien con un cierto conocimiento previo sobre ordenadores y programación necesita adentrarse plenamente en los fundamentos de un motor gráfico para poder llegar a usarlo correctamente.

La evidencia de que este parece ser el caso se encuentra en los commits[1] que han ido ocurriendo a lo largo de la elaboración del código que acompaña este treball de recerca. Es muy fácil apreciar que en cuestión de semanas se pasa de utilizar operaciones ineficientes, guardar información inútil y en general hacer un mal uso de las funciones creadas a utilizar matrices y punteros para optimizar la velocidad y incluso aportar un cierto nivel de modularidad.

Por lo tanto, me parece justo establecer que el mundo de los gráficos computables requiere que el programador sea altamente consciente del funcionamiento interno del mismo para poder hacer un uso óptimo.

La prueba definitiva de que mi hipótesis está en lo ciertose halla en la evolución de OpenGL como librería gráfica. En las versiones originales de OpenGL (1 y 2), la misma librería gráfica, además de gestionar la comunicación con la tarjeta gráfica, hace todos los cálcuos y gestiona los vértices. Sin embargo, en la versión más reciente, OpenGL ha quedado completamente relegado a comunicación directa con la tarjeta gráfica, y todas las funciones de más (justamente las mismas que hemos creado nosotros en nuestra librería gráfica) han creado deprecadas y ahora es tarea del equipo de desarrollo que usa OpenGL crear las suyas propias, pensadas para funcionar de forma óptima en ese proyecto, pues un motor, o incluso librería, con uso general, parece ser cada vez más inviable.

8.3 Opinión personal

Estoy muy satisfecho con el resultado de este Treball de Recerca. Aunque he intentado dejar este documento lo más teórico posible, la mayoría de horas se han invertido en progaramar el motor gráfico utilizando la teoría que se ha expuesto. Espero poder continuar investigando sobre el complejo mundo de los motores gráficos más adelante, pero ciertamente este primer trabajo ha servido para tener una (limitada, pero no negligible) idea de cómo funciona la investigación.

[1] actualización que hace un programador del código de un repositorio

8.4 Bibliografía

Diversos de estos recursos se han ido usando simultáneamente a lo largo del Treball de Recerca

- Wiki de SDL2
  `https://wiki.libsdl.org`
- Página web de Song Ho Ahn sobre OpenGL
  `https://www.songho.ca/opengl/index.html`
- Página web de referencia oficial sobre OpenGL de Khronos
  `https://www.khronos.org/registry/OpenGL-Refpages/`
- Wiki de LaTeX
  `https://en.wikibooks.org/wiki/LaTeX`
- El repositorio del Treball de Recerca creado por mi, crucial para la comprensión del mismo
  `https://github.com/gorostuck/treball-recerca`