

Universidad Nacional Autónoma de
México

Facultad de estudios superiores Acatlán

Materia:

Graficación por computadora .

Proyecto Final: Shooter

Alumnos:

Escobar Loera Alma Angélica

González Aguilar Juan Carlos

Hernandez Ramirez Diego

Palacio Colocho Walter Jonathan

Introducción

En este documento se explicará sobre la elaboración del programa que fue realizado aplicando los conocimientos adquiridos durante todo el curso de Graficación por computadoras.

El Programa fue realizado con OpenGL (**Open Graphics Library**) , que fue el que utilizamos durante todo el curso por recomendación del profesor.

El Proyecto escogido por los integrantes del equipo fue un shooter, ya que este sentimos que implica un poco mas de interacción con el usuario.

Shooter

Es un género aplicado en películas o videojuegos. En videojuegos se caracteriza comúnmente por permitir controlar un personaje que, por norma general, dispone de un arma de fuego (pistola, escopeta, etc). que puede ser disparada a voluntad.

El shooter elaborado en teoría es algo sencillo, se maneja una o varias miras que van a depender de la cantidad de número de objetos a apuntar, haciendo manejo de las prácticas elaboradas, se decidió que los objeto a eliminar serian cubos de rubik.

Indice

Introducción

Indice

Antecedentes

Compilación

Requisitos

Compilación

Arquitectura de la aplicación

Descripción del cubo

Numeración de caras

Numeración de colores

Colorear cada cuadro de las caras

Numeración de renglones y columnas

Descripción de la Función principal

Descripción de algunas de las funciones anexadas en el código:

Algoritmos de luz

Algoritmo para aplicar transparencia al cubo.

Algoritmo para generar mira

Algoritmo para disparar

Conclusiones

Antecedentes

Puntos: Se especifican a partir de su localización y color.

Segmentos de recta: Son esenciales para la mayor parte de las entidades. Se especifican a partir de un par de puntos que representan sus extremos.

Circunferencias: En algunos casos representar entidades curvadas con segmentos poligonales puede ser inadecuado o costoso, por lo que en la prácticas las circunferencias o círculos se adoptan como primitivas. Se especifican con la posición de su centro y su radio.

Polígonos: Son indispensables para representar entidades sólidas. Se representan a partir de la secuencia de puntos que determina la poligonal de su perímetro.

Para el momento de escoger un método de discretización para una primitiva gráfica, es indispensable contar con criterios que permitan evaluar y comparar las ventajas y desventajas de las distintas alternativas.

Apariencia: Es la especificación más obvia, aunque no es fácil describirla en términos formales. Normalmente se espera que un segmento de recta tenga una “apariencia recta” más allá de que se hayan escogido los pixeles matemáticamente más adecuados. Tampoco debe tener discontinuidades. Debe pasar por la discretización del primer y último punto del segmento. Debe ser uniforme, etc.

Simetría e invariancia geométrica: Esta especificación se refiere a que un método de discretización debe producir resultados equivalentes si se modifican algunas propiedades geométricas de la primitiva que se está discretizando. Por ejemplo, la discretización de un segmento no debe variar si dicho segmento se traslada a otra localización en el espacio, o si es rotado, etc.

Simplicidad y velocidad de cómputo: Como los métodos tradicionales de discretización de primitivas se desarrollaron hace tres décadas, en momentos en que las posibilidades del hardware y software eran muy limitadas, los resultados eran muy sensibles al uso de memoria u operaciones aritméticas complejas. Por lo tanto, los métodos tienden a no depender de estructuras complejas y a ser directamente implementables en hardware específico de baja complejidad.

Primitivas de gráficas

Componentes gráficos básicos con lo cual se construyen las escenas (ver [Figura 1](#)):

Puntos

Líneas

Líneas conectadas

Polígonos

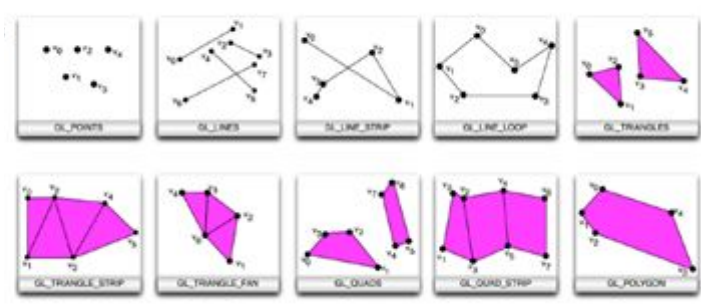


Figura 1 : Primitivas gráficas

Líneas

La constante GL_LINES interpreta una lista de puntos como puntos extremos de segmentos de línea separados. Si es impar ignora el último valor.

GL_LINE_STRIP produce una polilínea abierta.

GL_LINE_LOOP produce una polilínea cerrada.

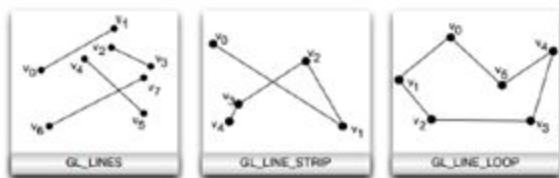


Figura 2 : Tipos de líneas

Transformaciones geométricas

Necesidad de repositonar a los objetos o re-escalarlos.

Queremos visualizar una escena en el Sistema de mundo para desplegarlo en un dispositivo particular.

Necesarios en aplicaciones como CAD y animación.

Las transformaciones son usadas para construir una escena o bien dar la descripción jerárquica de un objeto complejo compuesto de varias partes.

Traslación

Se realiza una transformación mediante la adición de un desplazamiento a sus coordenadas para generar una nueva posición.

Sobre un objeto se realiza esta adición a todos los puntos que describen al objeto.

Rotación

Para generar una rotación debemos especificar un eje de rotación y un ángulo de rotación.

Todos los puntos del objeto son transformados a sus nuevas posiciones al girarlos respecto al eje el número de ángulos.

Una rotación se obtiene al reposicionar al objeto a lo largo de una trayectoria circular en un plano.

La vista

Para obtener un despliegue de una escena en un mundo3D, se debe configurar un marco de referencia para la vista (ojo o cámara).

Dichos parámetros deben definir la posición y orientación de un plano de vista o plano de proyección que corresponde con el plano de la lente de la cámara.

Operaciones de control

2D / 3D

Modelos geométricos e imágenes digitalizadas:

- Rasterizados: Generados por pixel, describen colores en cada punto, y si aplica, la transparencia.
- Vectorizados: Determinados por primitivas geométricas parametrizadas y operaciones que las combinan para lograr composiciones complejas.

Coordenadas

- De modelo: Descripciones geométricas de los objetos que serán desplegados.
- De mundo: Define la localización de los objetos en una escena.
- De vista: La vista que se requiere en función de una cámara hipotética.
- Normalizadas: Las posiciones son transformadas a una proyección 2D normalizadas.
- De dispositivo: Coordenadas del dispositivo en donde se desplegará la escena.

Pipeline de OpenGL

La **representación de la tubería** es la secuencia de pasos que toma OpenGL al representar objetos. Esta visión general proporcionará una descripción de alto nivel de los pasos de la pipeline (ver [Figura 3](#)).

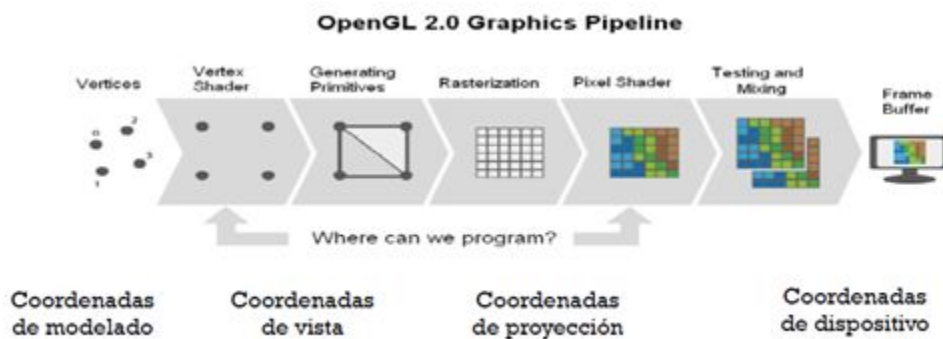


Figura 3 : Pipeline 2D

Viewports.

Un viewport es una región rectangular visualización en gráficos de computadora, o un término utilizado para los componentes ópticos. Tiene varias definiciones en diferentes contextos:

En gráficos 3D por computadora se refiere al rectángulo 2D utilizado para proyectar la escena 3D en la posición de una cámara virtual. Un viewport es una región de la pantalla que se utiliza para mostrar una parte de la imagen total que shown.

En los escritorios virtuales, la vista es la parte visible de un área 2D que es más grande que el dispositivo de visualización. En los navegadores web, la vista es la parte visible de la canvas.

Iluminación y sombreado

OpenGL calcula el color de cada píxel en una escena final, se muestra que se mantiene en el uso de este dispositivo. Parte de este cálculo depende de lo que la iluminación se utiliza en la escena y de cómo los objetos en la escena reflejan o absorben la luz. Como ejemplo de esto, recordar que el océano tiene un color diferente en un día brillante y soleado que lo hace en un día nublado gris. La presencia de la luz del sol o las nubes determina si se ve el océano como turquesa brillante o turbia de color verde grisáceo. De hecho, la mayoría de los objetos ni siquiera se ven en tres dimensiones hasta que estén encendidos. La siguiente figura muestra dos versiones de la misma escena exacta (una sola esfera), una con y otra sin iluminación.

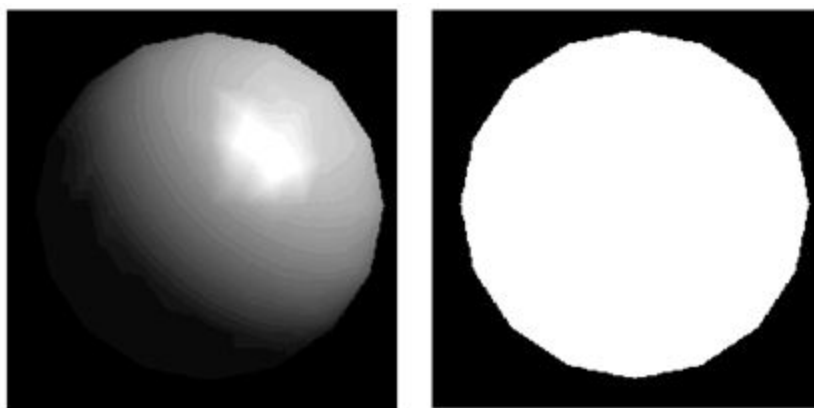


Figura 4 : Un lit y una esfera Unlit

Como se puede ver, una esfera sin luz no es diferente de un disco de dos dimensiones. Esto demuestra lo importante que esta interacción entre los objetos y la luz es en la creación de una escena tridimensional.

Con OpenGL, puede manipular la iluminación y objetos en una escena para crear muchos tipos diferentes de efectos.

Cuando nos fijamos en una superficie física, la percepción del ojo del color depende de la distribución de energías de fotones que llegan y desencadenan sus células del cono. Esos fotones provienen de una fuente de luz o combinación de fuentes, algunas de las cuales son absorbidos y algunos de los cuales son reflejados por la superficie. Además, las diferentes superficies pueden tener propiedades muy diferentes - algunos son brillantes y preferentemente reflejan la luz en ciertas direcciones, mientras que otros se dispersan la luz que entra por igual en todas las direcciones. La mayoría de las superficies están en algún punto intermedio.

Ambiente, difusa y especular Luz

Un ambiente iluminación es la luz que se ha dispersado tanto por el ambiente que su dirección es imposible determinar - que parece venir de todas las direcciones. Retroiluminación en una habitación tiene un componente ambiente grande, ya que la mayor parte de la luz que alcanza el ojo ha sido rebotada muchas superficies. Un centro de atención al aire libre tiene un componente ambiental pequeño; la mayor parte de la luz viaja en la misma dirección, y ya que estás al aire libre, muy poco de la luz alcanza el ojo después de rebotar en otros objetos. Cuando la luz ambiental golpea una superficie, que está dispersa por igual en todas las direcciones.

El componente difuso es la luz que proviene de una dirección, por lo que es más brillante si se trata de lleno sobre una superficie que apenas si rebota en la superficie. Una vez que golpea una superficie, sin embargo, se esparció por igual en todas las direcciones, por lo que parece igual de brillante, sin importar dónde se encuentra el ojo. Cualquier luz que viene de una posición o dirección en particular, probablemente tiene un componente difuso.

Finalmente, la luz especular proviene de una dirección particular, y tiende a rebotar en la superficie en una dirección preferida. Un rayo láser colimado bien-rebotando en un espejo de alta calidad produce casi el 100 por ciento de la reflexión especular. Brillante metal o de plástico tiene un alto componente especular, y la tiza o la alfombra tiene casi ninguna. Se puede pensar en especularidad como brillo.

Aunque una fuente de luz proporciona una única distribución de frecuencias, el ambiente, difusa, y los componentes especulares pueden ser diferentes. Por ejemplo, si usted tiene una luz blanca en una habitación con paredes de color rojo, la luz dispersada tiende a ser de color rojo, a pesar de la luz que incide directamente en los objetos es de color blanco. OpenGL permite configurar los valores de rojo, verde y azul para cada componente de la luz de forma independiente.

Refracción

La luz cambia de velocidad cuando pasa de un medio a otro. Esto afecta la Frecuencia de la luz pero si la longitud de onda.

Índice de refracción es la razón de la velocidad de la luz en el vacío con la Velocidad en un medio diferente.

Los rayos de luz se doblan cuando pasan de un medio a otro.

Objeto transparente si podemos ver las cosas situadas detrás de él, si no se pueden ver es opaco.

Incluir objetos de refracción produce escenas muy reales, alto procesamiento. Se usan aproximaciones.

Sin cambio en el índice de refracción.-Se asigna un coeficiente de transparencia para saber qué porcentaje de luz de los objetos de segundo plano hay que transmitir. Orden de profundidad. Búfer de profundidad modificado.

Compilación

Como muchos programas de computadora para la correcta instalación de las aplicaciones es necesario que nuestros equipos tengan una serie de requisitos además de un modo diferente de instalación.

Requisitos

- Tener instalado los compiladores g++ o en su defecto una alternativa para compilar archivos fuente en lenguaje c++
- Instalar librerías OpenGL y GLUT en sistemas linux existe openGLUT que funciona de igual manera para sustituir GLUT sin necesidad de cambiar nada en los codigos

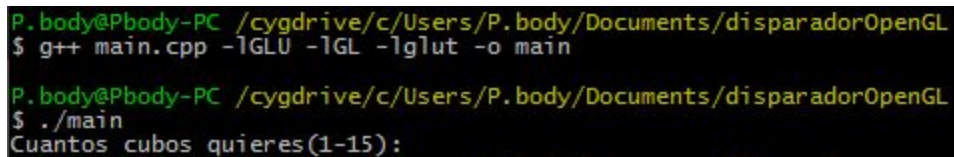
Compilación

1. Compilar código con las siguientes banderas:

```
g++ main.cpp -lGLU -lGL -lglut -o main
```

2. Probar aplicación: Ejecutarla aplicación (Linux)

```
./main
```



```
P.body@Pbody-PC /cygdrive/c/Users/P.body/Documents/disparadorOpenGL
$ g++ main.cpp -lGLU -lGL -lglut -o main

P.body@Pbody-PC /cygdrive/c/Users/P.body/Documents/disparadorOpenGL
$ ./main
Cuantos cubos quieres(1-15):
```

Figura 5 : Vista del resultado de los comandos

Utilizar la aplicación

1. Elegir el número de cubos que quiere, la aplicación permite el juego desde uno hasta quince cubos (ver [figura 5](#)).
2. Aparecerá una pantalla (interfaz del juego "disparador").
3. Al ir disparando (clickeando) sobre los cubos más pequeños que están en el centro de los cubos más grandes estos van desapareciendo por secciones; cada cubo está compuesto por cuadrados, estos son los que van desapareciendo a cada disparo(click) como se ve en la siguiente imagen después de dar varios clicks sobre el cubo. (ver [figura 6](#)).
4. Una vez que se acaben estos cuadrados por los que está compuesto el cubo, osea que se haya disparado varias veces a el cubo del centro y estos a su vez desaparecen al click.
5. Llegará un punto en el cual ya no haya más cuadrados que quitar y el cubo desaparecerá como se ve en la siguiente imagen, también se observa como en la descripción del juego te muestra tu puntaje; este es el número de segundos que has hecho, un contador. Del lado derecho en la descripción te dice el número de cubos al que te hace falta disparar (ver [figura 7](#)).
6. Después desapareceremos el último cubo disparando (haciendo clicks sobre él), una vez desaparecido ya no quedan más cubos y aparecerá una leyenda diciendo Ganaste!!. (ver [figura 8](#))
7. El paso más importante, ¡Diviértete!.

Ejemplos de la aplicación:

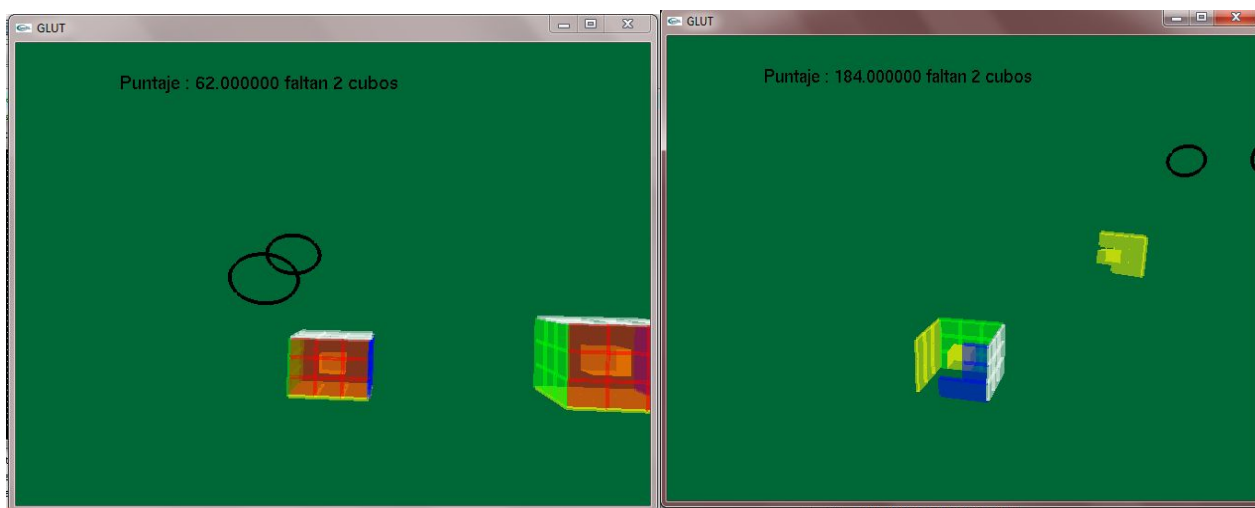


Figura 6 : Vista del resultado de seleccionar 2 objetos, como se muestra aparecen 2 miras y muestra despues de varios disparos.



Figura 7 : Vista del resultado de desaparecer un Cubo y la leyenda diciendonos que aun nos falta un cubo por derrotar.



Figura 8 : Vista resultante de eliminar todos los cubos.

Arquitectura de la aplicación

De manera general la aplicación consta de un archivos principales: main.cpp y este a su vez cuenta de diferentes componentes para su funcionamiento :

- ❖ Descripción del cubo

 - Numeración de caras

 - Numeración de colores

 - Colorear cada cuadro de las caras

 - Numeración de renglones y columnas

- ❖ Descripcion de la función Principal

 - Descripcion de algunas de las funciones anexadas en el código

- ❖ Algoritmos de luz

- ❖ Algoritmo para dar transparencia

- ❖ Algoritmo para generar mira

- ❖ Algoritmo para disparar

Descripción del cubo

Es la representación gráfica de un cubo de 6 caras de diferentes colores cada una, estas caras están formadas por 9 cuadrados cada una, que su vez estos cuadros están conformados por la unión de líneas cada cuadro, estas líneas son primitivas gráficas que nos ofrece la librería Glut.h, se hablará más adelante sobre la es estructura de esto.

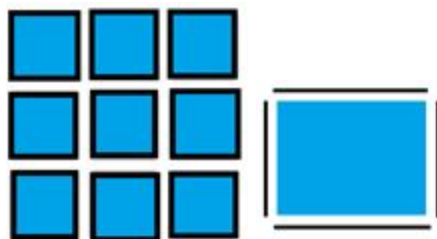


Figura 9 : Cubo.

Numeración de caras

El mapeo de las caras a números se hizo de la siguiente manera:

- Frontal -> 1
- Trasera -> 0
- Izquierda -> 5

- Derecha -> 4
- Piso -> 2
- Techo -> 3

Esto quiere decir que los vértices para pintarlo en OpenGL que le pertenecen a cada cara (ver [Figura 10](#)) está dado por el intervalo $\text{numeracionCara} * 9 - \text{numeracionCara} + 8$, posiciones del vector `vertices[][]` de vectores en 3D.

Numeración de colores

Para numerar los colores, se hizo un mapeo muy parecido al que se hizo al de las caras

- Amarillo -> 1
- Blanco -> 0
- Naranja -> 5
- Rojo -> 4
- Verde -> 2
- Azul -> 3
- Blanco -> 6

Este mapeo se guardó en el vector `color[]`, pero no quiere decir que sea lo mismo que el número de las caras, cualquier cara en un momento dado puede estar pintado indistintamente de los colores que sean, excepto negro claro.

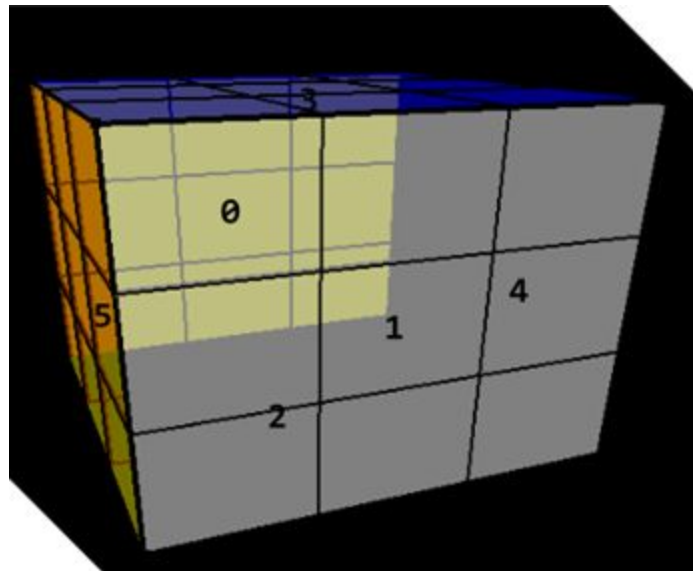


Figura 10 :Numeración de caras elegido

Colorear cada cuadro de las caras

Para saber de qué color debe ir cada cuadro, se hizo un arreglo llamado *colores* inicializado con los siguientes valores:

```
int colores[54]=  
{  
    0, 0, 0, 0, 0, 0, 0, 0, 0, //Cara 0  
    1, 1, 1, 1, 1, 1, 1, 1, 1, //Cara 1  
    2, 2, 2, 2, 2, 2, 2, 2, 2, //Cara 2  
    3, 3, 3, 3, 3, 3, 3, 3, 3, //Cara 3  
    4, 4, 4, 4, 4, 4, 4, 4, 4, //Cara 4  
    5, 5, 5, 5, 5, 5, 5, 5, 5 // Cara 5  
};
```

Eso quiere decir que la cara 1 es de amarillo, la 2 de blanco, así hasta la cara 5 de color naranja, esto se puede saber porque los valores dentro del arreglo representan el índice del color asociado a los cuadros de cada cara del cubo. Estos valores siempre van a ser los mismos.

Numeración de renglones y columnas

La numeración de los renglones se hizo de arriba hacia abajo, comenzando desde el 0 y terminando en 2, y la numeración de las columnas se hizo de izquierda a derecha de igual manera comenzando desde 0 y terminando en 2 (ver [Figura 11](#))

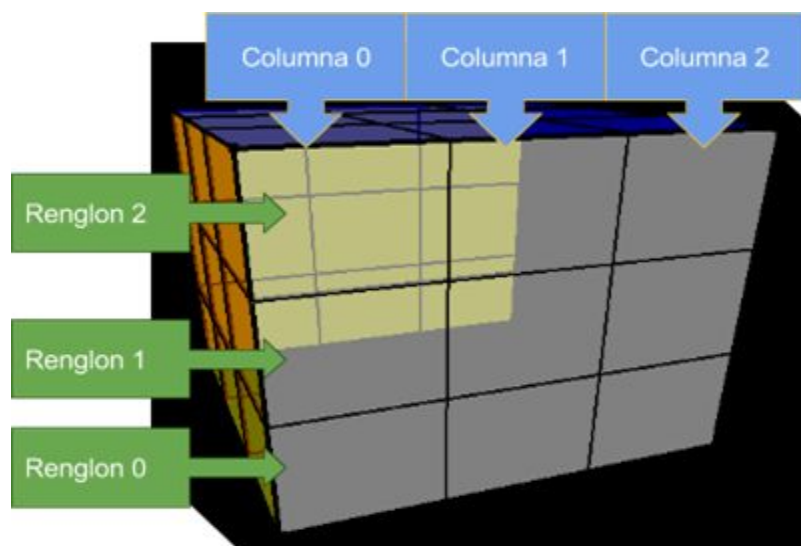


Figura 11 :Numeración de renglones y columnas

Descripción de la Función principal

```
int main( int argc, char **argv )  
  
{  
  
    cout<<"Cuantos cubos  
quieres(1-4):";  
  
    cin>>numeroCubos;  
  
    glutInit( &argc, argv );  
    glutInitDisplayMode( GLUT_RGBA |  
GLUT_DEPTH | GLUT_DOUBLE );  
  
    glutInitWindowSize( 640, 480 );  
  
    glutCreateWindow( "GLUT" );  
    glutDisplayFunc( display );  
  
    glutSpecialFunc( specialKeys );  
  
    glutKeyboardFunc(teclado);  
    glutMouseFunc(mouseClickHandler);  
  
    glutMotionFunc(motion);  
    glutPassiveMotionFunc(motion);  
  
    glEnable( GL_DEPTH_TEST );
```

En esta sección se pregunta al usuario cuantos cubos quiere para el juego

glutInit se utiliza para inicializar la biblioteca GLUT.

glutInitDisplayMode establece el *modo de visualización inicial*.

glutInitWindowSize establecen la posición de la ventana inicial y el tamaño respectivamente.

glutCreateWindow crea una ventana de nivel superior.

glutDisplayFunc establece la devolución de llamada de visualización de la *ventana actual*.

glutSpecialFunc establece la devolución de llamada teclado especial para la *ventana actual*.

glutMouseFunc establece la devolución de llamada del ratón para la *ventana actual*.

glutMotionFunc y glutPassiveMotionFunc establecer el movimiento y el movimiento pasivo, respectivamente, las devoluciones de llamada para la *ventana actual*.


```

    glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);

    glEnable( GL_BLEND );

    glutSetCursor(GLUT_CURSOR_NONE);


    init();

    inicializaLuz();

    glutTimerFunc(0, timer, 0);

    glutMainLoop();

    return 0;

```

glEnable - activar o desactivar funciones de GL del lado del servidor

glBlendFunc - especificar la aritmética de píxeles

glutSetCursor cambia la imagen del cursor de la *ventana actual*.

init() y inicializaLuz() se tratará más adelante

glutTimerFunc registra una devolución de llamada de temporizador, que se activará en un número especificado de milisegundos.

glutMainLoop entra en el bucle de procesamiento de eventos de GLUT.

Función init(); de la función principal main ()

- Gráfica el número de cubos que quiere el usuario (1-4)

Se utiliza la siguiente `dibujaVerticeLista()`

`dibujaVerticeLista()`

- Dibuja un cuadrado entero, con sus vértices respectivos formando estos polígonos

La función `dibujaVerticeLista()` usa las siguientes funciones importantes

- `glBegin(GL_QUADS);`
- `glVertex`
- `glEnd();`
- `glLineWidth(4.0);`
- `glBegin(GL_LINE_STRIP);`
- `glVertex` – specify a vertex

`glLineWidth` - Especificar el ancho de las líneas rasterizadas

Especifica el primitivo o primitivas que se crea a partir de los vértices que se presentan entre `glBegin` y la posterior `glEnd`. Se aceptan las diez constantes simbólicas:

`glBegin(GL_LINE_STRIP);` -Forma con los vértices un primitivo o un grupo de primitivas en este caso `GL_LINE_STRIP`

Función `display()` de la función principal `main ()`

`glutDisplayFunc(display);`

Descripcion de algunas de las funciones anexadas en el código:

moverAleatoriamenteCubos();

Esta función hace que los cubos se mueven aleatoriamente en la pantalla la pantalla, a través de un algoritmo

glClearColor(0.0,1.0,1.0, 1);

glClearColor - especificar valores claros para los buffers de color.

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

Limpia los valores prefijados bit a bit de las mask especificadas

glMatrixMode(GL_PROJECTION);

Especificar qué matriz es la matriz actual. Se aplica operaciones de matriz posteriores a la pila de matriz de proyección

glLoadIdentity();

Sustituir la matriz actual con la matriz de identidad.

glutGet(GLUT_WINDOW_WIDTH);

Recupera el estado de GLUT sencilla representada por números enteros.

Anchura en píxeles de la ventana actual.

glutGet(GLUT_WINDOW_HEIGHT);

Recupera el estado de GLUT sencilla representada por números enteros. Altura en píxeles de la ventana actual.

gluPerspective(60, w / h, 0.1, 300);

gluPerspective - establecer una matriz de proyección en perspectiva

gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

fovy

Especifica el ángulo del campo de visión, en grados, en la dirección y.

aspect

Especifica la relación de aspecto que determina el campo de visión en la dirección x. La relación de aspecto es la relación de x (anchura) a Y (altura).

zNear

Especifica la distancia del espectador con respecto al plano de delimitación cercano (siempre positivo).

zFar

Especifica la distancia del espectador con respecto al plano de delimitación lejano (siempre positivo).

glMatrixMode(GL_MODELVIEW);

Especificar qué matriz es la matriz actual. Aplica operaciones con matrices posteriores a la pila matriz MODELVIEW.

gluLookAt (40, 4, 10, 0, 0, 0, 0, 0, 1);

Definir una transformación de la imagen. Crea una matriz de visualización derivado de un punto de ojo, un punto de referencia que indica el centro de la escena, y un vector de UP.

```
void gluLookAt( GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ, GLdouble  
centerX, GLdouble centerY, GLdouble centerZ, GLdouble upX, GLdouble upY, GLdouble  
upZ);
```

eyeX, ojo, EYEZ

Especifica la posición del punto de vista.

centerx, centery, centery

Especifica la posición del punto de referencia.

UPX, Upy, UPZ

Especifica la dirección del vector de arriba.

glTranslatef(translate_x, translate_y, 0.0)

Multiplicar la matriz actual por una matriz de traducción

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

Specify the x , y , and z coordinates of a translation vector.

glTranslate

Produce una traducción hecha por x y z . La matriz actual (ver `glMatrixMode`) se multiplica por esta matriz de traducción, con el producto de sustitución de la matriz actual, como si `glMultMatrix` se llama con la siguiente matriz para su argumento:

cuboRubik();

glPointSize(7.0);

especificar el diámetro de los puntos rasterizadas

GENERAMIRA();

Esta función se explicará más adelante.

Contiene funciones como:

glMaterialfv(GL_FRONT, GL_AMBIENT, color[7]);

glMaterialfv(GL_FRONT, GL_DIFFUSE, color[7]);

glMaterialfv(GL_FRONT, GL_SPECULAR, color[7]);

glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);

Especifica los parámetros de materiales para el modelo de iluminación.

```
void glMaterialf( GLenum  face, GLenum  pname, GLfloat  param);
```

face

Especifica qué cara o caras están siendo actualizadas. Debe ser uno de GL_FRONT, GL_BACK, o GL_FRONT_AND_BACK.

pname

Especifica el parámetro del material de un solo valor de la cara o caras que se está actualizando. Debe ser GL_SHININESS.

param

Especifica el valor de ese parámetro GL_SHININESS se ajustará a.

glutSwapBuffers();

intercambia los buffers de la *ventana actual* si hay doble almacenamiento intermedio.

Algoritmos de luz

Función inicializaLuz()

```
void inicializaLuz(){
    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    GLfloat light_ambient[] = { 0.8, 0.8, 0.8, 1.0 };
    GLfloat light_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 0.0, 0.0, 500.0, 1.0 };

    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

}
```

Se utilizan las siguientes funciones:

- **glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);**

glLightModeli() fija el parámetro de modelo de iluminación. Pname nombres de parámetros de modelo de iluminación. Hay tres modelos de iluminación

GL_LIGHT_MODEL_LOCAL_VIEWER

Params es un valor entero o de coma flotante que especifica cómo se calculan los ángulos de reflexión especular. Si params es 0 (o 0,0), los ángulos de reflexión especular toman la dirección de la vista para ser paralelo a y en la dirección del eje z, independientemente de la ubicación del vértice en el ojo coordenadas. De lo contrario, las reflexiones especulares se calculan a partir del origen del sistema de coordenadas del ojo. El valor por defecto es 0.

● **glEnable(GL_LIGHTING);**

Si se habilita y ningún vertex shader está activo, utilice los parámetros de iluminación actuales para calcular el color de vértice o índice. De lo contrario, sólo tiene que asociar el color actual o índice con cada vértice.

● **glEnable(GL_LIGHT0);**

Si está activado, la luz incluir i en la evaluación del cociente de iluminación.

● **GLfloat light_ambient[] = { 0.8, 0.8, 0.8, 1.0};**

La intensidad de la luz ambiental RGBA.

● **GLfloat light_diffuse[] = { 0.8, 0.8, 0.8, 1.0};**

La intensidad de la luz difusa RGBA

● **GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0};**

La intensidad de la luz especular RGBA

● **GLfloat light_position[] = { 0.0, 0.0, 500.0, 1.0};**

(X, y, z, w) La posición de la luz.

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);

glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);

glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

glLightfv(GL_LIGHT0, GL_POSITION, light_position);

Como se puede ver, las matrices se definen por los valores de los parámetros, y glLightfv () es llamado varias veces para ajustar los distintos parámetros. En este ejemplo, las tres primeras llamadas a glLightfv () son superfluas, ya que están siendo utilizados para especificar los valores predeterminados para los parámetros GL_AMBIENT, GL_DIFFUSE, y GL_SPECULAR.

Estos parámetros interactúan con las que definen el modelo global de iluminación de una escena en particular y las propiedades del material de un objeto.

Algoritmo para aplicar transparencia al cubo.

OpenGL no admite una interfaz directa para la prestación de los primitivos translúcidos (parcialmente opacos). Sin embargo, se puede crear un efecto de transparencia con la función de BLEND y ordenar cuidadosamente sus datos primitivos.

Una aplicación OpenGL suele permitir la mezcla de la siguiente manera:

```
glEnable (GL_BLEND);  
  
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Después de que BLEND está habilitada, como se muestra arriba, el color primitivo entrante se mezcla con el color ya almacenada en el uso de este dispositivo. glBlendFunc () controla cómo se produce esta mezcla. El uso típico descrito anteriormente modifica el color de entrada por su valor alpha asociado y modifica el color de destino por uno menos el valor alfa entrante. La suma de estos dos colores se escribe de nuevo en el uso de este dispositivo.

La opacidad de la primitiva se especifica utilizando glColor4 (). RGB especifica el color, y el parámetro alpha especifica la opacidad.

Algoritmo para generar mira

```
void generaMira(){  
    glMaterialfv(GL_FRONT, GL_AMBIENT, color[7]);  
    glMaterialfv(GL_FRONT, GL_DIFFUSE, color[7]);  
    glMaterialfv(GL_FRONT, GL_SPECULAR, color[7]);  
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);  
  
    glColor3f(1.0,1.0,1.0);  
    set<int> listaMiras;  
    std::set<int>::iterator it;  
    for(int i=0;i<numeroCubos;i++){  
        if(tamanoCubos[i]>0){  
            listaMiras.insert(desplazamientosCubo[i][0]);  
        }  
    }  
  
    for (it=listaMiras.begin(); it!=listaMiras.end(); ++it){  
        glBegin(GL_LINE_LOOP);  
        for(int i =0; i <= 300; i++){  
            double angle = 2 * PI * i / 300;  
            double x = cos(angle);  
            double y = sin(angle);  
            glVertex3d(*it,miraX+2*x,miraY+2*y);  
        }  
    }
```



```
    glEnd();  
  }  
}
```

generaMira

Lo primero que hace es que le da color a todo lo que pinte a partir de ahí de negro, después lo que hace es hacer un conjunto de miras dependiendo del número de cubos que aún siguen en el juego y le da la profundidad de cada cubo.

Recordando que en el programa el eje x es decir la posición 0 del arreglo de desplazamientos es la profundidad, es un conjunto por si hay 2 cubos a la misma profundidad, luego va pintando un círculo a esa profundidad pero en la posición del mouse.

Resumiendo el proceso se asigna a cada cubo una mira, por lo que no con todas las miras se elimina se elimina un determinado cubo.

Se nos hizo más fácil diseñarlo de esta manera ya que aunque se vean que la mira le está dando al cubo del centro este puede que no esté a su misma profundidad, por eso se pintó una mira por cada cubo o en otras palabras la escala de la mira real que es la línea recta generada por el centro de todos los círculos de la mira.

Algoritmo para disparar

```
void mouseClickedHandler(int button, int state, int  
x, int y)  
{  
    myMouse(x,y);  
    for(int i=0;i<numeroCubos;i++){  
        if(pegarCubo(i)){  
            tamanoCubos[i]--;  
            rotacionesCubo[i].second+=20.0;  
        }  
    }  
    glutPostRedisplay();  
}
```

Función mouseClickHandler()

Aquí se utilizan las funciones:

- `glutPostRedisplay ()` -Marca la *ventana actual* como la necesidad de que se vuelva a mostrar.
- `pegarCubo()` -Evalúa si se ha disparado al cubo, se ser así se reduce el cubo, también el cubo va rotando cada vez que le dispaes.

Conclusiones

Para la realización de esta aplicación, como se mencionaba se tomaron en cuenta los conocimientos obtenidos durante el curso como la elaboración de primitivas, la rotación y traslación del conjunto de estas, el color, la sombra y la luz de los objetos que aparecen en escena, etc.

Como podemos darnos cuenta, la aplicación puede ejecutarse desde la consola, en donde se pedirá, se ingrese el número de cubos que se desea aparezcan en pantalla, posteriormente en las ventanas siguientes, podemos ver la cantidad de cubos seleccionados con anterioridad pero ahora en escena, para que estos sean destruidos por la mira.

En algún momento del desarrollo de este proyecto, se consideró la idea de hacer un cubo Rubik el cual fuera aprendiendo mediante movimientos y de esta manera este pudiera decirnos cómo armarse solo gracias al algoritmo de programación utilizado, pero esta idea fue descartada, porque el cubo no mostraba rotación ni traslación en los objetos que componían la escena de la aplicación.

Para ver ese proyecto también: <https://github.com/juanunam/proyectoGraficacion>.

No obstante se ocupando el diseño del cubo rubik ya elaborado y lo implementamos para poder utilizarlo en el Nuevo proyecto.

Posteriormente, se llegó a la conclusión de que la manera más fácil de poder implementar todos los puntos que se requerían para esta proyecto, era la realización del Shooter ya que este sí involucraba el movimiento (rotación y tralación), de sus elementos en escena.

Tipo de archivo : C++

Desarrollado en: C++11 con OpenGL

Sistema operativo en el que se realizó: Ubuntu de 64 Bits.

Peso del archivo: 30.0 KB

Repositorio GITHUB: <https://github.com/juanunam/disparadorOpenGL>