

Universidad Nacional Autónoma de
México

Facultad de estudios superiores Acatlán

Materia:

Sistemas Inteligentes.

Proyecto Final: Cubo Rubik

Proyecto en equipo de tres:

González Aguilar Juan Carlos

Introducción

El programa fue realizado con OpenGL. Aplicando los conocimientos de la Materia de Sistemas inteligentes y parte de los conocimientos adquiridos en Graficación por computadoras.

Como lo mencionamos el algoritmo aplicado es propio,

Los tipos de conocimiento que maneja son los siguientes:

- ❖ **Habitación:** Es una Respuesta que decae ante un Conjunto de Estímulos Repetidos (o Continuos) No Asociados a Ningún Tipo de Recompensa o Refuerzo. Implica Tendencia a Borrar todo Tipo de Respuesta ante un Estímulo que No tiene Importancia para la Supervivencia. Sirve como Filtro a un Conjunto de Estímulos No Relevantes.
- ❖ **Intuitiva:** Implica Copiar una Conducta, Acción o Expresión Nueva o que resulta Imprescindible de Aprender si no es Copiada de otro Individuo.

Tiene parte de Habitación porque como lo mencionamos antes el programa va aprendiendo de lo que nosotros le vamos enseñando cuando movemos las piezas y eliminando lo todo aquello que no le sirve (porque hay una opción más rápida de llegar a la solución) o es erróneo (no se llegó a una solución).

Índice

Introducción

Índice

Compilación

Requisitos

Compilación

Utilizar la aplicación

Ejemplos de rotaciones y movimientos en el cubo

Arquitectura de la aplicación

Sobre los archivos

Arquitectura de “generaCoordenadas.cpp “

Arquitectura de” pruebaMain.cpp”

Numeración de caras

Numeración de colores

Numeración de cada cuadro de cada cara

Colorear cada cuadro de las caras

Numeración de renglones y columnas

Algoritmos para girar una columna o renglón

Función giraDerecha y giralzquierda

Función giraArriba y giraAbajo

Algoritmo para guardar en el archivo

Función guardararchivo

Función guardarPasos

Algoritmo para aplicar transparencia al cubo.

Algoritmo para aprendizaje.

Conclusiones

Referencias OpenGL

Correos

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 códigos

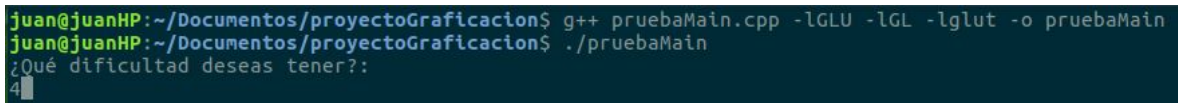
Compilación

1. Compilar código con las siguientes banderas:

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

2. Probar aplicación: Ejecutarla aplicación (Linux)

```
./pruebaMain
```



```
juan@juanHP:~/Documentos/proyectoGraficacion$ g++ pruebaMain.cpp -lGLU -lGL -lglut -o pruebaMain
juan@juanHP:~/Documentos/proyectoGraficacion$ ./pruebaMain
¿Qué dificultad deseas tener?:
4
```

Figura 1 : Vista del resultado de los comandos

Utilizar la aplicación

1. Elegir dificultad (ver [figura 1](#))
2. Para mover fragmentos del cubo clicar la sección de tres pequeños cuadros que queramos mover, manteniendo el click arrastrar el mouse hacia donde lo queramos mover hay dos tipos de movimientos que se pueden hacer con el mouse:
 - a. Horizontal: Se logra haciendo click y soltando en dos cuadros de la cara frontal del cubo que están en el mismo renglón (ver figura 2)
 - b. Vertical: Se logra haciendo click y soltando en dos cuadros de la cara frontal del cubo que están en la misma columna (ver figura 3).
3. Para “girar” todo el cubo hacia alguna de las direcciones usar teclas arrows (ver Figura 4)
4. Para usar el aprendizaje del programa usar leyenda “pista” que se muestra sobre el cubo, esta describe primero la acción a realizar y si es necesario el parámetro con el que se tiene que realizar la acción (ver [Figuras 3,4](#)).
5. En caso de no haber leyenda pista, significar que es una configuración no conocida para el programa, entonces intentar armar el cubo ya sea con algoritmos convencionales, para enseñar al programa a resolver esa configuración, algunas fuentes para aprender a hacerlo son:
 - a. <http://www.rubikaz.com/resolucion.php>
6. El paso mas importante, ¡diviértete!.

Ejemplos de rotaciones y movimientos en el cubo

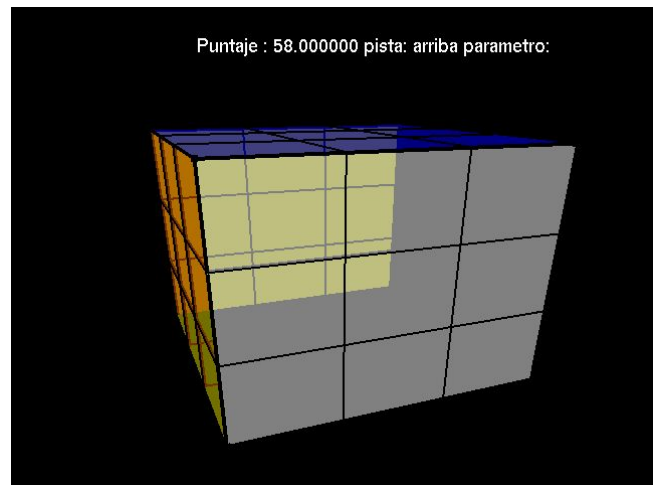
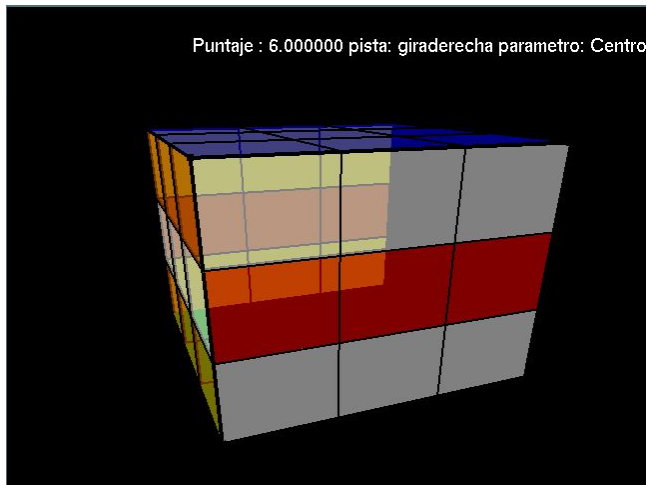


Figura 2 : Movimiento de girar derecha (Movimiento Horizontal) aplicado a renglón del centro

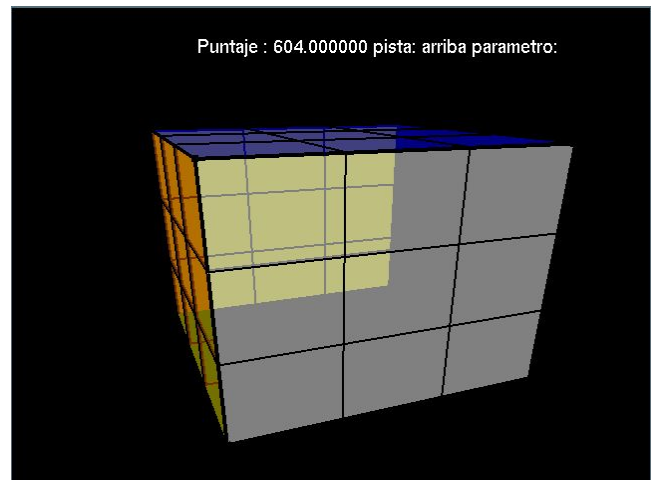
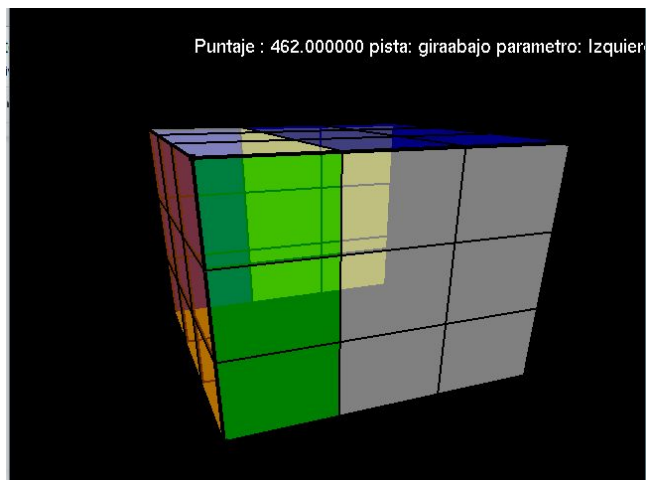


Figura 3 : Movimiento de girar abajo(Movimiento Vertical) aplicado a columna izquierda

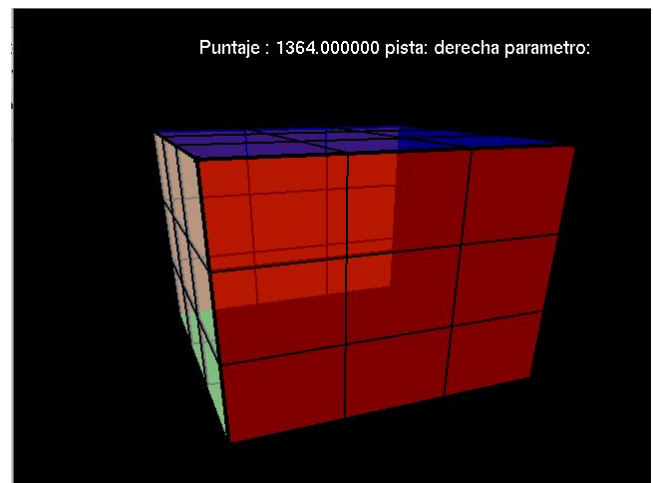
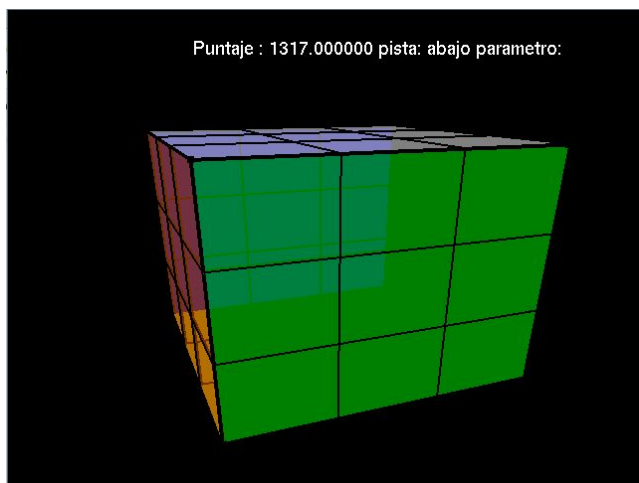


Figura 4 : Ejemplos de rotaciones de todo el cubo (Vertical y Horizontal)

Arquitectura de la aplicación

De manera general la aplicación consta de dos archivos principales: pruebaMain.cpp y mapa.txt, y a su vez existen otros dos archivos que se utilizaron para poder generar las coordenadas del cubo de manera fácil, se podría decir que solo son necesarios pruebaMain.cpp y mapa.txt, los otros dos archivos solo se incluyeron en el repositorio del proyecto, porque es un referente y un antecedente importante de la aplicación.

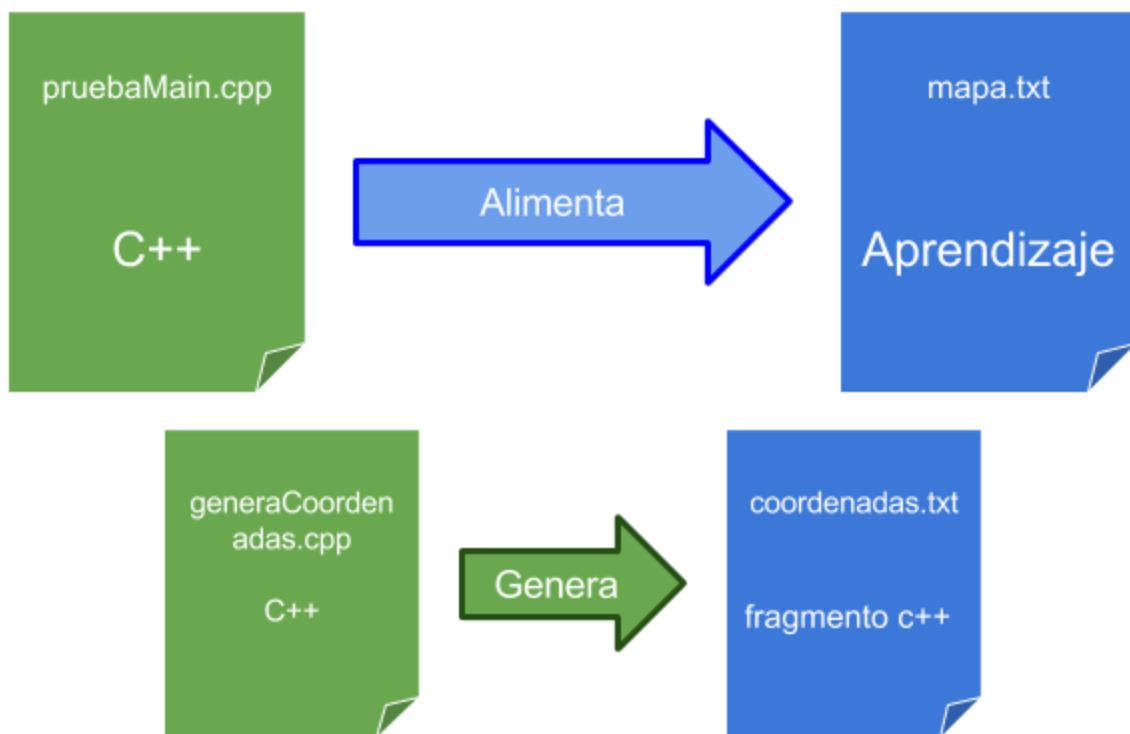


Diagrama 1: Diagrama explicando las relaciones entre los archivos

Sobre los archivos

La función de los código fuente es de servir de lógica para nuestra aplicación, o es un precedente de la principal (pruebaMain.cpp), más sin embargo, archivo mapa.txt representa el aprendizaje de la aplicación, si se borrara la aplicación empezará de cero, o si agregáramos líneas lógicas la aplicación entendería que sabe resolver algunos casos, a continuación una explicación sobre las arquitecturas de 2 de las aplicaciones.

Arquitectura de “generaCoordenadas.cpp”

Esta aplicación imprime una cadena, que representa las coordenadas de los nodos del cubo en un espacio 3D, lo que hace el programa es imprimirlo en orden de caras, además facilita la tarea de generar el cubo, este programa fué muy importante porque ahorró mucho tiempo, en cuanto a que no se tuvo que escribir uno por uno cada posición del cubo, la forma de compilarlo y de correrlo es la siguiente:

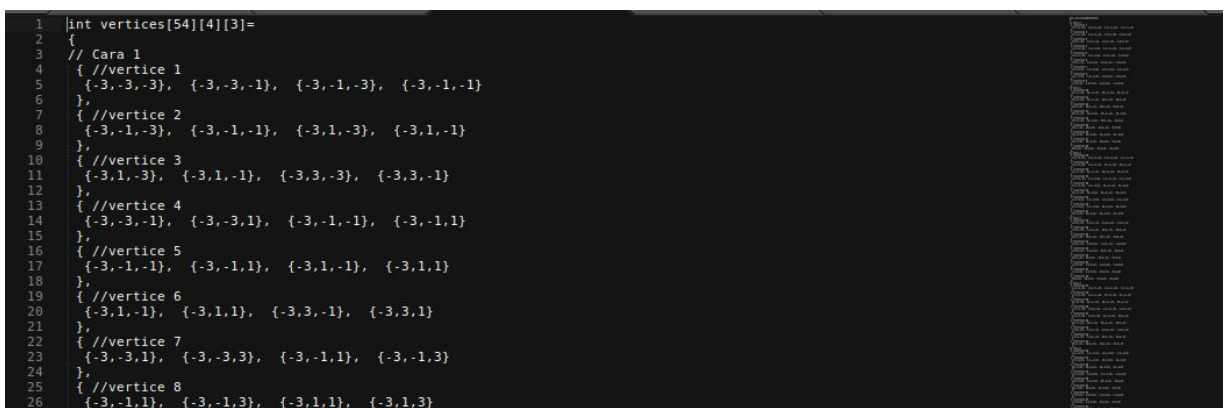
#Así compilamos

```
$g++ generaCoordenadas.cpp -o generaCoordenadas
```

#Así corremos

```
$. ./generaCoordenadas > coordenadas.txt
```

Con el parámetro > le decimos a la aplicación que guarde la salida del programa en dicho archivo, esto para luego poder extraer el código generado de una manera sencilla (ver [Figura 5](#)).



```
1 |int vertices[54][4][3]=
2 |{
3 |// Cara 1
4 |{ //vertex 1
5 |{-3,-3,-3}, {-3,-3,-1}, {-3,-1,-3}, {-3,-1,-1}
6 |},
7 |{ //vertex 2
8 |{-3,-1,-3}, {-3,-1,-1}, {-3,1,-3}, {-3,1,-1}
9 |},
10|{ //vertex 3
11|{-3,1,-3}, {-3,1,-1}, {-3,3,-3}, {-3,3,-1}
12|},
13|{ //vertex 4
14|{-3,-3,-1}, {-3,-3,1}, {-3,-1,-1}, {-3,-1,1}
15|},
16|{ //vertex 5
17|{-3,-1,-1}, {-3,-1,1}, {-3,1,-1}, {-3,1,1}
18|},
19|{ //vertex 6
20|{-3,1,-1}, {-3,1,1}, {-3,3,-1}, {-3,3,1}
21|},
22|{ //vertex 7
23|{-3,-3,1}, {-3,-3,3}, {-3,-1,1}, {-3,-1,3}
24|},
25|{ //vertex 8
26|{-3,-1,1}, {-3,-1,3}, {-3,1,1}, {-3,1,3}
```

Figura 5 :Contenido de archivo coordenadas.txt después de la corrida de generaCoordenadas

Arquitectura de "pruebaMain.cpp"

En cuanto pruebaMain.cpp este consta de distintos componentes que hacen posible su funcionamiento, estos son :

- a) Numeración de caras.
- b) Numeración de colores
- c) Numeración de cada cuadro de cada cara
- d) Colorear cada cuadro de las caras.
- e) Numeración de renglones y columnas
- f) Algoritmos para girar una columna o renglón.
- g) Algoritmo para guardar en el archivo
- h) Algoritmo de aprendizaje
- i) Algoritmo para aplicar transparencia al cubo

Numeración de caras

Sobre las caras se decidió que la posición de las caras nunca cambiará y entonces en realidad lo único que cambiaría sería el color de los cuadros sobre ellas, dando una ilusión de movimiento y además facilitando la programación del algoritmo de aprendizaje.

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 6](#)) 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
- Negro -> 7

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.

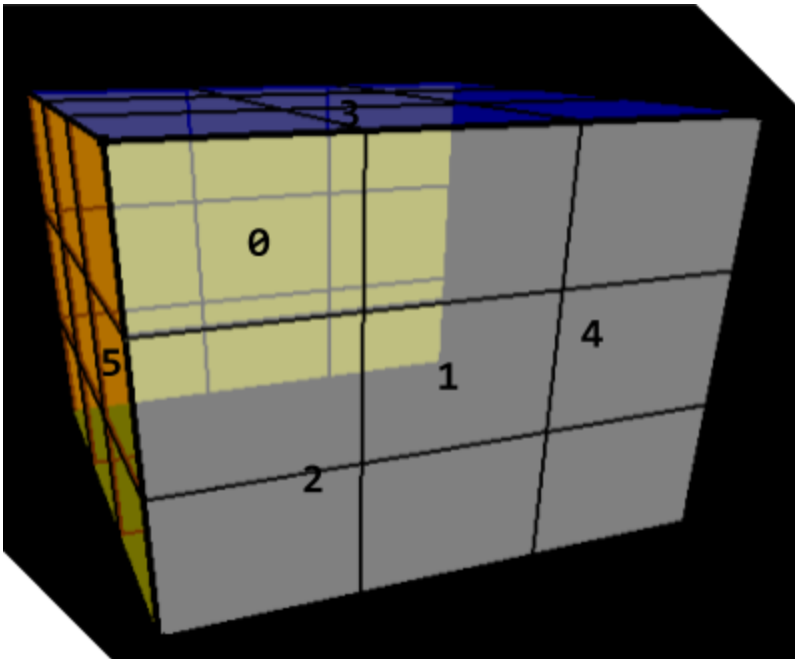


Figura 6 :Numeración de caras elegido

Numeración de cada cuadro de cada cara

La numeración de cada cara del cuadro se hizo de la siguiente manera, explicada en la siguiente figura:

						7	8	9						
						4	5	6						
						1	2	3						
7	8	9	7	8	9	7	8	9	7	8	9			
4	5	6	4	5	6	4	5	6	4	5	6			
1	2	3	1	2	3	1	2	3	1	2	3			
						7	8	9						
						4	5	6						
						1	2	3						

Lo cual implica que las rotaciones y/o movimientos del cubo van a depender de esta numeración, mientras movimientos como el de mover el cubo horizontalmente parecieran sencillos, movimientos como los verticales comienzan a tener su complejidad ya que por ejemplo si quisiéramos hacer una rotación hacia arriba en la columna derecha implicaría:

Lo cual programacional-mente hablando hace más complejo nuestro problema, pero de todos los

						7	8	9				
						4	5	6				
						1	2	3				
7	8	9	7	8	9	7	8	9	7	8	9	
4	5	6	4	5	6	4	5	6	4	5	6	
1	2	3	1	2	3	1	2	3	1	2	3	
						7	8	9				
						4	5	6				
						1	2	3				

Dirección de la rotación

Dirección de la rotación

modelos posibles analizados este es de los más sencillos de trabajar, ya que solo son casos donde se complica el trabajo con este modelo, y no en general todos los casos.

Colorear cada cuadro de las caras

Para saber de que 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, 0, //Cara 0
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, //Cara 1
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, //Cara 2
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, //Cara 3
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, //Cara 4
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5 // Cara 5
};
```

Eso quiere decir que inicialmente la cara 1 es de amarillo blanco, 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 no siempre van a ser los mismos, y este arreglo es el que nos dice que configuración de cubo tenemos, cuando rotamos o hacemos cualquier otra acción, lo que cambia es este array, lo que hace que el cubo se pinte de distinta manera.

Si queremos saber cual es el color de el cuadro n, de la cara m bastaria con hacer

```
colorCuadro=colores[m*9+n]
```

donde n esta dado de 0-9 (recordando que en c++ los arreglos comienzan desde 0).

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 7](#))

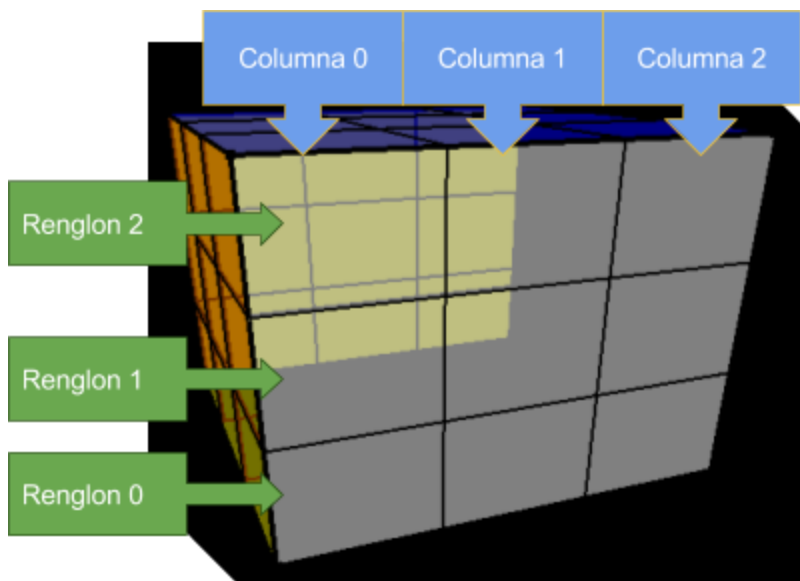


Figura 7 :Numeración de renglones y columnas

Algoritmos para girar una columna o renglón

Como ya vimos en la sección donde numeramos los cuadros de las caras del cubo, en orden de simplicidad las rotaciones por columnas son las más complicadas de programar, pero primero hay que tomar en cuenta varios factores:

- a) ¿Como gira un cubo de rubik real?
- b) ¿Se puede determinar de una manera algorítmica?¿Como?
- c) ¿Como hacer que algo que funciona para un caso funcione para todos?

La respuesta para todas las preguntas anteriores se fueron resolviendo a la marcha, lo primero que se programó fue el giraDerecha con parámetro 1 (centro), ya que como veremos enseguida girar el piso o el techo y en general cualquier columna o renglón que este junto a más de una cara se vuelve en un problema complejo.

Función giraDerecha y giralzquierda

Estas funciones reciben como parámetro el renglón donde se va a aplicar.

```
void giraDerecha(int renglon){
    renglon=2-renglon;
    permutaVerticesRenglon(1,4,renglon);
    permutaVerticesRenglon(5,1,renglon);
    permutaVerticesRenglon(5,0,renglon);
    int numero=-1;
    int contador=2;
    if(renglon==0){
        numero=2*9; contador=6;
    }
    else if(renglon==2)
        numero=3*9;
    while(numero!=-1 and contador--){
        int auxiliar=colores[numero+0];
        colores[numero+0]=colores[numero+3];
        ;
        int auxiliar2=colores[numero+1];
        colores[numero+1]=auxiliar;
        auxiliar=colores[numero+2];
        colores[numero+2]=auxiliar2;
        auxiliar2=colores[numero+5];
        colores[numero+5]=auxiliar;
        auxiliar=colores[numero+8];
        colores[numero+8]=auxiliar2;
        auxiliar2=colores[numero+7];
        colores[numero+7]=auxiliar;
        auxiliar=colores[numero+6];
        colores[numero+6]=auxiliar2;
        auxiliar2=colores[numero+3];
        colores[numero+3]=auxiliar;
    }
}
```

El algoritmo giraDerecha consta de permutaciones entre caras en el renglón que tiene como argumento, el código de permutaVerticesRenglon es:

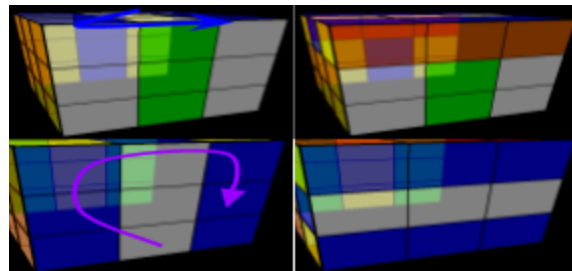
```
void permutaVerticesRenglon(int a,int b,int renglon){
    a=a*9; b=b*9;
    for(int i=renglon*3;i<3*(renglon+1);i++){
        int aux=colores[a+i];
        colores[a+i]=colores[b+i];
        colores[b+i]=aux;
    }
}
```

De manera muy sencilla es fácil ver que esta función no tiene su complejidad solo intercambia los colores de 1 renglón de 2 caras diferentes, como gira derecha intercambia de la siguiente manera:

1<->4 , 5<->1 ,5<->0

lo cual da como resultado: 0=4,1=5,4=1,5=0

lo que según la sección de [Numeración de caras](#) sería el equivalente a un giro a la derecha. pero también habría que rotar el piso y el techo cuando el valor fuera 0 o 2 en el renglón, lo cual se programó casi de manera manual, se hizo el mismo trozo de código para ambos casos, solo cambiando las repeticiones;



giralzquierda=3xgiraDerecha

Función giraArriba y giraAbajo

Estas funciones reciben como parámetro la columna donde se va a aplicar

```
void giraArriba(int columna){
    permutaVerticesColumna(3,0,columna);
    permutaVerticesColumna(1,3,columna);
    permutaVerticesColumna(2,1,columna);
    int numero=-1; int contador=6 ;
    if(columna==0){
        numero=5*9; contador=2;
    }
    else if(columna==2)
        numero=4*9;
    while(numero!=-1 and contador--){
        int auxiliar=colores[numero+0];
        colores[numero+0]=colores[numero+3];
        int auxiliar2=colores[numero+1];
        colores[numero+1]=auxiliar;
        auxiliar=colores[numero+2];
        colores[numero+2]=auxiliar2;
        auxiliar2=colores[numero+5];
        colores[numero+5]=auxiliar;
        auxiliar=colores[numero+8];
        colores[numero+8]=auxiliar2;
        auxiliar2=colores[numero+7];
        colores[numero+7]=auxiliar;
        auxiliar=colores[numero+6];
        colores[numero+6]=auxiliar2;
        auxiliar2=colores[numero+3];
        colores[numero+3]=auxiliar;
    }
}
```

El algoritmo **giraArriba** consta de permutaciones entre caras en el la columna que tiene como argumento, el código de permutaVerticesColumna es:

```
void permutaVerticesColumna(int a,int b,int columna){
    int originalA=a; a=a*9; b=b*9;
    for(int i=columna;i<9;i=i+3){
        if(originalA==3 && b==0){
            int aux=colores[a+i];
            colores[a+i]=colores[b+8-i];
            colores[b+8-i]=aux;
        }else{
            int aux=colores[a+i];
            colores[a+i]=colores[b+i];
            colores[b+i]=aux;
        }
    }
}
```

Esta función a diferencia de la re los renglones es más compleja porque la numeración cambia cuando cambiamos de cara, eso se menciona en la parte de [Numeración de caras](#), las permutaciones que hace son:

3<->0 , 1<->3 ,2<->1

lo cual da como resultado: 0=3,1=2,2=0,3=1

Lo que según la sección de [Numeración de caras](#) sería el equivalente a un giro a la haia arriba. pero también habría que rotar el lado derecho o izquierdo en los casos 0 y 2, lo cual se resuelve parecido a lo visto en la sección anterior con giraDerecha;

giraAbajo=3xgiraArriba;

Algoritmo para guardar en el archivo

Esta parte de nuestro programa es de suma importancia ya que aquí se va almacenado el conocimiento o lo aprendido por nuestro cubo.

Función guardararchivo

Esta función se crea el archivo mapa.txt de escritura, en el caso de que no existiese o se sobrescribe si existe.

```
void guardarArchivo(){
    std::map<string,string>::iterator it;
    FILE* archivoTres;
    // cout<<"sa"<<endl;
    archivoTres = fopen("mapa.txt", "w");
    // cout<<"sa"<<endl;
    for ( it = mapaAccion.begin(); it!=mapaAccion.end(); ++it){
        string numerosDelMapa=it->first;
        // cout<<numerosDelMapa<<endl;
        fprintf(archivoTres, "%s %d %d
%s\n",numerosDelMapa.c_str(),mapaCaminos[numerosDelMapa],mapa
Parametro[numerosDelMapa],mapaAccion[numerosDelMapa].c_str()
    );
    }
    fclose(archivoTres);
}
```


Función guardarPasos

En esta parte utilizamos el manejo de las pilas, para cada que se realiza un movimiento este se va almacenando, para que sea más fácil al momento de rastrearlo.

```
void guardarPasos(){

    string pasosCadena="";
    string numeracionCadena="";
    int tamanoPila=vectorPasos.size();
    for (int i = 0; i < tamanoPila; ++i)
    {
        numeracionCadena=vectorColores.top();
        pasosCadena=vectorPasos.top();
        if(mapaCaminos[numeracionCadena]!=0){
            if(mapaCaminos[numeracionCadena]>i+1){
                mapaCaminos[numeracionCadena]=i+1;

mapaParametro[numeracionCadena]=vectorParametros.top();
                mapaAccion[numeracionCadena]=vectorPasos.top();
            }
        }else{
            mapaCaminos[numeracionCadena]=i+1;
            mapaParametro[numeracionCadena]=vectorParametros.top();
            mapaAccion[numeracionCadena]=vectorPasos.top();
            printf("Aprendi!!\n");
        }
        vectorPasos.pop();
        vectorColores.pop();
        vectorParametros.pop();
    }
    guardarArchivo();
}
```

Se va Introduciendo el conjunto de números (cada cual corresponde a un color) correspondientes a cada cara, ejemplo.

000000004 411111111 333333033 422222222 555555555 442444143 8 0 giraderecha

El estado guardado en el archivo de texto de una configuración del cubo Rubick ya guardada en el pasado de una partida por otro jugador y que puedes tomar para llegar a un resultado en un determinado número de pasos. Son los primeros seis conjuntos de números correspondiente a cada cara del cubo. Estos son introducidos a un archivo de texto llamado "mapa.txt"

000000004 411111111 333333033 422222222 555555555 442444143 8 0 giraderecha

las últimas características representan:

el primer número (en este caso es en 8 puede ser 120 tal vez) es el camino que podemos tomar, en éste caso si tomamos este camino tendremos un camino con longitud de 8 pasos.

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:

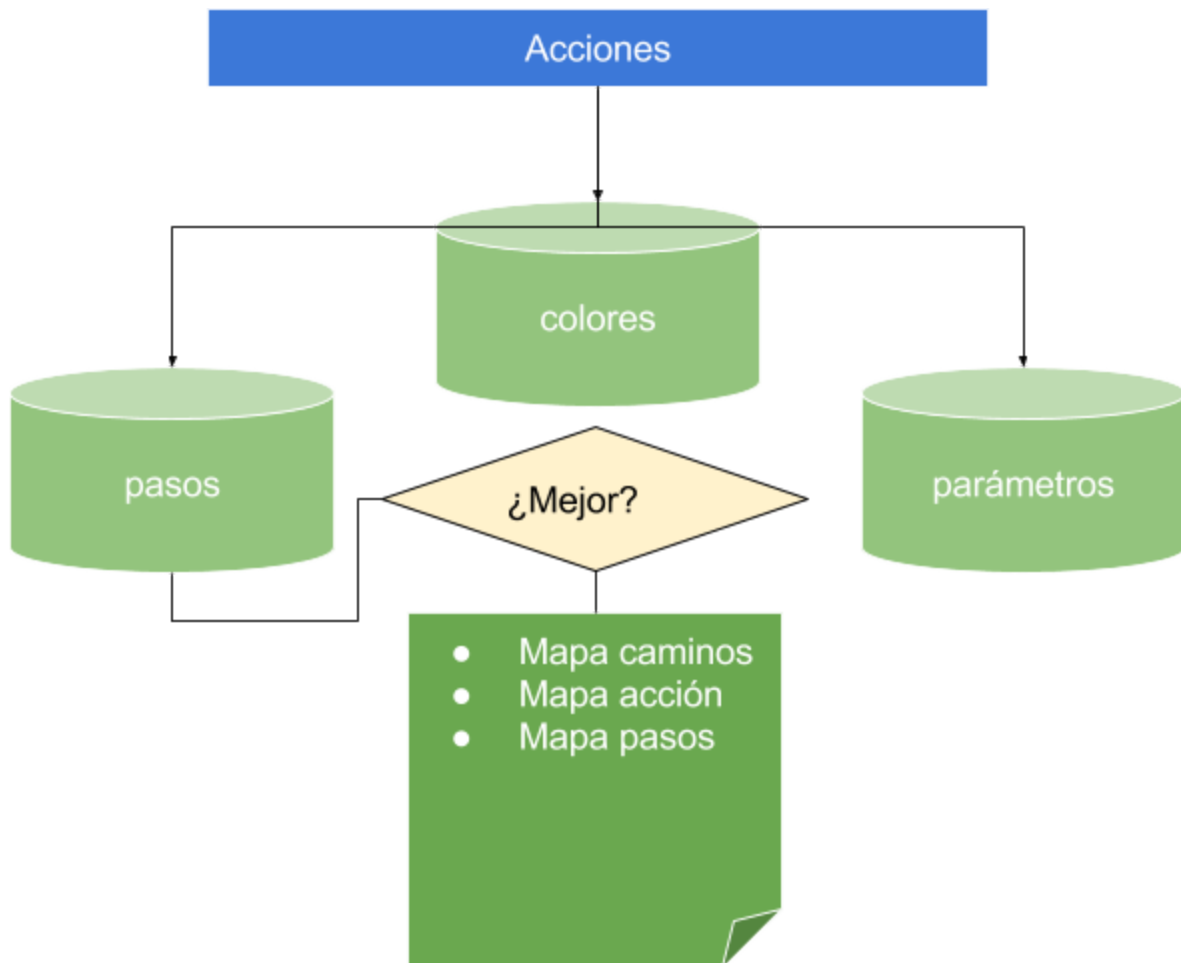
```
glEnable (GL_BLEND);  
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

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 aprendizaje.

El aprendizaje en la aplicación se abordó de una manera compleja estructuralmente pero sencilla en cuanto a su evaluación, había muchas opciones para hacer que el cubo nos sugiriera movimientos para resolverlo, por ejemplo una opción era guardar los movimientos hechos cuando se desarmaba el cubo para así solo aplicar **backtracking** y así sugerir el antecesor como movimiento a seguir teniendo la certeza de que llegaríamos a la solución final pero, esta no se nos hizo una solución innovadora o que requiriera de inteligencia de parte de la aplicación, así que recordando algunos conceptos sobre aprendizaje de máquina se optó por generar aprendizaje a partir de las personas que ya puedan armarlo (Nosotros), los únicos conocimientos que requería entonces la aplicación sería saber cuando un cubo está armado.



Lo primero que hace la aplicación al iniciar es revisar si tiene conocimiento aprendido

```
while(file>>lineaNumeros){  
    file>>peso;  
    file>>parametro;  
    file>>lineaAccion;  
    mapaCaminos[lineaNumeros]=peso;  
    mapaParametro[lineaNumeros]=parametro;  
    mapaAccion[lineaNumeros]=lineaAccion;  
}
```

Lo almacena en los mapas, asignándoles como llave la configuración de colores asociada a la configuración del cubo que tenía en el momento que hizo esa acción que resulto ser parte de una ruta que llego a la solución final del cubo, lo cual demostraremos mas tarde.

Después de este paso nuestros mapas cuentan con el conocimiento que tenemos, ha que decir que cuando el cubo hace un movimiento todos los movimientos se ven almacenados en las pilas de la ilustración de arriba, ya que todos pasan por la siguiente función:

```
void relizarAccion(string accion,int parametro){  
    . . .  
    vectorPasos.push(accion);  
    vectorParametros.push(parametro);  
    vectorColores.push(convierteColores());  
    . . .  
}
```

Lo cual va creando una “bitacora” de lo que está haciendo el usuario, una vez que llega a la solución final pasa por la función:

```
void guardarPasos(){
    string pasosCadena="";string numeracionCadena="";int tamanoPila=vectorPasos.size();
    for (int i = 0; i < tamanoPila; ++i)
    {
        numeracionCadena=vectorColores.top();
        pasosCadena=vectorPasos.top();
        if(mapaCamino[s numeracionCadena]!=0){
            if(mapaCamino[s numeracionCadena]>i+1){
                mapaCamino[s numeracionCadena]=i+1;
                mapaParametro[s numeracionCadena]=vectorParametros.top();
                mapaAccion[s numeracionCadena]=vectorPasos.top();
            }
        }else{
            mapaCamino[s numeracionCadena]=i+1;
            mapaParametro[s numeracionCadena]=vectorParametros.top();
            mapaAccion[s numeracionCadena]=vectorPasos.top();
            printf("Aprendi!!\n");
        }
        vectorPasos.pop();vectorColores.pop();vectorParametros.pop();
    }
    guardarArchivo();
}
```

En la función guardarPasos() se analiza si la solución actual, desde la configuración de cada paso ya existía en nuestro archivo de conocimiento (“mapa.txt”), esto lo hace con el if:

```
if(mapaCamino[s numeracionCadena]!=0){
    if(mapaCamino[s numeracionCadena]>i+1){
        mapaCamino[s numeracionCadena]=i+1;mapaParametro[s numeracionCadena]=vectorParametros.top();
        mapaAccion[s numeracionCadena]=vectorPasos.top();
    }
}
```

Cuando se da cuenta que ya existía analiza si es más eficiente que la que tenía, esto lo hace comparándolo con lo que sea recorrido de la pila, sabemos que ese es el tamaño del camino hacia la solución por obvias razones, si fuera mas eficiente reemplaza su lugar en el mapa, lo cual implica que también su lugar en el archivo.

En caso de ser una nueva solución a este le basta con introducir la solución en el mapa

```
mapaCamino[numeracionCadena]=i+1;

mapaParametro[numeracionCadena]=vectorParametros.top();

mapaAccion[numeracionCadena]=vectorPasos.top();
```

Lo cual garantiza su almacenamiento en la memoria de la aplicación.

Por último para hacer sugerencias, bastaría en todo momento con revisar si la configuración actual de colores del cubo existe en el mapa, como ya sabemos en el mapa solo estan las configuraciones del cubo de las cuales se conoce la ruta ganadora o que resuelve el cubo en otras palabras por lo tanto se reduce a las siguientes líneas de código:

```
if(mapaCamino[numeroColores]!=0){

    accionPosible = mapaAccion[numeroColores];

    int parametroNumero=mapaParametro[numeroColores];

    if(accionPosible == "giraizquierda" || accionPosible == "giraderecha"){

        parametroCadena=parametroNumero==0?"Techo":(parametroNumero==1?"Centro":"Piso");

    }else if(accionPosible == "giraarriba" || accionPosible == "giraabajo"){

        parametroCadena=parametroNumero==0?"Izquierda":(parametroNumero==1?"Centro":"Derecha");

    }else{

        parametroCadena="";

    }

    cadena=cadena+" pista: "+accionPosible+" parametro: "+parametroCadena;

}
```

Que de manera simple solo es una evaluación en uno de los mapas , si este devuelve algo diferente a 0 quiere decir que conocemos la solución.

Conclusiones

La manera más eficiente que encontramos para resolver el problema del objetivo de nuestra aplicación que es que supiera armar el cubo, fue que aprendiera a hacerlo, porque en realidad la programación no es tan complicada a diferencia de otras alternativas que analizamos que era usar un algoritmo ya conocido, lo cual no hacía que nuestro cubo se volviera más eficiente con el tiempo, o generar el árbol de búsqueda que para nuestro problema era imposible computacionalmente por la cardinalidad del número de permutaciones diferentes en un cubo rubik.

Aprendizaje de máquina es una excelente opción cuando tenemos problemas de este tipo donde el árbol de soluciones es no computable y donde no existe un método óptimo para todos los casos, es una excelente opción cuando las aplicaciones tienen de donde aprender, en este caso de nosotros mismos, probablemente el paso a seguir sería ponerlo a correr con un programa que sepa armar el cubo desde cualquier punto, para que absorbiera su información sin necesidad de programar ni de ver en su código.

Referencias OpenGL

- <http://sabia.tic.udc.es/gc/Tutorial%20OpenGL/tutorial/>
- <http://www.lighthouse3d.com/tutorials/glut-tutorial/>
- <http://www.glprogramming.com/red/>

Correos

carlosgonzagular@gmail.com

Tipo de archivo : C++

Desarrollado en: C++11

Creado con: OpenGL

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

Peso del archivo: 85.0 KB (87,127 bytes)

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