

CUBO RUBIK

Mauricio Carazas Segovia





Las dependencias son las siguientes:

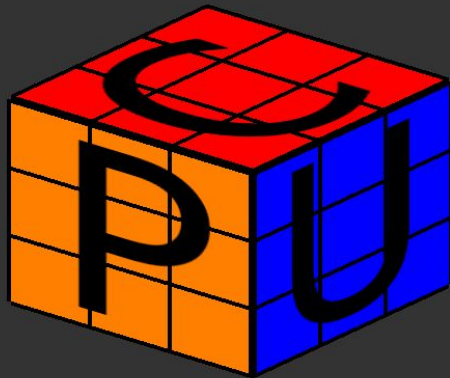
GLFW
CONTENTO
GLM
FREEIMAGE
FREEGLUT
GLEW

D:\LabCompiler_CG\glfw-master\OwnProjects\rubick_finalista\main.cpp - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Folder as Workspace

LearnOpenGL



Con la tecla K Se resolvera el cubo

> - cac
> - CMakeFiles
 CMakeLists.txt
 cmake_install.cmake
 Cubo.h
 glad.c

```
35     "    TexCoord = aTexCoord;\n"36     "}\n";3738     const char *fragmentShaderSource = "#version 330 core\n"39     "out vec4 FragColor;\n"40     "in vec3 ourColor;\n"
```

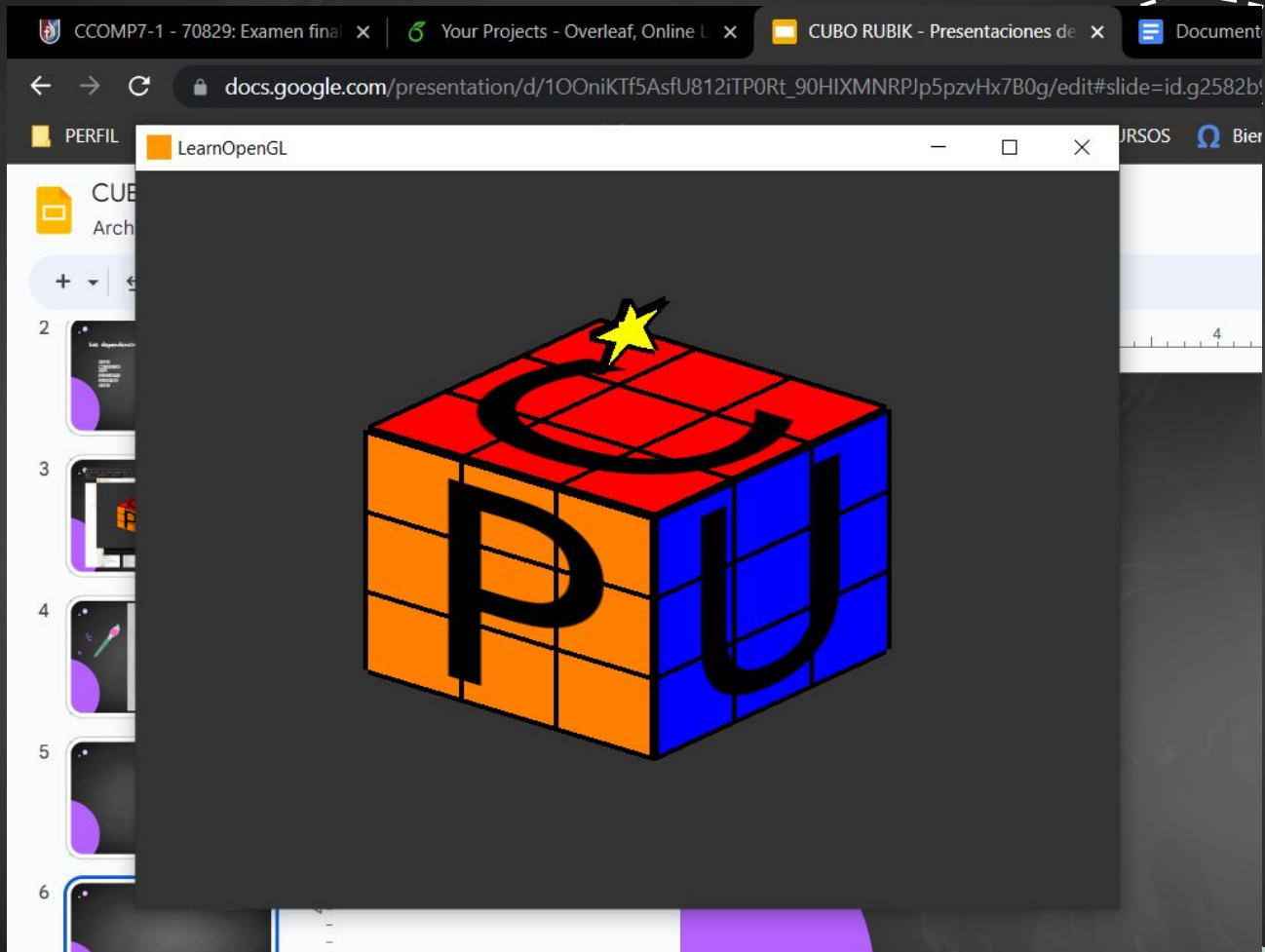


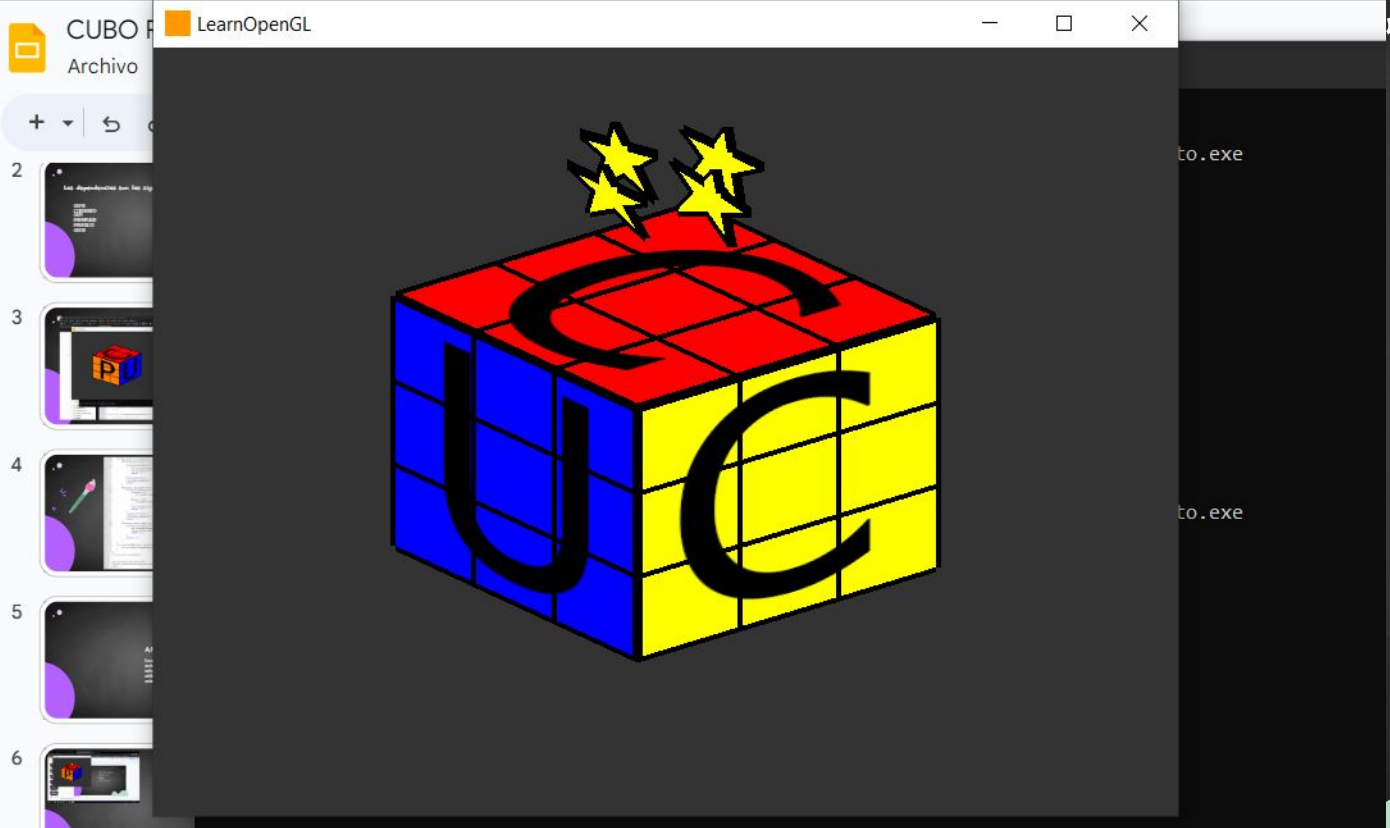
```
81 string animacion_estrella(int pos_negativo, string eje){
82     if(estado_estrella=="M"){
83         //if(reg_movimientos!=2700){//0.002
84         if(reg_movimientos!=2000){
85             pol_estrella->nob_translate(0, velocidad_animacion, 0);
86             reg_movimientos+=1;
87             return "M";
88         }
89         reg_movimientos=0;
90         velocidad_animacion=0.03;
91         return "G";
92     }
93     if(estado_estrella=="G"){
94         if(reg_movimientos<3000){
95             if(eje=="X"){
96                 pol_estrella->pro_rotation_x(pos_negativo * velocidad_animacion);
97             }
98             if(eje=="Z"){
99                 pol_estrella->pro_rotation_z(pos_negativo * velocidad_animacion);
100            }
101            reg_movimientos+=1;
102            return "G";
103        }
104        reg_movimientos=0;
105        velocidad_animacion=0.003;
106        return "B";
107    }
108    if(estado_estrella=="B"){
109        if(reg_movimientos!=2000){
110            pol_estrella->nob_translate(0, -velocidad_animacion, 0);
111            reg_movimientos+=1;
112            return "B";
113        }
114        return "N";
115    }
116 }
117 void draw(GLFWwindow* window, bool wired=false){
118     pol_estrella->draw(window, wired);
119 }
120 };
121 int pos_vect_estrellas=0;
122
123
124 vector<estrella*> estrellas;
125 Polygon estrella(vertices_estrella, 1, 1, 0);
126 Rubik c;
127
```



ANIMACIÓN

La animación de las estrellas en este código se implementa utilizando la estructura estrella. Cada objeto estrella tiene sus propios atributos y métodos para controlar su animación.







ANIMACIÓN

Cada objeto estrella tiene un estado de animación que puede ser "M" (movimiento), "G" (giro) o "B" (movimiento inverso).

El método `animacion_estrella` se encarga de realizar la animación de la estrella en función de su estado actual y los parámetros recibidos. La animación se divide en tres fases: movimiento, giro y movimiento inverso. Dependiendo del estado actual de la estrella, se realizan diferentes operaciones de traslación y rotación en el objeto Polygon asociado.


```

219 void processInput(GLFWwindow* window, int key, int scancode, int action, int mods){
220     if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
221         glfwSetWindowShouldClose(window, true);
222     ///////////////////////////////////////////////////////////////////
223     if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS){
224         pitch -= (cameraSpeed );
225     }
226     if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS){
227         pitch += (cameraSpeed );
228     }
229     if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS){
230         yaw += (cameraSpeed );
231     }
232     if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS){
233         yaw -= (cameraSpeed );
234     }
235     ///////////////////////////////////////////////////////////////////
236     if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS && condition_input=="N"){
237         reg_mov.push_back("R");
238         cout<<"INICIA ANIMACION R"<<endl;
239         condition_input="R";
240     }
241     if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS && condition_input=="N"){
242         reg_mov.push_back("L");
243         cout<<"INICIA ANIMACION L"<<endl;
244         condition_input="L";
245     }
246     if (glfwGetKey(window, GLFW_KEY_U) == GLFW_PRESS && condition_input=="N"){
247         reg_mov.push_back("U");
248         cout<<"INICIA ANIMACION U"<<endl;
249         condition_input="U";
250     }
251     if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS && condition_input=="N"){
252         reg_mov.push_back("D");
253         cout<<"INICIA ANIMACION D"<<endl;
254         condition_input="D";
255     }
256     if (glfwGetKey(window, GLFW_KEY_F) == GLFW_PRESS && condition_input=="N"){
257         reg_mov.push_back("F");
258         cout<<"INICIA ANIMACION F"<<endl;
259         condition_input="F";
260     }
261     if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS && condition_input=="N"){
262         reg_mov.push_back("B");
263         cout<<"INICIA ANIMACION B"<<endl;
264         condition_input="B";
265     }
266 }

```



Funcionalidades

Control de movimiento de cámara: Si se presionan las teclas de flecha hacia arriba (GLFW_KEY_UP), hacia abajo (GLFW_KEY_DOWN), hacia la izquierda (GLFW_KEY_LEFT) o hacia la derecha (GLFW_KEY_RIGHT), los valores de pitch y yaw se ajustan según la velocidad de la cámara (cameraSpeed). Estas variables probablemente se utilizan para controlar la orientación de la cámara y se actualizan para lograr el movimiento deseado.

Animaciones: Si se presionan ciertas teclas (R, L, U, D, F, B) y la variable condition_input tiene un valor específico ("N"), se agrega una cadena correspondiente ("R", "L", "U", etc.) al vector reg_mov y se imprime un mensaje indicando el inicio de una animación particular. Además, se actualiza condition_input con el valor de la tecla presionada para indicar que se ha iniciado una animación.

```

int main(){

    estrellas.reserve(12);
    for(int i=0;i<12;++i){
        estrella *estrellas_point=new estrella;
        estrellas.push_back(estrellas_point);
    }

    // glfw: initialize and configure
    // -----
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // glfw window creation
    // -----
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL){
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // -----
    if (!gladLoadGL(glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }
    //-----
    // vertex shader
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);
    // check for shader compile errors
    int success;
    char infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success){
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
}

```

Funcion MAIN

El código main proporcionado es la función principal de un programa en C++ que utiliza OpenGL y GLFW para crear una ventana de renderizado y realizar operaciones gráficas en 3D. A continuación, se explica el flujo de ejecución del código:

- Se reserva espacio en memoria para un vector llamado estrellas que almacenará punteros a objetos de la clase estrella.
- Se inicializa GLFW y se configuran algunas opciones para la ventana de renderizado.
- Se crea la ventana utilizando la función glfwCreateWindow, especificando el ancho, alto y título de la ventana.
- Se verifica si la creación de la ventana fue exitosa. Si no, se muestra un mensaje de error y se termina el programa.
- Se establece la ventana creada como el contexto de renderizado actual y se configura una función de devolución de llamada para cambiar el tamaño del framebuffer.
- Se carga la biblioteca GLAD para obtener los punteros a las funciones OpenGL.
- Se crean los shaders de vértices y fragmentos y se compilan.
- Se verifica si hubo errores de compilación en los shaders.
- Se crea un programa de shaders, se adjuntan los shaders compilados y se enlazan.
- Se verifica si hubo errores de enlace en el programa de shaders.

Funcion MAIN

- Se habilita la prueba de profundidad y la textura 2D, y se elimina los shaders ya que ya no son necesarios.
- Se generan los objetos de búfer de vértices (VBO) y de arreglo de vértices (VAO) para almacenar los datos de vértices y atributos necesarios para el renderizado.
- Se establecen los atributos de posición, color y coordenadas de textura en los datos del búfer de vértices.
- Se genera y carga una textura desde un archivo de imagen.
- Se muestra un menú con instrucciones para interactuar con el programa.
- Comienza el bucle principal del programa (`while (!glfwWindowShouldClose(window))`), que se ejecuta hasta que se solicite el cierre de la ventana.
- Dentro del bucle principal, se manejan las entradas del teclado y se realizan animaciones según las teclas presionadas.
- Se borra el búfer de color y de profundidad y se configura la vista de la cámara.
- Se dibujan los objetos en el búfer de visualización, incluyendo el cubo (`c.draw(window)`) y las estrellas (`estrellas[i]->draw(window)`).
- Se intercambia el búfer de renderizado y se procesan los eventos de la ventana.
- Cuando se sale del bucle principal, se liberan los recursos utilizados por los búferes y se termina GLFW.


Este código muestra un programa básico que crea una ventana de renderizado, carga shaders, texturas y realiza animaciones gráficas utilizando OpenGL y GLFW.

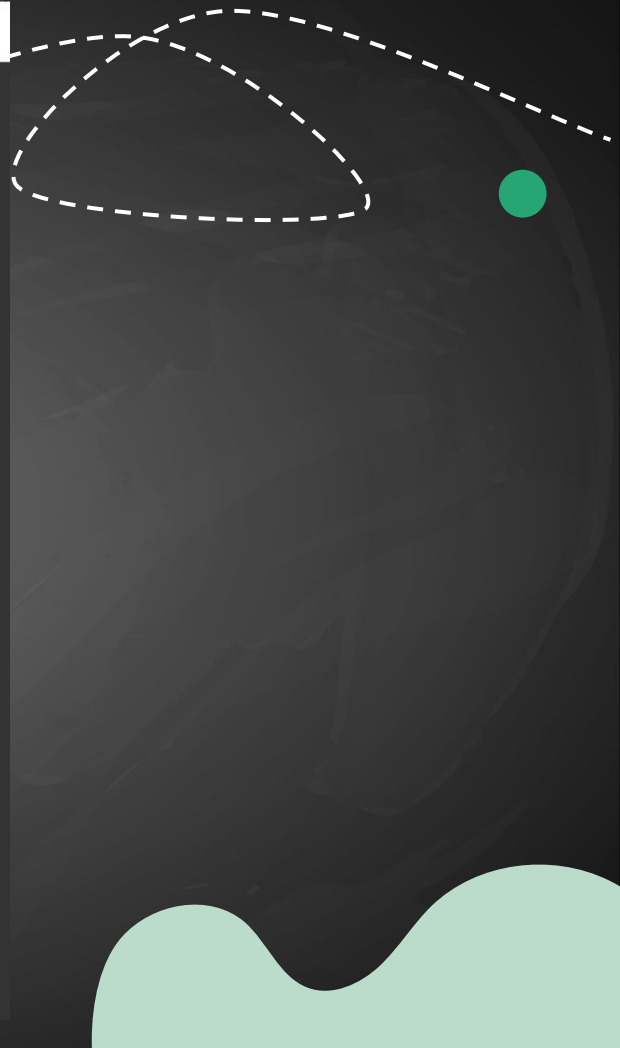
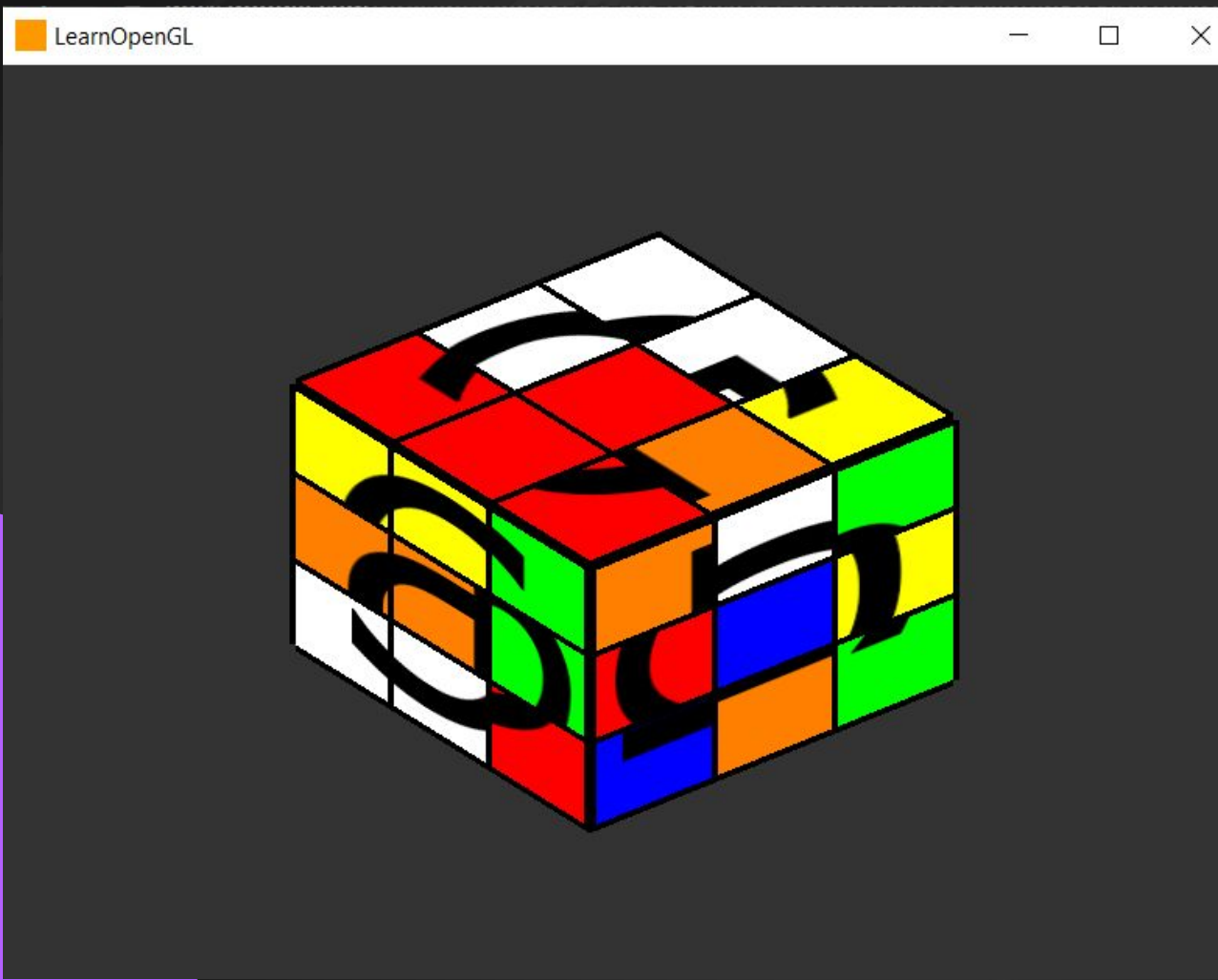


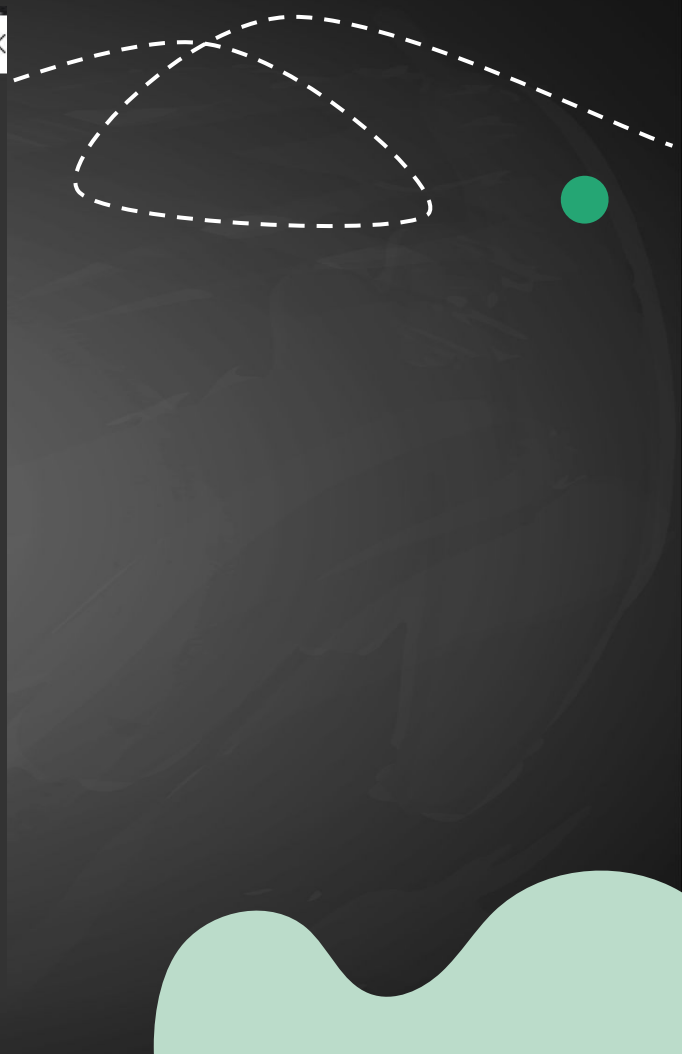
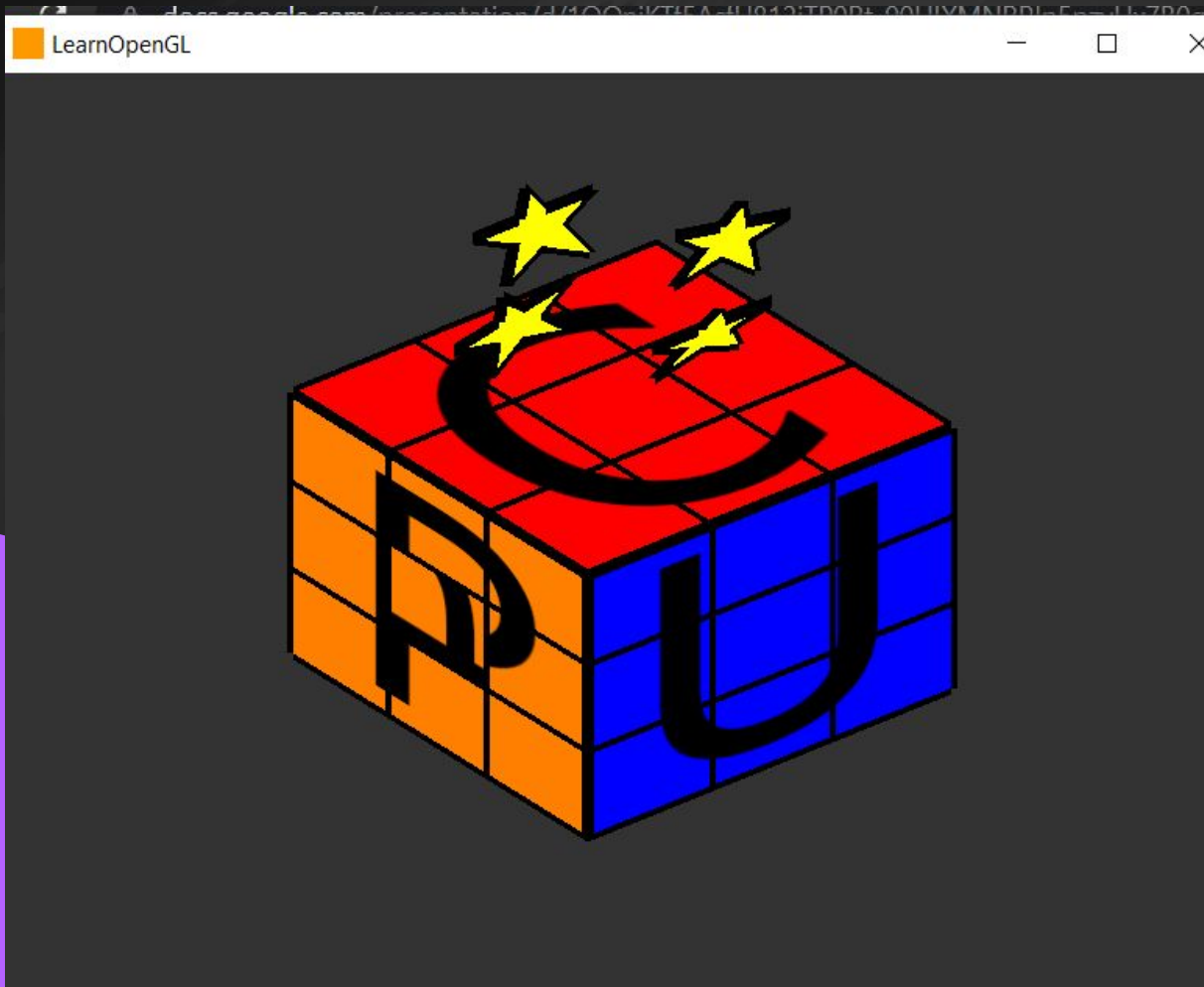
SOLVER



Utilizamos el algoritmo de dos fases de Kociemba modificado para obtener la solución en 20 movimientos o menos, que se encuentra [aquí](#)









<https://github.com/mauriciocarazas/Computaci-n-Gr-fica>