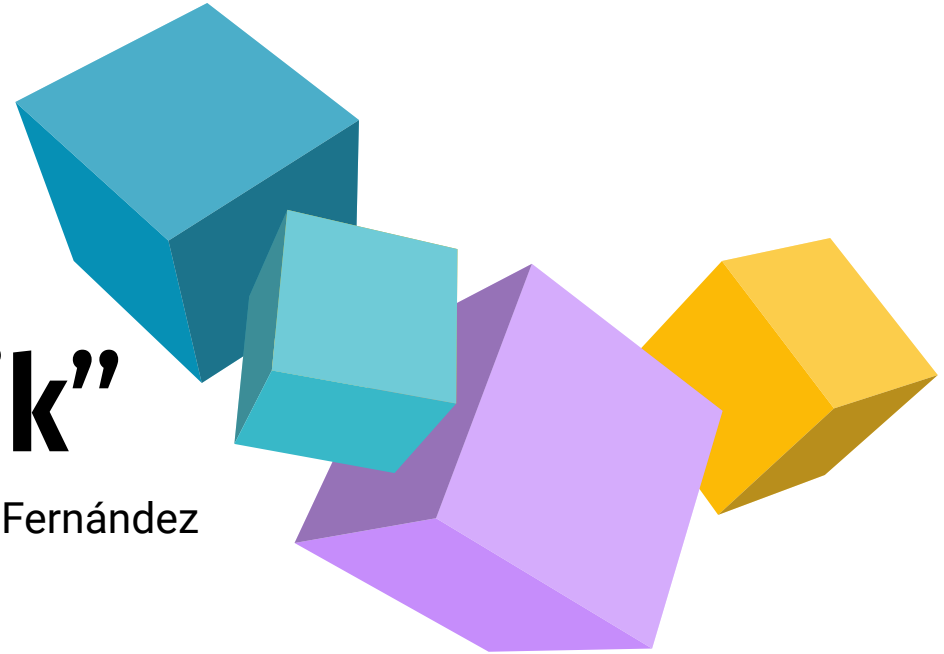
Proyecto “Cubo Rubik”

Andrés Cusirramos - Anthony Fernández



Computación Gráfica - CCOMP - 2022-1

ÍNDICE DE CONTENIDOS

1

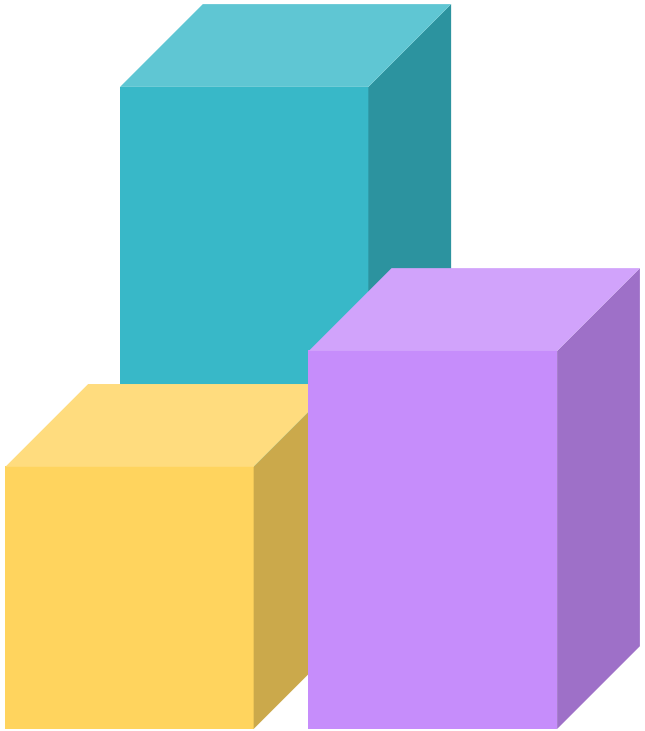
Estructura de Datos

2

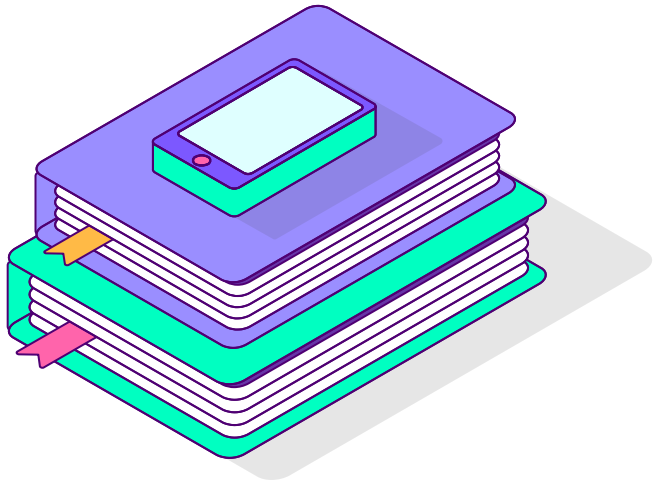
Animaciones

3

Conclusiones



1. ESTRUCTURA DE DATOS



01

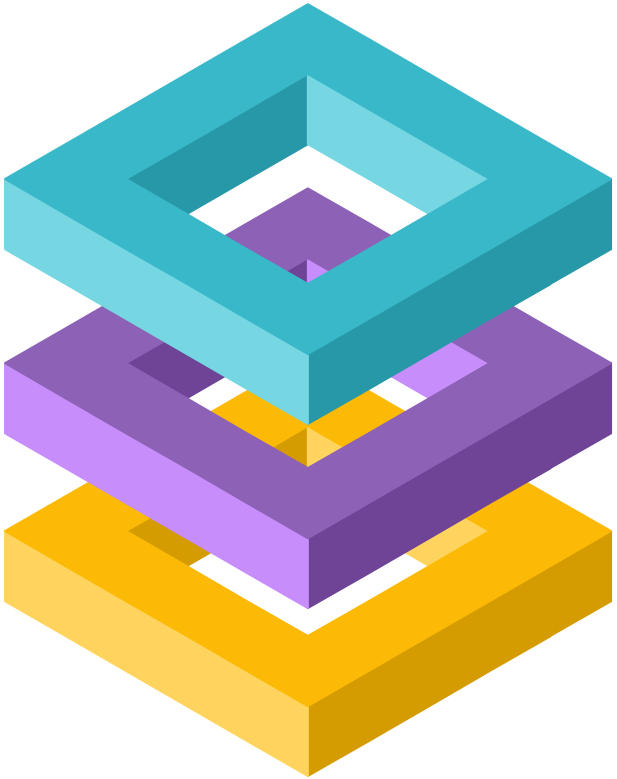
Shader: vertex - fragment, texture, perspective.

02

CCube: instancias de SLittleCube, establece parámetros, genera movimientos, realiza animaciones.

03

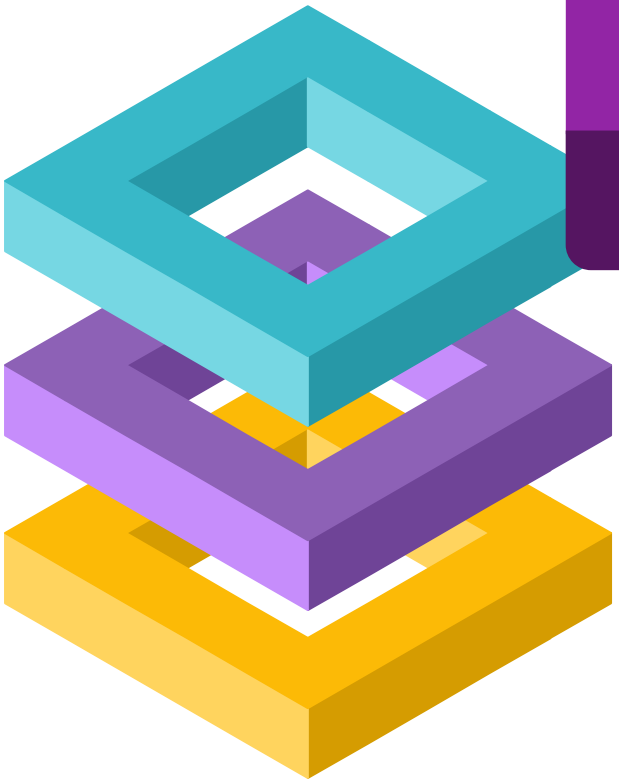
SLittleCube: vértices, colores y texturas, parámetros de animación.



01

Shader: vertex - fragment, texture, perspective.

```
class Shader
{
public:
    unsigned int ID;
    // constructor generates the shader on the fly
    // ------------------------------------------------------------------------
    Shader(const char* vertexPath, const char* fragmentPath, const char* geometryPath = nullptr){ ... }
    // activate the shader
    // ------------------------------------------------------------------------
    void use(){ ... }
    // ...
    void setBool(const std::string& name, bool value) const{ ... }
    // ------------------------------------------------------------------------
    void setInt(const std::string& name, int value) const{ ... }
    // ------------------------------------------------------------------------
    void setFloat(const std::string& name, float value) const{ ... }
    // ------------------------------------------------------------------------
    void setVec2(const std::string& name, const glm::vec2& value) const{ ... }
    void setVec2(const std::string& name, float x, float y) const{ ... }
    // ------------------------------------------------------------------------
    void setVec3(const std::string& name, const glm::vec3& value) const{ ... }
    void setVec3(const std::string& name, float x, float y, float z) const{ ... }
    // ------------------------------------------------------------------------
    void setVec4(const std::string& name, const glm::vec4& value) const{ ... }
    void setVec4(const std::string& name, float x, float y, float z, float w){ ... }
    // ------------------------------------------------------------------------
    void setMat2(const std::string& name, const glm::mat2& mat) const{ ... }
    // ------------------------------------------------------------------------
    void setMat3(const std::string& name, const glm::mat3& mat) const{ ... }
    // ------------------------------------------------------------------------
    void setMat4(const std::string& name, const glm::mat4& mat) const{ ... }
```



Vectores

Vertices, IPLittleCubes
createLittleCubes(),
drawCube(...)

+

Movimientos

moveR/L/U/D/F/B(...)

+

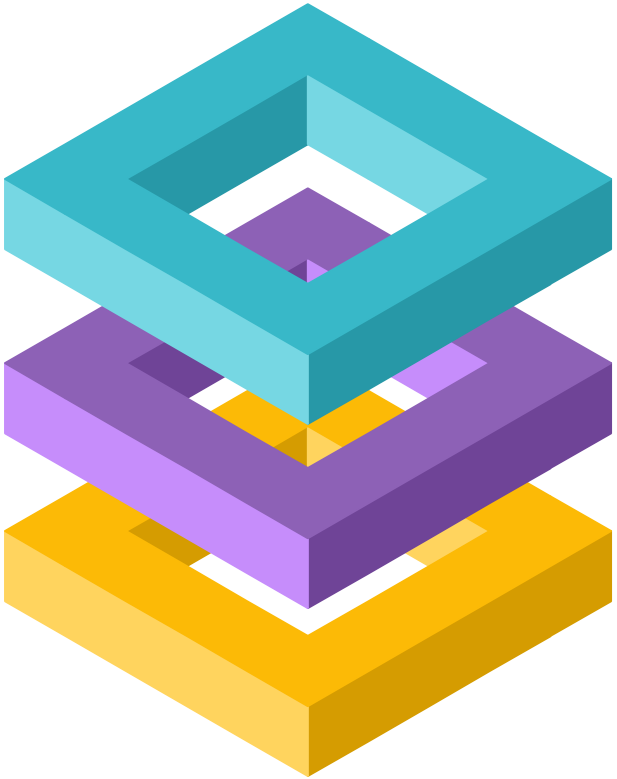
Animaciones

animation1/2(...)

CCube

02

CCube: instancias de SLittleCube, establece parámetros, genera movimientos, realiza animaciones.

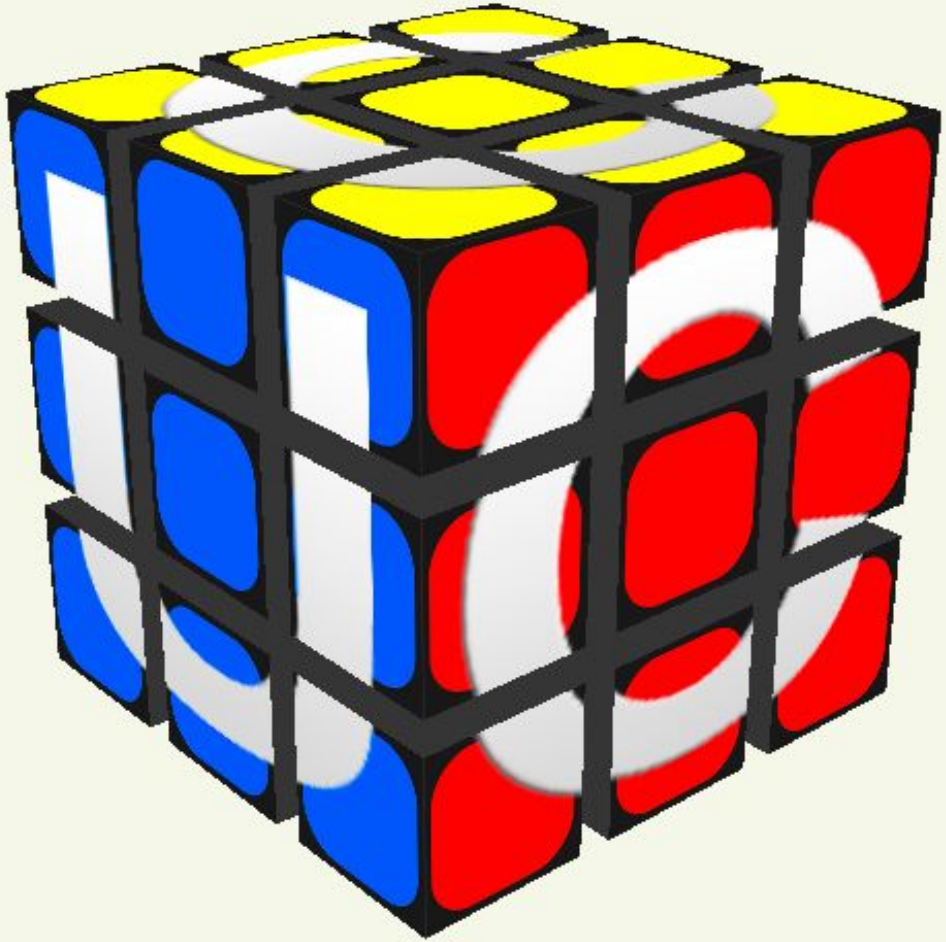


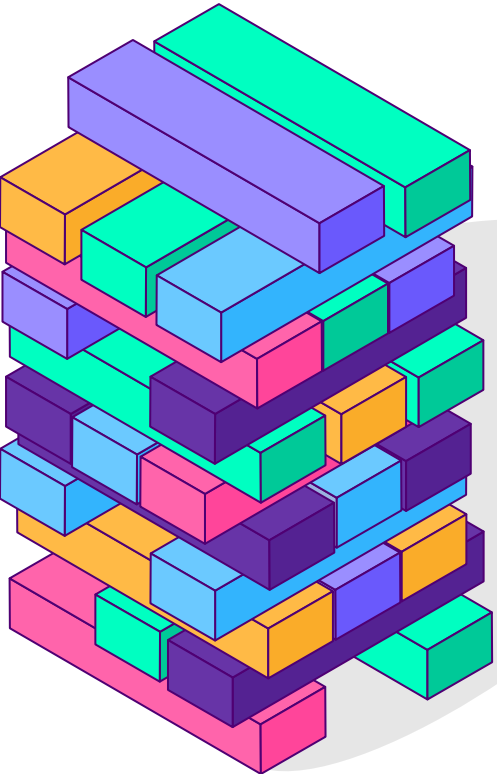
```
struct SLittleCube
{
    int id;
    glm::vec3 initialPosition;
    vector<float> vertices;
    vector<vector<float>> listOfColors;
    vector<char> camadas;
    vector<float> textureIndices;
    vector<float> animation2Parameters; //ac - alfa - b0
    glm::mat4 modelo;
    glm::vec3 centro_cubo;
    SLittleCube(int _id, glm::vec3 iP, vector<float> _vertices) {
        id = _id;
        initialPosition = iP;
        vertices = _vertices;
    };
    void Alejarse(glm::vec3 centro, float distancia);
};
```

03

SLittleCube: vértices, colores y texturas, parámetros de animación.

Valores propios de cada instancia generan una estructura completa.

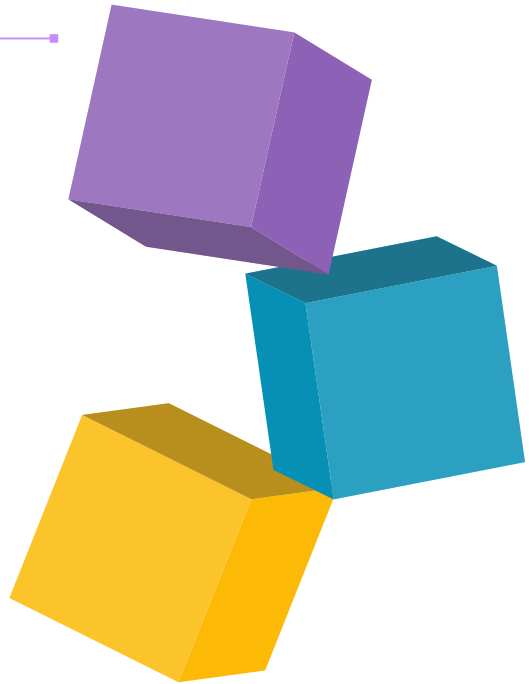




2. ANIMACIONES

A

Movimientos para solución del Cubo



B

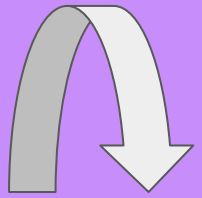
Animación de la Cámara

C

Animaciones de los cubos

main.cpp

vector<char> movementsList
Solver

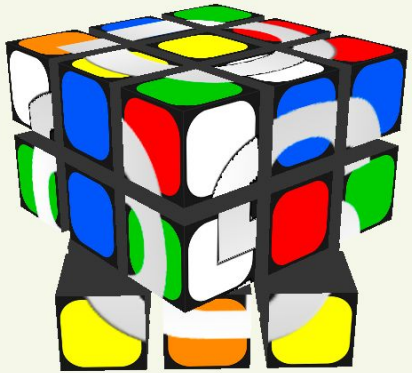


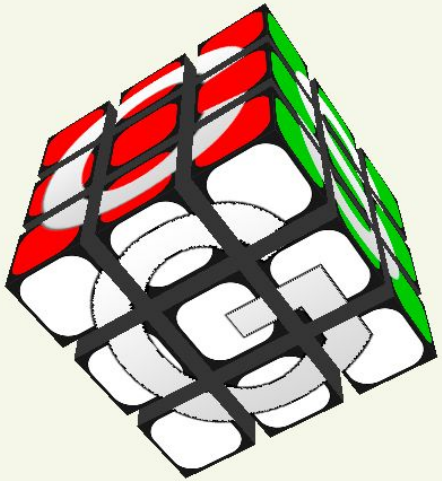
- ❏ Variar angulo
- ❏ Activar movimiento
- ❏ Actualizar camadas
- ❏ Corregir centros

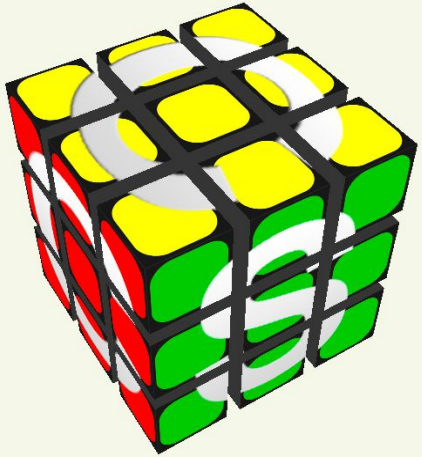
cube_structure.h

- ❖ Identificar cubos por camada
- ❖ Generar rotaciones

transformations.h



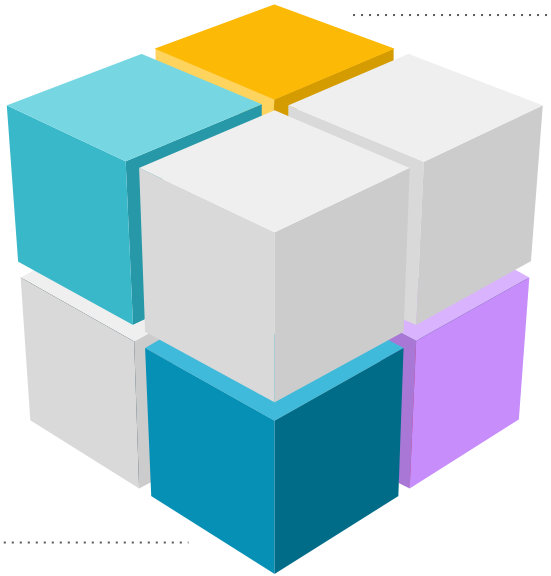




```
if (activateCameraX) {
    camY = sin(glfwGetTime()) * xRadius;
    camX = cos(glfwGetTime()) * xRadius;
    camZ = 4.0;
    cameraPos = glm::vec3(camX, camY, camZ);
}
if (activateCameraY) {
    camZ = cos(glfwGetTime()) * yRadius;
    camY = sin(glfwGetTime()) * yRadius;
    camX = 4.0;
    cameraPos = glm::vec3(camX, camY, camZ);
}
if (activateCameraZ) {
    camX = sin(glfwGetTime()) * zRadius;
    camZ = cos(glfwGetTime()) * zRadius;
    camY = 4.0;
    cameraPos = glm::vec3(camX, camY, camZ);
}
if (activateEllipticalCamera) {
    if (cycleForEllipC) {
        if (camY < 18.0) { ... }
        else
            cycleForEllipC = false;
    }
    else {
        if (camY > -18.0) { ... }
        else
            cycleForEllipC = true;
    }
    camX = cos(glfwGetTime()) * zRadius;
    camZ = sin(glfwGetTime()) * zRadius;
    cameraPos = glm::vec3(camX, camY, camZ);
}
```

Respiración de cubos

Transformaciones para la Respiración

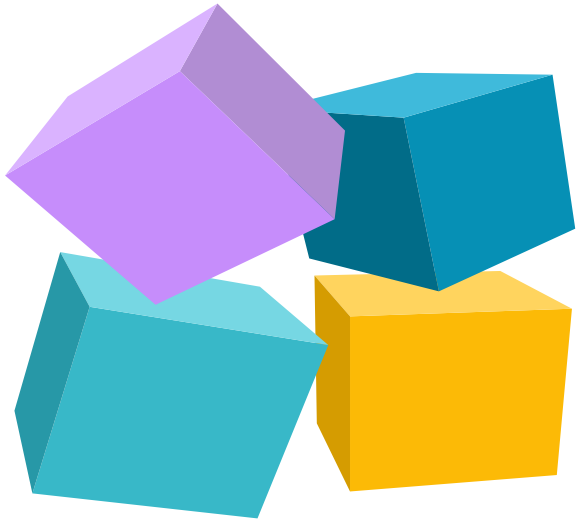


Traslacion

Variar la distancia entre los cubitos

Escala

Aumentar el tamaño de los cubitos



Render Loop

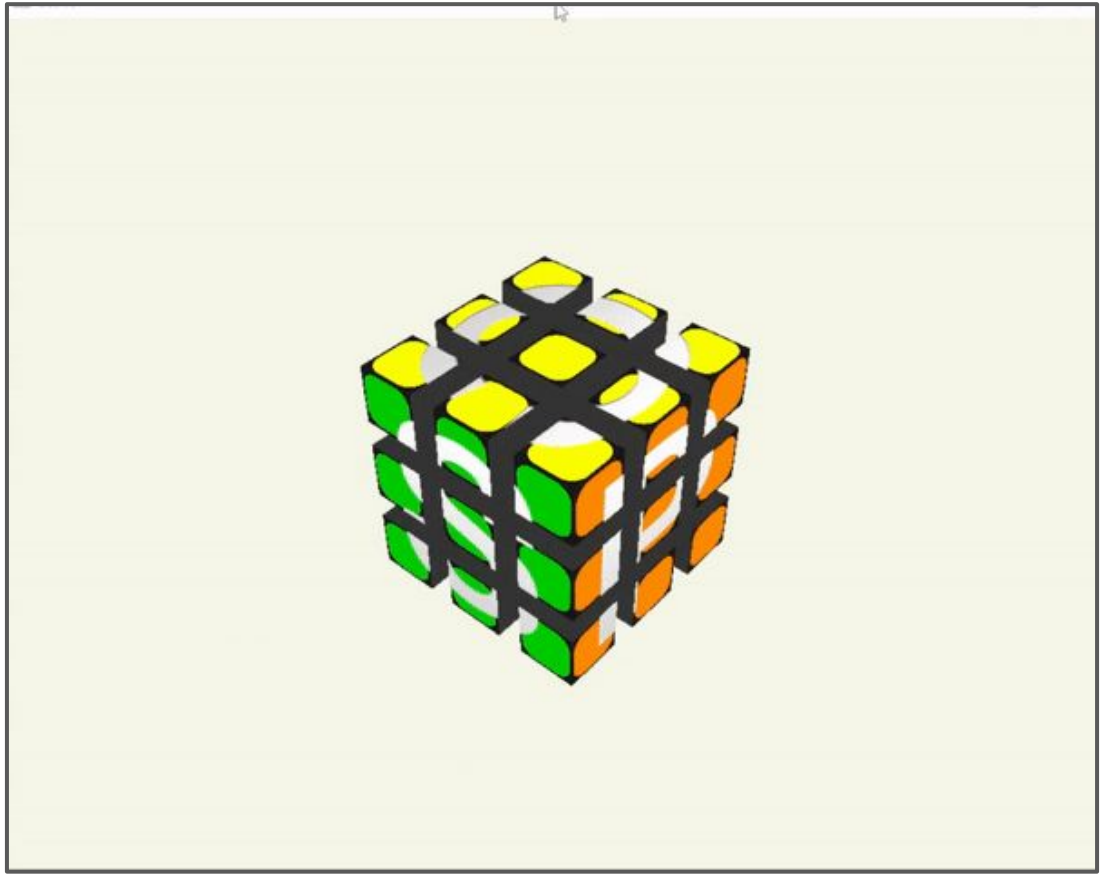
Al presionar las teclas modificamos los valores de la funciones que se encuentran en el render loop

Traslación y Escala

```
model = glm::translate(glm::mat4(1.0), (MagicCube.littleCubes[i].initialPosition) * distancia);
model = glm::scale(glm::mat4(1.0), scaleVector)* model;
```

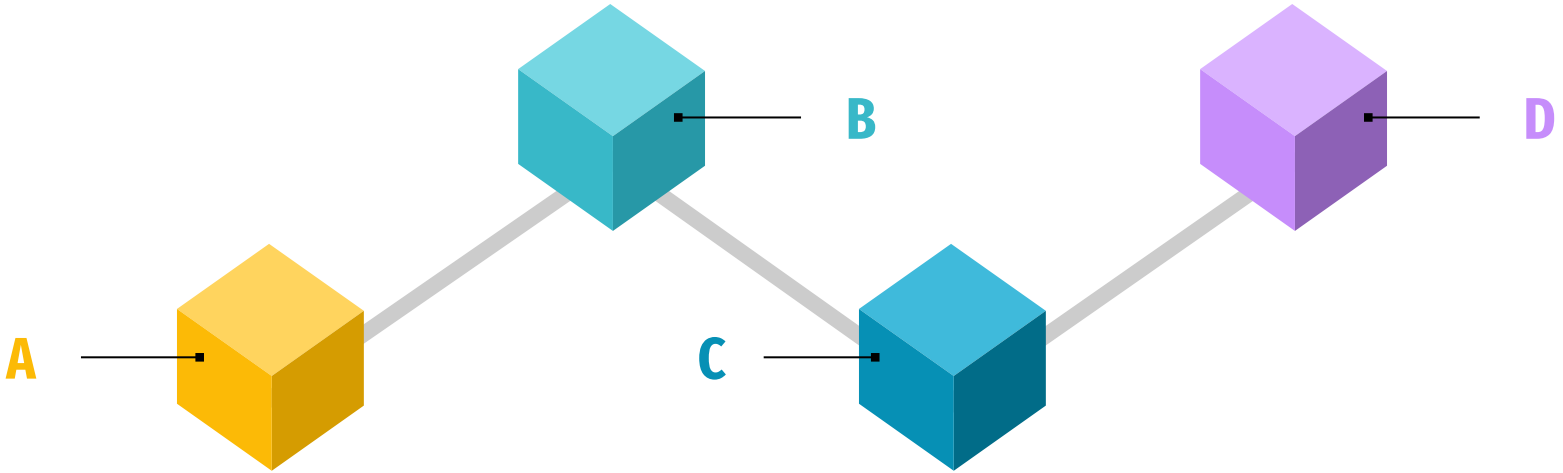
Shader

```
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
```



Cubos desordenados

Los cubitos se dispersan y caen a una cierta distancia con un movimiento elíptico



Elipsoide

x, y, z, a, b, c

Parametrización

Ángulo alfa, theta

Derivación

Traslación

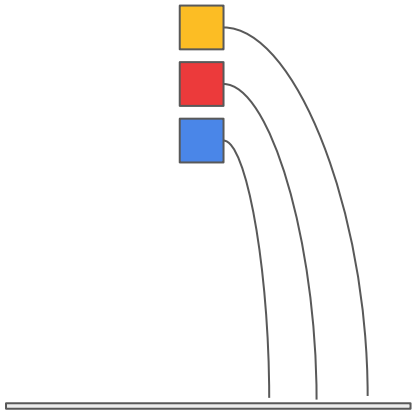
Parámetros en estructura

Distintos movimientos

Elipsoide

x, y, z, a, b, c

$$\frac{x - xo}{a^2} + \frac{y - yo}{b^2} + \frac{z - zo}{c^2} = 1$$



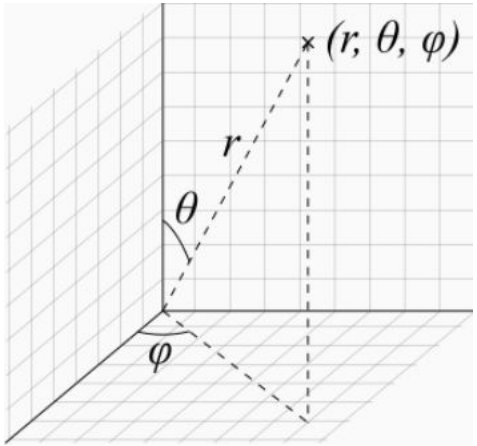
Parametrización

Ángulo alfa, theta

$$x = xo + a \cdot sen\alpha \cdot sen\theta$$

$$y = yo + b \cdot \ cos\theta$$

$$z = zo + c \cdot \ cos\alpha \cdot sen\theta$$



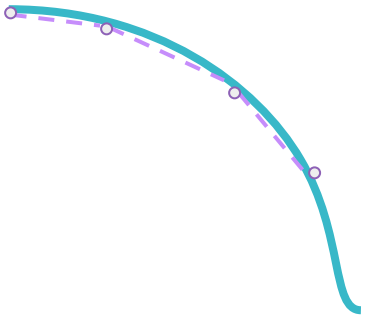
Derivación

Traslación

$$\frac{dx}{d\theta} = (a \cdot sen\alpha \cdot cos\theta)$$

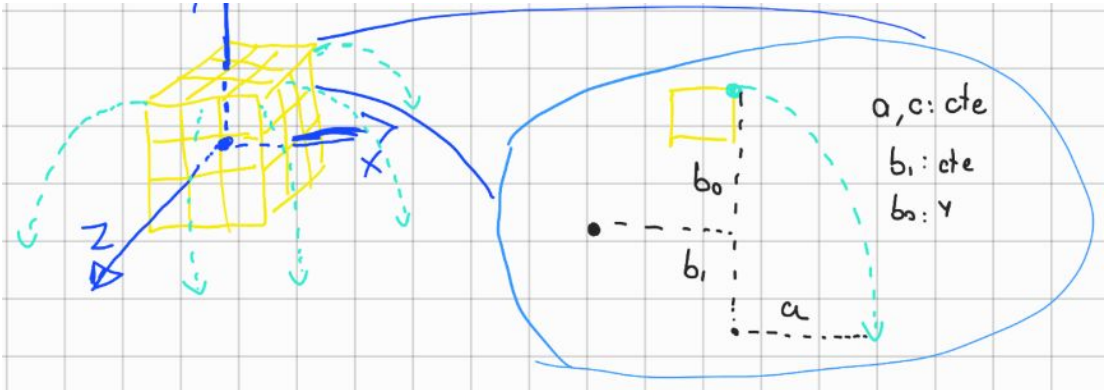
$$\frac{dy}{d\theta} = (-b \cdot sen\theta)$$

$$\frac{dz}{d\theta} = (c \cdot cos\alpha \cdot cos\theta)$$



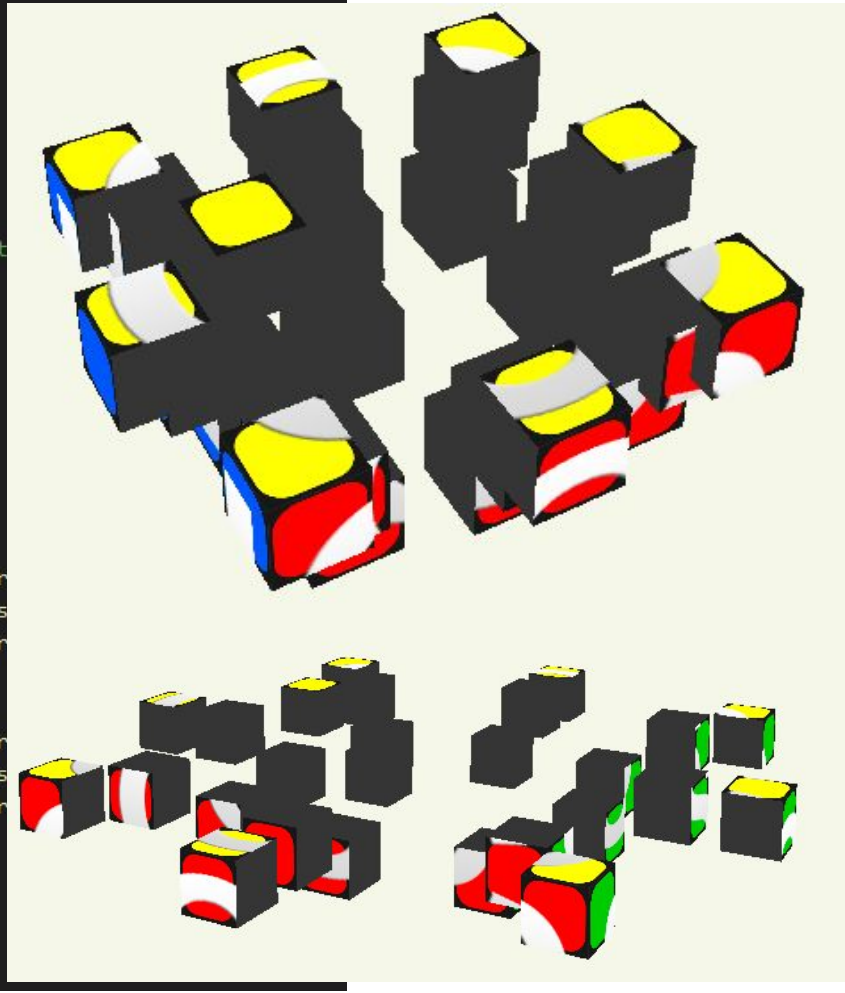
Parámetros en estructura

Distintos movimientos

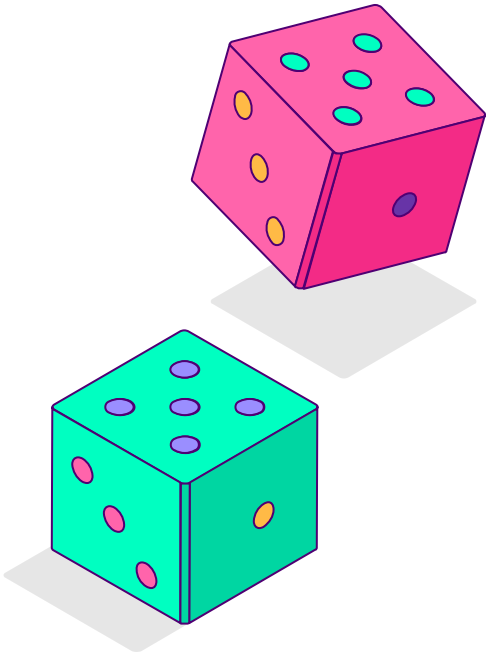


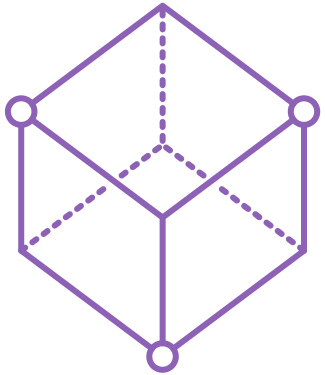

```
struct SLittleCube
{
    int id;
    glm::vec3 initialPosition;
    vector<float> vertices;
    vector<vector<float>> listOfColors;
    vector<char> camadas;
    vector<float> textureIndices;
    vector<float> animation2Parameters; //ac - alfa - b0
```

```
void CCube::animation2(bool reverse, float b1, float angle) {
    for (unsigned int indexCube = 0; indexCube < 26; indexCube++)
    {
        //it was used parametric ecuations and derivation
        /* ... */
        //calculate b0 only for the first time
        if (angle == 1.0 && !reverse) {
            float b0 = numeric_limits<float>::min();
            for (int vertexIndex = 1; vertexIndex < 288; vertexIndex += 8)//it
            {
                if (b0 < littleCubes[indexCube].vertices[vertexIndex])
                    b0 = littleCubes[indexCube].vertices[vertexIndex];
            }
            littleCubes[indexCube].animation2Parameters[2] = b0;
        }
        //calculate variations of 'x' 'y' 'z'
        float difX, difY, difZ;
        if (!reverse) {
            difX = littleCubes[indexCube].animation2Parameters[0] * sin(glm::r
            difY = -(b1 + littleCubes[indexCube].animation2Parameters[2]) * (s
            difZ = littleCubes[indexCube].animation2Parameters[0] * cos(glm::r
        }
        else {
            difX = littleCubes[indexCube].animation2Parameters[0] * sin(glm::r
            difY = -(b1 + littleCubes[indexCube].animation2Parameters[2]) * (s
            difZ = littleCubes[indexCube].animation2Parameters[0] * cos(glm::r
        }
        translatePoints(littleCubes[indexCube].vertices, difX, difY, difZ);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
}
```



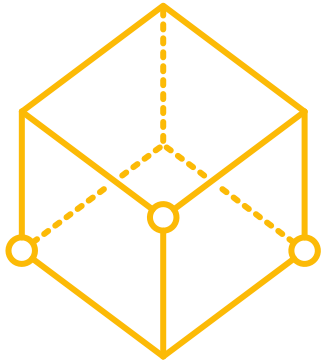
3. CONCLUSIONES





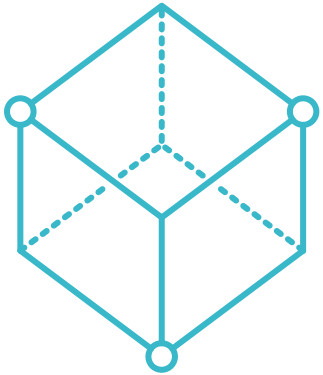
OpenGL + math

El entorno de OpenGL es bastante amigable para modelar, renderizar y animar objetos en dos y tres dimensiones. El conocimiento de conceptos de álgebra y geometría ayuda bastante a que dicho entorno sea más aprovechable y encontrar soluciones a problemas propios de este ámbito.



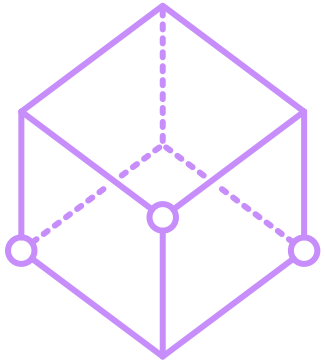
Estructura

La utilización de estructuras de datos permite encapsular valores comunes entre los objetos y que sea más sencilla su manipulación, así como la aplicación de funciones.



Modelado+renderizado

El modelado y renderizado se encuentran bastante relacionados. La disposición de los vectores debe considerar la apariencia que tendrá cada uno de ellos. Los shaders hacen posible que estos dos aspectos se relacionen y puedan ser aplicados.



Animación

La animación requiere una claridad en la manipulación de los vectores que se desean animar. OpenGL ofrece librerías como GLM, aunque de igual forma se pueden aplicar transformaciones elaborando las operaciones con sus matrices correspondientes.

Search or jump to... /

Pull requests Issues Marketplace Explore

Andrescmm / Rubick-Cube Public

Code Issues Pull requests Actions Projects Wiki Security Insights

main 1 branch 0 tags Go to file Add file Code

Andrescmm Update main.cpp 30f401a 14 minutes ago 10 commits

Rubick Cube	Update main.cpp	14 minutes ago
LICENSE	Initial commit	6 hours ago
README.md	Update README.md	1 hour ago

README.md

Rubik-s-Cube-in-OpenGL

This repository contains the Final Project of Computer Graphics UCSP 2022-1

Authors

- **Fernandez Sardon Anthony Paolo** - *San Pablo Catholic University* - GitHub
- **Cusirramos Marquez Mares Andres** - *San Pablo Catholic University* - GitHub

https://github.com/Andrescmm/Rubick-Cube

Gracias!