Informe Cubo de Rubik

Fabrizio Paul Rosado Málaga
Univesidad Católica San Pablo
Arequipa, Perú
Email: fabrizio.rosado@ucsp.edu.pe

Jean Carlo Cornejo Cornejo
Univesidad Católica San Pablo
Arequipa, Perú
Email: jean.cornejo@ucsp.edu.pe

***Abstract*—**

I. Introducción

II. Manejo de los elementos visuales

A. Clase Cubo

Para el manejo de las posiciones, se creo una clase cubo en la que se encuentran las funciones que establecen los vértices del cubo, junto a esta función tenemos una que dibuja nuestro cubo en las posiciones asignadas, declarando así mismo los VAOs (Vertex Array Object), EBOs (Element Buffer Object), y VBOs (Vertex Buffer Object), que corresponderían a cada cubo.

Dentro de la clase tenemos los índices que se utilizan para la definir la estructura del cubo, estos están ordenados de tal manera que el cubo siempre tendrá la forma adecuada para ser dibujado, sin provocar problemas que resulten de vértices mal ubicados y por ello mismo generando otra figura diferente a la del cubo. Esto se logra mediante la inserción de los vértices en un vector de tamaño 24, en la función *setVertex* de cada cubo.

Se utiliza la función *setGLint*, para asignar al *fragmentshader* del color al cubo. Esta es la estructura interna del cubo, fuera de esta misma es donde se instancian los cubos que componen al cubo de Rubik.

B. Sección Main

En esta nueva función de manejo de cubos se envía un puntero de puntero triple de la estructura Cube, a su vez que un arreglo de 27 punteros a Cube, que permitirán apuntar a cada cubo perteneciente a la estructura y serán utilizados para el intercambio de punteros, dentro de la misma estructura, todo esto cuando se realice una rotación del cubo. También se le añade otro arreglo de punteros a cubo, de esta manera se permite rastrear cada cubo con su ID, se pasaran 2 vectores de posiciones que permitirán la generación de los vértices del cubo en relación a dos vértices iniciales en posiciones estáticas.

```
void setVerticesCubes(Cube*** cubos_,Cube*
    cubosp_[27],Cube* cubosID_[27],std::vector
    <float> pos_,std::vector<float> pos1_,
    GLint uniColor_){
    int count=0;
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            for(int k=0;k<3;k++){
                std::vector<float> vertices_={
    pos_[0],pos_[1],pos_[2],
                                            pos1_
    [0],pos_[1],pos_[2],
                                            pos_
    [0],pos_[1],pos1_[2],
                                            pos1_
    [0],pos_[1],pos1_[2],
                                            pos_
    [0],pos1_[1],pos_[2],
                                            pos1_
    [0],pos1_[1],pos_[2],
                                            pos_
    [0],pos1_[1],pos1_[2],
                                            pos1_
    [0],pos1_[1],pos1_[2]};
                ((*(*(cubos_+i)+j)+k))->
    setGLint(uniColor_);
                ((*(*(cubos_+i)+j)+k))->
    setVertex(vertices_);
                ((*(*(cubos_+i)+j)+k))->
    createbindbuffers();
                ((*(*(cubos_+i)+j)+k))->setID(
    count);
                cubosp_[count]=((*(*(cubos_+i)
    +j)+k));
                cubosID_[count]=((*(*(cubos_+i
    )+j)+k));
                pos_={pos_[0]+0.3f,pos_[1],
    pos_[2]};
                pos1_={pos_[0]+0.3f,pos_
    [1]-0.3f,pos_[2]-0.3f};
                count++;
            }
            pos_={pos_[0]-0.9f,pos_[1]-0.3f,
    pos_[2]};
            pos1_={pos_[0]+0.3f,pos_[1]-0.3f,
    pos_[2]-0.3f};
        }
        pos_={pos_[0],pos_[1]+0.9f,pos_[2]-0.3
    f};
```

```
        pos1_={pos_[0]+0.3f,pos_[1]-0.3f,pos_
    [2]-0.3f};
    }
}
```

Listing 1. Generador de Vértices

C. Clase Camadas

La clase consiste de un arreglo de 9 punteros a cubos, además de un arreglo estático bidimensional que almacena los índices de los punteros a cubos que son apuntados por el arreglo *cubosp_* que se encuentra en el main como en [Listing 1], de esta manera, cada vez que se selecciona a alguna camada del cubo de Rubik, se accede a los punteros de los cubos que pertenecen a dicha camada, se modifican sus vértices y se actualizan los punteros a cubos que se encuentran en la nueva posición, así manteniendo la selección a su camada correspondiente.

A continuación tenemos el código perteneciente a la clase camadas.

```
int ccv[9][9]={
            {18,9,0,
             21,12,3,
             24,15,6}, //izquierda_vertical

            {19,10,1,
             22,13,4,
             25,16,7}, //centro_vertical

            {20,11,2,
             23,14,5,
             26,17,8}, //derecha_vertical

            {18,19,20,
             9,10,11,
             0,1,2,}, //top

            {21,22,23,
             12,13,14,
             3,4,5,}, //centro_horizontal

            {24,25,26,
             15,16,17,
             6,7,8,}, //bottom

            {0,1,2,
             3,4,5,
             6,7,8,}, //front

            {9,10,11,
             12,13,14,
             15,16,17,}, //cara_central

            {18,19,20,
             21,22,23,
             24,25,26}, //cara_trasera
    };
```

Listing 2. Arreglo estatico bidimensional

Y también se muestra la función de movimiento, con animación en cuanto a la rotación, para mostrar más del cubo y que no se vea solo como una rotación en donde se ve solo el inicio y el fin.

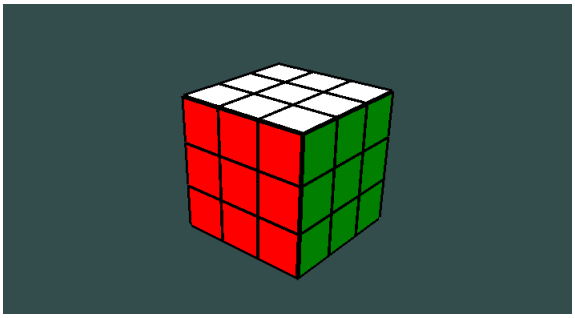

Fig. 1. Representación visual de nuestro Cubo.

```
    void movimiento(int sentido, Cube* cubesp_
[27], GLFWwindow* window) {
        float signo;
        if (sentido)
            signo = 1.0;
        else
            signo = -1.0;

        float vel = 18.0;
        glfwSetTime(0.0);
        float angle = 90.0 / vel;
        int count = 0;
        while (count < 90.0f) {
            for (int i = 0; i < 9; i++) {
                for (int j = 0; j < 24; j = j
+ 3) {
                    glm::vec4 vec(camadas[i]->
vertices[j], camadas[i]->vertices[j + 1],
camadas[i]->vertices[j + 2],
                                  1.0f);
                    glm::mat4 trans = glm::
mat4(1.0f);
                    glm::vec3 eje(0.0f, 0.0f,
0.0f);
                    eje[ejerotacion[indice]] =
 1.0f;

                    trans = glm::rotate(trans,
 glm::radians(signo * angle), eje);

                    vec = glm::vec4(0.0f, 0.0f
, 0.0f, 0.0f) + trans * vec;
                    camadas[i]->vertices[j] =
vec[0];
                    camadas[i]->vertices[j +
1] = vec[1];
                    camadas[i]->vertices[j +
2] = vec[2];
                }
            }
            glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
```

```
            for(int k=0;k<27;k++){
                cubesp_[k]->drawCube();
            }
            glfwSwapBuffers(window);

            count = count + angle;
        }

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        for(int k=0;k<27;k++){
            cubesp_[k]->drawCube();
        }
        glfwSwapBuffers(window);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        int modIndices[2][9] = {
                {6, 3, 0, 7, 4, 1, 8, 5, 2,}, //horario
                {2, 5, 8, 1, 4, 7, 0, 3, 6,}}; //antihorario

        if (indice > 2) {
            for (int i = 0; i < 9; i++) {
                cubesp_[ccv[indice][i]] = camadas[modIndices[sentido][i]];
            }
        } else {
            if (sentido == 1)
                sentido = 0;
            else
                sentido = 1;
            for (int i = 0; i < 9; i++) {
                cubesp_[ccv[indice][i]] = camadas[modIndices[sentido][i]];
            }
        }
    }
    void exeanimation(std::vector<std::string> str, GLFWwindow *window)
```

Listing 3. Movimiento a traves de las camadas

Es así como todo movimiento esta basado en índices y se utilizo las funciones en *GLM*, para permitir operaciones matemáticas sin necesidad de implementar las mismas. Y por último se implemento una cámara con la que se puede apreciar más ángulos del cubo al ser renderizado.

La integración del *solver* se realiza con esta función [Listing 4], mediante un parser el cual interpreta *strings* en funciones visuales para poder observar no solo la animación de los movimientos si no la resolución de un cubo desordenado, ya sea de manera manual al mover nosotros el cubo, o, en caso de obtener un cubo con movimientos aleatorios, ya que esto es posible mediante la librería del *solver*.

```
if (keyblock == false && key == GLFW_KEY_C && action == GLFW_PRESS){
        string tempo1= to_cube_not(movreg);
        movreg.clear();
        solvedCube=get_solution(tempo1);
        for(int i=0;i<solvedCube.size();++i){
            cout<<solvedCube[i]<<" ";
        }
        cout<<endl;
    }
    if (keyblock == false && key == GLFW_KEY_Z && action == GLFW_PRESS){
        std::string temp = "";
        std::vector<std::string> tempo;
        std::string aux = "";
        temp=randomize();
        for(std::string::size_type i = 0; i < temp.size(); ++i) {
            aux=temp[i];
            cout<<temp[i]<<" ";
            tempo.push_back(aux);
        }
        cout<<endl;
        string tempo1=to_cube_not(tempo);
        exeanimation(tempo, window);
        solvedCube=get_solution(tempo1);
        for(int i=0;i<solvedCube.size();++i){
            cout<<solvedCube[i]<<" ";
        }
        cout<<endl;
    }

    if (keyblock == false && key == GLFW_KEY_X && action == GLFW_PRESS){
        exeanimation(solvedCube,window);
    }
```

Listing 4. Integración del *Solver*

Estos *keyblock* son específicos de como obtendremos la solución del cubo, con la tecla ”z” podemos obtener un string de movimientos aleatorios, ejecutar la animación, y obtener la solución de nuestro cubo. También tenemos la tecla ”x” con la que podemos ejecutar la animación de la solución y con la tecla ”c” obtenemos los movimientos que hemos realizado con el teclado y su solución.