Computación Gráfica: 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

I. Introducción

En el siguiente documento se detalla la propuesta de manejo de un cubo de Rubik, desde la lógica de los movimientos hasta la representación visual y el calculo que conlleva todo esto. Este trabajo emplea el uso de C++ con OpenGL, utilizando la biblioteca GLFW para la representación visual del cubo junto a la biblioteca GLM para cálculos matriciales, que son necesarios en la realización de animaciones.

Se emplea el uso de un solver que toma la estrategia Kociemba [1].

II. Descripción del Solver del Cubo de Rubik

Se emplea el uso de un solver basado en el algoritmo de Kociemba modificado, el uso del mismo implica la creación y manipulación de *strings* y vectores de *strings*, con los cuales obtendremos un cubo con movimientos aleatorios usando la función *randomize* y las soluciones a los mismos o a los movimientos que realizamos, utilizando *get_solution*, todo esto en base a la notación de movimientos para así evitar errores o movimientos equivocados.

III. Descripción de la parte gráfica

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]-0.3f};
        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.

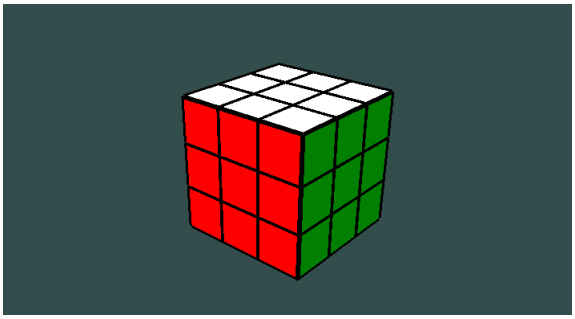

Fig. 1. Representación visual de nuestro Cubo.

```
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*.

IV. Implementación de la nueva animación propuesta

Como propuesta de animación para agregar al cubo de Rubik, se ha pensado en implementar tres animaciones que se dan al momento de interactuar con el cubo cuando se realiza una rotación de alguna de las camadas.

La primera es una traslación de los cubos para separarse y volverse a a colocar a la distancia original en la que estaban en un inicio.

La segunda, a la par de la traslación, al momento de separarse los cubos aumentaran de tamaño, esto se logra con una transformación de escala.

Tercero, se tiene pensado que los cubos giren sobre su propio eje a la vez que rotan en relación al origen.

Esto se realiza en una nueva función con el uso de la librería GLM, mientras que en la propuesta que menciona la intermitencia en los colores del cubo, se realizó con una variante de la función *drawcube*, llamada *draweffects*.

V. Funcionalidades de su programa

Para interactuar con el cubo de Rubik:

Para compilar el programa es necesario cambiar las direcciones del VertexShader y FragmentShader en el main

Para interactuar con el cubo de RUBIK:

Las teclas W, A, S y D permiten mover la camara con ayuda del mouse. Se puede hacer zoom utilizando la rueda del ratón.

Las caras del cubo se mueven con las teclas:

T,U,G,J,B,M para orientaciones negativas. Keypad7, Keypad9, Keypad4, Keypad6, Keypad1, Keypad3 de la parte derecha del teclado para orientaciones positivas.

Para utilizar el solver:

Presionar 'X' para resolver el cubo.

Presionar 'Z' para desordenar el cubo con movimientos aleatorios.

```
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.

VI. Problemas encontrados en la implementación

En el proceso de desarrollo del proyecto, se encontraron algunos contratiempos que afectaron al fluido desarrollo que se había tenido, pero se lograron solucionar gracias a aplicaciones de enfoques encontrados.

Un ejemplo de estos problemas fue al momento de actualizar los índices de las camadas a través de la estructura dinámica que se creo para albergar 27 cubos, para esto inicialmente se había pensado utilizar la estructura tridimensional con punteros al objeto cubo para actualizar sus propios índices, pero no se permitía realizar estos cambios debido a que no había forma de asignar datos a la estructura.

Por lo que se vio necesario crear una estructura externa estática que contuviera punteros a cada uno de los cubos del la estructura dinámica, evitando la tridimensionalidad de esta última, y utilizando los nuevos índices de esta estructura para reasignar los punteros cada vez que hubiera un cambio en las camadas.

References

[1] Muodov, "Kociemba solver." [Online]. Available: https://github.com/muodov/kociemba