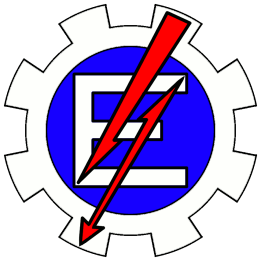

UNIVERSIDADE FEDERAL DE ITAJUBÁ - UNIFEI
Instituto de Matemática e Computação - IMC

CIC270 - Computação Gráfica
Relatório - Trabalho Prático 02

Dupla:
Flávio Mota Gomes - 2018005379
Rafael Antunes Vieira - 2018000980

1. Introdução

O presente trabalho objetiva desenvolver de uma aplicação gráfica 3D baseada na biblioteca OpenGL moderna. A aplicação consiste em uma animação.

Este relatório discorre acerca da implementação de um programa que apresenta um letreiro com a escrita “UNIFEI”. Esse letreiro consiste em uma animação que aplica todos os conceitos aprendidos na disciplina de Computação Gráfica, tais como: modelagem 3D, transformações geométricas, projeções e iluminação.

Para a confecção desta animação, fez-se uso da biblioteca gráfica OpenGL. A biblioteca gráfica OpenGL, cujo nome significa Open Graphical Library caracteriza-se por ser uma interface de software para aceleração da programação de dispositivos gráficos. Há uma permissividade de uso da OpenGL com várias linguagens de programação. Para o presente trabalho, utilizou-se a Linguagem C++.

Implementado em Linguagem C++, o programa utiliza o compilador g++ e é compilado através de um arquivo makefile por meio de qualquer terminal Linux.

A ideia inicial para este trabalho consistia numa animação de uma bomba. Contudo, prevendo dificuldades na produção desse projeto, dados os níveis de complexidade para a referida animação, optou-se por mudar a proposta e chegar à animação supracitada, que implementa todas as funcionalidades aprendidas durante a disciplina de Computação Gráfica.

2. Ferramentas, linguagens e bibliotecas

A animação foi implementada em Linguagem C++, utilizando-se da IDE Microsoft VSCode. Utilizou-se a biblioteca OpenGL moderna para as funcionalidades gráficas. O código foi produzido e compilado no Sistema Operacional Linux.

3. Detalhes da implementação

A ideia deste trabalho foi utilizar vários cubos posicionados de partes diferentes no plano x,y,z a fim de formar uma animação com a palavra “ UNIFEI”.

Nos tópicos a seguir, descreve-se detalhes das técnicas utilizadas neste trabalho.

3.1. Modelagem 3D

A forma geométrica modelada foi a do cubo. Para construí-lo, define-se as coordenadas de 2 triângulos que juntos formam uma face do cubo. Um cubo tem 6 faces, sendo elas: frontal, direita, posterior, esquerda e superior e inferior, totalizando 12 triângulos para formar o cubo, como mostra a Figura 1.

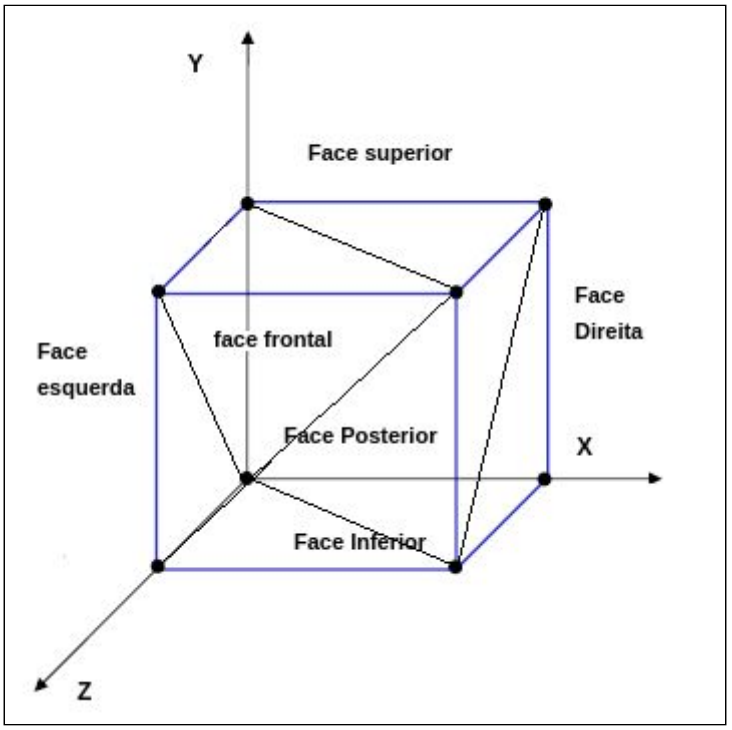

Figura 1. Representação do cubo desenhado na animação.

A seguir, apresenta-se as coordenadas de um cubo programadas no código da aplicação.

```
// Defina os vértices do Cubo.
   float dados_CUBO[] = {
       //Primeiro triângulo da face frontal
       // coordenada      // cor
```

```
        -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
        0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
        0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
        //Segundo triângulo da face frontal
        -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
        0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
        -0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
        //Primeiro triângulo da face direita.
        0.5f, -0.5f, 0.5f, 0.0f, 1.0f, 0.0f,
        0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
        0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
        // Segundo triângulo da face direita.
        0.5f, -0.5f, 0.5f, 0.0f, 1.0f, 0.0f,
        0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
        0.5f, 0.5f, 0.5f, 0.0f, 1.0f, 0.0f,
        // Primeiro triângulo da face posterior.
        0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 1.0f,
        -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 1.0f,
        -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 1.0f,
        // Segundo triângulo da face posterior.
        0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 1.0f,
        -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 1.0f,
        0.5f, 0.5f, -0.5f, 0.0f, 1.0f, 1.0f,
        // Primeiro triângulo da face esquerda.
        -0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f,
        -0.5f, -0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
        -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
        // Segundo triângulo da face esquerda.
        -0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f,
        -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 0.0f,
        -0.5f, 0.5f, -0.5f, 1.0f, 0.0f, 0.0f,
        // Primeiro triângulo da face superior.
        -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 1.0f,
        0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 1.0f,
        0.5f, 0.5f, -0.5f, 1.0f, 0.0f, 1.0f,
        // Segundo triângulo da face superior.
        -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, 1.0f,
        0.5f, 0.5f, -0.5f, 1.0f, 0.0f, 1.0f,
        -0.5f, 0.5f, -0.5f, 1.0f, 0.0f, 1.0f,
        // Primeiro triângulo da face inferior.
        -0.5f, -0.5f, 0.5f, 1.0f, 1.0f, 1.0f,
        -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f,
```

```
        0.5f, -0.5f, 0.5f, 1.0f, 1.0f, 1.0f,
        // Segundo triângulo da face inferior.
        -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f,
        0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f,
        0.5f, -0.5f, 0.5f, 1.0f, 1.0f, 1.0f};
```

Para formar a palavra “UNIFEI”, foi preciso criar e posicionar 57 cubos. As funções `cria_cubos()` e `coordenadas_do_cubo()` são responsáveis por fazer essa tarefa. A função `coordenadas_do_cubo()` contém todas as coordenadas do plano de todos os cubos e, assim, ela faz a chamada da função `cria_cubos()`, passando as coordenadas do plano onde o cubo será montado. Sendo assim, a função `cria_cubos()` recebe as coordenadas X, Y, Z e translada o cubo para essa posição. Também pode-se observar no código abaixo que além das três coordenadas passadas pela função `coordenadas_do_cubo()`, também é passado um quarto valor. Esse valor, informa a `cria_cubos()` se o cubo que será criado terá a animação da rotação ou não: 1 para aplicar a rotação na matriz do cubo e 0 para não aplicar a rotação na matriz do cubo.

Código da função `cria_cubos()`:

```
void cria_cubos(float x, float y, float z, int selec_rotacao)
{

    // Esta escala é comum para todos os objetos
    glm::mat4 Escala = glm::scale(glm::mat4(1.0f), glm::vec3(0.5,
0.5, ZoomZ));
    // Rotação para os objetos que se desejam ser rotacionados
    // Eixo Y
    glm::mat4 Ry = glm::rotate(glm::mat4(1.0f),
glm::radians(py_angle), glm::vec3(0.0f, 1.0f, 0.0f));
    // Eixo X
    glm::mat4 Rx = glm::rotate(glm::mat4(1.0f),
glm::radians(px_angle), glm::vec3(1.0f, 0.0f, 0.0f));

    // Utiliza o vetor com os dados do cubo
    glBindVertexArray(CUBO);

    // Definindo a posição que cada cubo vai ficar para formar a
palavra " UNIFEI " no final. (Como se fosse um LEGO)
    glm::mat4 T = glm::translate(glm::mat4(1.0f), glm::vec3(x, y,
```

```
z));

    // Criando a variável que vai guardar o calculo da matriz
    glm::mat4 M;

    // Faz a verificação se precisa aplicar a rotação no cubo que
será criado
    if (selec_rotacao == 1)
    {
        // Calculos da matriz com rotação
        M = T * Escala * Rx * Ry;
    }
    else
    {
        // Calculos da matriz sem rotação
        M = T * Escala;
    }

    // Escreve na tela após os cálculos da matriz
    unsigned int loc = glGetUniformLocation(program, "M");
    glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(M));
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Código da função `coordenadas_do_cubo()`:

```
// Função responsável por passar as coordenadas para a criação dos
cubos na tela
// Esta função chama a função que cria o cubo na tela, passando as
coordenadas corretas
void coordenadas_do_cubo(void)
{
    /* Informando as coordenadas dos cubos que formam a palavra
"UNIFEI" */
    // Cubo 1 -------------------------------
    cria_cubos(-5.0, 0.0, -0.5, 0);
    // Cubo 2 -------------------------------
    cria_cubos(-4.5, 0.0, -0.5, 0);
    // (...)
}
```

Na Figura 2 é apresentada a estrutura com os pontos dos 57 cubos interligados e na Figura 3 os mesmos cubos com as fases preenchidas.

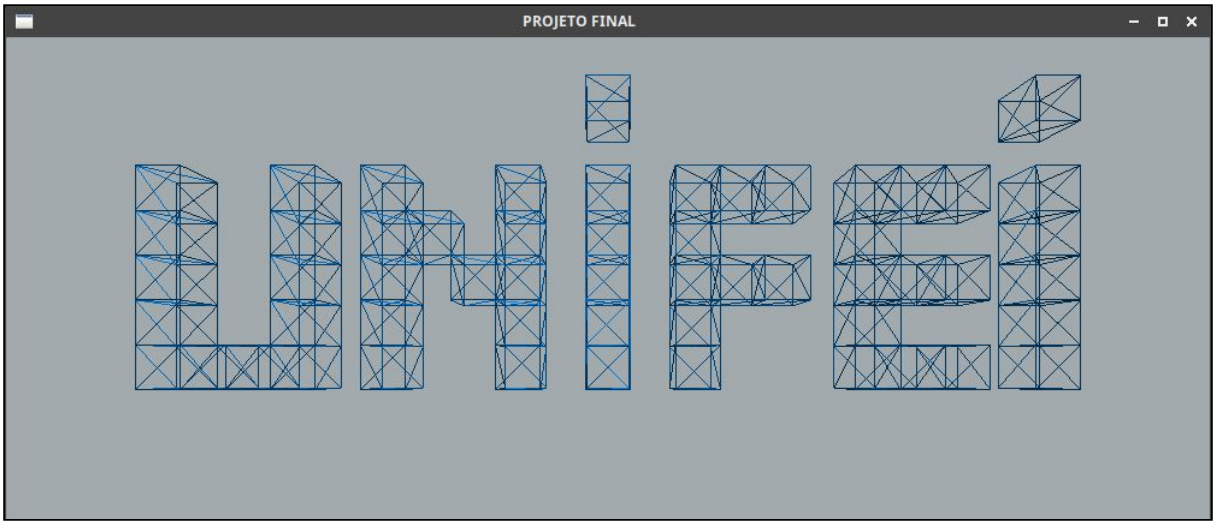

Figura 2: Estrutura da palavra “UNIFEI” com os pontos interligados.

Figura 3. Estrutura da palavra “UNIFEI” com as faces preenchidas.

3.2. Transformações Geométricas

Foram implementados três tipos de transformação geométrica neste trabalho, sendo eles: translação, rotação e escala. A seguir, descreve-se detalhes sobre cada um deles.

3.2.1. Translação

A translação está manifestada na posição de cada um dos cubos que compõem o letreiro UNIFEI. Os 57 cubos sofrem translação ao serem posicionados em seus respectivos locais para, juntos, formarem o letreiro. No trecho de código de cada um deles, há a definição da variável `T`, conforme ilustrado a seguir, aplicando-se o `translate` e posicionando conforme necessário.

```
// Definindo a posição que cada cubo vai ficar para formar a
palavra " UNIFEI " no final. (Como se fosse um LEGO)
    T = glm::translate(glm::mat4(1.0f), glm::vec3(-4.5f, 0.0f,
-0.5f));
```

3.2.2. Rotação

A função `rotacao()` é responsável por definir o ângulo da rotação dos objetos na tela. Ela define ângulos de rotação para os eixos Y e X. A rotação será aplicada em dois dos 57 cubos utilizados para formar o letreiro “UNIFEI”. Esses cubos são os dois “pingos” dos i’s.

```
void rotacao()
{
    // Angulo de rotação para o eixo Y
    py_angle = ((py_angle + py_inc) < 360.0f) ? py_angle + py_inc
: 360.0 - py_angle + py_inc;

    // Angulo de rotação para o eixo X
    px_angle = ((px_angle + px_inc) < 360.0f) ? px_angle + px_inc
: 360.0 - px_angle + px_inc;

    glutPostRedisplay();
}
```

O código a seguir demonstra o valor de ângulo e o local onde esse ângulo deve ser multiplicado para realizar a rotação. Veja que `Ry` tem zerados os valores de `x` e `z` porque realiza rotação em `y`; enquanto isso, `Rx` tem zerados os valores de `y` e `z` porque realiza rotação em `x`.

```
// Rotação para os objetos que se desejam ser rotacionados
    // Eixo Y
    glm::mat4 Ry = glm::rotate(glm::mat4(1.0f),
```

```
glm::radians(py_angle), glm::vec3(0.0f, 1.0f, 0.0f));
    // Eixo X
    glm::mat4 Rx = glm::rotate(glm::mat4(1.0f),
glm::radians(px_angle), glm::vec3(1.0f, 0.0f, 0.0f));
```

Dois cubos dos 57 realizam rotação. Para informar a função `cria_cubos()` que se deseja aplicar a rotação nesses cubos, é passado o valor "1" para a função. Sendo assim, a mesma consegue identificar e aplicar a rotação nos cubos. Abaixo apresenta-se o chamado da função para os cubos onde se deseja aplicar a rotação.

```
 // Cubo 31 pingo do i passa 1 no quarto elemento informando a
rotação no cubo  ------------------------

    cria_cubos(0.0, 3.0, -0.5, 1);

// (...)

    // Cubo 57 pingo do i passa 1 no quarto elemento informando a
rotação no cubo  ------------------------

    cria_cubos(5.0, 3.0, -0.5, 1);
```

Assim que a função recebe o valor "1", ela aplica os valores da função `rotate` na matriz do cubo. Veja no trecho do código abaixo:

```
// Faz a verificação se precisa aplicar a rotação no cubo que será
criado
    if (selec_rotacao == 1)
    {
        // Cálculos da matriz com rotação
        M = T * Escala * Rx * Ry;
    }
    else
    {
        // Cálculos da matriz sem rotação
        M = T * Escala;
    }
```

3.2.3. Escala

Todos os objetos aplicam o conceito de escala. Dentro da função `display()`, chama-se a função `coordenadas_do_cubo()`. Nela, serão construídos os cubos formadores da imagem. Também nela, antes da disposição dos cubos, cria-se a variável `Escala`, do tipo `glm::mat4`. Essa variável vai definir a escala comum para todos os objetos. Foram definidos os valores `0.5` para as dimensões x e y do cubo. A dimensão z é representada pela variável `ZoomZ`, relacionada à escala do zoom na aplicação, conforme o tópico 3.5. deste trabalho explicará com maior clareza. Ambos os valores correspondem à medida de cada uma dessas dimensões e, ao serem aplicados na escala, fazem com que os cubos se encontrem e sejam capazes de formar as letras. A seguir, apresenta-se o código da escala.

```
// Essa escala é comum para todos os objetos
   glm::mat4 Escala = glm::scale(glm::mat4(1.0f), glm::vec3(0.5,
0.5, ZoomZ));
```

Além disso, na mesma função define-se as variáveis `Rx` e `Ry` para rotação.

```
// Eixo Y
    glm::mat4 Ry = glm::rotate(glm::mat4(1.0f),
glm::radians(py_angle), glm::vec3(0.0f, 1.0f, 0.0f));
    // Eixo X
    glm::mat4 Rx = glm::rotate(glm::mat4(1.0f),
glm::radians(px_angle), glm::vec3(1.0f, 0.0f, 0.0f));
```

Junto disso, setam-se outros parâmetros.

```
// Utiliza o vetor com os dados do cubo
    glBindVertexArray(CUBO);

    // Definindo a posição que cada cubo vai ficar para formar a
palavra " UNIFEI " no final. (Como se fosse um LEGO)
    glm::mat4 T = glm::translate(glm::mat4(1.0f), glm::vec3(x, y,
z));

    // Criando a variável que vai guardar o cálculo da matriz
    glm::mat4 M;
```

A aplicação desta variável de escala em cada um dos cubos ocorre numa multiplicação com a posição, para todos os cubos. Isso pode ser visualizado a seguir. Note a existência de uma variável chamada `selec_rotacao`. Essa variável é um parâmetro passado na função `cria_cubo()` para informar quais cubos ficarão em rotação constante - no caso, apenas os pingos das letras I. Quando a rotação é aplicada, ocorre a multiplicação dos valores `Rx` e `Ry`. Todavia, para absolutamente todos os cubos, assim como a variável `T`, multiplica-se também a variável de `Escala`, ou seja, aplica-se a escala para todos os cubos que formam o letreiro.

```
// Faz a verificação se precisa aplicar a rotação no cubo que será criado
    if (selec_rotacao == 1)
    {
        // Cálculos da matriz com rotação
        M = T * Escala * Rx * Ry;
    }
    else
    {
        // Cálculos da matriz sem rotação
        M = T * Escala;
    }
```

3.3. Projeções

Na função `display()`, define-se características importantes também para a definição de projeção para os objetos 3D.

```
// Define a view
    glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -5.0f));
    unsigned int loc = glGetUniformLocation(program, "view");
    glUniformMatrix4fv(loc, 1, GL_FALSE, glm::value_ptr(view));

    // Define a projeção da animação com perspectiva
    glm::mat4 projection = glm::perspective(glm::radians(70.0f), (janela_largura / (float)janela_altura), 0.1f, 100.0f);
    loc = glGetUniformLocation(program, "projection");
```

```
    glUniformMatrix4fv(loc, 1, GL_FALSE,
glm::value_ptr(projection));
```

Para concretizar a questão da projeção, utilizou-se o shader disponibilizado nas aulas.

```
/** Vertex shader. */
const char *vertex_code = "\n"
                          "#version 330 core\n"
                          "layout (location = 0) in vec3
position;\n"
                          "layout (location = 1) in vec3
normal;\n"
                          "\n"
                          "uniform mat4 M;\n"
                          "uniform mat4 view;\n"
                          "uniform mat4 projection;\n"
                          "\n"
                          "out vec3 vNormal;\n"
                          "out vec3 fragPosition;\n"
                          "\n"
                          "void main()\n"
                          "{\n"
                          "    gl_Position = projection * view * M
* vec4(position, 1.0);\n"
                          "    vNormal = normal;\n"
                          "    fragPosition = vec3(M *
vec4(position, 1.0));\n"
                          "}\0";
```

3.4. Iluminação

A iluminação desta aplicação foi colocada no intuito de iluminar o letreiro. A iluminação da animação realiza movimentação no eixo x, de modo a iluminar todas as letras do letreiro, seguindo seu caminho da primeira à última, depois da última à primeira, sucessivamente. A seguir, apresenta-se a definição de parâmetros de iluminação. Definiu-se a cor do objeto para a iluminação, a cor da luz sobre o objeto e também a posição inicial dessa luz.

```
    // Define a cor do objeto para a iluminação.
    loc = glGetUniformLocation(program, "objectColor");
    glUniform3f(loc, 0.0, 0.24745, 0.43921)

    // Define a cor da luz sobre o objeto.
    loc = glGetUniformLocation(program, "lightColor");
    glUniform3f(loc, 1.0, 1.0, 1.0);

    // Define a posição da luz.
    loc = glGetUniformLocation(program, "lightPosition");
    glUniform3f(loc, 0.0, 3.0, 2.0);

   // Define a posição da câmera.
    loc = glGetUniformLocation(program, "cameraPosition");
    glUniform3f(loc,posicao_camera_X, 0.0, 0.0);
```

A questão da movimentação da luz foi implementada por intermédio da função `animacao_da_luz()`, chamada na função `display()` da aplicação. Essa função é responsável por animar a iluminação, alterando a posição da iluminação no eixo x entre as posições -5.0 e 5.0, respectivamente, entre as letras “U” e o segundo “I”. Veja, a seguir, a implementação desta função.

```
void animacao_da_luz(void)
{
    contador++;
    // Verifica o sentido que a iluminação vai percorrer
    // 1 ele avança positivamente no eixo X
    //-1 ele avança negativamente no eixo X
    if (sentido == 1)
    {
        posicao_camera_X = posicao_camera_X + 0.1;
    }
    else
    {
        posicao_camera_X = posicao_camera_X - 0.1;
    }
    if (contador == 100 && sentido == 1)
    {
        contador = 0;
```

```
        sentido = -1;
    }
    else if (contador == 100 && sentido == -1)
    {
        contador = 0;
        sentido = 1;
    }
}
```

Para as demais configurações de iluminação, utilizou-se do shader disponibilizado nas aulas.

```
/** Fragment shader. */
const char *fragment_code = "\n"
                            "#version 330 core\n"
                            "\n"
                            "in vec3 vNormal;\n"
                            "in vec3 fragPosition;\n"
                            "\n"
                            "out vec4 fragColor;\n"
                            "\n"
                            "uniform vec3 objectColor;\n"
                            "uniform vec3 lightColor;\n"
                            "uniform vec3 lightPosition;\n"
                            "uniform vec3 cameraPosition;\n"
                            "\n"
                            "void main()\n"
                            "{\n"
                            "    float ka = 0.5;\n"
                            "    vec3 ambient = ka *
lightColor;\n"
                            "\n"
                            "    float kd = 0.8;\n"
                            "    vec3 n = normalize(vNormal);\n"
                            "    vec3 l = normalize(lightPosition
- fragPosition);\n"
                            "\n"
                            "    float diff = max(dot(n,l),
0.0);\n"
                            "    vec3 diffuse = kd * diff *
lightColor;\n"
                            "\n"
```

```
                            "    float ks = 1.0;\n"
                            "    vec3 v = normalize(cameraPosition
- fragPosition);\n"
                            "    vec3 r = reflect(-l, n);\n"
                            "\n"
                            "    float spec = pow(max(dot(v, r),
0.0), 3.0);\n"
                            "    vec3 specular = ks * spec *
lightColor;\n"
                            "\n"
                            "    vec3 light = (ambient + diffuse +
specular) * objectColor;\n"
                            "    fragColor = vec4(light, 1.0);\n"
                            "}\0";
```

3.5. Outros

Descreve-se aqui alguns trechos de código importantes, a começar pela `main` do programa. Essa função é responsável por dar o início a todas as demais funções, chamando-as para que monte a animação construída. Além disso, chama a inicialização da janela e dá um nome a ela; no caso, “Projeto Final”.

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitContextVersion(3, 3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(janela_largura, janela_altura);
    glutCreateWindow("PROJETO FINAL");
    glewExperimental = GL_TRUE;
    glewInit();
    initData();
    initShaders();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutIdleFunc(rotacao);
    glutMainLoop();
```

```
}
```

Tem-se também a função `initShaders()`, chamada pela `main`, que é responsável por compilar os shaders criados pelo programa. Ela solicita um programa e slots de shader da GPU.

```
void initShaders()
{
    program = createShaderProgram(vertex_code, fragment_code);
}
```

Por se tratar de uma animação, o programa não é interativo no sentido de manipular a animação em si. Todavia, implementou-se a função keyboard, que é responsável por comandos no teclado, como para permitir a saída da janela de execução, saída essa executada por meio da tecla ‘q’ do teclado do computador. Além disso, permite-se, por meio das teclas 1 e 2, respectivamente, a interligação dos pontos do cubo, de modo a apresentar as linhas que formam os cubos; e o preenchimento dessas linhas, de modo a ter os cubos completos, tal qual apresentado na Figura 3. Também implementou-se uma manipulação de câmera, que permite a realização de *zoom in* e *zoom out* no eixo z da aplicação. Essa implementação pauta-se nas teclas ‘W’ para aproximação e ‘S’ para afastar.

```
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 27:
        exit(0);
        // Sai do programa
    case 'q':
    case 'Q':
        exit(0);
    case '1':
        // Interliga os pontos do cubo
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        break;
    case '2':
        // Preenche as faces do cubo
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

```
        break;
    case 'W':
    case 'w':
        // Aproxima do objeto
        ZoomZ = ZoomZ + 0.2;
        break;
        // Afasta do objeto
    case 'S':
    case 's':
        ZoomZ = ZoomZ - 0.2;
        break;
    }

    glutPostRedisplay();
}
```

Por fim, a função `reshape` cria a janela com as larguras e alturas definidas previamente. Essa definição dá-se por meio das variáveis globais. A janela tem dimensão 1000 x 550.

```
void reshape(int width, int height)
{
    janela_largura = width;
    janela_altura = height;
    glViewport(0, 0, width, height);
    glutPostRedisplay();
}
```

4. Manual

A compilação dos código dá-se por intermédio do arquivo *makefile* presente junto a ele. O programa é executável em terminal Linux. Para isso, deve-se abrir uma janela de terminal na pasta onde estão os arquivos do programa. No terminal, os comandos a serem informados são, nesta ordem:

```
make
./modelo
```

Após a execução dos comandos, a janela do programa será aberta e a animação apresentada.

A aplicação é uma animação, sendo assim apresenta somente 5 interações possíveis entre o usuário e o programa:

- TECLA ‘1’ : As bordas de limite do objeto são desenhadas como segmentos de linha, para que o usuário consiga ver a estrutura dos cubos posicionados;
- TECLA ‘2’: Preenche as faces dos objetos apresentados na animação, formando então a palavra UNIFEI;
- TECLA ‘W’ / ’w’: Esta tecla realiza alterações no eixo z, incrementando a variável de zoom;
- TECLA ‘S’ / ‘s’: Esta tecla realiza alterações no eixo z, decrementando a variável de zoom;
- TECLA ‘Q’ / ‘q’: Permite a saída da aplicação.

5. Conclusões

Conclui-se que o objetivo deste trabalho, ou seja, a confecção de uma animação utilizando os conceitos aprendidos durante as aulas do curso de Computação Gráfica, foi cumprido. Para comprovar, apresenta-se, a seguir, resultados obtidos na execução do código.

A visualização do resultado alcançado foi apresentada nas Figuras 2 e 3 deste trabalho. Através da execução do código, visualiza-se os movimentos destacados ao longo deste relatório, bem como as questões de iluminação e posicionamento com maior clareza.

Reitera-se que a visualização neste presente relatório não reflete fidedignamente à visão durante a execução em máquina, justamente por conta das limitações deste meio, o que impossibilita, por exemplo, a apresentação das movimentações. Para uma melhor visualização, recomenda-se a execução do código em C++.

No que tange às dificuldades encontradas, ressalta-se a dificuldade de trabalhar com diversos objetos na mesma tela e ao mesmo tempo. Embora contornada, a questão de movimentação de câmera também foi uma dificuldade.

Referências Bibliográficas

THORMÄHLEN, T. Graphics Programming Buffer Objects. *Universität Marburg*. Disponível em: <https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/graphics_8_1_eng_web.html#1>. Acesso em: 23 nov. 2020.

OPENGL. Colors. *OpenGL.* Disponível em: <https://learnopengl.com/Lighting/Colors>. Acesso em: 23 nov. 2020.

_____. Transformations. *OpenGL.* Disponível em: <https://learnopengl.com/Getting-started/Transformations>. Acesso em: 23 nov. 2020.

_____. Coordinate-Systems. *OpenGL.* Disponível em: <https://learnopengl.com/Getting-started/Coordinate-Systems>. Acesso em: 23 nov. 2020.

Anexo

No link a seguir, encontra-se um vídeo com a apresentação deste relatório:
<
https://drive.google.com/file/d/1EoOBpGZBqXkXvou7BU4JbsD-nKF7zs5l/view?usp=sharing >