Relatório do Trabalho 01 – Computação Gráfica (CIC270)

Nome: Rafael Greca Vieira **Matrícula:** 2018000434
Nome: Vítor Siqueira Lobão **Matrícula:** 2018004809

1. Introdução

Representar qualquer tipo de superfície no âmbito da computação gráfica é por muitas vezes, altamente custoso em níveis de capacidade computacional.

A fim de solucionar esse problema, existem duas soluções mais conhecidas, a representação de superfícies por meio de malha triangular ou nuvem de pontos. Ambos os métodos consistem em utilizar-se de métodos matemáticos para simplificar a representação das imagens e podem ser visualizados nas figuras 1 e 2.

Figura 1 - Representação por núvem de pontos

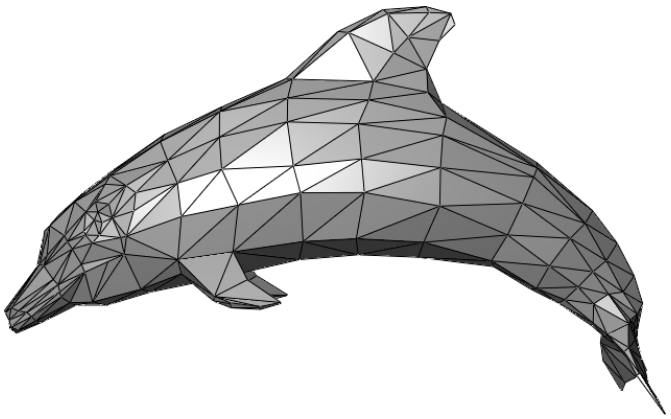

Figura 2 - Representação por malha triangular

O trabalho tem como objetivo utilizar a biblioteca OpenGL para a criação de uma aplicação que irá abrir imagens PGM e PPM e, a partir delas, criar os dois modelos de representação citadas anteriormente.

A implementação foi realizada utilizando a linguagem de programação C, compilado utilizando o arquivo Makefile e criado no sistema operacional Linux. A imagem que será utilizada deverá ser

passada como parâmetro durante a compilação do programa através do terminal. Ao passar a imagem, ela será lida pelo programa e os modelos serão apresentados na tela.

2. Descrição de como os modelos de malha triangular e nuvem de pontos foram computados a partir da imagem.

Antes de tudo, precisamos saber o que é malha triangular e nuvem de pontos. A representação por nuvem de pontos consistem em se utilizar de pontos retirados da superfície que se deseja representar, os pontos carregam dados como, cor e posição equivalentes à da superfícies desejada. O método de nuvem de pontos tem se mostrado muito útil atualmente devido ao advento de scanners 3d, que consistem em, por meio de câmeras especiais, capturar bilhões de bilhões de pontos de um objeto, e por meio do método de nuvem, representar o objeto.

Já a representação or malhar triangular consiste em se utilizar de diversos triângulos formados por vértices compartilhados e diagonais entre os vértices. O método de malha triangular é altamente utilizado em modelagens 3d, pois facilitam o processamento das estruturas diminuindo a quantidade de polígonos a serem renderizados.

Agora que já sabemos a definição dos conceitos iremos ver como eles são computados a partir de uma determinada imagem. Para que os modelos de malha triangular e nuvem de pontos sejam apresentados, é necessário obter as coordenadas de cada pixel da imagem (X, Y e Z, onde Z será sempre 0.0) e também os seus valores RGB. Após todos os valores serem coletados, eles precisarão ser normalizados. Os valores das cores (RGB), após ser aplicada a normalização, deverá apresentar um valor entre 0 e 1. Já os valores das coordenadas X e Y deverão apresentar um valor entre -1 e 1.

Para normalizarmos os valores RGB, iremos pegar o valor de cada pixel da imagem e dividir pelo valor máximo de nível de cinza (ou pixel de maior valor contido na imagem) que está no cabeçalho da imagem. Por exemplo: considerando uma imagem que possui 255 como o maior valor de nível de cinza e um pixel com valor de R igual a 120, G igual a 255 e B igual a 10, após ser normalizado seus valores serão 120/255=0.4706, 1.0 e 0.04, respectivamente.

Para normalizarmos as coordenadas X e Y não será tão intuitivo quanto ao do RGB. Primeiro precisamos pensar como as dimensões da imagem se encontram no plano cartesiano. Considerando a figura 3, podemos perceber que a largura da imagem corresponde ao eixo X do plano cartesiano e a altura ao eixo Y. Então se uma imagem possui dimensão (500 x 300), significa que sua largura tem valor de 500 e 300 de altura. Transformando em uma visão do plano cartesiano, o eixo X vai do ponto zero até 500 e o eixo Y vai do ponto até 300.

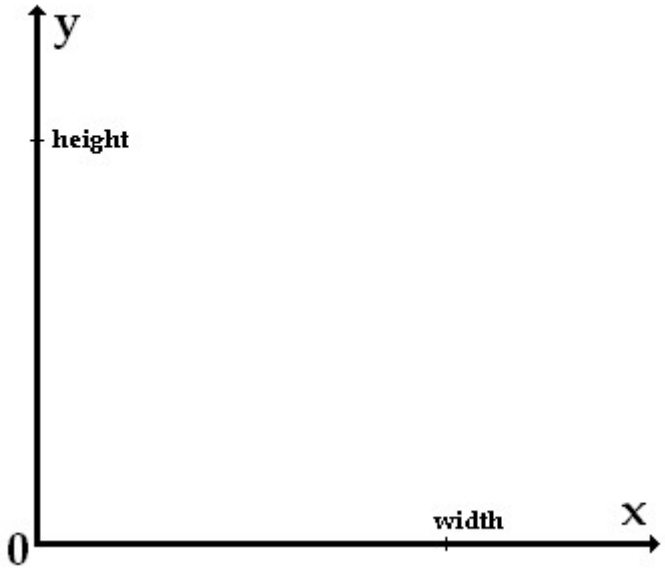

Figura 3 – Plano cartesiano considerando as dimensões da imagem

Como dito anteriormente, os valores normalizados, tanto para o eixo X quanto o Y, deverão estar entre -1 e 1, como mostra o novo plano cartesiano representado na figura 4.

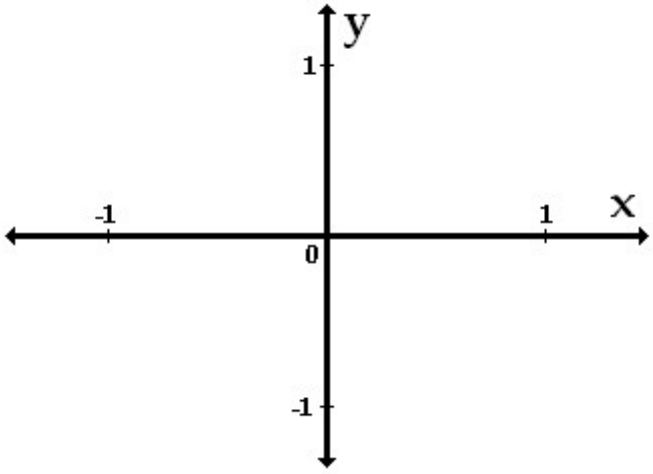

Figura 4 – Novo plano cartesianos considerando os pontos normalizados

Utilizando a imagem do exemplo anterior, com dimensão (500x300), podemos observar que o ponto central (250x150) deverá ter um valor igual a (0, 0) depois de ser normalizado. Agora, considerando os pontos das extremidades da imagem (0, 300), (500, 0), (0,0) e (500, 300), após normalizados deverão ter os respectivos valores: (-1, 1), (1, -1), (-1, -1) e (1, 1). As fórmulas que serão utilizadas na normalização do eixo X e Y serão:

Eixo X => ((coordenada/(largura_imagem - 1.0f)) * 2.0f – 1.0f)
Eixo Y => (((altura_imagem - 1.0f - coordenada)/(altura_imagem - 1.0f)) * 2.0f – 1.0f)

A letra “f” contida nos números é para converter o valor float para double.

Utilizando as fórmulas citadas anteriormente iremos comprovar o resultado normalizado obtido no vértice (250, 150) para exemplificar.

Ponto (250, 150):
Eixo X => ((250/(500 - 1.0f)) * 2.0f – 1.0f) =>0.5 * 2f – 1f = 1 – 1f = 0
Eixo Y => (((300 - 1.0f - 150)/(300 - 1.0f)) * 2.0f – 1.0f) = 0.4983 * 2f – 1f = 0

3. Explicação dos detalhes importantes da implementação usando a biblioteca gráfica OpenGL.

3.1 Struct pixel

A struct pixel será utilizada para armazenar o valor RGB de um determinado pixel. Cada valor será armazenado em variáveis float separadas. Como mostra a figura 5. Veremos mais para frente que essa estrutura será utilizada para armazenar as cores de cada pixel da imagem dentro de uma matriz.

```
//irá armazenar os valores RGB de cada pixel
typedef struct pixel{
  float r, g, b;

}pixel;
```

Figura 5 – Struct pixel

3.2 Variáveis globais

A figura 6 mostra as variáveis globais mais importantes do código. A primeira, chamada “type_primitive” será utilizada para a visualização dos modelos de malha triangular e nuvem de pontos. Inicializando o valor dela como “GL_POINTS” fará com que a nuvem de pontos seja apresentada assim que a aplicação for inicializada. As variáveis “win_width” e “win_height” armazenarão, respectivamente, o valor da propriedade de largura e altura do tamanho da janela de visualização da imagem junto com os modelos.

```
/* Variáveis globais */

int type_primitive = GL_POINTS;
int largura_imagem;
int altura_imagem;
int win_width  = 900;
int win_height = 700;
int program;
unsigned int VAO;
unsigned int VBO;
```

Figura 6 – Variáveis globais do código

3.3 Função de normalização dos eixos X e Y

Como foi dito na seção anterior, assim como os valores da cor do pixel, as coordenadas X e Y também precisam ser normalizadas. As funções que realizarão a normalização estão representadas na figura 7. A função “normaliza_eixo_x” irá receber como parâmetro a coordenada X do pixel e irá retornar o seu valor normalizado considerando a largura total da imagem. A função “normaliza_eixo_y” tem o mesmo objetivo, porém receberá a coordenada Y e irá considerar a altura total da imagem.

```
//normaliza a coordenada do eixo x
float normaliza_eixo_x(float coordenada){

  return ((coordenada/(largura_imagem - 1.0f)) * 2.0f - 1.0f);
}

//normaliza a coordenada do eixo y
float normaliza_eixo_y(float coordenada){

  return (((altura_imagem - 1.0f - coordenada)/(altura_imagem - 1.0f)) * 2.0f - 1.0f);
}
```

Figura 7 – Funções de normalização das coordenadas do eixo X e Y

3.4 Função de leitura da imagem

Nessa função será realizada a leitura da imagem que for passada como parâmetro e irá retornar uma matriz de pixel (a struct utilizada no começo da seção). Antes de começar a leitura, será feita uma verificação se a imagem passada como parâmetro é válida, como mostra a figura 8. A imagem será aberta em formato binário e, se não for uma imagem válida, o programa irá retorna uma mensagem de erro e irá fechar a aplicação.

```
/*Lê a imagem que é passada como parâmetro e retorna sua matriz de pixels*/
pixel **le_imagem(char nome_imagem[]){

  char id[2];
  int valor_maximo_pixel;
  pixel **matriz;

  //Abre a imagem
  FILE *imagem = fopen(nome_imagem, "rb");
  if(!imagem){
    printf("Problema com o arquivo. \n");
    exit(-1);
  }
```

Figura 8 – Função que realizará a leitura da imagem

Se a imagem for válida, o próximo passo será ler o cabeçalho da imagem, como apresentado na figura 9. Toda imagem PGM/PPM possui o mesmo padrão de organização do cabeçalho. A primeira linha representa o tipo, a segunda tem o valor da largura e altura da imagem, respectivamente, na terceira irá conter o seu valor máximo de nível de cinza. E, por último, estarão todos os pixels da imagem em um formato binário ou em texto, dependendo de qual for o seu tipo. A imagem pode conter comentários entre as linhas. O código só trata comentários de uma linha.

```
//P2 e P3 = imagem que está no formato texto (ASCII)
//P5 e P6 = imagem que está no formato binário
//PBM (preto e branco)
//PGM (colorida)
fscanf(imagem, "%s\n", id);
le_comentarios(imagem);
fscanf(imagem, "%d %d\n", &largura_imagem, &altura_imagem);
le_comentarios(imagem);
fscanf(imagem, "%d\n", &valor_maximo_pixel);
le_comentarios(imagem);
```

Figura 9 – Leitura do cabeçalho da imagem

Uma verificação para saber se o tamanho da imagem é menor que o da janela será realizado. Se for válido, outra verificação será realizada para saber se o cabeçalho lido é válido, como mostra a figura 10.

```
//verifica se a imagem é maior que a janela
if((largura_imagem > win_width) || (altura_imagem > win_height)){

  printf("A resolucao da imagem eh maior que a da janela!\n");
  exit(-1);
}
```

Figura 10 – Verificação das dimensões da imagem

Se o cabeçalho for válido, será realizado a alocação dinâmica da matriz que irá armazenar os valores de cada pixel da imagem. Como mostra a figura 11.

```
//verifica se a imagem possui um cabeçalho válido
if(largura_imagem > 0 && altura_imagem > 0 && valor_maximo_pixel <= 255){

  matriz = malloc (altura_imagem * sizeof(pixel));

  //aloca a matriz
  for(int i=0; i<altura_imagem; i++){
    matriz[i] = malloc (largura_imagem * sizeof(pixel));
  }
```

Figura 11 – Alocação dinâmica da matriz de pixels

Agora será necessário identificar qual é o tipo da imagem, já que serão utilizados maneiras diferentes de leitura dependendo que tipo for. Se a imagem for P2, quer dizer que seus pixels estão armazenados no arquivo como um texto e se trata de uma imagem PGM (preto e branco). Isso quer dizer que cada pixel terá apenas um valor RGB, então eles serão normalizados e armazenados nas três variáveis, como mostra a figura 12.

```
//verifica se o identificador é p2
if(strcmp(id, "P2") == 0){

  for(int i=0; i<altura_imagem; i++){
    for(int j=0; j<largura_imagem; j++){
      float rgb;
      fscanf(imagem, "%f", &rgb);

      //insere os valores rgb já normalizados (com valor entre 0 e 1)
      matriz[i][j].r = rgb/valor_maximo_pixel;
      matriz[i][j].g = rgb/valor_maximo_pixel;
      matriz[i][j].b = rgb/valor_maximo_pixel;
    }
  }

  fclose(imagem);
  return matriz;
}
```

Figura 12 – Leitura de uma imagem que possui o tipo “P2”

Se a imagem for P3, quer dizer que seus pixels estão armazenados no arquivo como um texto e se trata de uma imagem PPM (colorida). Isso quer dizer que cada pixel terá três valores (um para cada canal de cor), então eles serão normalizados e armazenados nas suas respectivas variáveis, como mostra a figura 13.

```
//verfica se o identificador é p3
if(strcmp(id, "P3") == 0){
  for(int i=0; i<altura_imagem; i++){
    for(int j=0; j<largura_imagem; j++){
      float r, g, b;
      fscanf(imagem, "%f %f %f", &r, &g, &b);

      //insere os valores rgb já normalizados (com valor entre 0 e 1)
      matriz[i][j].r = r/valor_maximo_pixel;
      matriz[i][j].g = g/valor_maximo_pixel;
      matriz[i][j].b = b/valor_maximo_pixel;
    }
  }

  fclose(imagem);
  return matriz;
}
```

Figura 13 – Leitura de uma imagem que possui o tipo “P3”

Se a imagem for P5, quer dizer que seus pixels estão armazenados no arquivo em binário e se trata de uma imagem PGM (preto e branco). Isso quer dizer que cada pixel terá apenas um valor RGB, então eles serão normalizados e armazenados nas três variáveis, como mostra a figura 14.

```
//verifica se o identificador é p5
if(strcmp(id, "P5") == 0){
  for(int i=0; i<altura_imagem; i++){
    for(int j=0; j<largura_imagem; j++){
      int rgb;
      fread(&rgb, 1, 1, imagem);

      //insere os valores rgb já normalizados (com valor entre 0 e 1)
      matriz[i][j].r = rgb/(1.0 * valor_maximo_pixel);
      matriz[i][j].g = rgb/(1.0 * valor_maximo_pixel);
      matriz[i][j].b = rgb/(1.0 * valor_maximo_pixel);
    }
  }

  fclose(imagem);
  return matriz;
}
```

Figura 14 – Leitura de uma imagem que possui o tipo “P5”

Se a imagem for P6, quer dizer que seus pixels estão armazenados no arquivo em binário e se trata de uma imagem PPM (colorida). Isso quer dizer que cada pixel terá três valores (um para cada canal de cor), então eles serão normalizados e armazenados nas suas respectivas variáveis, como mostra a figura 15.

```
//verifica se o identificador é p6
if(strcmp(id, "P6") == 0){
  for(int i=0; i<altura_imagem; i++){
    for(int j=0; j<largura_imagem; j++){
      unsigned char currentPixel[3];
      fread(currentPixel, 3, 1, imagem);

      //insere os valores rgb já normalizados (com valor entre 0 e 1)
      matriz[i][j].r = currentPixel[0]/(1.0 * valor_maximo_pixel);
      matriz[i][j].g = currentPixel[1]/(1.0 * valor_maximo_pixel);
      matriz[i][j].b = currentPixel[2]/(1.0 * valor_maximo_pixel);
    }
  }

  fclose(imagem);
  return matriz;
}
```

Figura 15 – Leitura de uma imagem que possui o tipo “P6”

Se o arquivo possuir um cabeçalho inválido, será retornado uma mensagem de erro e a aplicação será fechada.

3.5 Função initData

A função initData será responsável por realizar a criação dos vértices, através das coordenadas X, Y e valores das cores de cada pixel, que serão utilizados na visualização dos modelos. A função receberá como parâmetro a matriz de pixels criada na função de leitura da imagem, apresentada na subseção anterior. Primeiramente será realizada alocação dinâmica no vetor de vértices. Como cada pixel terá seis vértices (1 para canto e os dois da diagonal secundário repetidos), cada vértice terá seis valores (as coordenadas X, Y e Z, onde Z sempre será 0.0, os valores do R, G e B). A alocação será realizada como mostra a figura 16.

```
void initData(pixel **matriz){

  //cada pixel terá 6 vértices
  //como cada vértice terá seis valores, como mostrado abaixo:
  //ordem dos valores do vértice: X, Y, Z (0.0), R, G, B
  //o tamanho do vértice será: altura_imagem * largura_imagem * 36
  float *vertices;
  int quantidade_vet = 0;

  vertices = (float *) malloc(altura_imagem * largura_imagem * 36 * sizeof(float));

  if(!vertices){
    printf("Erro ao alocar memória!\n");
    exit(-1);
  }
```

Figura 16 – Alocação dinâmica do array de vértices

Depois de realizar a alocação dinâmica, a matriz será percorrida em cada pixel para criar os seix vértices que cada um terá. Os vértices deverão ser criados no senti anti-horário para que a visualização dos modelos e das cores da imagem sejam visualizadas da maneira correta. O primeiro triângulo da malha triangular composto pelos três primeiros vértices e que possuem as coordenadas (j, i), (j, i+1) e (j+1, i), respectivamente, como mostra a figura 17.

```
for(int i=0; i<altura_imagem-1; i++){
  for(int j=0; j<largura_imagem-1; j++){

    //para cada pixel teremos 6 vértices
    //um para cada canto do quadrado (pixel) e dois repetidos para a diagonal secundária

    //vértice 1
    float vet_aux[6] = {normaliza_eixo_x(j), normaliza_eixo_y(i), 0.0, matriz[i][j].r, matriz[i][j].g,
     matriz[i][j].b};

    for(int k=0; k<6; k++){
      vertices[quantidade_vet++] = vet_aux[k];
    }

    //vértice 2
    float vet_aux2[6] = {normaliza_eixo_x(j), normaliza_eixo_y(i+1), 0.0, matriz[i+1][j].r,
     matriz[i+1][j].g, matriz[i+1][j].b};

    for(int k=0; k<6; k++){
      vertices[quantidade_vet++] = vet_aux2[k];
    }

    //vértice 3
    float vet_aux3[6] = {normaliza_eixo_x(j+1), normaliza_eixo_y(i), 0.0, matriz[i][j+1].r,
     matriz[i][j+1].g, matriz[i][j+1].b};

    for(int k=0; k<6; k++){
      vertices[quantidade_vet++] = vet_aux3[k];
    }
```

Figura 17 – Criação dos vértices do primeiro triângulo da malha triangular

Para cada vértice será passado primeiro o valor normalizado do eixo x, seguido do valor normalizado do eixo y, 0.0 (que representa o valor de Z) e os valores das cores que já foram normalizados previamente (seguindo a ordem do RGB). O segundo triangulo da malha triangular composto pelos três últimos vértices e que possuem as coordenadas (j, i+1), (j+1, i+1) e (j+1, i), respectivamente, como mostra a figura 18.

```
//vértice 4
float vet_aux4[6] = {normaliza_eixo_x(j), normaliza_eixo_y(i+1), 0.0, matriz[i+1][j].r,
 matriz[i+1][j].g, matriz[i+1][j].b};

for(int k=0; k<6; k++){
  vertices[quantidade_vet++] = vet_aux4[k];
}

//vértice 5
float vet_aux5[6] = {normaliza_eixo_x(j+1), normaliza_eixo_y(i+1), 0.0, matriz[i+1][j+1].r,
 matriz[i+1][j+1].g, matriz[i+1][j+1].b};

for(int k=0; k<6; k++){
  vertices[quantidade_vet++] = vet_aux5[k];
}

//vértice 6
float vet_aux6[6] = {normaliza_eixo_x(j+1), normaliza_eixo_y(i), 0.0, matriz[i][j+1].r,
 matriz[i][j+1].g, matriz[i][j+1].b};

for(int k=0; k<6; k++){
  vertices[quantidade_vet++] = vet_aux6[k];
}
```

Figura 18 – Criação dos vértices do segundo triângulo da malha triangular

As funções que seguem no final da função initData são nativas do OpenGL e irão trabalhar com o array dos vértices, buffers e atributos. Como representado na figura 19.

```
//Vertex array.
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

//Vertex buffer
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * altura_imagem * largura_imagem * 36 , vertices,
 GL_STATIC_DRAW);

//Set attributes.
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6*sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6*sizeof(float), (void*)(3*sizeof(float)));
glEnableVertexAttribArray(1);

//Unbind Vertex Array Object.
glBindVertexArray(0);
```

Figura 19 – Funções nativas do OpenGL que trabalha com o array de vértices

3.6 Função main

A função main será apresentada para poder visualizar melhor como está organizado e sendo chamado nossas funções. Ela tratará problemas em relação a como os parâmetros foram passados, depois chamará a função de leitura da imagem, como mostra a figura 20.

```
pixel **matriz_pixels;

if (argc != 2){
  printf("Erro ao passar os parâmetros!\n");
  printf("Exemplo: ./modelo images/nome_da_imagem.pgm(ou ppm)");
  return 0;
}

char nome_imagem[250];

strcpy(nome_imagem, argv[1]);

matriz_pixels = le_imagem(nome_imagem);
```

Figura 20 – Verificação dos argumentos que foram passados como parâmetro

Depois serão utilizadas funções nativas do OpenGL para a criação da janela da aplicação, a matriz de pixels obtida será passada para a initData e, depois, serão utilizadas outras funções nativas para a criação dos shaders, display da janela e configuração da função das teclas do teclado, como mostra a figura 21.

```
glutInit(&argc, argv);
glutInitContextVersion(3, 3);
glutInitContextProfile(GLUT_CORE_PROFILE);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
glutInitWindowSize(win_width, win_height);
glutCreateWindow("Lista 1 - Computacao Grafica");
glewExperimental = GL_TRUE;
glewInit();

initData(matriz_pixels);
initShaders();

glutReshapeFunc(reshape);
glutDisplayFunc(display);
glutKeyboardFunc(keyboard);

glutMainLoop();

//libera o espaço alocado pela matriz de pixels
free(matriz_pixels);
return 0;
```

Figura 21 – Funções nativas do OpenGL usadas no display da janela, shaders e teclado

4. Apresentação das escolhas feitas para gerar uma boa visualização dos modelos.

A função keyboard será responsável pela visualização dos modelos. Dentro dela serão configuradas as teclas responsáveis por ativar o modelo de malha triangular ou nuvem de pontos, como podemos visualizar na figura 22. Essa função será utilizada dentro da função nativa do OpenGL, a glutKeyboardFunc.

```
//configura as teclas necessárias do teclado
void keyboard(unsigned char key, int x, int y){

  switch(key){

    case 27:
      exit(0);
    case 'q':
    case 'Q':
      exit(0);
      break;
    case 'v':
    case 'V':
      if(type_primitive == GL_POINTS){
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        type_primitive = GL_TRIANGLES;
      }else{
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        type_primitive = GL_POINTS;
      }

      break;
  }

  glutPostRedisplay();
}
```

Figura 22 – Função de configuração das teclas do teclado

As teclas “Q” e “q” foram configuradas para que seja possível finalizar a tela de execução do programa. As teclas “v” e “V” serão utilizadas para poder alternar entre a visualização dos dois modelos. Como mostrado na seção anterior, a variável “type_primitive” será global e inicializada com o valor “GL_POINTS”. Isso significa que o primeiro modelo mostrado assim que for feita a execucação do programa será o de nuvem de pontos. Caso o usuário pressione a tecla responsável por mudar a visualização, como o valor da variável “type_primitive” é “GL_POINTS”, o modelo que será apresentado na imagem será o de malha triangular e o valor da variável “type_primitive” será alterado para “GL_TRIANGLES”. Com essa configuração será possível alternar entre os dois modelos de forma instantânea e fechar a aplicação de uma maneira mais fácil.

5. Discussão sobre as conclusões e dificuldades encontradas e falhas do seu programa.

Podemos concluir que o trabalho cumpre o que foi proposto. Consegue ler tanto imagens PGM quanto PPM e apresenta corretamente a malha triangular e nuvem de pontos para as diferentes imagens. A maior dificuldade ao decorrer do trabalho foi encontrar uma fórmula de normalização das coordenadas e que apresentasse uma boa visualização dos modelos. O nosso programa falha em reconhecer imagens que possuam mais de uma linha de comentário seguidas, porém funciona corretamente caso não tenha comentários ou comentários de apenas uma linha.

Para finalizar, será mostrado a malha triangular e a nuvem de pontos em duas imagens diferentes. Iremos notar que quanto maior a imagem, mais difícil fica para poder visualizar com clareza o

modelo de malha triangular. Quanto menor a imagem, os modelos serão apresentados de uma forma mais clara e detalhada.

Figura 23 - Imagem colorida (512x512) original

Figura 24 - Imagem colorida (512x512) nuvem de pontos

Figura 25 - Imagem colorida (512x512) malha triangular

6. Vídeo da apresentação

O vídeo da apresentação do trabalho pode ser acessado utilizando o link a seguir: https://drive.google.com/drive/folders/1SUE9Rg7YoHUMlOiBAnR5LEjoTVjEPc8_?usp=sharing

7. Referências

BURKARDT, John; FSU. PGMA Files. Disponível em: <https://people.sc.fsu.edu/~jburkardt/data/pgma/pgma.html>. Acessado em 08/10/2020.

Coordinate Systems. Learn OpenGL. Disponível em: <https://learnopengl.com/Getting-started/Coordinate-Systems>. Acessado em: 07/10/2020.

Hello Triangle. Learn OpenGL. Disponível em: <https://learnopengl.com/Getting-started/Hello-Triangle>. Acessado em: 06/10/2020.

MOTA, Kleber. OpenGL Desenhando Polígonos. Disponível em: <https://klebermota.eti.br/2015/05/08/opengl-desenhando-poligonos/>. Acessado em: 06/10/2020.

Normalized Device Coordinates - Interactive 3D Graphics. Udacity. Disponível em: <https://www.youtube.com/watch?v=Ck1SH7oYRFM>. Acessado em: 07/10/2020.

OpenGL Documentation. Khronos. Disponível em: <https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/>. Acessado em: 09/10/2020.