Relatorio Trabalho CG

Bruno P. Larsen, 9283872
Luiz Eduardo Dorici, 4165850

April 2019

Contents

1 Introdução

1.1 Separação do código e interação entre alunos

O projeto foi divido em front-end e back-end, para facilitar o desenvolvimento. Bruno ficou responsável pelo front-end e Luiz ficou responsável pelo back-end.

O front-end consistia do gerenciador gráfico, a implementação de botões e gerenciamento de input; O back-end consistia da implementação da ET, da AET e o algoritmo de preenchimento em si.

A comunicação dos alunos ocorreu através do app Telegram, e o codigo foi desenvolvido usando GitHub.

1.2 Visão Geral

O projeto foi feito primariamente em OpenGL, com apoio da biblioteca glfw3 e a biblioteca auxiliar glad.

As únicas bibliotecas necessárias para rodar o programa são OpenGL versao 4.5, glfw versao 3 e as bibliotecas padrões de C++. No entanto, ele só foi testado no sistema Linux, na distribuição Antergos.

1.3 Compilação e execução

O projeto contém um arquivo makefile, permitindo que o projeto inteiro seja compilado usando o comando "make".

Da mesma maneira, para executar o programa basta usar o comando "make run".

1.4 interface Gráfica

A interface gráfica é exemplificado na imagem 1.

A interface pode ser divida em 4 partes, esquerda, direita e central na parte inferior, e o resto da tela.

O ícone inferior esquerdo é usado para resetar os vértices; O ícone inferior direito termina a seção de input e chama a funcao de preenchimento do polígono; a parte inferior central é a paleta de cores usada para pintar o poligono desenhado.

O resto da tela é o quadro em que o usuário pode usar para entrar os vértices do polígono. A partir do terceiro clique, o poligono começa a ser delineado, sem preenchimento.

O último detalhe importante da interface gráfica é que uma vez que o botão de finalização (preenchimento do poligono) é pressionado, todos os botões são desabilitados, com excecao do botão de reset.

1.5 Algoritmo de Preenchimento

O algoritmo de preenchimento recebe um vector do tipo float da interface e transforma em um vector do tipo int nomeado pontos que por sua vez é armazenado em um vector do tipo arestas. A classe arestas é utilizada para ar-

mazenar os dados necessários para a criação e utilização das estruturas ET(edges table) e AET(active edge-table), contendo xmax, xmin, ymax e ymin das arestas além das variaveis xval(x do vertice ymin) e incremento(1/m). A classe arestas também contém um metodo para cálculo da variavel incremento que é chamado toda vez que uma nova aresta é criada passando o par (x1,y1)(x2,y2).

Os vetores do tipo vector(arestas) também utilizam a estrutura de tuplas do c++ armazenando cada vertice como uma tupla (x,y) e cada aresta como um conjunto de duas tuplas(x1,y1)(x2,y2). Dessa forma podemos criar as estruturas de maneira facil e rodar o algoritmo de preenchimento de poligonos em cima delas, ao processar cada bloco de pixels que devem ser pintados o algoritmo armazena esses pixels em um vector do tipo float, e no final, após preencher todo o poligono retorna o vetor com todos os pontos para a interface poder desenhar o poligono na tela.

2 Modularização

O código foi altamente modularizado para facilitar o desenvolvimento em paralelo e a escalabilidade do código para projetos futuros. Cada subseção será dedicada as classes criadas.

2.1 VAO_INFO

Essa struct foi criada para armazenar todas as informações importantes dos buffers usados para desenhar cada uma das figuras na tela.

Uma struct foi escolhida, pois implementação de um destrutor que desaloca os Buffers cusa problemas, devido a passagem de objetos por valor.

Os métodos implementados pela struct são os seguintes:

- start: Implementado como um construtor para a struct; Esse método tem overload para que pudessemos ter a mesma chamada para desenhos usando vértices ou índices
- finish: Implementado como um destrutor; Desaloca os buffers necessarios
- updateVertex: faz o update dos buffers.
- setColor: define a cor a ser usada no buffer.
- operator ==: overload de igualdade entre structs; A única parte importante da struct é o ID do VAO, por isso é o único membro analisado.
- operator =: overload de atribuição entre structs; Aceita apenas os valores que podem ser mudados sem perda de informacao (cor,contagem de vertices e qual topologia usar)

2.2 DisplayManager

Essa classe é a principal do trabalho. Ela contém todas as informações pertinentes a janela que está sendo usada, e gerencia todas as chamadas diretas às bibliotecas gráficas, com exceção da geração de buffers.

Os métodos implementados para essa classe são:

- Construtor: Inicia os membros da classe com valores impossíveis, e inicia a biblioteca GLFW, indicando a versão a ser usada e que a tela nao deve ser redimensionada
- Destrutor: Garante que todos os buffers de vertices são corretamente desalocados e termina o funcionamento da biblioteca GLFW
- init: Inicia a janela, faz a leitura e compilação do shader e arruma as variáveis de tamanho, tanto internas à classe quanto ao shader.
- run: Contém o loop principal do programa; Exterior ao loop, é definida a cor de fundo; no interior, processa-se os inputs, renderiza-se os vértices e troca os buffers para mostrar o que foi renderizado;

  O processo de renderização consiste de percorrer a lista de VAO_INFO registrados, obter as informações pertinentes a renderização e chamar a função desejada (renderizando a partir de VBO ou de EBO).
- register_VAO: Armazena uma cópia das informações de renderização de um objeto
- deregister_VAO: Remove a cópia da informação e desaloca os buffers;
- update_VAO: Atualiza as informações de renderização de um objeto, como contagem de vertices ou cor.
- registerMouseButtonCallback: Registra uma função com callback para quando houver um clique de Mouse;
- getWindow: Retorna a janela aberta
- getShader: Retorna o programa de shader sendo utilizado
- setClearColor: Modifica a cor de fundo da janela;

Além da classe, o arquivo *dm.cpp* também contém 3 funções auxiliares:

- processInput: função que processa input através de teclado;
- readShader: função que le um arquivo que contém o codigo para shaders que serão usados pelo programa
- compileShader: função que faz a compilação dos shaders lidos e lida com os erros possíveis;

2.3 Botao

Classe responsável por inserir responsividade a interface grafica. Essa classe sabe as coordenadas dos seus vértices, os valores maximos e minimos de X e Y (usados para determinar se o clique ocorreu dentro ou fora do botão), se o botão está ativo, qual a cor que o botão deve ter quando está ativo e um ponteiro para a função que define a ação do botão.

Foi usado um ponteiro para função por ter muito pouca reutilização de classes ou métodos, ou seja, cada botão seria uma derivação única, ou teria um método unico para realizar a ação.

Os métodos da classe são:

- Construtor: Recebe os vértices, os indíces, a cor e a ação que o botão deve ter;
- Destrutor: Não precisa fazer nada, já que o DisplayManager ja desaloca todos os buffers;
- set: usado pelo construtor, modifica o valor de todas as variaveis, atualizando os valores do botao.
- press: chama a função de ação;
- inside: Calcula se a posiçaõ passada está dentro do botão;
- disable: Troca a cor do botão para preto, e desabilita as ações
- enable: reverte o botao para a cor original e habilita as ações
- getID: retorna o VAO_INFO do botão. Antigamente retornava VAO_ID, por isso tem esse nome;
- get$XXXXX$: retorna o respectivo membro;

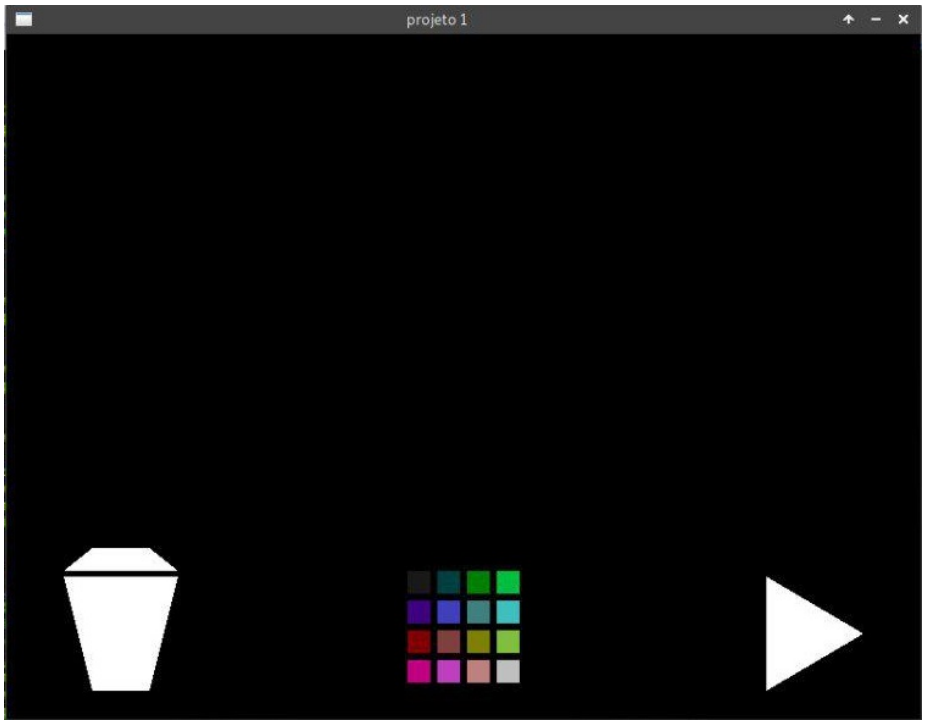

Figure 1: Exemplo da Interface Grafica