Visualizador de Objetos 3D OFF

Introdução:

OpenGL e *QT* são bibliotecas gráficas bem populares e interessantes de C++. A diferença das duas são o seu uso final, enquanto *OpenGL* tem como foco aplicações gráficas de Computação Gráficas e Jogos, *QT* tem o foco na criação de bibliotecas e interfaces.
Com isso em mente, o melhor a se fazer é juntas as duas bibliotecas e algumas técnicas de manipulação de memória do G++ 10, para criar uma aplicação com interface simples e limpa para visualizar objetos 3D, que possa servir como base para criação de visualizadores maiores como o *Blender*, por exemplo.

Implementação:

Primeiro é necessário criar um novo projeto, que deve ser do tipo *QT Gui Project*. Após nomeado e criado, alguns arquivos serão gerados pelo programa:

- **myqtglproject.pro:** Contém as configuração do projeto.
- **mainwindow.cpp/hpp:** Responsável pela janela principal do programa.
- **main.cpp:** Contém as funções de C++.
- **mainwindow.ui:** Arquivo XML com as informações da interface do programa.

Após criado, uma classe denominada *GLWidget* que estende a classe *QGLWidget* deve ser adicionada aos arquivos do projeto. Tal classe conterá os seguintes métodos, é possível visualizar o código fonte da aplicação nas referências.

1. **explicit GLWidget(QWidget *parent = 0):** Construtor padrão do que ira inicializar algumas variáveis padrões do programa.
2. **virtual ~*GLWidget*():** Destrutor do programa que destrói todos os recursos indesejados na memória ao fim da execução.
3. **void showFileOpenDialog():** *Slot* responsável de pegar a *string* com o nome do arquivo *OFF*, do objeto.
4. **void statusBarMessage(QString s):** Sinal para a interface, com o objetivo de mostrar a quantidade de faces e vértices do objeto renderizado.
5. **void *initializeGL*():** Inicializa alguma variáveis sempre que um novo objeto é escolhido.
6. **void *resizeGL*(int width, int height):** Efetua o redimensionamento da matriz de projeção sempre que o objeto sofre alguma interferência de zoom provido do mouse.
7. **void *paintGL*():** Renderiza todas as informações salvas do objeto.
8. **void *mouseMoveEvent*(QMouseEvent *event):** Cuida do movimento do mouse em relação a janela.
9. **void *mousePressEvent*(QMouseEvent *event):** Cuida do pressionar no mouse.
10. **void *mouseReleaseEvent*(QMouseEvent *event):** Cuida do mouse liberado.
11. **void *wheelEvent*(QWheelEvent *event):** Cuida do *scroll* do mouse.
12. **void readOFFFile(const QString &fileName):** Lê o objeto que sera renderizado na tela.
13. **void triangulation():** Transforma todos os polígonos do objeto em triângulos.
14. **std::vector<std::vector<unsigned int>> polygonTriangulation(std::vector<unsigned int> polygon):** Transforma um polígono qualquer em um triangulo.
15. **void genNormals():** Responsável por calcular os vetores normais de cada vértice do objeto.
16. **void genTextCoordsCylinder():** Responsável por calcular as coordenadas das texturas do objeto.
17. **void genTangents():** Responsável por calcular as as tangentes do objeto.

18. **void createShaders():** Responsável pela criação de cada um dos *shaders*.
19. **void destroyShaders():** Responsável pela destruição dos *shaders* utilizados no programa.
20. **void createVBOs():** Cria os *Vertex Buffer Objects*, que são responsáveis pelo objeto já pronto para ser renderizado criado de uma maneira otimizada.
21. **void destroyVBOs():** Remove qualquer *VBO* existente de outros objetos.
22. **void *keyPressEvent*(QKeyEvent *event):** Responsável pelos sinais do teclado, que são utilizados para a troca de *Shaders*.

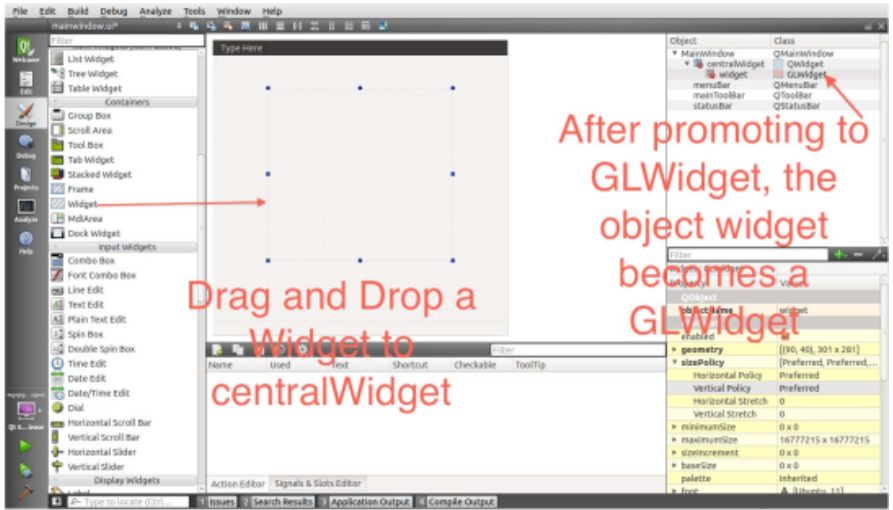
Além destes métodos, ele também possui alguns atributos listados a seguir:

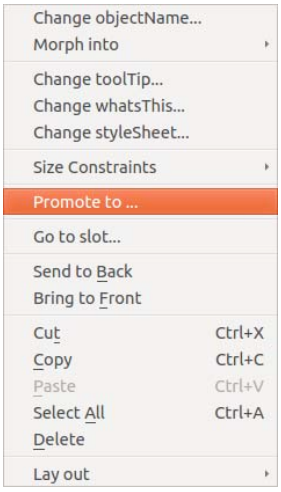
- **unsigned int numVertices:** Quantidade de vértices objeto.
- **unsigned int numFaces:** Quantidade de faces triangulares do objeto.
- **std::vector<QVector4D> vertices:** Todos os vértices do objeto, fisicamente.
- **std::vector<unsigned int> indices:** Todas as faces triangulares, representados por 3 pontos em sequencia no vetor.
- **std::vector<std::vector<unsigned int>> preIndices:** Todas as faces poligonais do programa.
- **std::vector<QVector3D> normals:** Vetores normais de cada um dos vértices.
- **std::vector<QVector2D> texCoords:** Todas as coordenadas de texturas do objeto.
- **std::vector<QVector4D> tangents:** Todas as tangentes da face de um objeto.
- **std::unique_ptr<QGLShader> vertexShader:** Contém o *shader* de transformação de vértices.
- **std::unique_ptr<QGLShader> fragmentShader:** Contém o *shader* que cria os fragmentos.
- **std::unique_ptr<QGLShaderProgram> shaderProgram:** Contém a junção do *shader* de vértices e fragmentos.
- **unsigned int currentShader:** Qual o *shader* está em uso.

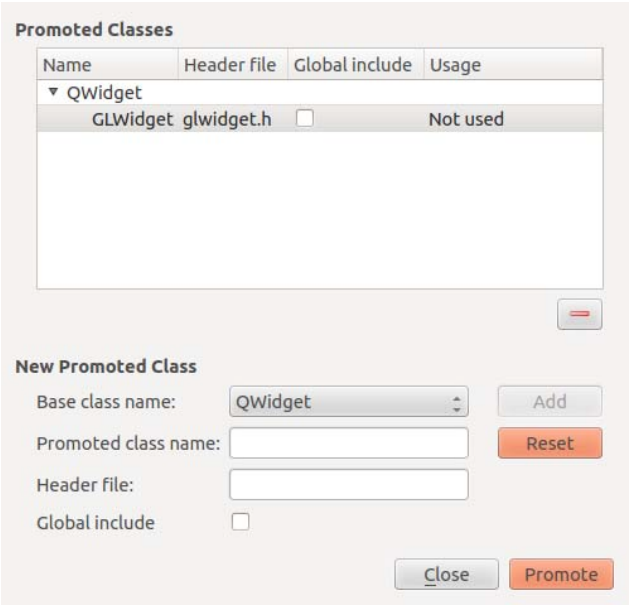
- **std::unique_ptr<QGLBuffer> vboVertices;**
- **std::unique_ptr<QGLBuffer> vboNormals;**
- **std::unique_ptr<QGLBuffer> vboTexCoords;**
- **std::unique_ptr<QGLBuffer> vboTangents;**
- **std::unique_ptr<QGLBuffer> vboIndices;**

- **QMatrix4x4 modelViewMatrix:** Contém a matriz do do objeto.
- **QMatrix4x4 projectionMatrix:** Contem a matriz de projeção do objeto.
- **int texID[2]:** Contém o id das texturas.
- **QTimer timer:** Contador de tempo.
- **Camera camera:** Informações sobre a câmera.
- **Light light:** Informações sobre a luz no objeto.
- **Material material:** Informações sobre a difusão natural do objeto.
- **TrackBall trackBall:** Classe que transforma as informações do mouse em valores e altera a matriz de projeção.
- **double zoom:** Salva o zoom atual do objeto.

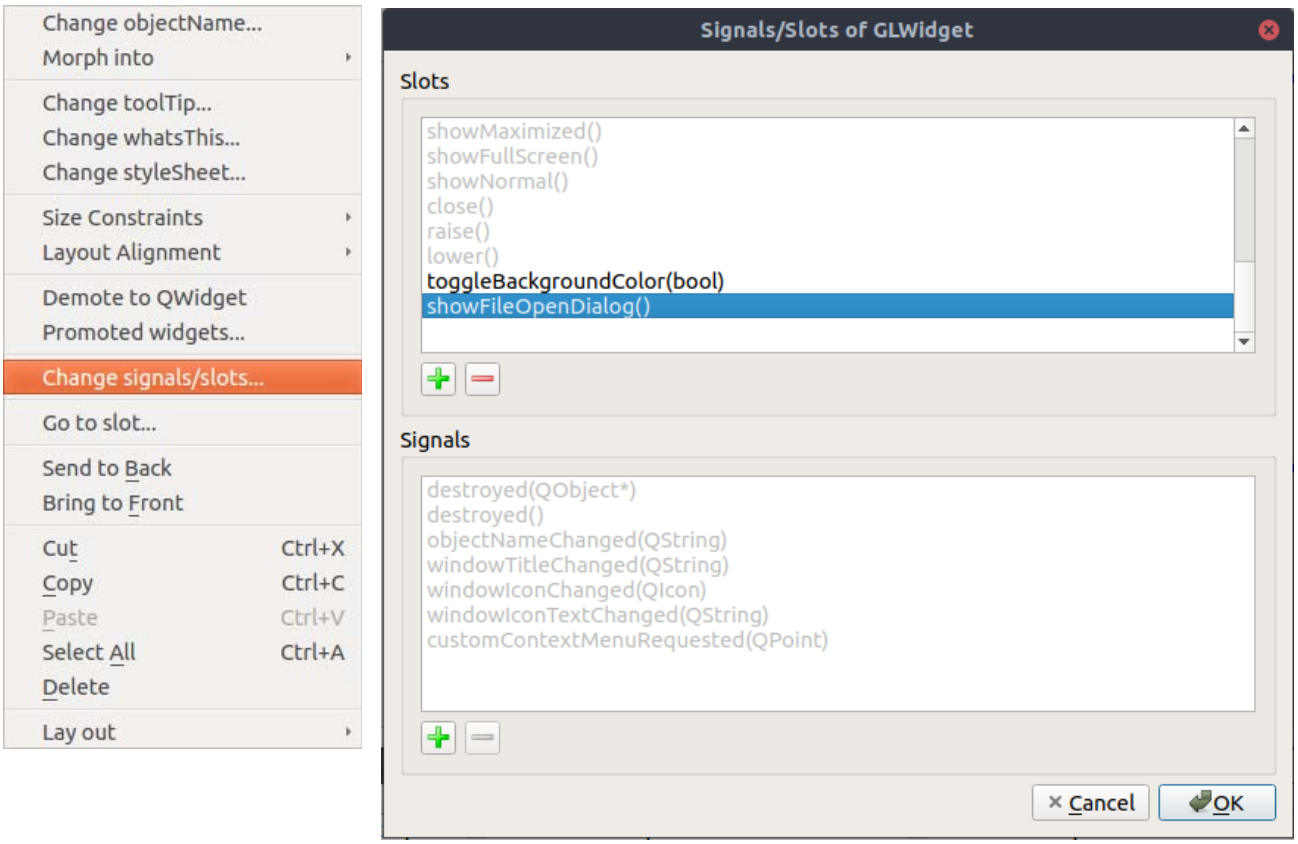
Após criada a classe principal e sua auxiliares, basta criar a interface para o programa.
Para editar a interface existem dois métodos, editar a interface diretamente pelo *mainwindow.ui*, ou dar um duplo clique no arquivo e editar pelo próprio editor do *QT Creato*r. Neste documento será utilizado o segundo método. Para criar o visualizador de objeto, basta selecionar um *widget* na divisão de *containers* e arrastar para a janela, em seguida promover o objeto para a classe *GLWidget*.







Por fim basta criar na interface a opção para carregar os arquivos do tipo *OFF*, para isto o primeiro passo é ir no gerenciador de interface do *QT Creator*, ir na parte superior da interface mostrada selecionar **"Type Here"** e digitar **"Arquivo"**, em seguida clicar na opção **"Arquivo"** → **"Type Here"**, e digitar **"Abrir"**. Logo após importar um novo sinal e *slot*, assim como mostrado nas imagens a seguir:

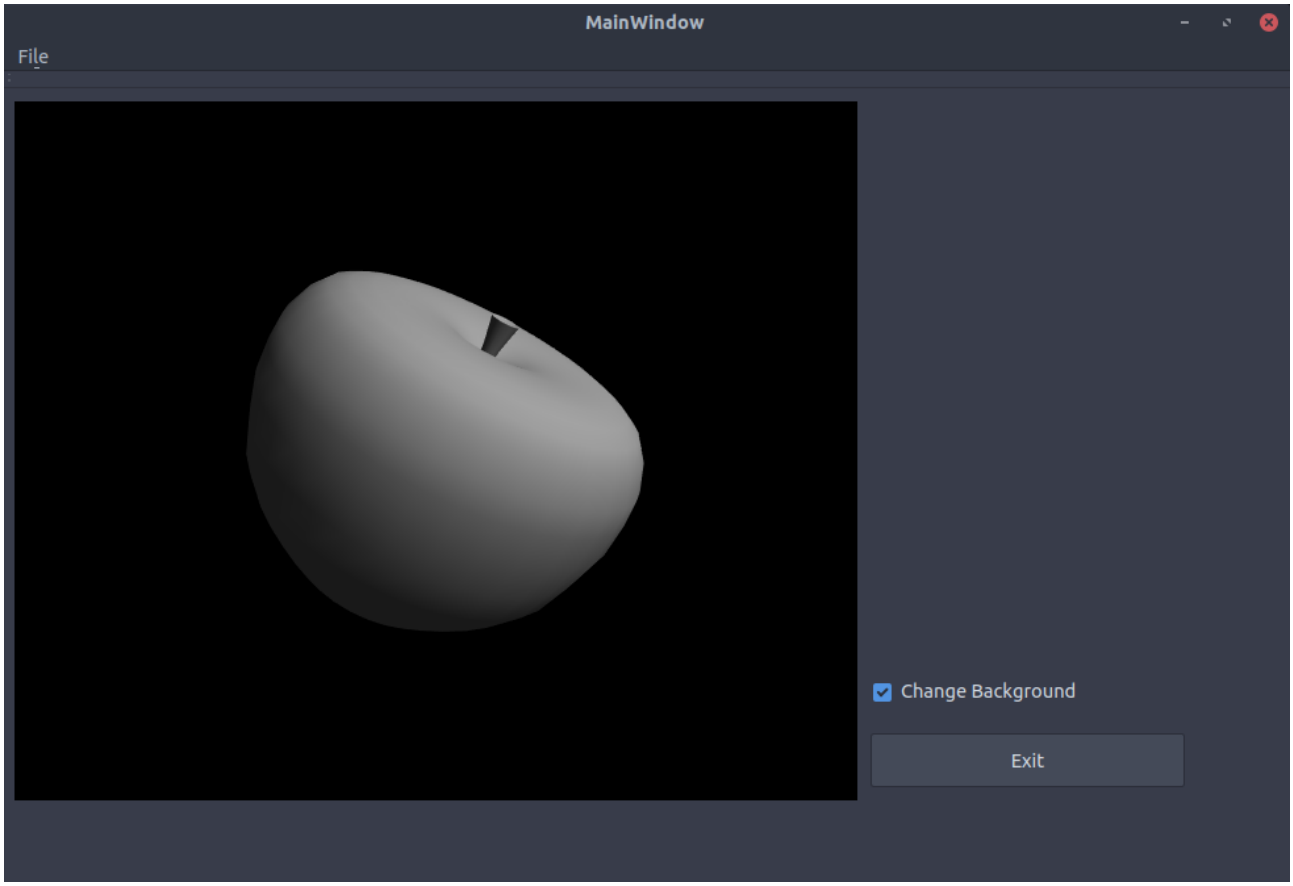


O QT automaticamente cria um método *actionOpen*, da classe *QAction*, onde os métodos destas classes são utilizados na função 3 da lista de métodos da classe *GLWidget*.

Testes e Resultados:

O programa foi testado em um *GNU/Linux Ubuntu 18.04.5 LTS*, com um processador *I3-3110M*, chip gráfico *Intel Graphics 4000* e 8 GB de RAM.
A IDE utilizada foi *QT Creator 4.5.2* e *QT 5.9.5*, além do compilador na versão 10.1.0 e OpenGL na versão 3.0 Mesa 20.0.8.

A metodologia de teste foi simples, abrir um arquivo e visualizar o modelo por inteiro e verificar se existe algum erro no visualizador.



Conclusão

QT e OpenGL formam um ótima dupla, não só no contexto profissional ou desenvolvimento acadêmico, mas também com o intuito educacional, como mostrado neste documento.
Ainda assim a interface é bem simples, mas com pouco tempo de desenvolvimento é possível criar interfaces mais interessantes e algumas outras funcionalidades.

Referências

Gois, Joao & Batagelo, Harlen. (2009). Interactive Graphics Applications with OpenGL Shading Language and Qt. 10.1109/SIBGRAPI-T.2012.10.