Relatório Técnico Final – Editor de Imagens SVG

Autor: Gustavo do Nascimento Pereira

Matrícula: 2024001644

- **1. Introdução**
- Este relatório detalha a implementação final do projeto da disciplina de Programação Orientada a Objetos, cujo tema escolhido foi um Editor de Imagens SVG com Polígonos. O resultado é uma aplicação com interface gráfica (GUI) construída com C++ e o framework Qt 6. O programa permite ao usuário criar, selecionar, editar (arrastando vértices) e salvar polígonos de forma interativa, além de importar formas de arquivos SVG existentes, cumprindo todos os requisitos obrigatórios da especificação.

- ***2. Arquitetura Final: Model-View-Controller (MVC)***

- A arquitetura Model-View-Controller (MVC) foi a base para o design do software, garantindo a separação de responsabilidades e a manutenibilidade do código. As classes do projeto foram organizadas da seguinte forma:
- **Modelo (Model):** Camada responsável pela representação dos dados e pela lógica de negócio.
    - Classes: Canvas, Shape (abstrata), Polygon, Point.
- **Visão (View):** Camada responsável pela apresentação dos dados e pela interface com o usuário.
    - Classes: MainWindow, CanvasWidget.
- **Controlador (Controller):** Camada que atua como intermediária, recebendo a entrada do usuário, manipulando o Modelo e atualizando a Visão.
    - Classe: EditorController.
- A comunicação entre a Visão e o Modelo é sempre mediada pelo Controlador. O método paintEvent da CanvasWidget é responsável por renderizar o estado atual do Modelo, garantindo a separação das camadas.

- ***3. Mapeamento dos Requisitos de Programação Orientada a Objetos***

- A seguir, um mapa detalhado de como cada requisito de POO, conforme a seção 3 da especificação do projeto, foi atendido no código-fonte.
- **1. Abstração & Encapsulamento**
- **Implementação:** Os dados das classes do Modelo, como o vetor de vértices em Polygon, são mantidos como private. O acesso a esses dados é controlado por uma interface de métodos public (ex: getVertices(), moveVertex(), setSelected()), que garante a manipulação segura e controlada do estado interno dos objetos.

- **2. Classes e Objetos**
- **Implementação:** O projeto foi modularizado em classes com responsabilidades únicas e coesas, seguindo o padrão MVC. A CanvasWidget tem a única responsabilidade de desenhar e capturar eventos de mouse, enquanto a EditorController lida com a lógica de aplicação (ex: importSvgFromFile).
- **3. Herança & Polimorfismo**
- **Implementação:** A herança é central na arquitetura, com Polygon herdando da classe base abstrata Shape. As classes da GUI também herdam de classes do Qt (MainWindow de QMainWindow, CanvasWidget de QWidget). O polimorfismo é demonstrado com métodos virtual como setSelected() na classe Shape e paintEvent() e os eventos de mouse em CanvasWidget.
- **4. Composição vs. Herança**
- **Implementação:** A composição foi usada para modelar relações "tem-um". A MainWindow *tem um* CanvasWidget. O Canvas *tem uma* coleção de Shapes. Um Polygon *tem uma* coleção de Points.
- **5. Polimorfismo Dinâmico (Ligação Tardia)**
- **Implementação:** A lógica de seleção e edição em CanvasWidget::mousePressEvent utiliza dynamic_cast para inspecionar o tipo real do Shape clicado em tempo de execução. Isso permite aplicar a lógica de manipulação de vértices apenas se o objeto for um Polygon, um exemplo claro de polimorfismo dinâmico.
- **6. Gerenciamento de Recursos (RAII)**
- **Implementação:** O Canvas utiliza um std::vector<std::unique_ptr<Shape>> para gerenciar o ciclo de vida das formas. O std::unique_ptr garante que a memória de cada objeto Shape seja liberada automaticamente, prevenindo vazamentos de memória e aplicando o princípio RAII.
- **7. Templates e STL**
- **Implementação:** A Standard Template Library (STL) é usada extensivamente. std::vector é o principal container para a lista de vértices em Polygon e para a lista de formas em Canvas. std::unique_ptr, std::string e std::stringstream também são utilizados.
- **8. Sobrecarga de Operadores**
- **Implementação:** A classe Point sobrecarrega os operadores operator+, operator- e operator==, permitindo a manipulação de coordenadas de forma intuitiva, como visto na lógica de movimentação de vértices.
- **9. Tratamento de Exceções**
- **Implementação:** Na versão CLI (Etapa 2), um bloco try/catch foi implementado no main.cpp para o parsing de comandos do usuário. Ele captura exceções (std::exception) lançadas por std::stod ao tentar converter uma entrada de texto inválida para um número.
- **10. Documentação Técnica e UML**

- **Implementação:** Este relatório e o Diagrama de Classes UML atualizado (uml_diagram_final.png) cumprem este requisito.
- **11. Build Automatizado**
- **Implementação:** O projeto utiliza o **CMake** para um processo de build multiplataforma. O arquivo CMakeLists.txt foi configurado para encontrar e vincular automaticamente os módulos do Qt 6 necessários (Widgets e Xml) durante a configuração.

- ***4. Conclusão***

- O desenvolvimento do projeto permitiu a aplicação prática de um vasto conjunto de conceitos de Programação Orientada a Objetos em um cenário realista. A arquitetura MVC provou-se eficaz em separar as responsabilidades e facilitar a implementação de funcionalidades complexas, como a edição interativa e a importação/exportação de dados. O resultado final é uma aplicação funcional que cumpre todos os requisitos propostos.
-

10. Documentação Técnica e UML

- **Implementação:** Este relatório e o Diagrama de Classes UML (uml_diagram_final.png) cumprem este requisito de documentação.

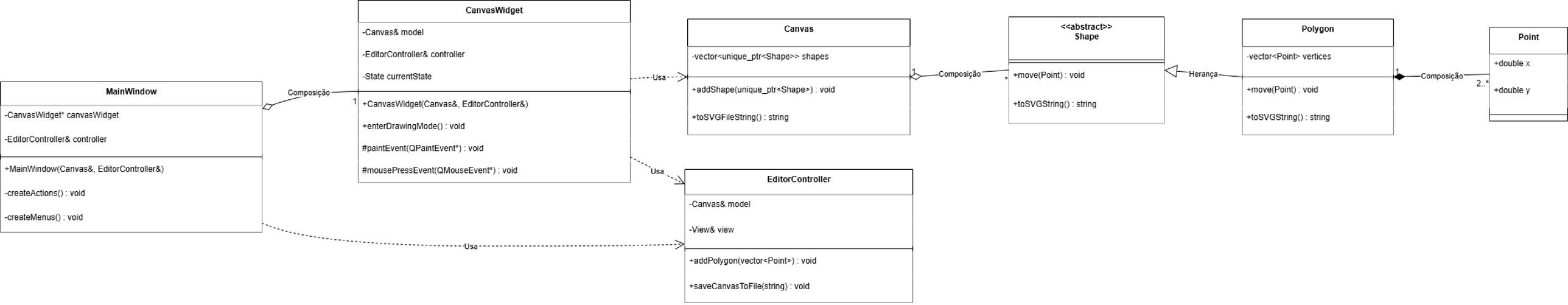
11. Build Automatizado

- **Implementação:** O projeto utiliza o **CMake** para gerenciar o processo de build de forma automatizada e multiplataforma. O arquivo CMakeLists.txt, localizado na raiz do projeto, contém todas as instruções necessárias para configurar e compilar a aplicação com suas dependências (como o Qt 6).

4. Conclusão

O desenvolvimento deste projeto permitiu a aplicação prática de um vasto conjunto de conceitos de Programação Orientada a Objetos em um cenário realista. A arquitetura MVC provou-se eficaz em separar as responsabilidades e facilitar a transição de uma aplicação de terminal para uma com interface gráfica. O resultado final é uma aplicação funcional que cumpre todos os requisitos propostos, solidificando o conhecimento em C++ moderno e no desenvolvimento de GUIs com o framework Qt.

5. Diagrama UML das relações


MainWindow
-CanvasWidget* canvasWidget
-EditorController& controller
+MainWindow(Canvas&, EditorController&)
-createActions() : void
-createMenus() : void
Composição
1
CanvasWidget
-Canvas& model
-EditorController& controller
-State currentState
+CanvasWidget(Canvas&, EditorController&)
+enterDrawingMode() : void
#paintEvent(QPaintEvent*) : void
#mousePressEvent(QMouseEvent*) : void
Usa
Usa
Usa
Canvas
-vector<unique_ptr<Shape>> shapes
+addShape(unique_ptr<Shape>) : void
+toSVGFileString() : string
1
Composição
*
<<abstract>>
Shape
+move(Point) : void
+toSVGString() : string
Herança
Polygon
-vector<Point> vertices
+move(Point) : void
+toSVGString() : string
1
Composição
2..*
Point
+double x
+double y
EditorController
-Canvas& model
-View& view
+addPolygon(vector<Point>) : void
+saveCanvasToFile(string) : void