**ОГЛАВЛЕНИЕ**

[***// Начало Calculations.h //*** 2](#_Toc198831300)

[***// Начало Calculations.cpp //*** 3](#_Toc198831301)

[***// Начало TrajectoryVisualizer.h //*** 6](#_Toc198831302)

[***// Начало TrajectoryVisualizer.cpp //*** 9](#_Toc198831303)

[***// Начало UserInterface.h //*** 17](#_Toc198831304)

[***// Начало UserInterface.cpp //*** 20](#_Toc198831305)

[***// Начало main.cpp //*** 34](#_Toc198831306)

# ***// Начало Calculations.h //***

#ifndef CALCULATIONS\_H

#define CALCULATIONS\_H

#include <vector>

#include <string>

#include <cmath> // Для std::sqrt

#include <iostream> // Для std::cerr

// Параметры симуляции

struct SimulationParameters {

double G = 1.0;

double M = 1.0;

double CENTRAL\_BODY\_RADIUS = 0.01;

double DRAG\_COEFFICIENT = 0.05;

double THRUST\_COEFFICIENT = 0.00;

double DT = 0.001;

int STEPS = 100000;

struct InitialStateParams {

double x = 1.5;

double y = 0.0;

double vx = 0.0;

double vy = 0.8;

} initialState;

};

// Состояние системы

struct State {

double x, y, vx, vy;

};

class Calculations {

public:

Calculations(); // Конструктор по умолчанию

// Основной метод для запуска симуляции

std::vector<State> runSimulation(const SimulationParameters& params);

private:

// Правая часть системы дифференциальных уравнений

static State derivatives(const State& s, const SimulationParameters& params);

// Один шаг интегрирования методом Рунге-Кутты 4-го порядка

static State rungeKuttaStep(const State& s, double dt, const SimulationParameters& params);

};

#endif // CALCULATIONS\_H

***// Конец Calculations.h //***

# ***// Начало Calculations.cpp //***

#include "Calculations.h"

Calculations::Calculations() {

// Конструктор может быть пустым, если нет специфической инициализации

}

// Основной метод для запуска симуляции

std::vector<State> Calculations::runSimulation(const SimulationParameters& params) {

State currentState;

currentState.x = params.initialState.x;

currentState.y = params.initialState.y;

currentState.vx = params.initialState.vx;

currentState.vy = params.initialState.vy;

std::vector<State> trajectoryStates; // Теперь храним полные состояния

trajectoryStates.reserve(static\_cast<size\_t>(params.STEPS) + 1);

trajectoryStates.push\_back(currentState); // Добавляем начальное состояние

double initial\_r\_squared = currentState.x \* currentState.x + currentState.y \* currentState.y;

if (initial\_r\_squared < params.CENTRAL\_BODY\_RADIUS \* params.CENTRAL\_BODY\_RADIUS) {

std::cout << "Столкновение: начальная позиция (" << currentState.x << ", " << currentState.y

<< ") внутри радиуса центрального тела (" << params.CENTRAL\_BODY\_RADIUS << ").\n";

return trajectoryStates;

}

for (int i = 0; i < params.STEPS; ++i) {

currentState = rungeKuttaStep(currentState, params.DT, params); // Передаем params явно

trajectoryStates.push\_back(currentState); // Добавляем полное состояние

double r\_squared = currentState.x \* currentState.x + currentState.y \* currentState.y;

if (r\_squared < params.CENTRAL\_BODY\_RADIUS \* params.CENTRAL\_BODY\_RADIUS) {

std::cout << "Столкновение обнаружено на шаге " << i + 1

<< " после вычисления. Координаты: (" << currentState.x << ", " << currentState.y

<< "), r = " << std::sqrt(r\_squared) << "\n";

break;

}

}

return trajectoryStates;

}

// Правая часть системы дифференциальных уравнений

State Calculations::derivatives(const State& s, const SimulationParameters& params) {

double r\_squared = s.x \* s.x + s.y \* s.y;

if (r\_squared == 0) {

return { s.vx, s.vy, 0, 0 };

}

double r = std::sqrt(r\_squared);

double r\_cubed = r\_squared \* r;

double common\_factor\_gravity = -params.G \* params.M / r\_cubed;

double net\_propulsion\_factor = params.THRUST\_COEFFICIENT - params.DRAG\_COEFFICIENT;

double ax = common\_factor\_gravity \* s.x + net\_propulsion\_factor \* s.vx;

double ay = common\_factor\_gravity \* s.y + net\_propulsion\_factor \* s.vy;

return { s.vx, s.vy, ax, ay };

}

// Один шаг интегрирования методом Рунге-Кутты 4-го порядка

State Calculations::rungeKuttaStep(const State& s, double dt, const SimulationParameters& params) {

State k1 = derivatives(s, params);

State s\_temp\_k2 = {

s.x + dt \* k1.x / 2.0,

s.y + dt \* k1.y / 2.0,

s.vx + dt \* k1.vx / 2.0,

s.vy + dt \* k1.vy / 2.0

};

State k2 = derivatives(s\_temp\_k2, params);

State s\_temp\_k3 = {

s.x + dt \* k2.x / 2.0,

s.y + dt \* k2.y / 2.0,

s.vx + dt \* k2.vx / 2.0,

s.vy + dt \* k2.vy / 2.0

};

State k3 = derivatives(s\_temp\_k3, params);

State s\_temp\_k4 = {

s.x + dt \* k3.x,

s.y + dt \* k3.y,

s.vx + dt \* k3.vx,

s.vy + dt \* k3.vy

};

State k4 = derivatives(s\_temp\_k4, params);

return {

s.x + dt / 6.0 \* (k1.x + 2.0 \* k2.x + 2.0 \* k3.x + k4.x),

s.y + dt / 6.0 \* (k1.y + 2.0 \* k2.y + 2.0 \* k3.y + k4.y),

s.vx + dt / 6.0 \* (k1.vx + 2.0 \* k2.vx + 2.0 \* k3.vx + k4.vx),

s.vy + dt / 6.0 \* (k1.vy + 2.0 \* k2.vy + 2.0 \* k3.vy + k4.vy)

};

}

***// Конец Calculations.cpp //***

# ***// Начало TrajectoryVisualizer.h //***

#pragma once

#ifndef TRAJECTORYVISUALIZER\_H

#define TRAJECTORYVISUALIZER\_H

#include <SFML/Graphics.hpp>

#include <vector>

#include <string>

#include <cmath> // Для std::sqrt, std::min, std::max

#include <iostream> // Для std::cerr, std::cout

#include <fstream> // Для std::ifstream, std::ofstream

#include <sstream> // Для std::istringstream, std::ostringstream

#include <iomanip> // Для std::fixed, std::setprecision

#include <algorithm> // Для std::min, std::max (дублирование, но не страшно)

// Используем тот же тип данных, что и в Calculations.h

// Если вынести в общий "types.h", то можно будет включать его.

using WorldTrajectoryPoint = std::pair<double, double>;

using WorldTrajectoryData = std::vector<WorldTrajectoryPoint>;

class TrajectoryVisualizer {

public:

TrajectoryVisualizer(unsigned int width, unsigned int height, const std::string& windowTitle = "Trajectory Visualizer");

void setData(const WorldTrajectoryData& data);

bool loadDataFromFile(const std::string& filename);

void run();

void resetViewAndAnimation();

bool saveTrajectoryToFile(const std::string& filename) const;

private:

// --- Константы визуализации ---

// Их можно сделать static constexpr членами класса или оставить как есть, если они не меняются

static constexpr float DEFAULT\_SCALE = 150.0f;

static constexpr unsigned int DEFAULT\_POINTS\_PER\_FRAME = 1u;

static constexpr unsigned int MIN\_POINTS\_PER\_FRAME = 1u;

static constexpr unsigned int MAX\_POINTS\_PER\_FRAME = 2048u;

static constexpr unsigned int ANIMATION\_SPEED\_MULTIPLIER = 2;

const std::string FONT\_FILENAME = "arial.ttf";

static constexpr unsigned int INFO\_TEXT\_CHAR\_SIZE = 16;

static constexpr float CENTER\_POINT\_RADIUS = 5.0f;

static constexpr float TRAJECTORY\_START\_POINT\_RADIUS = 2.0f;

static constexpr float ZOOM\_FACTOR\_STEP = 1.3f;

sf::RenderWindow m\_window;

WorldTrajectoryData m\_worldTrajectoryData;

std::vector<sf::Vertex> m\_screenTrajectory;

float m\_scale;

sf::Vector2f m\_offset;

sf::Vector2f m\_screenCenter;

size\_t m\_currentPointIndex;

unsigned int m\_pointsPerFrame;

bool m\_isPaused;

bool m\_showAllPointsImmediately;

sf::Font m\_font;

sf::Text m\_infoText;

bool m\_isDragging;

sf::Vector2i m\_lastMousePos;

// Приватные методы

sf::Vector2f toScreenCoords(double worldX, double worldY) const;

sf::Vector2f toWorldCoords(sf::Vector2f screenPos) const;

void recalculateScreenTrajectory();

void setupInfoText();

void updateInfoText();

void handleEvent(const sf::Event& event);

void handleKeyPress(const sf::Event::KeyEvent& keyEvent);

void updateAnimation();

void draw();

};

#endif // TRAJECTORYVISUALIZER\_H

***// Конец TrajectoryVisualizer.h //***

# ***// Начало TrajectoryVisualizer.cpp //***

#include "TrajectoryVisualizer.h"

// --- Инициализация статических констант (если они объявлены как static в .h) ---

// constexpr float TrajectoryVisualizer::DEFAULT\_SCALE; // И т.д. для всех static constexpr

// Но для литеральных типов static constexpr можно инициализировать прямо в .h (C++17+)

// Если компилятор старый, то так:

// const float TrajectoryVisualizer::DEFAULT\_SCALE = 150.0f;

// const unsigned int TrajectoryVisualizer::DEFAULT\_POINTS\_PER\_FRAME = 1u;

// ... и так далее для других ...

TrajectoryVisualizer::TrajectoryVisualizer(unsigned int width, unsigned int height, const std::string& windowTitle)

: m\_window(sf::VideoMode(width, height), windowTitle, sf::Style::Default), // Используем L"" для кириллицы в заголовке, если нужно

m\_scale(DEFAULT\_SCALE),

m\_offset(0.f, 0.f),

m\_screenCenter(static\_cast<float>(width) / 2.f, static\_cast<float>(height) / 2.f),

m\_currentPointIndex(0),

m\_pointsPerFrame(DEFAULT\_POINTS\_PER\_FRAME),

m\_isPaused(false),

m\_showAllPointsImmediately(false),

m\_isDragging(false) {

m\_window.setFramerateLimit(60);

setupInfoText();

}

void TrajectoryVisualizer::setData(const WorldTrajectoryData& data) {

m\_worldTrajectoryData = data;

resetViewAndAnimation();

// recalculateScreenTrajectory(); // Вызывается внутри resetViewAndAnimation

}

bool TrajectoryVisualizer::loadDataFromFile(const std::string& filename) {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "TrajectoryVisualizer: Ошибка: не удалось открыть файл траектории " << filename << "\n";

return false;

}

WorldTrajectoryData data;

std::string line;

while (std::getline(file, line)) {

std::istringstream iss(line);

double x, y;

if (iss >> x >> y) {

data.emplace\_back(x, y);

}

else {

std::cerr << "TrajectoryVisualizer: Предупреждение: неверный формат строки в файле: " << line << "\n";

}

}

file.close(); // Закрываем файл

if (data.empty()) {

std::cerr << "TrajectoryVisualizer: Ошибка: файл " << filename << " пуст или не содержит корректных данных.\n";

return false;

}

setData(data);

return true;

}

void TrajectoryVisualizer::run() {

if (m\_worldTrajectoryData.empty()) {

std::cerr << "TrajectoryVisualizer: Нет данных для визуализации. Загрузите данные.\n";

// Можно просто показать пустое окно с сообщением

bool dataNotLoaded = true;

while (m\_window.isOpen() && dataNotLoaded) {

sf::Event event{};

while (m\_window.pollEvent(event)) {

if (event.type == sf::Event::Closed) m\_window.close();

if (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Escape) m\_window.close();

}

updateInfoText(); // Обновит текст, который может содержать сообщение об ошибке

m\_window.clear(sf::Color::Black);

m\_window.draw(m\_infoText); // Показать инфо-текст (можно изменить его содержимое)

m\_window.display();

if (!m\_worldTrajectoryData.empty()) dataNotLoaded = false; // Если данные загрузились в другом потоке/способом

}

if (!m\_window.isOpen()) return; // Если окно было закрыто

}

while (m\_window.isOpen()) {

sf::Event event{};

while (m\_window.pollEvent(event)) {

handleEvent(event);

}

updateAnimation();

updateInfoText();

draw();

}

}

void TrajectoryVisualizer::resetViewAndAnimation() {

m\_scale = DEFAULT\_SCALE;

m\_offset = { 0.f, 0.f };

m\_isPaused = false;

m\_showAllPointsImmediately = false;

m\_pointsPerFrame = DEFAULT\_POINTS\_PER\_FRAME;

m\_currentPointIndex = m\_worldTrajectoryData.empty() ? 0 : 1;

recalculateScreenTrajectory();

}

sf::Vector2f TrajectoryVisualizer::toScreenCoords(double worldX, double worldY) const {

return {

m\_screenCenter.x + m\_offset.x + static\_cast<float>(worldX) \* m\_scale,

m\_screenCenter.y + m\_offset.y - static\_cast<float>(worldY) \* m\_scale

};

}

sf::Vector2f TrajectoryVisualizer::toWorldCoords(sf::Vector2f screenPos) const {

return {

(screenPos.x - m\_screenCenter.x - m\_offset.x) / m\_scale,

-(screenPos.y - m\_screenCenter.y - m\_offset.y) / m\_scale

};

}

void TrajectoryVisualizer::recalculateScreenTrajectory() {

m\_screenTrajectory.clear();

if (m\_worldTrajectoryData.empty()) return;

m\_screenTrajectory.reserve(m\_worldTrajectoryData.size());

for (const auto& world\_point : m\_worldTrajectoryData) {

m\_screenTrajectory.emplace\_back(toScreenCoords(world\_point.first, world\_point.second), sf::Color::White);

}

if (!m\_showAllPointsImmediately) {

m\_currentPointIndex = std::min(m\_currentPointIndex, m\_screenTrajectory.size());

if (m\_currentPointIndex == 0 && !m\_screenTrajectory.empty()) {

m\_currentPointIndex = 1;

}

}

else {

m\_currentPointIndex = m\_screenTrajectory.size();

}

}

void TrajectoryVisualizer::setupInfoText() {

if (!m\_font.loadFromFile(FONT\_FILENAME)) {

std::cerr << "TrajectoryVisualizer: Ошибка: не удалось загрузить шрифт " << FONT\_FILENAME << "\n";

}

m\_infoText.setFont(m\_font);

m\_infoText.setCharacterSize(INFO\_TEXT\_CHAR\_SIZE);

m\_infoText.setFillColor(sf::Color::Yellow);

m\_infoText.setPosition(10.f, 10.f);

}

void TrajectoryVisualizer::updateInfoText() {

std::ostringstream oss;

oss << std::fixed << std::setprecision(2);

oss << "Scale: " << m\_scale << "\n";

oss << "Offset: (" << m\_offset.x << ", " << m\_offset.y << ")\n";

oss << "Points drawn: " << m\_currentPointIndex << "/" << m\_worldTrajectoryData.size() << "\n";

oss << "Animation: " << (m\_isPaused ? "Paused" : "Running")

<< " (" << m\_pointsPerFrame << " pts/frame)\n";

oss << "Controls:\n";

oss << " Mouse Wheel: Zoom\n";

oss << " Right Mouse Drag: Pan\n";

oss << " P: Pause/Resume animation\n";

oss << " F: Toggle full trajectory\n";

oss << " +/-: Change animation speed\n";

oss << " R: Reset view & animation\n";

oss << " Esc: Exit";

m\_infoText.setString(oss.str()); // Для sf::Text лучше использовать sf::String или L"" если есть кириллица

// но здесь только ASCII, так что oss.str() должен работать.

// Для надежности можно: m\_infoText.setString(sf::String::fromUtf8(oss.str().c\_str()));

}

void TrajectoryVisualizer::handleEvent(const sf::Event& event) {

switch (event.type) {

case sf::Event::Closed:

m\_window.close();

break;

case sf::Event::Resized:

{

sf::FloatRect visibleArea(0, 0, static\_cast<float>(event.size.width), static\_cast<float>(event.size.height));

m\_window.setView(sf::View(visibleArea));

m\_screenCenter = { event.size.width / 2.f, event.size.height / 2.f };

recalculateScreenTrajectory();

}

break;

case sf::Event::KeyPressed:

handleKeyPress(event.key);

break;

case sf::Event::MouseWheelScrolled:

if (event.mouseWheelScroll.wheel == sf::Mouse::VerticalWheel && event.mouseWheelScroll.delta != 0) { // Проверяем тип колеса

sf::Vector2f worldPosBeforeZoom = toWorldCoords(static\_cast<sf::Vector2f>(sf::Mouse::getPosition(m\_window)));

float zoomFactor = (event.mouseWheelScroll.delta > 0) ? ZOOM\_FACTOR\_STEP : 1.0f / ZOOM\_FACTOR\_STEP;

m\_scale \*= zoomFactor;

sf::Vector2f worldPosAfterZoom = toWorldCoords(static\_cast<sf::Vector2f>(sf::Mouse::getPosition(m\_window)));

m\_offset.x += (worldPosAfterZoom.x - worldPosBeforeZoom.x) \* m\_scale;

m\_offset.y += (worldPosAfterZoom.y - worldPosBeforeZoom.y) \* m\_scale;

recalculateScreenTrajectory();

}

break;

case sf::Event::MouseButtonPressed:

if (event.mouseButton.button == sf::Mouse::Right) {

m\_isDragging = true;

m\_lastMousePos = sf::Mouse::getPosition(m\_window);

}

break;

case sf::Event::MouseButtonReleased:

if (event.mouseButton.button == sf::Mouse::Right) {

m\_isDragging = false;

}

break;

case sf::Event::MouseMoved:

if (m\_isDragging) {

sf::Vector2i newMousePos = sf::Mouse::getPosition(m\_window);

sf::Vector2f delta = static\_cast<sf::Vector2f>(newMousePos - m\_lastMousePos);

m\_offset += delta;

m\_lastMousePos = newMousePos;

recalculateScreenTrajectory();

}

break;

default:

break;

}

}

void TrajectoryVisualizer::handleKeyPress(const sf::Event::KeyEvent& keyEvent) {

if (keyEvent.code == sf::Keyboard::Escape) m\_window.close();

if (keyEvent.code == sf::Keyboard::P) m\_isPaused = !m\_isPaused;

if (keyEvent.code == sf::Keyboard::F) {

m\_showAllPointsImmediately = !m\_showAllPointsImmediately;

if (m\_showAllPointsImmediately) {

m\_currentPointIndex = m\_screenTrajectory.size();

}

else {

m\_currentPointIndex = m\_screenTrajectory.empty() ? 0 : 1;

}

}

if (keyEvent.code == sf::Keyboard::Add || keyEvent.code == sf::Keyboard::Equal) { // Equal это + на основной клавиатуре

m\_pointsPerFrame = std::min(m\_pointsPerFrame \* ANIMATION\_SPEED\_MULTIPLIER, MAX\_POINTS\_PER\_FRAME);

}

if (keyEvent.code == sf::Keyboard::Subtract || keyEvent.code == sf::Keyboard::Hyphen) { // Hyphen это - на основной клавиатуре

m\_pointsPerFrame = std::max(m\_pointsPerFrame / ANIMATION\_SPEED\_MULTIPLIER, MIN\_POINTS\_PER\_FRAME);

}

if (keyEvent.code == sf::Keyboard::R) resetViewAndAnimation();

}

void TrajectoryVisualizer::updateAnimation() {

if (!m\_isPaused && !m\_showAllPointsImmediately && m\_currentPointIndex < m\_screenTrajectory.size()) {

m\_currentPointIndex = std::min(m\_screenTrajectory.size(), m\_currentPointIndex + m\_pointsPerFrame);

}

}

void TrajectoryVisualizer::draw() {

m\_window.clear(sf::Color::Black);

sf::CircleShape centerMassShape(CENTER\_POINT\_RADIUS);

centerMassShape.setFillColor(sf::Color::Red);

centerMassShape.setOrigin(CENTER\_POINT\_RADIUS, CENTER\_POINT\_RADIUS);

centerMassShape.setPosition(toScreenCoords(0, 0));

m\_window.draw(centerMassShape);

if (!m\_screenTrajectory.empty()) {

size\_t pointsToDraw = std::min(m\_currentPointIndex, m\_screenTrajectory.size());

if (pointsToDraw >= 2) {

m\_window.draw(&m\_screenTrajectory[0], pointsToDraw, sf::LineStrip);

}

else if (pointsToDraw == 1) {

sf::CircleShape firstPointShape(TRAJECTORY\_START\_POINT\_RADIUS);

firstPointShape.setFillColor(sf::Color::White);

firstPointShape.setOrigin(TRAJECTORY\_START\_POINT\_RADIUS, TRAJECTORY\_START\_POINT\_RADIUS);

firstPointShape.setPosition(m\_screenTrajectory[0].position);

m\_window.draw(firstPointShape);

}

}

m\_window.draw(m\_infoText);

m\_window.display();

}

bool TrajectoryVisualizer::saveTrajectoryToFile(const std::string& filename) const {

if (m\_worldTrajectoryData.empty()) {

std::cerr << "TrajectoryVisualizer: Нет данных траектории для сохранения в файл '" << filename << "'.\n";

return false; // Возвращаем false, если данных нет

}

std::ofstream outputFile(filename);

if (!outputFile.is\_open()) {

std::cerr << "TrajectoryVisualizer: Ошибка: не удалось открыть файл '" << filename << "' для записи.\n";

return false;

}

// Устанавливаем форматирование для вывода чисел с плавающей точкой

// для лучшей читаемости и точности в файле

outputFile << std::fixed << std::setprecision(10); // 10 знаков после запятой

for (const auto& point : m\_worldTrajectoryData) {

// m\_worldTrajectoryData это std::vector<std::pair<double, double>>

// point.first это x, point.second это y

outputFile << point.first << " " << point.second << "\n";

}

outputFile.close(); // Закрываем файл

if (outputFile.fail()) { // Проверяем, не возникло ли ошибок при записи или закрытии

std::cerr << "TrajectoryVisualizer: Ошибка при записи или закрытии файла '" << filename << "'.\n";

return false;

}

std::cout << "TrajectoryVisualizer: Траектория (" << m\_worldTrajectoryData.size()

<< " точек) успешно сохранена в файл '" << filename << "'.\n";

return true;

}

***// Конец TrajectoryVisualizer.cpp //***

// ========================================================================//

# ***// Начало UserInterface.h //***

#pragma once

#ifndef USERINTERFACE\_H

#define USERINTERFACE\_H

#include <SFML/Graphics.hpp>

#include <TGUI/TGUI.hpp>

#include "Calculations.h" // Включаем Calculations.h для доступа к State

#include <vector>

#include <string>

#include <iomanip>

#include <sstream>

struct TableRowData {

float h\_sec;

float x, y;

float Vx, Vy;

};

class UserInterface {

public:

UserInterface();

void run();

private:

static constexpr float INPUT\_FIELD\_WIDTH = 180.f;

static constexpr float INPUT\_ROW\_HEIGHT = 30.f;

static constexpr float PANEL\_PADDING = 10.f;

static constexpr float WIDGET\_SPACING = 10.f;

static constexpr float HEADER\_HEIGHT = 30.f;

static constexpr float TITLE\_HEIGHT = 30.f; // Увеличил для лучшей читаемости заголовков

static constexpr float SCROLLBAR\_WIDTH\_ESTIMATE = 18.f;

void initializeGui();

void loadWidgets();

void loadLeftPanelWidgets();

void loadRightPanelWidgets();

void loadTrajectoryWidgets(tgui::Panel::Ptr parentPanel);

void loadTableWidgets(tgui::Panel::Ptr parentPanel);

void setupLayout();

void connectSignals();

void handleEvents();

void update();

void render();

void onCalculateButtonPressed();

void populateTable(const std::vector<TableRowData>& data);

void drawTrajectoryOnCanvas(sf::RenderTarget& target\_rt); // Изменено имя аргумента

void prepareTrajectoryForDisplay();

sf::RenderWindow m\_window;

tgui::Gui m\_gui;

tgui::Label::Ptr m\_inputTitleLabel;

tgui::EditBox::Ptr m\_edit\_m;

tgui::EditBox::Ptr m\_edit\_M;

tgui::EditBox::Ptr m\_edit\_V0;

tgui::EditBox::Ptr m\_edit\_T;

tgui::EditBox::Ptr m\_edit\_k;

tgui::EditBox::Ptr m\_edit\_F;

tgui::Button::Ptr m\_calculateButton;

tgui::Grid::Ptr m\_inputControlsGrid;

tgui::Panel::Ptr m\_leftPanel;

tgui::Panel::Ptr m\_rightPanel;

tgui::Panel::Ptr m\_trajectoryContainerPanel;

tgui::Panel::Ptr m\_tableContainerPanel;

tgui::Label::Ptr m\_trajectoryTitleLabel;

tgui::Canvas::Ptr m\_trajectoryCanvas;

sf::Font m\_sfmlFont;

bool m\_trajectoryAvailable;

tgui::Label::Ptr m\_tableTitleLabel;

tgui::Grid::Ptr m\_tableHeaderGrid;

tgui::ScrollablePanel::Ptr m\_tableDataPanel;

tgui::Grid::Ptr m\_tableDataGrid;

std::vector<TableRowData> m\_currentTableData;

std::vector<State> m\_calculatedStates;

std::vector<sf::Vertex> m\_trajectoryDisplayPoints; // Остается для отрисовки

};

#endif // USERINTERFACE\_H

***// Конец UserInterface.h //***

# ***// Начало UserInterface.cpp //***

#include "UserInterface.h"

#include <iostream> // Для отладки

#include <algorithm> // Для std::min\_element, std::max\_element

#if defined(\_MSC\_VER)

#pragma execution\_character\_set("utf-8")

#endif

// --- Вспомогательная функция для создания строки ввода ---

std::pair<tgui::Label::Ptr, tgui::EditBox::Ptr> createInputRowControls(const sf::String& labelText, float editBoxWidth, float rowHeight) {

auto label = tgui::Label::create(tgui::String(labelText)); // sf::String с L"" хорошо работает с tgui::Label

if (label) {

label->getRenderer()->setTextColor(tgui::Color::Black);

label->setVerticalAlignment(tgui::Label::VerticalAlignment::Center);

}

auto editBox = tgui::EditBox::create();

if (editBox) {

editBox->setSize({ editBoxWidth, rowHeight });

}

return { label, editBox };

}

// --- Конструктор и инициализация ---

UserInterface::UserInterface()

: m\_window({ 1200, 800 }, L"Расчет траектории движения тела"),

m\_gui(m\_window),

m\_trajectoryAvailable(false) {

m\_gui.setFont("arial.ttf");

// Загрузка шрифта для SFML (используется на Canvas)

if (!m\_sfmlFont.loadFromFile("arial.ttf")) {

std::cerr << "SFML: Error - Failed to load font 'arial.ttf' for SFML rendering!\n";

}

initializeGui();

}

void UserInterface::initializeGui() {

std::cout << "DEBUG: Initializing GUI..." << std::endl;

loadWidgets();

setupLayout(); // Вызываем setupLayout после loadWidgets

connectSignals();

populateTable({}); // Начальное пустое состояние таблицы

std::cout << "DEBUG: GUI Initialized." << std::endl;

}

// --- Загрузка виджетов ---

void UserInterface::loadWidgets() {

std::cout << "DEBUG: Loading all widgets..." << std::endl;

loadLeftPanelWidgets();

loadRightPanelWidgets();

std::cout << "DEBUG: All widgets loaded." << std::endl;

}

void UserInterface::loadLeftPanelWidgets() {

m\_leftPanel = tgui::Panel::create();

if (!m\_leftPanel) { std::cerr << "Error: Failed to create m\_leftPanel" << std::endl; return; }

m\_leftPanel->getRenderer()->setBackgroundColor(tgui::Color(220, 220, 220));

m\_leftPanel->getRenderer()->setBorders({ 1, 1, 1, 1 });

m\_leftPanel->getRenderer()->setBorderColor(tgui::Color::Black);

m\_gui.add(m\_leftPanel);

// 1. Заголовок "Исходные значения"

m\_inputTitleLabel = tgui::Label::create(L"Исходные значения");

if (!m\_inputTitleLabel) { std::cerr << "Error: Failed to create m\_inputTitleLabel" << std::endl; return; }

m\_inputTitleLabel->getRenderer()->setTextStyle(tgui::TextStyle::Bold);

m\_inputTitleLabel->setHorizontalAlignment(tgui::Label::HorizontalAlignment::Center);

m\_inputTitleLabel->getRenderer()->setTextColor(tgui::Color::Black);

m\_inputTitleLabel->setSize({ "100% - " + tgui::String::fromNumber(2 \* PANEL\_PADDING), TITLE\_HEIGHT });

m\_inputTitleLabel->setPosition({ PANEL\_PADDING, PANEL\_PADDING });

m\_leftPanel->add(m\_inputTitleLabel);

// 2. Grid для полей ввода

m\_inputControlsGrid = tgui::Grid::create(); // Используем член класса

if (!m\_inputControlsGrid) { std::cerr << "Error: Failed to create m\_inputControlsGrid" << std::endl; return; }

m\_inputControlsGrid->setPosition({ PANEL\_PADDING, tgui::bindBottom(m\_inputTitleLabel) + WIDGET\_SPACING });

m\_leftPanel->add(m\_inputControlsGrid);

unsigned int currentRow = 0;

// Лямбда для добавления строки в inputControlsGrid

auto addInputRowToGrid = [&](const sf::String& text, tgui::EditBox::Ptr& editBoxMember) {

auto pair = createInputRowControls(text, INPUT\_FIELD\_WIDTH, INPUT\_ROW\_HEIGHT);

if (!pair.first || !pair.second) {

std::cerr << "Error: Failed to create input pair for: " << text.toAnsiString() << std::endl;

return;

}

editBoxMember = pair.second;

m\_inputControlsGrid->addWidget(pair.first, currentRow, 0);

m\_inputControlsGrid->addWidget(editBoxMember, currentRow, 1);

m\_inputControlsGrid->setWidgetPadding(currentRow, 0, { 5, 5, 5, 0 }); // Label: T,R,B,L (0 слева, т.к. грид имеет свой отступ)

m\_inputControlsGrid->setWidgetPadding(currentRow, 1, { 5, 0, 5, 5 }); // EditBox: T,R,B,L (0 справа)

currentRow++;

};

addInputRowToGrid(L"m (масса, кг):", m\_edit\_m);

addInputRowToGrid(L"M (масса, кг):", m\_edit\_M);

addInputRowToGrid(L"V0 (скорость, м/с):", m\_edit\_V0);

addInputRowToGrid(L"T (время, сек):", m\_edit\_T);

addInputRowToGrid(L"k (безразмерная):", m\_edit\_k);

addInputRowToGrid(L"F (безразмерная):", m\_edit\_F);

// 3. Кнопка "Рассчитать траекторию!"

m\_calculateButton = tgui::Button::create(L"Рассчитать траекторию!");

if (!m\_calculateButton) { std::cerr << "Error: Failed to create m\_calculateButton" << std::endl; return; }

m\_calculateButton->getRenderer()->setRoundedBorderRadius(15);

m\_calculateButton->setSize({ "100% - " + tgui::String::fromNumber(2 \* PANEL\_PADDING), 40 });

m\_calculateButton->setPosition({ PANEL\_PADDING, tgui::bindBottom(m\_inputControlsGrid) + WIDGET\_SPACING \* 2 }); // Больший отступ для кнопки

m\_leftPanel->add(m\_calculateButton);

}

void UserInterface::loadRightPanelWidgets() {

m\_rightPanel = tgui::Panel::create();

if (!m\_rightPanel) { std::cerr << "Error: Failed to create m\_rightPanel" << std::endl; return; }

m\_gui.add(m\_rightPanel); // Сначала добавляем, потом настраиваем содержимое

loadTrajectoryWidgets(m\_rightPanel);

loadTableWidgets(m\_rightPanel);

}

void UserInterface::loadTrajectoryWidgets(tgui::Panel::Ptr parentPanel) {

m\_trajectoryContainerPanel = tgui::Panel::create();

if (!m\_trajectoryContainerPanel) { std::cerr << "Error: Failed to create m\_trajectoryContainerPanel" << std::endl; return; }

m\_trajectoryContainerPanel->getRenderer()->setBorders({ 1,1,1,1 });

m\_trajectoryContainerPanel->getRenderer()->setBorderColor(tgui::Color::Black);

m\_trajectoryContainerPanel->getRenderer()->setBackgroundColor(tgui::Color::White);

parentPanel->add(m\_trajectoryContainerPanel);

m\_trajectoryTitleLabel = tgui::Label::create(L"Траектория движения тела");

if (!m\_trajectoryTitleLabel) { std::cerr << "Error: Failed to create m\_trajectoryTitleLabel" << std::endl; return; }

m\_trajectoryTitleLabel->getRenderer()->setTextStyle(tgui::TextStyle::Bold);

m\_trajectoryTitleLabel->setHorizontalAlignment(tgui::Label::HorizontalAlignment::Center);

m\_trajectoryTitleLabel->getRenderer()->setTextColor(tgui::Color::Black);

m\_trajectoryTitleLabel->setSize({ "100%", TITLE\_HEIGHT });

m\_trajectoryContainerPanel->add(m\_trajectoryTitleLabel, "TrajectoryTitle"); // Используем имя для позиционирования канваса

m\_trajectoryCanvas = tgui::Canvas::create();

if (!m\_trajectoryCanvas) { std::cerr << "Error: Failed to create m\_trajectoryCanvas" << std::endl; return; }

m\_trajectoryCanvas->setSize({ "100%", "100% - " + tgui::String::fromNumber(TITLE\_HEIGHT) });

m\_trajectoryCanvas->setPosition({ 0, "TrajectoryTitle.bottom" });

m\_trajectoryContainerPanel->add(m\_trajectoryCanvas);

}

void UserInterface::loadTableWidgets(tgui::Panel::Ptr parentPanel) {

m\_tableContainerPanel = tgui::Panel::create();

if (!m\_tableContainerPanel) { std::cerr << "Error: Failed to create m\_tableContainerPanel" << std::endl; return; }

m\_tableContainerPanel->getRenderer()->setBorders({ 1,1,1,1 });

m\_tableContainerPanel->getRenderer()->setBorderColor(tgui::Color::Black);

m\_tableContainerPanel->getRenderer()->setBackgroundColor(tgui::Color::White);

parentPanel->add(m\_tableContainerPanel);

m\_tableTitleLabel = tgui::Label::create(L"Таблица координат и скоростей");

if (!m\_tableTitleLabel) { std::cerr << "Error: Failed to create m\_tableTitleLabel" << std::endl; return; }

m\_tableTitleLabel->getRenderer()->setTextStyle(tgui::TextStyle::Bold);

m\_tableTitleLabel->setHorizontalAlignment(tgui::Label::HorizontalAlignment::Center);

m\_tableTitleLabel->getRenderer()->setTextColor(tgui::Color::Black);

m\_tableTitleLabel->setSize({ "100%", TITLE\_HEIGHT });

m\_tableContainerPanel->add(m\_tableTitleLabel, "TableTitle");

m\_tableHeaderGrid = tgui::Grid::create();

if (!m\_tableHeaderGrid) { std::cerr << "Error: Failed to create m\_tableHeaderGrid" << std::endl; return; }

m\_tableHeaderGrid->setSize({ "100% - " + tgui::String::fromNumber(SCROLLBAR\_WIDTH\_ESTIMATE), HEADER\_HEIGHT });

m\_tableHeaderGrid->setPosition({ 0, "TableTitle.bottom" });

std::vector<sf::String> headers = { L"h, сек", L"x", L"y", L"Vx", L"Vy" };

for (size\_t i = 0; i < headers.size(); ++i) {

auto headerLabel = tgui::Label::create(tgui::String(headers[i]));

if (!headerLabel) { std::cerr << "Error: Failed to create headerLabel " << i << std::endl; continue; }

headerLabel->getRenderer()->setTextColor(tgui::Color::Black);

headerLabel->getRenderer()->setBorders({ 0,0,0,1 }); // Только нижняя граница

headerLabel->getRenderer()->setBorderColor(tgui::Color::Black);

headerLabel->setHorizontalAlignment(tgui::Label::HorizontalAlignment::Center);

headerLabel->setVerticalAlignment(tgui::Label::VerticalAlignment::Center);

m\_tableHeaderGrid->addWidget(headerLabel, 0, i);

// Можно добавить отступы для headerLabel, если нужно

// m\_tableHeaderGrid->setWidgetPadding(0, i, {2,5,2,5}); // T,R,B,L

}

m\_tableContainerPanel->add(m\_tableHeaderGrid);

m\_tableDataPanel = tgui::ScrollablePanel::create();

if (!m\_tableDataPanel) { std::cerr << "Error: Failed to create m\_tableDataPanel" << std::endl; return; }

m\_tableDataPanel->setSize({ "100%", "100% - " + tgui::String::fromNumber(TITLE\_HEIGHT + HEADER\_HEIGHT) });

m\_tableDataPanel->setPosition({ 0, tgui::bindBottom(m\_tableHeaderGrid) });

m\_tableDataPanel->getRenderer()->setBackgroundColor(tgui::Color(245, 245, 245));

m\_tableContainerPanel->add(m\_tableDataPanel);

m\_tableDataGrid = tgui::Grid::create();

if (!m\_tableDataGrid) { std::cerr << "Error: Failed to create m\_tableDataGrid for data" << std::endl; return; }

// Ширина и высота m\_tableDataGrid будут управляться ScrollablePanel и его содержимым

m\_tableDataPanel->add(m\_tableDataGrid);

}

// --- Компоновка ---

void UserInterface::setupLayout() {

std::cout << "DEBUG: Setting up layout..." << std::endl;

// Левая панель

m\_leftPanel->setSize({ "30%", "100%" }); // Немного шире для комфорта

m\_leftPanel->setPosition({ 0, 0 });

// Правая панель

m\_rightPanel->setSize({ "70%", "100%" });

m\_rightPanel->setPosition({ "30%", 0 });

// Контейнеры внутри правой панели

const float rightPanelPadding = PANEL\_PADDING;

const float verticalSpacing = WIDGET\_SPACING / 2.f;

m\_trajectoryContainerPanel->setSize(

{ tgui::bindWidth(m\_rightPanel) - 2 \* rightPanelPadding,

"60% - " + tgui::String::fromNumber(rightPanelPadding + verticalSpacing / 2.f)

}

);

m\_trajectoryContainerPanel->setPosition({ rightPanelPadding, rightPanelPadding });

m\_tableContainerPanel->setSize(

{ tgui::bindWidth(m\_rightPanel) - 2 \* rightPanelPadding,

"40% - " + tgui::String::fromNumber(rightPanelPadding + verticalSpacing / 2.f)

}

);

m\_tableContainerPanel->setPosition({ rightPanelPadding, tgui::bindBottom(m\_trajectoryContainerPanel) + verticalSpacing });

std::cout << "DEBUG: Layout setup finished." << std::endl;

}

// --- Подключение сигналов ---

void UserInterface::connectSignals() {

if (m\_calculateButton) {

// Используем .connect() для TGUI 0.9.x

m\_calculateButton->onPress.connect(&UserInterface::onCalculateButtonPressed, this);

}

else {

std::cerr << "Error: m\_calculateButton is null in connectSignals! Cannot connect." << std::endl;

}

}

// --- Обработчики и логика ---

void UserInterface::onCalculateButtonPressed() {

std::cout << "Calculate button pressed!" << std::endl;

SimulationParameters paramsFromUI;

try {

if (m\_edit\_M && !m\_edit\_M->getText().empty()) paramsFromUI.M = std::stod(m\_edit\_M->getText().toStdString());

if (m\_edit\_V0 && !m\_edit\_V0->getText().empty()) {

double v0\_val = std::stod(m\_edit\_V0->getText().toStdString());

paramsFromUI.initialState.vy = v0\_val;

paramsFromUI.initialState.vx = 0.0;

}

if (m\_edit\_T && !m\_edit\_T->getText().empty()) {

double total\_time = std::stod(m\_edit\_T->getText().toStdString());

if (paramsFromUI.DT > 0.000001) { // Защита от деления на очень малое число или ноль

paramsFromUI.STEPS = static\_cast<int>(total\_time / paramsFromUI.DT);

if (paramsFromUI.STEPS <= 0) paramsFromUI.STEPS = 1;

}

else {

paramsFromUI.STEPS = 1000; // Значение по умолчанию, если DT некорректен

std::cerr << "Warning: Invalid DT, using default STEPS." << std::endl;

}

}

if (m\_edit\_k && !m\_edit\_k->getText().empty()) paramsFromUI.DRAG\_COEFFICIENT = std::stod(m\_edit\_k->getText().toStdString());

if (m\_edit\_F && !m\_edit\_F->getText().empty()) paramsFromUI.THRUST\_COEFFICIENT = std::stod(m\_edit\_F->getText().toStdString());

}

catch (const std::exception& e) {

std::cerr << "Error parsing input values: " << e.what() << std::endl;

if (m\_inputTitleLabel) m\_inputTitleLabel->setText(L"Ошибка ввода параметров!");

m\_trajectoryAvailable = false; m\_calculatedStates.clear();

prepareTrajectoryForDisplay(); populateTable({});

return;

}

if (m\_inputTitleLabel) m\_inputTitleLabel->setText(L"Исходные значения");

Calculations calculator;

std::cout << "DEBUG: Running simulation with STEPS=" << paramsFromUI.STEPS

<< ", DT=" << paramsFromUI.DT << std::endl;

m\_calculatedStates = calculator.runSimulation(paramsFromUI);

m\_currentTableData.clear();

if (!m\_calculatedStates.empty()) {

m\_trajectoryAvailable = true;

double currentTime = 0.0;

const size\_t maxTableEntries = 2000;

size\_t step = 1;

if (m\_calculatedStates.size() > maxTableEntries) {

step = m\_calculatedStates.size() / maxTableEntries;

if (step == 0) step = 1; // На случай, если calculatedStates.size() < maxTableEntries но не 0

}

for (size\_t i = 0; i < m\_calculatedStates.size(); i += step) {

const auto& state = m\_calculatedStates[i];

m\_currentTableData.push\_back({

static\_cast<float>(i \* paramsFromUI.DT), // Более точное время

static\_cast<float>(state.x), static\_cast<float>(state.y),

static\_cast<float>(state.vx), static\_cast<float>(state.vy)

});

}

}

else {

m\_trajectoryAvailable = false;

}

prepareTrajectoryForDisplay(); // Подготовка вершин и настройка View для канваса

populateTable(m\_currentTableData);

}

void UserInterface::prepareTrajectoryForDisplay() {

m\_trajectoryDisplayPoints.clear();

if (!m\_trajectoryAvailable || m\_calculatedStates.empty()) {

std::cout << "DEBUG: No trajectory to prepare for display." << std::endl;

return;

}

m\_trajectoryDisplayPoints.reserve(m\_calculatedStates.size());

for (const auto& state : m\_calculatedStates) {

m\_trajectoryDisplayPoints.emplace\_back(

sf::Vector2f(static\_cast<float>(state.x), static\_cast<float>(-state.y)), // Y инвертируется для отображения

sf::Color::Blue // Цвет линии траектории

);

}

std::cout << "DEBUG: Trajectory display points prepared. Count: " << m\_trajectoryDisplayPoints.size() << std::endl;

}

void UserInterface::drawTrajectoryOnCanvas(sf::RenderTarget& canvasRenderTarget) {

sf::View originalView = canvasRenderTarget.getView();

sf::View fittedView; // View, который будет подогнан под канвас

if (m\_trajectoryAvailable && !m\_trajectoryDisplayPoints.empty()) {

float min\_x\_world = m\_trajectoryDisplayPoints[0].position.x;

float max\_x\_world = m\_trajectoryDisplayPoints[0].position.x;

float min\_y\_world = m\_trajectoryDisplayPoints[0].position.y;

float max\_y\_world = m\_trajectoryDisplayPoints[0].position.y;

for (const auto& vertex : m\_trajectoryDisplayPoints) {

min\_x\_world = std::min(min\_x\_world, vertex.position.x);

max\_x\_world = std::max(max\_x\_world, vertex.position.x);

min\_y\_world = std::min(min\_y\_world, vertex.position.y);

max\_y\_world = std::max(max\_y\_world, vertex.position.y);

}

min\_x\_world = std::min(min\_x\_world, 0.0f); max\_x\_world = std::max(max\_x\_world, 0.0f);

min\_y\_world = std::min(min\_y\_world, 0.0f); max\_y\_world = std::max(max\_y\_world, 0.0f);

float content\_w = max\_x\_world - min\_x\_world;

float content\_h = max\_y\_world - min\_y\_world;

// Если контент это точка или линия, дадим ему небольшой размер для View

if (content\_w < 0.001f) content\_w = 1.0f;

if (content\_h < 0.001f) content\_h = 1.0f;

float paddingFactor = 0.1f; // 10% отступ

float view\_w = content\_w \* (1.0f + 2.0f \* paddingFactor);

float view\_h = content\_h \* (1.0f + 2.0f \* paddingFactor);

sf::Vector2f view\_center(min\_x\_world + content\_w / 2.0f, min\_y\_world + content\_h / 2.0f);

// Получаем размер RenderTarget канваса

sf::Vector2u canvasSize = canvasRenderTarget.getSize();

if (canvasSize.x == 0 || canvasSize.y == 0) { // Защита от деления на ноль

canvasRenderTarget.setView(originalView);

return;

}

float canvasAspectRatio = static\_cast<float>(canvasSize.x) / canvasSize.y;

float contentAspectRatio = view\_w / view\_h;

if (canvasAspectRatio > contentAspectRatio) { // Канвас шире, чем контент -> подгоняем по высоте контента

view\_w = view\_h \* canvasAspectRatio;

}

else { // Канвас выше (или такой же), чем контент -> подгоняем по ширине контента

view\_h = view\_w / canvasAspectRatio;

}

fittedView.setSize(view\_w, view\_h);

fittedView.setCenter(view\_center);

canvasRenderTarget.setView(fittedView);

// --- Отрисовка ---

float centralBodyDisplayRadius = std::min(view\_w, view\_h) \* 0.01f;

if (centralBodyDisplayRadius < 0.001f) centralBodyDisplayRadius = 0.01f;

sf::CircleShape centerBody(centralBodyDisplayRadius);

centerBody.setFillColor(sf::Color::Red);

centerBody.setOrigin(centralBodyDisplayRadius, centralBodyDisplayRadius);

centerBody.setPosition(0.f, 0.f);

canvasRenderTarget.draw(centerBody);

if (m\_trajectoryDisplayPoints.size() >= 1) {

canvasRenderTarget.draw(m\_trajectoryDisplayPoints.data(), m\_trajectoryDisplayPoints.size(), sf::LineStrip);

}

}

else {

canvasRenderTarget.setView(canvasRenderTarget.getDefaultView());

sf::Text placeholderText;

if (m\_sfmlFont.hasGlyph(L'Т')) {

placeholderText.setFont(m\_sfmlFont);

placeholderText.setString(L"Траектория не рассчитана.\nНажмите 'Рассчитать траекторию!'");

}

else {

placeholderText.setString("Trajectory not calculated.\nPress 'Calculate Trajectory!'");

}

placeholderText.setCharacterSize(16);

placeholderText.setFillColor(sf::Color(105, 105, 105));

sf::FloatRect textRect = placeholderText.getLocalBounds();

placeholderText.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top + textRect.height / 2.0f);

placeholderText.setPosition(static\_cast<float>(canvasRenderTarget.getSize().x) / 2.0f,

static\_cast<float>(canvasRenderTarget.getSize().y) / 2.0f);

canvasRenderTarget.draw(placeholderText);

}

canvasRenderTarget.setView(originalView);

}

void UserInterface::populateTable(const std::vector<TableRowData>& data) {

if (!m\_tableDataGrid) { std::cerr << "Error: m\_tableDataGrid is null in populateTable!" << std::endl; return; }

m\_tableDataGrid->removeAllWidgets();

if (data.empty()) {

auto emptyLabel = tgui::Label::create(L"Нет данных для отображения");

if (emptyLabel) {

emptyLabel->setHorizontalAlignment(tgui::Label::HorizontalAlignment::Center);

m\_tableDataGrid->addWidget(emptyLabel, 0, 0);

// Чтобы emptyLabel занимал все 5 колонок, если API Grid не позволяет colspan:

// for (unsigned int j = 1; j < 5; ++j) {

// m\_tableDataGrid->addWidget(tgui::Label::create(""), 0, j); // Пустые метки

// }

}

if (m\_tableDataPanel) m\_tableDataPanel->setContentSize({ 0,0 });

return;

}

for (size\_t i = 0; i < data.size(); i += (data.size() / 100)) {

const auto& rowData = data[i];

std::stringstream ss\_h, ss\_x, ss\_y, ss\_vx, ss\_vy;

ss\_h << std::fixed << std::setprecision(2) << rowData.h\_sec;

ss\_x << std::fixed << std::setprecision(2) << rowData.x;

ss\_y << std::fixed << std::setprecision(2) << rowData.y;

ss\_vx << std::fixed << std::setprecision(2) << rowData.Vx;

ss\_vy << std::fixed << std::setprecision(2) << rowData.Vy;

std::vector<tgui::String> rowStrings = {

tgui::String(ss\_h.str()), tgui::String(ss\_x.str()), tgui::String(ss\_y.str()),

tgui::String(ss\_vx.str()), tgui::String(ss\_vy.str()) };

for (size\_t j = 0; j < rowStrings.size(); ++j) {

auto cellLabel = tgui::Label::create(rowStrings[j]);

if (cellLabel) {

cellLabel->getRenderer()->setTextColor(tgui::Color::Black);

cellLabel->setHorizontalAlignment(tgui::Label::HorizontalAlignment::Center);

m\_tableDataGrid->addWidget(cellLabel, i, j);

m\_tableDataGrid->setWidgetPadding(i, j, { 2, 5, 2, 5 });

}

}

}

if (m\_tableDataPanel && m\_tableDataGrid) {

m\_tableDataPanel->setContentSize(m\_tableDataGrid->getSize());

}

}

// --- Главный цикл и обработка событий ---

void UserInterface::run() {

m\_window.setFramerateLimit(60); // Ограничение FPS для плавности и снижения нагрузки

while (m\_window.isOpen()) {

handleEvents();

update();

render();

}

}

void UserInterface::handleEvents() {

sf::Event event;

while (m\_window.pollEvent(event)) {

m\_gui.handleEvent(event);

if (event.type == sf::Event::Closed) {

m\_window.close();

}

}

}

void UserInterface::update() {

// Например, анимация или другие обновления состояния, не связанные с вводом пользователя

}

void UserInterface::render() {

if (m\_trajectoryCanvas) {

sf::RenderTexture& canvasRT = m\_trajectoryCanvas->getRenderTexture();

canvasRT.clear(sf::Color(250, 250, 250)); // Фон канваса

drawTrajectoryOnCanvas(canvasRT); // Этот метод теперь сам устанавливает и сбрасывает View

m\_trajectoryCanvas->display();

}

m\_window.clear(sf::Color(220, 220, 220));

m\_gui.draw();

m\_window.display();

}

***// Конец UserInterface.cpp //***

# ***// Начало main.cpp //***

#include "Calculations.h" // Для расчетов

#include "TrajectoryVisualizer.h" // Для визуализации

#include "UserInterface.h" // Для вашего TGUI интерфейса

#include <iostream>

#include <string>

#include <stdexcept> // Для tgui::Exception и std::exception

#include <iomanip> // Для std::fixed, std::setprecision в saveTrajectoryToFile

#include <fstream> // Для std::ofstream в saveTrajectoryToFile

void saveTrajectoryToFile(const WorldTrajectoryData& trajectoryData, const std::string& filename);

int main() {

setlocale(LC\_ALL, "Rus");

// 1. ВИЗУАЛИЗАЦИЯ ТРАЕКТОРИИ //

Calculations calculator;

SimulationParameters params;

std::cout << "Запуск симуляции...\n";

std::vector<State> trajectory = calculator.runSimulation(params);

std::cout << "Симуляция завершена. Получено " << trajectory.size() << " точек траектории.\n";

WorldTrajectoryData worldTrajectory;

worldTrajectory.reserve(trajectory.size());

for (const auto& s : trajectory) {

worldTrajectory.emplace\_back(s.x, s.y);

}

TrajectoryVisualizer visualizer(1000, 800); // Создаем окно визуализатора

// Передаем данные траектории напрямую в визуализатор

visualizer.setData(worldTrajectory);

// Или загружаем из файла (если нужно протестировать загрузку или использовать ранее сохраненные данные)

//if (!visualizer.loadDataFromFile("trajectory.txt")) {

// std::cerr << "Не удалось загрузить траекторию из файла, выход.\n";

// return 1;

//}

visualizer.run(); // Запускаем главный цикл визуализации

//return 0;

// ======================================================================================================== //

// 2. ОКНО ПРОГРАММЫ //

try {

UserInterface uiApp;

uiApp.run();

}

catch (const tgui::Exception& e) {

std::cerr << "TGUI Exception: " << e.what() << std::endl;

return EXIT\_FAILURE;

}

catch (const std::runtime\_error& e) {

std::cerr << "Runtime Exception: " << e.what() << std::endl;

return EXIT\_FAILURE;

}

catch (const std::exception& e) {

std::cerr << "Standard Exception: " << e.what() << std::endl;

return EXIT\_FAILURE;

}

catch (...) {

std::cerr << "An unknown C++ exception occurred." << std::endl;

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}

void saveTrajectoryToFile(const WorldTrajectoryData& trajectoryData, const std::string& filename) {

std::ofstream fout(filename);

if (!fout.is\_open()) {

std::cerr << "Ошибка: не удалось открыть файл '" << filename << "' для записи.\n";

return;

}

// Используем высокую точность для сохранения данных

fout << std::fixed << std::setprecision(10);

for (const auto& point : trajectoryData) {

fout << point.first << " " << point.second << "\n";

}

fout.close();

std::cout << "Результаты симуляции (" << trajectoryData.size() << " точек) записаны в " << filename << "\n";

}

***// Конец main.cpp //***