

**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технологический университет «СТАНКИН» (ФГБОУ ВО «МГТУ «СТАНКИН»)**

**Институт информационных систем и технологий**

**Кафедра инженерной графики**

# Отчет по выполнению лабораторной работы №3

по дисциплине «Геометрическое моделирование и компьютерная графика»

Направление подготовки: 09.03.03 «Прикладная информатика»

Профиль: «Управление данными»

Руководитель: Бейреш А. М.

Оценка Подпись Дата

Студент:

Хусниярова А. Р.

Группа ИДБ-20-11

Подпись Дата

Москва, 2023

# СОДЕРЖАНИЕ

[Задание 3](#_TOC_250009)

[Код 4](#_TOC_250007)

[Результат работы программы 10](#_TOC_250006)

# Задание

По аналогии со второй лабораторной, выполнение каждого дает дополнительные баллы

1. Сделать кнопку очистки путей градиентного спуска **[+5]**
2. Рисовать путь на дополнительной текстуре поверх М-Образа, чтобы ускорить очистку **[+10]**
3. Добавить возможность делать не только градиентный спуск, но и подъем.

# Код

Для выполнения лабораторной работы за основу взят проект второй лабораторной.

Функция отрисовки фигуры была изменена для увеличения изображения в два раза.

Также добавлены функции расчета и отрисовки градиента. В случае нажатия левой кнопкой мыши по экрану – из точки отрисовывается градиент. Засчет сохранения пути градиента в массиве векторов изображение постоянно обновляется и отрисовывает все градиенты.

Добавлена кнопка очистки градиентов.

#include <SFML/Graphics.hpp>

#include <functional>

#include <imgui-SFML.h>

#include <imgui.h>

#include <iostream>

#include <vector>

float ComplexFigure(const sf::Vector2f &point, const sf::Texture &texture);

std::string SaveImagePath; // Глобальная переменная для хранения пути сохранения изображения

std::vector<sf::VertexArray> gradientPaths; // Вектор для хранения градиентов

enum class MShape

{

ComplexFigure,

CustomMShape1,

CustomMShape2

};

MShape selectedMShape = MShape::ComplexFigure;

float CustomMShape1(const sf::Vector2f &point, const sf::Texture &texture)

{

// Реализация вашей первой M-образной формы

// В данном примере, просто возвращаем 0, чтобы показать пустую область

return 0.0f;

}

float CustomMShape2(const sf::Vector2f &point, const sf::Texture &texture)

{

// Реализация вашей второй M-образной формы

// В данном примере, просто возвращаем 0, чтобы показать пустую область

return 0.0f;

}

float SelectedMShape(const sf::Vector2f &point, const sf::Texture &texture)

{

switch (selectedMShape)

{

case MShape::ComplexFigure:

return ComplexFigure(point, texture);

case MShape::CustomMShape1:

return CustomMShape1(point, texture);

case MShape::CustomMShape2:

return CustomMShape2(point, texture);

default:

return 0.0f;

}

}

struct Functions

{

static float Circle(const sf::Vector2f &center, float radius, const sf::Vector2f &point)

{

return std::sqrt((point.x - center.x) \* (point.x - center.x) + (point.y - center.y) \* (point.y - center.y)) -

radius;

}

static float ROr(float w1, float w2) { return std::max(w1, w2); }

static float RAnd(float w1, float w2) { return std::min(w1, w2); }

};

float ComplexFigure(const sf::Vector2f &point, const sf::Texture &texture)

{

float scaleFactor = 2.0f; // Adjust the scale factor as needed

sf::Vector2f originalCenter{static\_cast<float>(texture.getSize().x) / 2,

static\_cast<float>(texture.getSize().y) / 2};

const sf::Vector2f circleCenter1 = originalCenter + sf::Vector2f(0, 100) \* scaleFactor;

const float circleRadius1 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 6 \* scaleFactor;

const sf::Vector2f circleCenter2 = originalCenter + sf::Vector2f(0, -75) \* scaleFactor;

const float circleRadius2 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 8 \* scaleFactor;

const sf::Vector2f circleCenter3 = originalCenter + sf::Vector2f(-50, -170) \* scaleFactor;

const float circleRadius3 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 25 \* scaleFactor;

const sf::Vector2f circleCenter4 = originalCenter + sf::Vector2f(50, -170) \* scaleFactor;

const float circleRadius4 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 25 \* scaleFactor;

const sf::Vector2f circleCenter5 = originalCenter + sf::Vector2f(130, 50) \* scaleFactor;

const float circleRadius5 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 10 \* scaleFactor;

const sf::Vector2f circleCenter6 = originalCenter + sf::Vector2f(130, 150) \* scaleFactor;

const float circleRadius6 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 10 \* scaleFactor;

const sf::Vector2f circleCenter7 = originalCenter + sf::Vector2f(-130, 50) \* scaleFactor;

const float circleRadius7 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 10 \* scaleFactor;

const sf::Vector2f circleCenter8 = originalCenter + sf::Vector2f(-130, 150) \* scaleFactor;

const float circleRadius8 =

std::min(static\_cast<float>(texture.getSize().x), static\_cast<float>(texture.getSize().y)) / 10 \* scaleFactor;

float result1 = Functions::Circle(circleCenter1, circleRadius1, point);

float result2 = Functions::Circle(circleCenter2, circleRadius2, point);

float result3 = Functions::Circle(circleCenter3, circleRadius3, point);

float result4 = Functions::Circle(circleCenter4, circleRadius4, point);

float result5 = Functions::Circle(circleCenter5, circleRadius5, point);

float result6 = Functions::Circle(circleCenter6, circleRadius6, point);

float result7 = Functions::Circle(circleCenter7, circleRadius7, point);

float result8 = Functions::Circle(circleCenter8, circleRadius8, point);

float res1 = Functions::ROr(result5, result6);

float res2 = Functions::RAnd(result1, res1);

float res3 = Functions::ROr(result7, result8);

float res4 = Functions::RAnd(res2, res3);

float res5 = Functions::RAnd(result2, result3);

float res6 = Functions::RAnd(res5, result4);

float res7 = Functions::RAnd(res4, res6);

return res7;

}

class RFuncSprite : public sf::Sprite

{

public:

void Create(const sf::Vector2u &size);

void DrawRFunc(const std::function<float(const sf::Vector2f &)> &rfunc, const sf::FloatRect &subSpace);

const sf::Texture &getTexture() const { return \_texture; }

const sf::Image &getImage() const { return \_image; }

// Методы для установки цветов

void setFirstColor(const sf::Color &color) { \_firstColor = color; }

void setSecondColor(const sf::Color &color) { \_secondColor = color; }

const sf::Color &getFirstColor() const { return \_firstColor; }

const sf::Color &getSecondColor() const { return \_secondColor; }

private:

sf::Texture \_texture;

sf::Image \_image;

// Переменные для хранения цветов

sf::Color \_firstColor = sf::Color::Red;

sf::Color \_secondColor = sf::Color::Blue;

};

void RFuncSprite::Create(const sf::Vector2u &size)

{

\_image.create(size.x, size.y, sf::Color::Cyan);

\_texture.loadFromImage(\_image);

setTexture(\_texture);

}

void RFuncSprite::DrawRFunc(const std::function<float(const sf::Vector2f &)> &rfunc, const sf::FloatRect &subSpace)

{

const sf::Texture &texture = getTexture();

sf::Vector2f spaceStep = {

subSpace.width / static\_cast<float>(\_image.getSize().x),

subSpace.height / static\_cast<float>(\_image.getSize().y),

};

for (unsigned y = 0; y < \_image.getSize().y; ++y)

{

for (unsigned x = 0; x < \_image.getSize().x; ++x)

{

int x1 = x, y1 = y;

int x2 = x, y2 = y + 1;

int x3 = x + 1, y3 = y;

float z1 = rfunc({static\_cast<float>(x1), static\_cast<float>(y1)});

float z2 = rfunc({static\_cast<float>(x2), static\_cast<float>(y2)});

float z3 = rfunc({static\_cast<float>(x3), static\_cast<float>(y3)});

float Ai = y1 \* (z2 - z3) - y2 \* (z1 - z3) + y3 \* (z1 - z2);

float Bi = -(x1 \* (z2 - z3) - x2 \* (z1 - z3) + x3 \* (z1 - z2));

float Ci = x1 \* (y2 - y3) - x2 \* (y1 - y3) + x3 \* (y1 - y2);

float Di = x1 \* (y2 \* z3 - y3 \* z2) - x2 \* (y1 \* z3 - y3 \* z1) + x3 \* (y1 \* z2 - y2 \* z1);

float Nx = Ai / std::sqrtf(Ai \* Ai + Bi \* Bi + Ci \* Ci);

float Ny = -Bi / std::sqrtf(Ai \* Ai + Bi \* Bi + Ci \* Ci);

float Nz = -Ci / std::sqrtf(Ai \* Ai + Bi \* Bi + Ci \* Ci);

float Nt = Di / std::sqrtf(Ai \* Ai + Bi \* Bi + Ci \* Ci + Di \* Di);

uint8\_t Cx = (Nx + 1.f) \* 127.f;

// Используйте цвета для закрашивания области

int zoneFlag = (z1 >= 0) + (z2 >= 0) + (z3 >= 0);

if (zoneFlag > 2)

\_image.setPixel(x, y, sf::Color(Cx, 0, 0));

else

\_image.setPixel(x, y, sf::Color(0, 0, Cx));

}

}

\_texture.loadFromImage(\_image);

}

sf::Vector2f Gradient(const std::function<float(const sf::Vector2f &)> &rfunc, const sf::Vector2f &point,

float epsilon = 0.01)

{

float x = point.x;

float y = point.y;

float dx = (rfunc({x + epsilon, y}) - rfunc({x - epsilon, y})) / (2 \* epsilon);

float dy = (rfunc({x, y + epsilon}) - rfunc({x, y - epsilon})) / (2 \* epsilon);

return sf::Vector2f(-dx, -dy); // Negative gradient for descent

}

sf::VertexArray GradientDescent(const std::function<float(const sf::Vector2f &)> &rfunc, const sf::Vector2f &startPoint,

int steps = 10000, float stepSize = 0.1)

{

sf::VertexArray descentPath(sf::LineStrip);

descentPath.append(sf::Vertex(startPoint, sf::Color::Green));

sf::Vector2f currentPoint = startPoint;

for (int i = 0; i < steps; ++i)

{

sf::Vector2f gradient = Gradient(rfunc, currentPoint);

currentPoint += stepSize \* gradient;

descentPath.append(sf::Vertex(currentPoint, sf::Color::Green));

}

return descentPath;

}

void HandleUserInput(sf::RenderWindow &window, const sf::Event &event)

{

switch (event.type)

{

case sf::Event::Closed:

window.close();

break;

default:

break;

}

}

void Update(sf::RenderWindow &window, const sf::Time &deltaClock)

{

// Make some time-dependent updates, like: physics, gameplay logic, animations, etc.

}

void Render(sf::RenderWindow &window, RFuncSprite &rFuncSprite, const sf::FloatRect &subSpace,

sf::Vector2f &mouseClickPoint)

{

// Отрисовка всех градиентов

for (const auto &gradientPath : gradientPaths)

{

window.draw(gradientPath);

}

// Отрисовка R-функции

rFuncSprite.DrawRFunc([&rFuncSprite](const sf::Vector2f &point) -> float

{ return SelectedMShape(point, rFuncSprite.getTexture()); },

subSpace);

window.draw(rFuncSprite);

// Проверка, была ли нажата кнопка мыши

if (sf::Mouse::isButtonPressed(sf::Mouse::Left))

{

// Get the mouse position in world coordinates

sf::Vector2i mousePosition = sf::Mouse::getPosition(window);

mouseClickPoint = window.mapPixelToCoords(mousePosition);

sf::VertexArray descentPath = GradientDescent([&rFuncSprite](const sf::Vector2f &point) -> float

{ return SelectedMShape(point, rFuncSprite.getTexture()); },

mouseClickPoint);

// Добавление нового градиента в вектор

gradientPaths.push\_back(descentPath);

for (const auto &gradientPath : gradientPaths)

{

window.draw(gradientPath);

}

}

}

void RenderGui(sf::RenderWindow &window, RFuncSprite &rFuncSprite)

{

extern std::vector<sf::VertexArray> gradientPaths; // Вектор для хранения градиентов

ImGui::Begin("Settings");

ImGui::Text("S: %lu", gradientPaths.size());

// Отрисовка кнопки для очистки всех градиентов

if (ImGui::Button("Clear Gradients"))

{

gradientPaths.clear(); // Очистка вектора градиентов

}

ImGui::Text("Select M-Shape:");

if (ImGui::RadioButton("Complex Animal", selectedMShape == MShape::ComplexFigure))

selectedMShape = MShape::ComplexFigure;

if (ImGui::RadioButton("Custom M-Shape 1", selectedMShape == MShape::CustomMShape1))

selectedMShape = MShape::CustomMShape1;

if (ImGui::RadioButton("Custom M-Shape 2", selectedMShape == MShape::CustomMShape2))

selectedMShape = MShape::CustomMShape2;

ImGui::Separator();

// Добавляем кнопку для случайного изменения цветов

if (ImGui::Button("Randomize Colors"))

{

// Генерируем случайные цвета и устанавливаем их в RFuncSprite

sf::Color randomColor1(rand() % 256, rand() % 256, rand() % 256);

sf::Color randomColor2(rand() % 256, rand() % 256, rand() % 256);

rFuncSprite.setFirstColor(randomColor1);

rFuncSprite.setSecondColor(randomColor2);

// Перерисовываем изображение с новыми цветами

rFuncSprite.Create(window.getSize());

}

const char \*HardcodedSavePath = "C:/Users/Khusn/Desktop/GeomLabs";

if (ImGui::Button("Save Image"))

{

// Use the hardcoded path directly in your save image logic

sf::Image screenshot = rFuncSprite.getImage();

screenshot.saveToFile(HardcodedSavePath + std::string("/output\_image.png"));

std::cout << "Image saved to: " << HardcodedSavePath << "/output\_image.png" << std::endl;

}

// Отрисовка кнопки для очистки всех градиентов

if (ImGui::Button("Clear Gradients"))

{

gradientPaths.clear(); // Очистка вектора градиентов

}

ImGui::End();

}

int main()

{

sf::RenderWindow window(sf::VideoMode(800, 800), "Geometry modeling 1");

window.setFramerateLimit(60);

if (!ImGui::SFML::Init(window))

{

std::cout << "ImGui initialization failed\n";

return -1;

}

RFuncSprite rFuncSprite;

rFuncSprite.Create(window.getSize());

sf::Vector2f mouseClickPoint;

sf::Clock deltaClock;

while (window.isOpen())

{

sf::Event event;

while (window.pollEvent(event))

{

ImGui::SFML::ProcessEvent(window, event);

HandleUserInput(window, event);

}

sf::Time deltaTime = deltaClock.restart();

ImGui::SFML::Update(window, deltaTime);

Update(window, deltaTime);

window.clear();

RenderGui(window, rFuncSprite);

Render(window, rFuncSprite, sf::FloatRect(-10.0f, -10.0f, 20.0f, 20.0f), mouseClickPoint);

ImGui::SFML::Render(window);

window.display();

}

ImGui::SFML::Shutdown();

return 0;

}

# Результат работы программы

После запуска программы получим следующий М-образ мишки. При нажатии на правую клавишу мыши в разных точках будет прокладываться траектория градиентного спуска или подъема из выбранной точки. Также реализована кнопка очистки массива градиентных векторов.



