МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра вычислительной техники

ОТЧЁТ
по лабораторной работе №5
«Алгоритмы заполнения многоугольников»

по дисциплине: «Компьютерная графика»

Выполнил(а):
Факультет: АВТФ
Группа: АВТ-418
Студенты:
Куриленко Артём
Юрков Владислав

Проверил(а):

Надежницкая
Вероника Анатольевна

«16» декабря 2025 г.

«___» _____ 2025

_____
(оценка, подпись)

Новосибирск
2025

Цель работы: изучить алгоритмы построчного заполнения многоугольников.

1. Теоретическая часть

1.1. Постановка задачи

Заполнение многоугольников – фундаментальная задача компьютерной графики, требующая эффективных алгоритмов для определения пикселей, принадлежащих внутренней области многоугольника произвольной формы.

1.2. Изучаемые алгоритмы

1.2.1. Алгоритм построчного заполнения

Принцип работы:

- Для каждой сканирующей строки определяются точки пересечения с рёбрами многоугольника.
- Точки пересечения сортируются по возрастанию координаты X.
- Закрашиваются отрезки между парами точек пересечения.

Преимущества:

- Высокая эффективность за счёт когерентности строк.
- Минимальное количество операций доступа к пикселям.
- Простота реализации.

Особенности обработки:

- Учёт локальных экстремумов (вершин, являющихся максимумами или минимумами).
- Корректная обработка вершин на границах сканирующих строк.

1.2.2. Алгоритм заполнения по рёбрам

Принцип работы:

- Для каждого ребра многоугольника инвертируются все пиксели справа от точки пересечения.

- Горизонтальные рёбра пропускаются.
- Многократная инверсия приводит к правильному заполнению внутренней области.

Преимущества:

- Наглядная демонстрация процесса заполнения.
- Простота понимания принципа работы.
- Не требует сложных вычислений.

Недостатки:

- Многократная обработка одних и тех же пикселей.
- Большое количество операций инверсии.
- Менее эффективен по сравнению с построчным заполнением.

2. Практическая часть

2.1. Структура программы

Программа реализует визуализацию обоих алгоритмов с возможностью интерактивного переключения между ними и наблюдения процесса заполнения в реальном времени.

2.2. Реализованные функции

2.2.1. fillPolygonScanline() – алгоритм построчного заполнения

- Определяет минимальную и максимальную Y-координаты многоугольника.
- Для каждой строки находит пересечения с рёбрами.
- Сортирует пересечения и закрашивает отрезки между парами точек.

2.2.2. fillPolygonByEdges() – алгоритм заполнения по рёбрам

- Обрабатывает каждое ребро многоугольника (кроме горизонтальных).
- Для каждой точки пересечения инвертирует пиксели справа.
- Визуализирует процесс инверсии в реальном времени.

2.2.3. render() – основная функции управления программой

- Обрабатывает пользовательский ввод.
- Переключает между алгоритмами.

2.3. Особенности реализации

- Использован тестовый многоугольник с обычными вершинами и экстремумами.
- Визуализации процесса заполнения с задержками для наглядности.
- Интерактивный интерфейс с подсказами управления.

Задание:

1. Построить многоугольник (при этом должны быть использованы обычные вершины и локальные максимумы или минимумы координат вершин).
2. Используя один из алгоритмов закрасить многоугольник.
3. При этом ввести программную задержку после прохождения очередной сканирующей строки или ребра.
4. Сравните алгоритмы.

Результат выполнения программы:

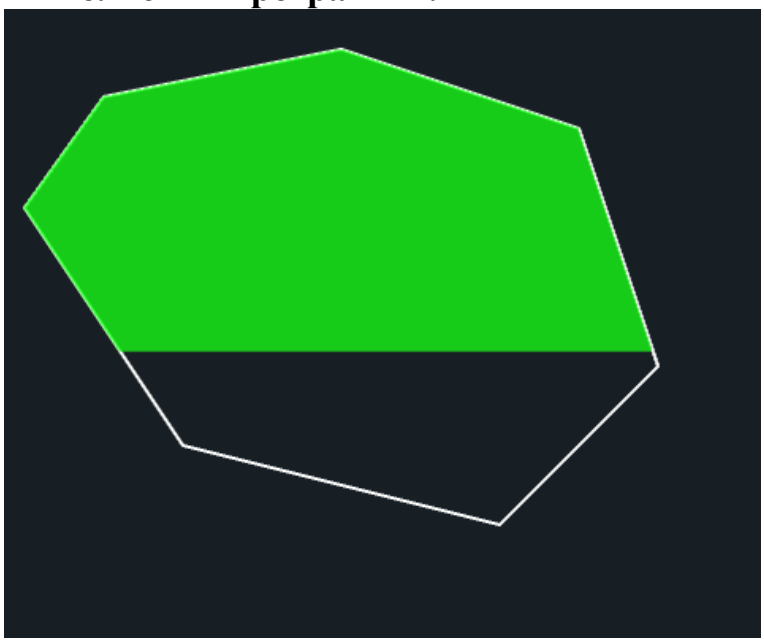

Рисунок 1 Построчное заполнение многоугольника

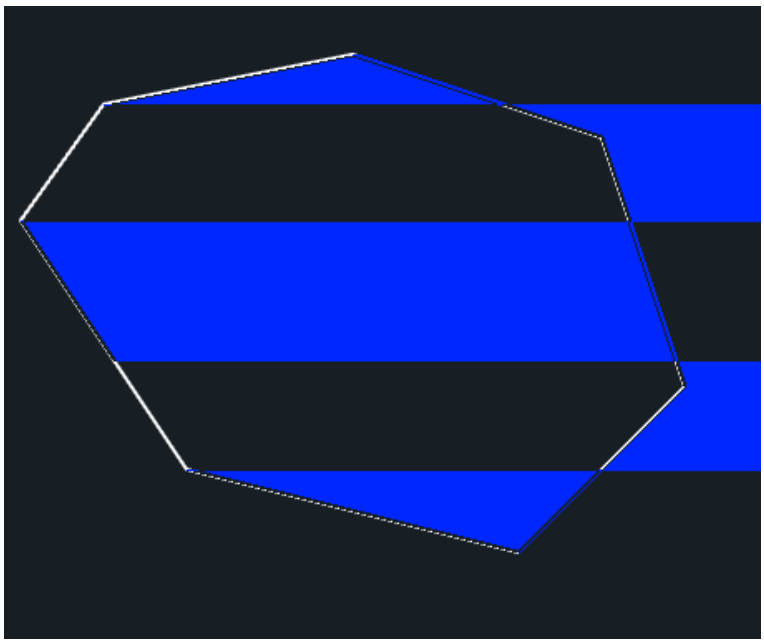

Рисунок 2 Заполнение многоугольника по рёбрам

3. Сравнительный анализ алгоритмов

3.1. Производительность

- Построчное заполнение: более эффективно, минимальное количество операций с пикселями.
- Заполнение по рёбрам: менее эффективно, многократная обработка пикселей.

3.2. Визуализация процесса

- Построчное заполнение: последовательное заполнение строка за строкой.
- Заполнение по рёбрам: наглядное отображение инверсии от каждого ребра.

3.3. Сложность реализации

- Построчное заполнение: требует корректного определения и сортировки пересечений.
- Заполнение по рёбрам: проще в реализации, но требует аккуратной обработки рёбер.

Вывод

В ходе выполнения лабораторной работы оба алгоритма успешно решают задачу заполнения многоугольников произвольной формы. Алгоритм построчного заполнения более эффективен для практического применения, а алгоритм заполнения по рёбрам предоставляет наглядную демонстрацию процесса работы.

Корректная обработка особых случаев (экстремумы, горизонтальные рёбра) является важным аспектом реализации. Визуализация процесса заполнения позволяет лучше понять принципы работы алгоритмов.

Полученные знания и навыки являются фундаментальными для дальнейшей работы в области компьютерной графики и обработки изображений.

ПРИЛОЖЕНИЕ

```ts
async function fillPolygonByEdges(ctx: CanvasRenderingContext2D): Promise<void> {
    if (polygon.length === 0) return;

    const imageData = ctx.getImageData(0, 0, ctx.canvas.width, ctx.canvas.height);
    const data: number[] = Array.from(imageData.data);

    const getAlpha = (x: number, y: number): number => {
        if (x < 0 || x >= ctx.canvas.width || y < 0 || y >= ctx.canvas.height) {
            return -1;
        }
        const index = (Math.floor(y) * ctx.canvas.width + Math.floor(x)) * 4;
        const alpha = data[index + 3];
        return alpha ?? 0;
    };

    const setRGBA = (x: number, y: number, r: number, g: number, b: number, a: number = 255): void => {
        if (x < 0 || x >= ctx.canvas.width || y < 0 || y >= ctx.canvas.height) return;
        const index = (Math.floor(y) * ctx.canvas.width + Math.floor(x)) * 4;
        data[index] = r;
        data[index + 1] = g;
        data[index + 2] = b;
        data[index + 3] = a;
    };

    const invertPixelsRight = (x: number, y: number): void => {
        const startX = Math.floor(x);
        const row = Math.floor(y);
        for (let invertX = startX; invertX < ctx.canvas.width; invertX++) {
            const alpha = getAlpha(invertX, row);
            if (alpha === -1) {
                continue;
            }
            if (alpha === 0) {
                setRGBA(invertX, row, 0, 0, 255, 255);
            }
            else {
                setRGBA(invertX, row, 0, 0, 0, 0);
            }
        }
    };

    for (let i = 0; i < polygon.length; i++) {
        const [vx, vy] = polygon[i] as [number, number];
        const [, py] = polygon[(i - 1 + polygon.length) % polygon.length] as [number, number];
        const [, ny] = polygon[(i + 1) % polygon.length] as [number, number];

        const prevHor = py === vy;
        const nextHor = vy === ny;
        if (prevHor || nextHor) continue;

        const isMax = py < vy && ny < vy;
        const isMin = py > vy && ny > vy;
        if (!isMax && !isMin) {
            invertPixelsRight(vx, vy);
        }
    }

    const updated = new Uint8ClampedArray(data);
    ctx.putImageData(new ImageData(updated, imageData.width, imageData.height), 0, 0);

    for (let i = 0; i < polygon.length; i++) {
        let [x1, y1] = polygon[i] as [number, number];
        let [x2, y2] = polygon[(i + 1) % polygon.length] as [number, number];

        if (y1 === y2) continue;

        if (y1 > y2) {
            [x1, y1, x2, y2] = [x2, y2, x1, y1];
        }

        const startY = Math.floor(y1);
        const endY = Math.floor(y2);
        if (startY > endY) continue;
        const slope = (x2 - x1) / (y2 - y1);

        for (let y = startY; y <= endY; y++) {
            const x = x1 + (y - y1) * slope;
            invertPixelsRight(x, y);

            await sleep();
            const frameData = new Uint8ClampedArray(data);
            ctx.putImageData(new ImageData(frameData, imageData.width, imageData.height), 0, 0);
        }
    }
}
```

```ts
async function fillPolygonScanline(ctx: CanvasRenderingContext2D): Promise<void> {
    if (polygon.length === 0) return;

    const minY = Math.min( ...polygon.map(p => p[1]));
    const maxY = Math.max( ...polygon.map(p => p[1]));

    for (let y = Math.floor(minY); y <= Math.ceil(maxY); y++) {
        const intersections: number[] = [];

        for (let i = 0; i < polygon.length; i++) {
            const [x1, y1] = polygon[i] as [number, number];
            const [x2, y2] = polygon[(i + 1) % polygon.length] as [number, number];

            if ((y1 <= y && y < y2) || (y2 <= y && y < y1)) {
                if (y1 !== y2) {
                    const x = x1 + (x2 - x1) * (y - y1) / (y2 - y1);
                    intersections.push(x);
                }
            }
        }

        intersections.sort((a, b) => a - b);

        ctx.strokeStyle = COLORS.GREEN;
        ctx.lineWidth = 1;
        for (let i = 0; i + 1 < intersections.length; i += 2) {
            const left = intersections[i];
            const right = intersections[i + 1];
            if (left === undefined || right === undefined) continue;

            const startX = Math.floor(left);
            const endX = Math.floor(right);

            ctx.beginPath();
            ctx.moveTo(startX, y);
            ctx.lineTo(endX, y);
            ctx.stroke();
        }

        await sleep();
    }
}
```

```ts
export default async function render(
    ctx: CanvasRenderingContext2D,
    algo: FillAlgorithm = FillAlgorithm.SCANLINE,
): Promise<void> {
    ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);

    drawPolygonOutline(ctx);

    switch (algo) {
        case FillAlgorithm.SCANLINE:
            await fillPolygonScanline(ctx);
            break;
        case FillAlgorithm.EDGE:
            await fillPolygonByEdges(ctx);
            break;
    }

    drawPolygonOutline(ctx);
}
```