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 №6
«Заливка области с затравкой»

по дисциплине: «Компьютерная графика»

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

Проверил(а):

Надежницкая
Вероника Анатольевна

«16» декабря 2025 г.

«___» _____ 2025
_____
(оценка, подпись)

Новосибирск
2025

Цель работы: изучить алгоритмы заливки области с затравкой [1, 3].

1. Теоретическая часть

1.1. Постановка задачи

Заливка области с затравкой – это процесс заполнения связной области изображения, ограниченной определённой границей, начиная с заданной начальной точки (затравкой).

1.2. Классификация областей

1.2.1. Гранично-определённые области

- Задаются замкнутой границей.
- Коды пикселов границы отличны от кодов внутренней части.
- При заливке перекрашиваются пикселы с кодом, отличным от кода границы.

1.2.2. Внутренне-определённые области

- Нарисованы одним определённым кодом пиксела.
- При заливке этот код заменяется на новый код закраски.

1.3. Типы связности

1.3.1. 4-связные области

Доступ к соседним пикселам осуществляется по 4 направлениям:

- Горизонтальное влево и вправо.
- Вертикально вверх и вниз.

1.3.2. 8-связные области

Доступ к соседним пикселам осуществляется по 8 направлениям (4 основных + 4 диагональных).

1.4. Изучаемые алгоритмы

1.4.1. Простой алгоритм заливки

Принцип работы:

- Используется рекурсивный или итеративный подход со стеком.
- Для каждого пиксела проверяются 4 или 8 соседей.
- Подходит для небольших областей.

Недостатки:

- Большие затраты памяти на стек.
- Многократная обработка одних и тех же пикселов.
- Риск переполнения стека для больших областей.

1.4.2. Построчный алгоритм заливки

Принцип работы:

- Использует пространственную когерентность.
- Закрашивает целые отрезки строк.
- На каждый закрашиваемый фрагмент хранится только одна затравка.

Преимущества:

- Существенное уменьшение размера стека.
- Высокая производительность.
- Эффективное использование памяти.

2. Практическая часть

2.1. Структура программы

Программа реализует визуализацию обоих алгоритмов с возможностью создания областей с отверстиями и наблюдения процесса заполнения.

2.2. Реализованные функции

2.2.1. simpleFloodFillInternal – простой алгоритм заливки

- Использует стек для хранения координат пикселов.
- Проверяет 4-связных или 8-связных соседей.
- Рекурсивно заполняет область.

2.2.2. scanlineFloodFillInternal – построчный алгоритм заливки

- Закрашивает целые отрезки строк.
- Использует затравки для незакрашенных фрагментов.
- Оптимизирует использование памяти.

2.2.3. drawPlayground – создание тестовой сцены

- Рисует область с отверстиями.
- Добавляет различные препятствия.

2.2.4. floodFill – основная функции управления программой

- Обрабатывает пользовательский ввод.
- Переключает между алгоритмами.
- Управляет визуализацией.

Задание:

1. Построить область с отверстиями, произвести заполнения, используя алгоритм простой затравки.
2. Построить область с отверстиями, произвести заполнение, используя алгоритм построчной затравки.
3. Сравните два алгоритма, найдите недостатки и преимущества.
4. Чем отличаются четырёх- и восьмисвязные алгоритмы (и области). Преимущества и недостатки.

Результат выполнения программы:

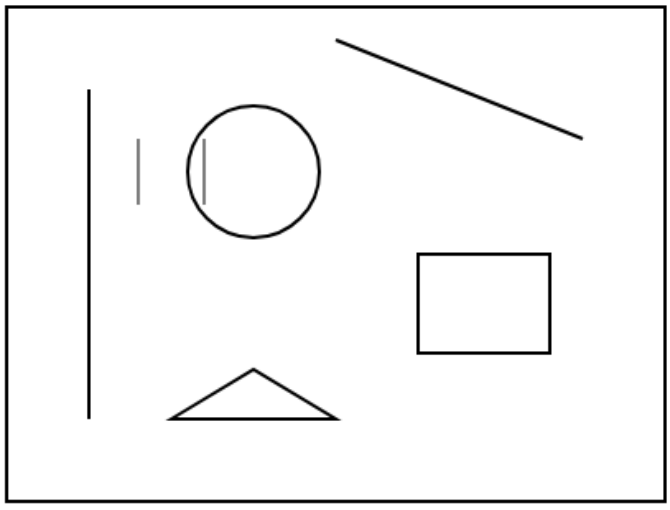

Рисунок 1 Начальный экран простой (4-связной) заливки области с затравкой

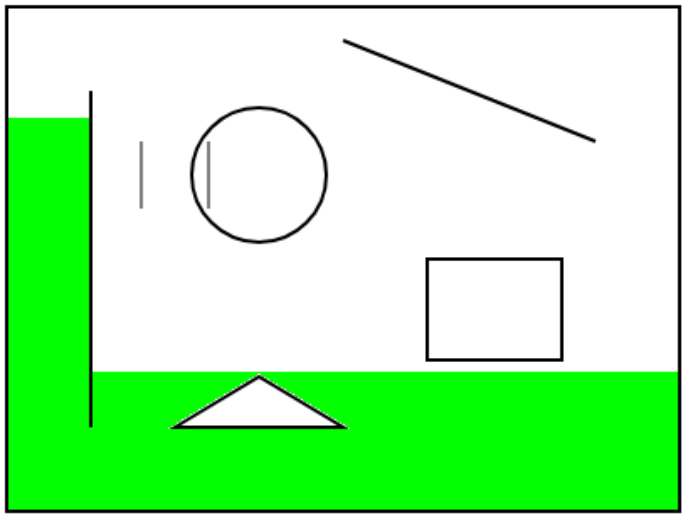

Рисунок 2 Построчная (4-связная) заливки области с затравкой

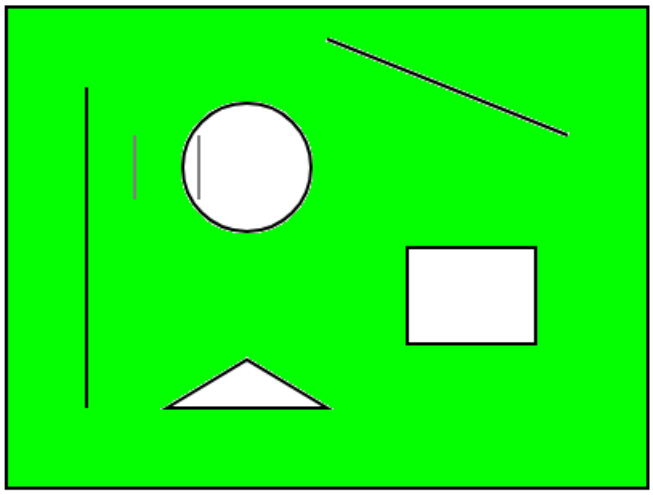

Рисунок 3 Построчная (4-связная) заливка области с затравкой

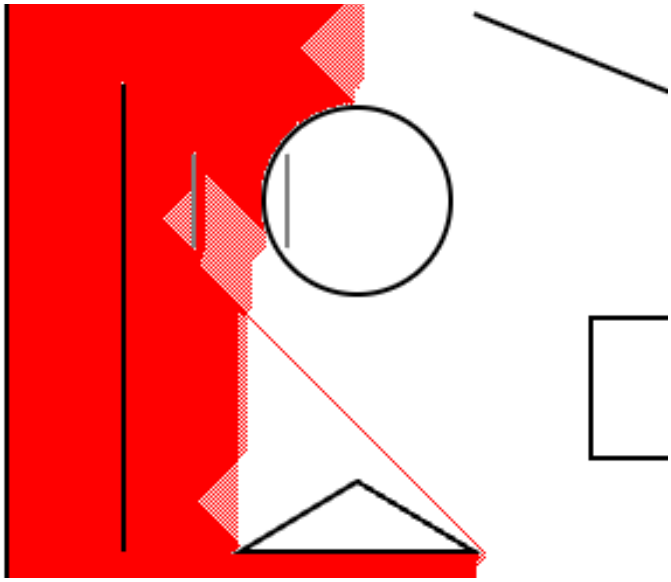

Рисунок 4 Простая (8-связная) заливка области с затравкой

3. Сравнительный анализ алгоритмов

3.1. Использование памяти

- Простая заливка: большой расход памяти на стек, риск переполнения.
- Построчная заливка: минимальное использование памяти, хранит только затравки.

3.2. Производительность

- Простая заливка: медленнее из-за многократной обработки пикселов.
- Построчная заливка: высокая скорость за счёт закрашивания целых отрезков.

3.3. Надёжность

- Простая заливка: может вызвать переполнение стека для больших областей.
- Построчная заливка: стабильная работа для областей любого размера.

4. Особенности реализации

4.1. Обработка граничных условий – корректная проверка выхода за границы изображения.

4.2. Оптимизация производительности – визуализация процесса с минимальными задержками.

4.3. Интерактивный интерфейс – возможность переключения алгоритмов и сброса сцены.

4.4. Визуальная обратная связь – отображение текущего алгоритма и инструкций по управлению.

Данная реализация демонстрирует все ключевые аспекты алгоритмов заливки с затравкой и позволяет наглядно сравнить их работу.

Вывод

В ходе выполнения лабораторной работы были успешно изучены и реализованы два алгоритма заливки области с затравкой: простой алгоритм заливки и построчный алгоритм заливки.

Проведённые исследования показали, что оба алгоритма эффективно решают поставленную задачу по заполнению связных областей, однако обладают различными характеристиками производительности и областью применения.

Построчный алгоритм заливки демонстрирует значительно лучшую производительность и эффективность использования памяти благодаря оптимизированному подходу к обработке целых отрезков строк. Этот алгоритм минимизирует количество операций с пикселями и существенно сокращает требования к объёму памяти, что делает его предпочтительным для работы с крупными областями.

Простой алгоритм заливки, несмотря на свою ограниченную производительность и повышенные требования к памяти, остаётся ценным с образовательной точки зрения. Его простота реализации и наглядность работы делают его идеальным инструментом для первоначального изучения принципов заливки областей.

Критически важным аспектом для обоих алгоритмов является корректная обработка граничных условий и особых случаев, таких как области со сложной геометрией и наличием отверстий.

Выбор конкретного алгоритма для практического применения должен осуществляться на основе анализа требований к производительности, объёма обрабатываемых данных и доступных вычислительных ресурсов. Для задач, требующих высокой скорости обработки и работы с большими областями, рекомендуется использование построчного алгоритма, тогда как для учебных целей и работы с небольшими областями может быть достаточно простого алгоритма заливки.

ПРИЛОЖЕНИЕ

```
function checkScanline(
    stack: [number, number][],
    data: Uint8ClampedArray,
    width: number,
    height: number,
    leftX: number,
    rightX: number,
    y: number,
    targetColor: [number, number, number, number],
): void {
    let x = leftX;
    while (x <= rightX) {
        const color = getPixelColor(data, width, height, x, y);
        if (color && colorsEqual(color, targetColor)) {
            const segmentStart = x;

            while (x <= rightX) {
                const c = getPixelColor(data, width, height, x, y);
                if (!c || !colorsEqual(c, targetColor)) {
                    break;
                }
                x++;
            }

            stack.push([segmentStart, y]);
        }
        else {
            x++;
        }
    }
}

async function scanlineFloodFillInternal(
    ctx: CanvasRenderingContext2D,
    x: number,
    y: number,
    targetColor: [number, number, number, number],
    replacementColor: [number, number, number, number],
    connectivity: ConnectivityType,
): Promise<FloodFillStats> {
    if (colorsEqual(targetColor, replacementColor)) {
        return { pixelsFilled: 0, maxStackSize: 0, time: 0 };
    }

    const startTime = performance.now();
    const imageData = ctx.getImageData(0, 0, ctx.canvas.width, ctx.canvas.height);
    const data = imageData.data;
    const width = ctx.canvas.width;
    const height = ctx.canvas.height;

    const stack: [number, number][] = [[x, y]];
    let filledPixels = 0;
    let maxStackSize = 0;

    while (stack.length > 0) {
        const [seedX, seedY] = stack.pop()!;

        if (seedX < 0 || seedX >= width || seedY < 0 || seedY >= height) {
            continue;
        }

        const currentColor = getPixelColor(data, width, height, seedX, seedY);

        if (!currentColor || !colorsEqual(currentColor, targetColor)) {
            continue;
        }

        let leftX = seedX;
        while (leftX > 0) {
            const color = getPixelColor(data, width, height, leftX - 1, seedY);
            if (!color || !colorsEqual(color, targetColor)) {
                break;
            }
            leftX--;
        }

        let rightX = seedX;
        while (rightX < width - 1) {
            const color = getPixelColor(data, width, height, rightX + 1, seedY);
            if (!color || !colorsEqual(color, targetColor)) {
                break;
            }
            rightX++;
        }

        for (let xPos = leftX; xPos <= rightX; xPos++) {
            setPixelColor(
                data,
                width,
                height,
                xPos,
                seedY,
                replacementColor[0],
                replacementColor[1],
                replacementColor[2],
                replacementColor[3],
            );
            filledPixels++;
        }

        if (connectivity === 4) {
            if (seedY > 0) {
                checkScanline(stack, data, width, height, leftX, rightX, seedY - 1, targetColor);
            }
            if (seedY < height - 1) {
                checkScanline(stack, data, width, height, leftX, rightX, seedY + 1, targetColor);
            }
        }
        else {
            for (const dy of [-1, 1]) {
                const newY = seedY + dy;
                if (newY >= 0 && newY < height) {
                    const extendedLeft = Math.max(0, leftX - 1);
                    const extendedRight = Math.min(width - 1, rightX + 1);
                    checkScanline(stack, data, width, height, extendedLeft, extendedRight, newY, targetColor);
                }
            }
        }

        maxStackSize = Math.max(maxStackSize, stack.length);

        ctx.putImageData(imageData, 0, 0);
        await sleep(1);
    }

    ctx.putImageData(imageData, 0, 0);

    return {
        pixelsFilled: filledPixels,
        maxStackSize,
        time: (performance.now() - startTime) / 1000,
    };
}
```

```
async function simpleFloodFillInternal(
    ctx: CanvasRenderingContext2D,
    x: number,
    y: number,
    targetColor: [number, number, number, number],
    replacementColor: [number, number, number, number],
    connectivity: ConnectivityType,
): Promise<FloodFillStats> {
    if (colorsEqual(targetColor, replacementColor)) {
        return { pixelsFilled: 0, maxStackSize: 0, time: 0 };
    }

    const startTime = performance.now();
    const imageData = ctx.getImageData(0, 0, ctx.canvas.width, ctx.canvas.height);
    const data = imageData.data;
    const width = ctx.canvas.width;
    const height = ctx.canvas.height;

    const stack: [number, number][] = [[x, y]];
    let filledPixels = 0;
    let maxStackSize = 0;

    const directions: [number, number][]
        = connectivity === 4
            ? [[1, 0], [-1, 0], [0, 1], [0, -1]]
            : [[1, 0], [-1, 0], [0, 1], [0, -1], [1, 1], [1, -1], [-1, 1], [-1, -1]];

    while (stack.length > 0) {
        const [currentX, currentY] = stack.pop()!;

        if (currentX < 0 || currentX >= width || currentY < 0 || currentY >= height) {
            continue;
        }

        const currentColor = getPixelColor(data, width, height, currentX, currentY);

        if (!currentColor || !colorsEqual(currentColor, targetColor)) {
            continue;
        }

        setPixelColor(
            data,
            width,
            height,
            currentX,
            currentY,
            replacementColor[0],
            replacementColor[1],
            replacementColor[2],
            replacementColor[3],
        );
        filledPixels++;

        for (const [dx, dy] of directions) {
            stack.push([currentX + dx, currentY + dy]);
        }

        maxStackSize = Math.max(maxStackSize, stack.length);

        if (filledPixels % 50 === 0) {
            ctx.putImageData(imageData, 0, 0);
            await sleep(1);
        }
    }

    return {
        pixelsFilled: filledPixels,
        maxStackSize,
        time: (performance.now() - startTime) / 1000,
    };
}
```

```ts
export function drawPlayground(ctx: CanvasRenderingContext2D): void {
    const canvas = ctx.canvas;
    const width = canvas.width;
    const height = canvas.height;

    ctx.fillStyle = COLORS.WHITE;
    ctx.fillRect(0, 0, width, height);

    ctx.strokeStyle = COLORS.BLACK;
    ctx.lineWidth = 2;
    ctx.strokeRect(100, 100, 400, 300);

    ctx.beginPath();
    ctx.arc(250, 200, 40, 0, Math.PI * 2);
    ctx.stroke();

    ctx.strokeRect(350, 250, 80, 60);

    ctx.beginPath();
    ctx.moveTo(200, 350);
    ctx.lineTo(250, 320);
    ctx.lineTo(300, 350);
    ctx.closePath();
    ctx.stroke();

    ctx.lineWidth = 2;
    ctx.beginPath();
    ctx.moveTo(150, 150);
    ctx.lineTo(150, 350);
    ctx.stroke();

    ctx.beginPath();
    ctx.moveTo(300, 120);
    ctx.lineTo(450, 180);
    ctx.stroke();

    ctx.lineWidth = 1;
    ctx.beginPath();
    ctx.moveTo(180, 180);
    ctx.lineTo(180, 220);
    ctx.stroke();

    ctx.beginPath();
    ctx.moveTo(220, 180);
    ctx.lineTo(220, 220);
    ctx.stroke();
}
```

```ts
export async function floodFill(
    ctx: CanvasRenderingContext2D,
    x: number,
    y: number,
    algo: FloodFillAlgorithm,
): Promise<FloodFillStats> {
    const imageData = ctx.getImageData(x, y, 1, 1);
    const data = imageData.data;
    const r = data[0] ?? 0;
    const g = data[1] ?? 0;
    const b = data[2] ?? 0;
    const a = data[3] ?? 0;
    const targetColor: [number, number, number, number] = [r, g, b, a];

    const colorMap: Record<FloodFillAlgorithm, [number, number, number, number]> = {
        [FloodFillAlgorithm.SIMPLE_4]: [255, 0, 0, 255], // Синий
        [FloodFillAlgorithm.SIMPLE_8]: [255, 0, 0, 255], // Красный
        [FloodFillAlgorithm.SCANLINE_4]: [0, 255, 0, 255], // Зелёный
        [FloodFillAlgorithm.SCANLINE_8]: [255, 255, 0, 255], // Жёлтый
    };

    const replacementColor = colorMap[algo];

    // Выполняем заливку в зависимости от алгоритма
    switch (algo) {
        case FloodFillAlgorithm.SIMPLE_4:
            return await simpleFloodFillInternal(ctx, x, y, targetColor, replacementColor, 4);
        case FloodFillAlgorithm.SIMPLE_8:
            return await simpleFloodFillInternal(ctx, x, y, targetColor, replacementColor, 8);
        case FloodFillAlgorithm.SCANLINE_4:
            return await scanlineFloodFillInternal(ctx, x, y, targetColor, replacementColor, 4);
        case FloodFillAlgorithm.SCANLINE_8:
            return await scanlineFloodFillInternal(ctx, x, y, targetColor, replacementColor, 8);
    }
}
```