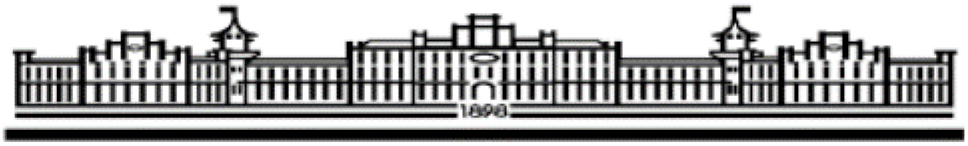


Національний технічний університет України
«Київський політехнічний інститут»

Факультет Прикладної Математики
Кафедра Системного Програмування і Спеціалізованих Комп'ютерних Систем

ПРАКТИЧНА РОБОТА №3

з дисципліни

«Комп'ютерна Графіка»

ТЕМА: «Алгоритми заливки заданих контурів (областей)»

Група: КВ-11

Виконав: Брюханов О.

Оцінка: ___

Київ – 2023

Теоретичні відомості

В цій лабораторній роботі досліджується швидкодія чотирьох наступних алгоритмів:

Довільні контури на довільному зображенні	Заливка методом повені (Flood Fill)
	Модифікована заливка методом повені (Boundary Fill)
	Заливка зправа наліво (Span Fill)
Довільні контури по точкам	Scanline Fill

Перші три алгоритми реалізовані в ітеративному варіанті, а не рекурсивному, з метою запобігання ситуації з переповненням стеку на великих зображеннях.

Flood Fill

Один із розповсюджених алгоритмів заливки, якій працює наступним чином: обирається піксель з якого починається заливка, цей піксель закрашується та його колір зберігається. Поітм усі пікселі, які знаходяться поряд (у цій лабораторній роботі додаються 4 пікселі з кожної сторони, «діагональні сусіди» не враховуються), додаються до черги. Алгоритм послідовно бере з черги піксель та, якщо його колір співпадає з кольором, який був у першого залитого пікселя, закрашує його та додає пікселі, що поряд, до черги. В решті нових пікселів для додавання не залишиться та алгоритм зупиниться через пусту чергу. Коли таке відбувається, можна сказати, що заливка була виконана успішно.

Boundary Fill

Цей алгоритм є модифікованою версією попереднього. У попередньому критерієм закрашування була відповідність кольору пікселя, а у цього критеріями виступають невідповідності кольору пікселя кольорам заливки та «границі». Таким чином, цей алгоритм буде заливати усі пікселі, що не належать границі, якою може виступати, наприклад, чорна лінія (у такому випадку кольор границі буде також чорний).

Span Fill

Цей алгоритм заливає лінію, в який знаходиться початковий піксель та визначає межі, де закінчується ця лінія, що має бути того ж самого кольору, що і початковий піксель. Потім алгоритм сканує по межам лінію вище та нижче. Знайдені пікселі додаються в чергу, з якої алгоритм бере пікселі для наступної ітерації. Критерій зупинки – черга стала пустою.

Scanline Fill

На відмінність від інших алгоритмів, що працюють в основному із пікселями на певному зображенні, цей алгоритм знаходить пікселі, які знаходились би у заданому многокутнику. Цей алгоритм, відповідно, не зможе залити довільний контур на вже існуючому зображенні, але може додати новий многокутник на нього, якщо дати йому координати усіх вершин того многокутника. Цей алгоритм має бути швидший за усі інші,

хоча й має суттєві обмеження. Цей алгоритм будує сторони многокутника по вершинам та потім проводить пряму по всій області, у якій визначений многкутник (умовний прямокутник, в який вписаний заданий вершинами многокутник), та знаходить перетини сторін із прямою, після чого додає собі в список координати перетинів. Між кординатами перетинів потім заливаються усі пікселі. Таким чином, многокутник заливається «построчно».

Програмна реалізація алгоритмів

Алгоритми реалізовані як класи в мові програмування Java, де їх ітераційна частина винесена в окремий метод, який викликається «планувальником заливки», це зроблено для можливості коригування швидкості заливки, яку було додано до ЛР з міркувань полегшення пошуку помилок. Більшість алгоритмів наслідуються від базового класу, який має наступну допоміжну функцію, яка повертає `true`, якщо по переданим в неї кординатам знаходиться піксель заданого кольору:

```
protected boolean pixelExists(int x, int y, Color color) {
    boolean exists = y >= 0 && y < mImage.getHeight() && x >= 0
            && x < mImage.getWidth();

    if(color != null)
        exists = exists && mImage.getRGB(x, y) == color.getRGB();

    return exists;
}
```

Наведений нижче код є неповним та включає лише «важливі» частини з визначеням змінних, коду ініціалізації та заливки. Повний код можна продивитись за посиланням: https://github.com/cithis/cg-labs

Flood Fill

```
public class FloodFill extends FillAlgorithm {
    protected Color mStartColor;
    protected LinkedBlockingQueue<Pixel> mPixels;

    public FloodFill(BufferedImage image, int x, int y, Color color) {
        super(image, x, y, color);
        mStartColor = new Color(mImage.getRGB(x, y));
        mPixels     = new LinkedBlockingQueue<>();

        mPixels.offer(new Pixel(x, y));
    }

    @Override
    public FillResult step() {
        if (mPixels.isEmpty())
            return FillResult.FINISHED;

        Pixel currentPixel = mPixels.poll();
        int x = currentPixel.x, y = currentPixel.y;
        if (pixelExists(x, y, mStartColor)) {
            // pixel matches start color and should be colored
            mImage.setRGB(x, y, mColor.getRGB());

            if (pixelExists(x - 1, y, null))
                mPixels.offer(new Pixel(x - 1, y));
            if (pixelExists(x + 1, y, null))
```

```
                mPixels.offer(new Pixel(x + 1, y));
            if (pixelExists(x, y - 1, null))
                mPixels.offer(new Pixel(x, y - 1));
            if (pixelExists(x, y + 1, null))
                mPixels.offer(new Pixel(x, y + 1));
        }

        return FillResult.CONTINUE;
    }

    // ...
}
```

Boundary Fill

```
public class BoundaryFloodFill extends FillAlgorithm {
    protected Color mBoundaryColor;
    protected LinkedBlockingQueue<Pixel> mPixels;

    public BoundaryFloodFill(BufferedImage image, int x, int y, Color color,
            Color boundary) {
        super(image, x, y, color);
        mBoundaryColor = boundary;
        mPixels        = new LinkedBlockingQueue<>();

        mPixels.offer(new Pixel(x, y));
    }

    @Override
    public FillResult step() {
        if (mPixels.isEmpty())
            return FillResult.FINISHED;

        Pixel currentPixel = mPixels.poll();
        int x = currentPixel.x, y = currentPixel.y;
        if (!pixelExists(x, y, mColor) && !pixelExists(x, y, mBoundaryColor)) {
            // pixel isn't part of border and is not already coloured
            mImage.setRGB(x, y, mColor.getRGB());

            if (pixelExists(x - 1, y, null))
                mPixels.offer(new Pixel(x - 1, y));
            if (pixelExists(x + 1, y, null))
                mPixels.offer(new Pixel(x + 1, y));
            if (pixelExists(x, y - 1, null))
                mPixels.offer(new Pixel(x, y - 1));
            if (pixelExists(x, y + 1, null))
                mPixels.offer(new Pixel(x, y + 1));
        }

        return FillResult.CONTINUE;
    }
    // ...
}
```

Span Fill

```
public class SpanFill extends FillAlgorithm {
    protected Color mStartColor;
    protected ArrayDeque<Pixel> mPixels;

    public SpanFill(BufferedImage image, int x, int y, Color color) {
        super(image, x, y, color);
        mPixels = new ArrayDeque<>();
        mStartColor = new Color(mImage.getRGB(x, y));

        mPixels.offerFirst(new Pixel(x, y));
    }

    @Override
    public FillResult step() {
        if (mPixels.isEmpty())
            return FillResult.FINISHED;

        Pixel currentPixel = mPixels.pollFirst();
        int x = currentPixel.x, lx = x, rx = x + 1, y = currentPixel.y;
        while (pixelExists(lx--, y, mStartColor))
            mImage.setRGB(lx + 1, y, mColor.getRGB());

        while (pixelExists(rx++, y, mStartColor))
            mImage.setRGB(rx - 1, y, mColor.getRGB());

        scan(lx, rx - 1, y + 1);
        scan(lx, rx - 1, y - 1);

        return FillResult.CONTINUE;
    }

    protected void scan(int lx, int rx, int y) {
        for (int i = lx; i < rx; i++)
            if (pixelExists(i, y, mStartColor))
                mPixels.offerFirst(new Pixel(i, y));
    }

    // ...
}
```

Scanline Fill

```
public class ScanlineFill implements IFillAlgorithm {
    protected BufferedImage mImage;
    protected ArrayList<PolygonEdge> mEdges;
    protected int mMinY, mMaxY, mY;
    protected Color mColor;

    public ScanlineFill(BufferedImage image, Polygon poly, Color color) {
        mImage = image;
        mColor = color;
```

```
        mEdges = new ArrayList<>();

        prepareEdges(poly);
    }

    private void prepareEdges(Polygon poly) {
        int count = poly.getVerticesCount();
        for (int i = 0; i < count; i++) {
            Pixel a = poly.getVertex(i), b = poly.getVertex((i + 1) % count);
            int ax  = a.x, bx = b.x, ay = a.y, by = b.y;
            if (ay > by) {
                int tx = ax, ty = ay;
                ax = bx;
                bx = tx;
                ay = by;
                by = ty;
            }

            if (ay == by)
                continue;

            double slope     = (double) (bx - ax) / (by - ay);
            PolygonEdge edge = new PolygonEdge(ay, by, ax, slope);
            mEdges.add(edge);
        }

        mEdges.sort(Comparator.comparingInt((var e) -> e.yMin));
        mMinY = mEdges.get(0).yMin;
        mMaxY = mEdges.get(mEdges.size() - 1).yMax;
        mY    = mMinY;
    }

    public FillResult step() {
        if (mY > mMaxY)
            return FillResult.FINISHED;

        ArrayList<Integer> intersections = new ArrayList<>();
        for (var edge : mEdges) {
            if (mY < edge.yMin || mY >= edge.yMax)
                continue;

            int x = (int) (edge.x + edge.slope * (mY - edge.yMin));
            intersections.add(x);
        }

        Collections.sort(intersections);
        for (int i = 0; i < intersections.size(); i += 2) {
            int ax = intersections.get(i);
            int bx = intersections.get(i + 1);
            for (int j = ax; j <= bx; j++)
                mImage.setRGB(j, mY, mColor.getRGB());
        }

        mY++;
```

```
        return FillResult.CONTINUE;
    }
}
```

Методика тестування

Для тестування перших трьох алгоритмів буде використане монохромне зображення з чітко виділеними контурами для легкого порівняння.

Рис. 1 – Зображення «elf_monochrome.png»

Заливатися буде ділянка зображення, що містить лице персонажа. Кольором заливки буде (`#f3eae1`), а кольором границі чорний. Для перевірки правильності роботи алгоритмів, виконаємо заливку в програмі для малювання «KolourPaint» для подальшого порівняння:

Рис. 2 – Зображення з правильно виконанною заливкою

Варто звернути увагу на те, що алгоритм Scanline Fill, як нам відомо, не може заливати довільні зображення, а лише многокутники за вершинами. Тому, у порівнянні швидкодії на цьому зображенні він брати участі не буде. Натомість буде проведене окреме

порівняння швидкості заливок на многокутниках: Scanline Fill отримає на вхід вершини, а решта алгоритмів – зображення многокутника:

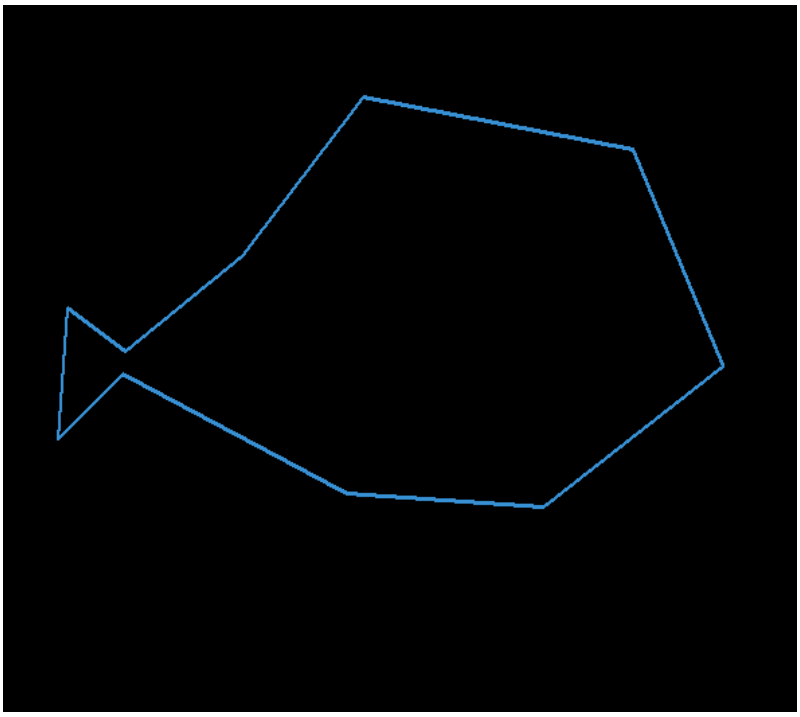
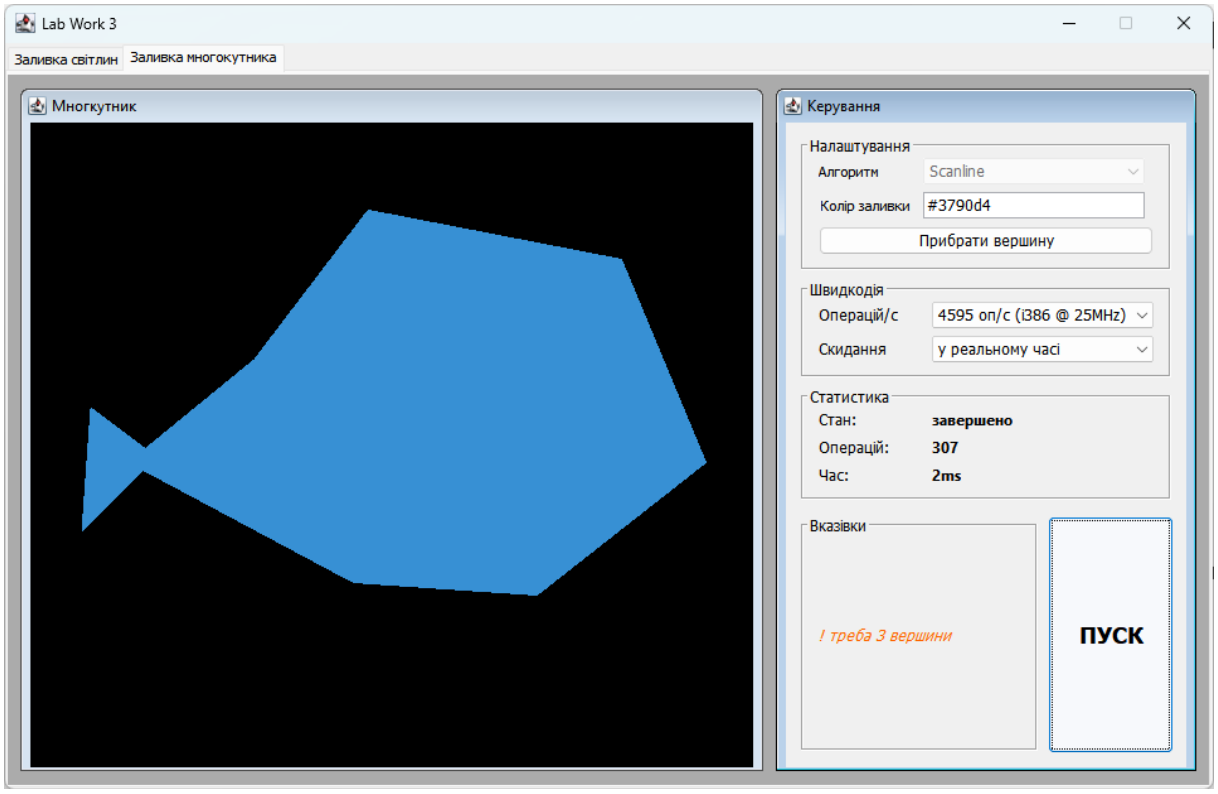
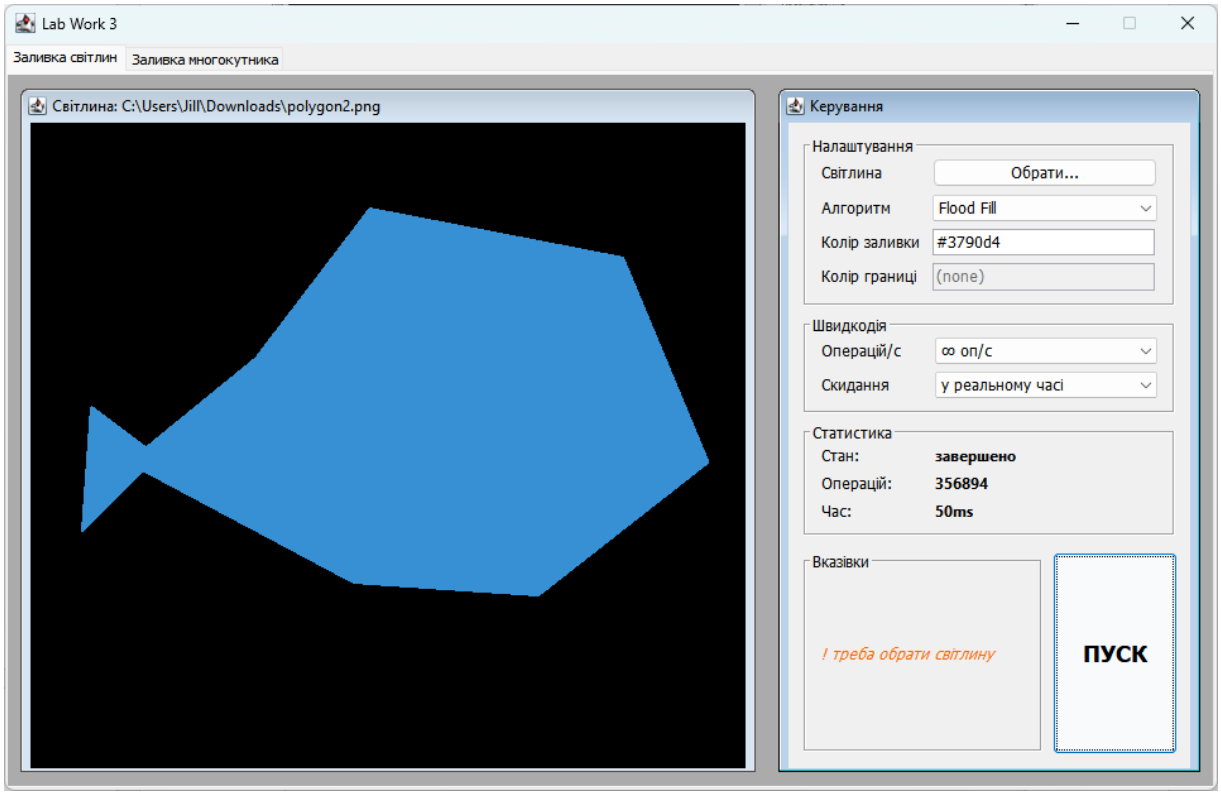
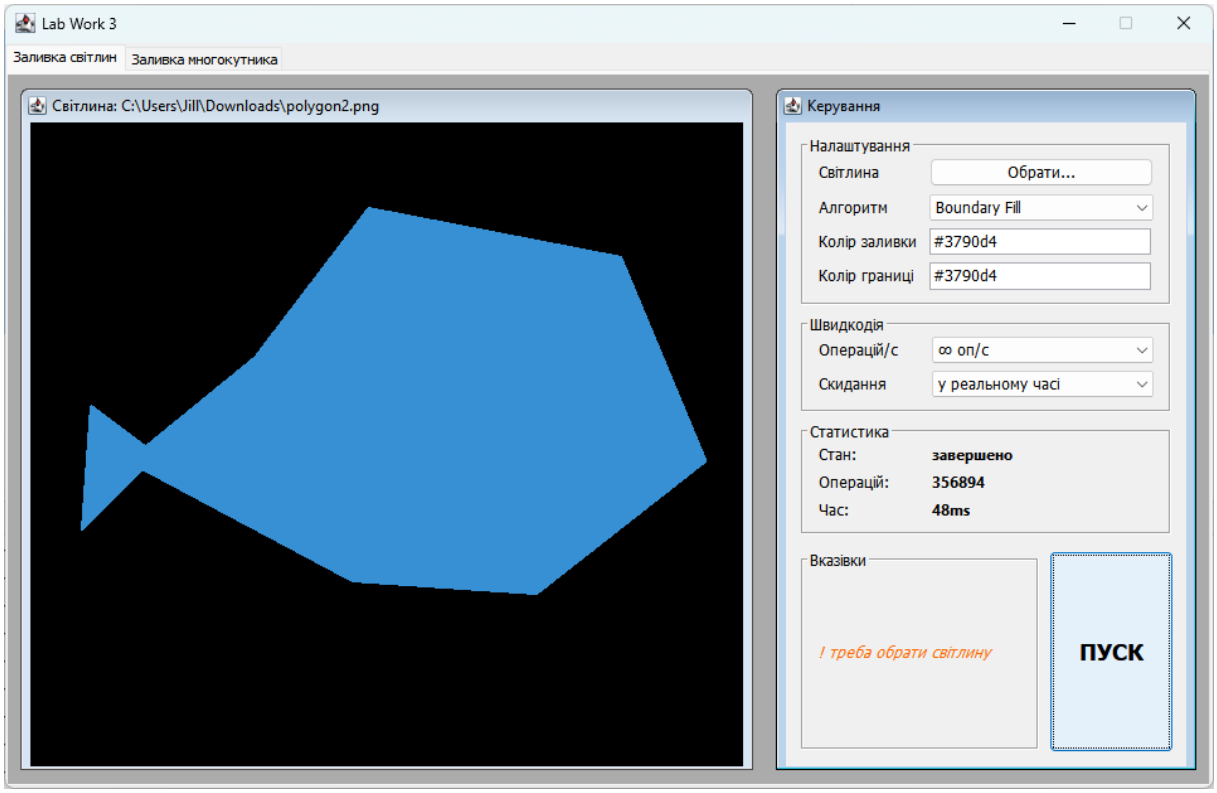


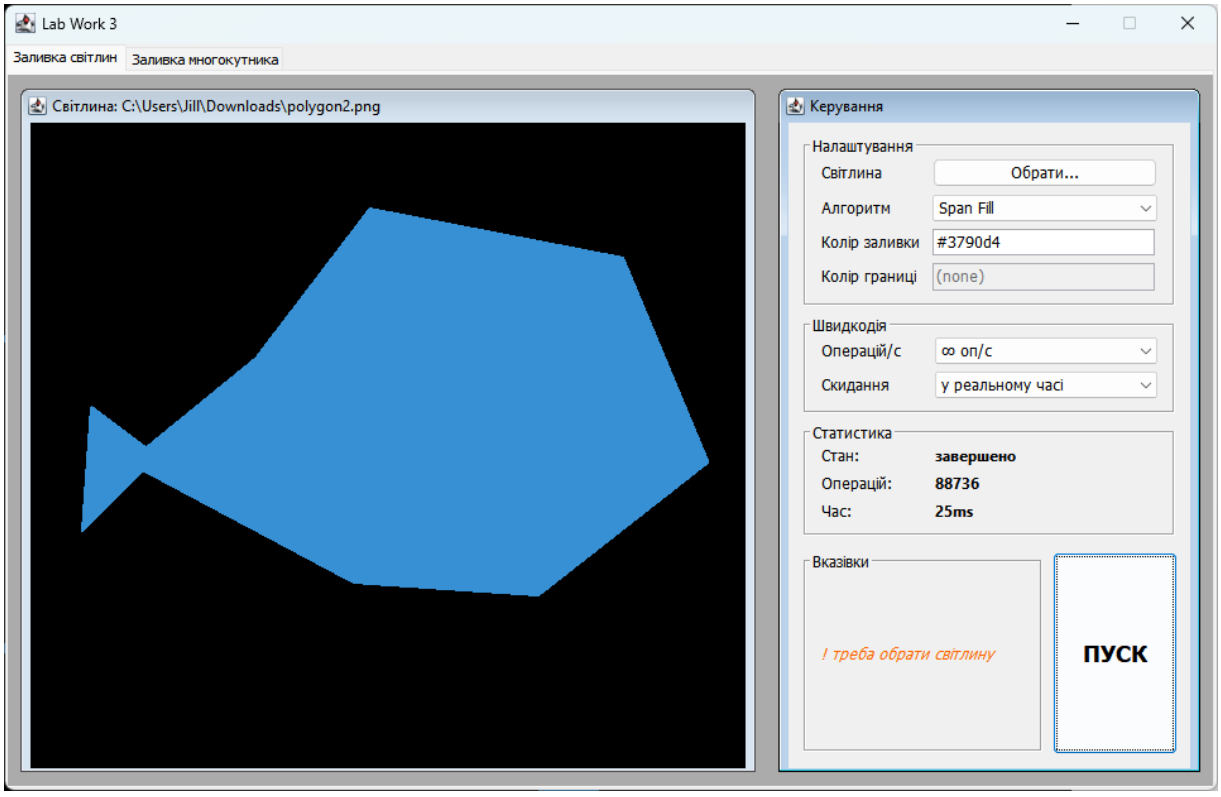
Рис. 3 – Многокутник для порівняння

Тестування: многокутник



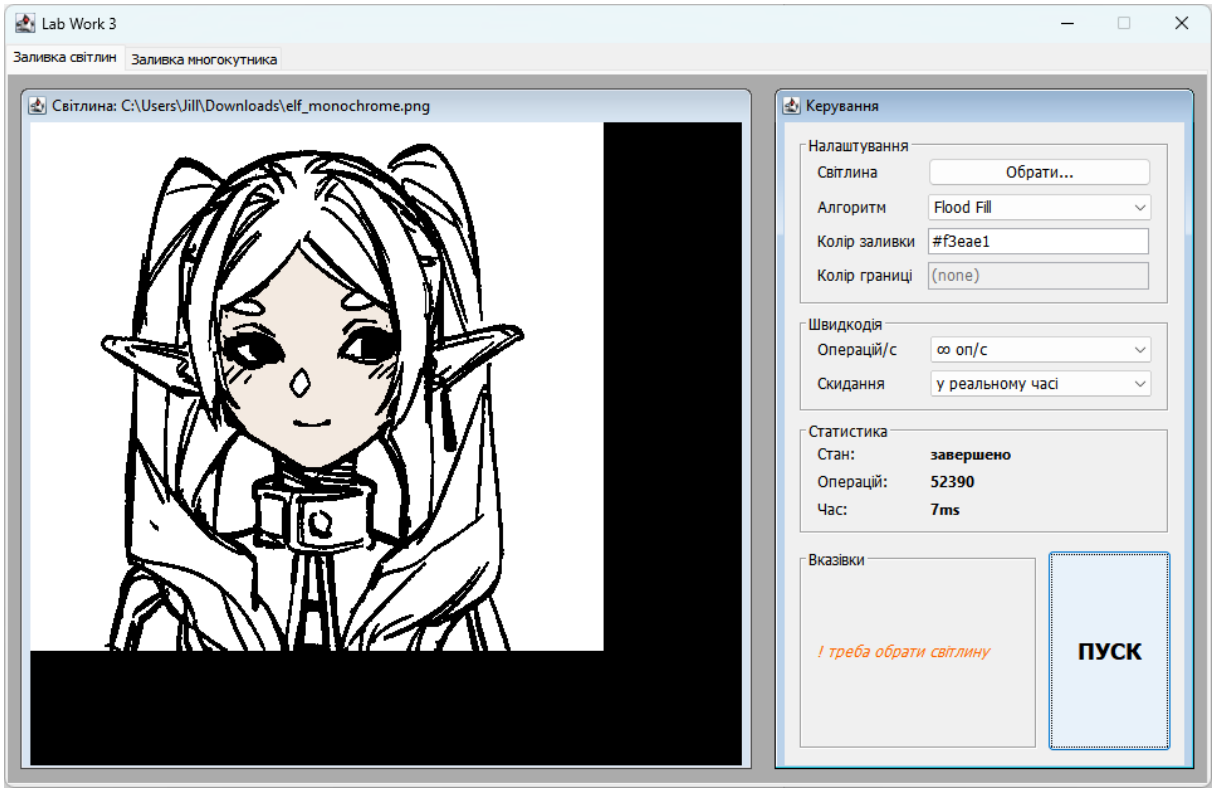


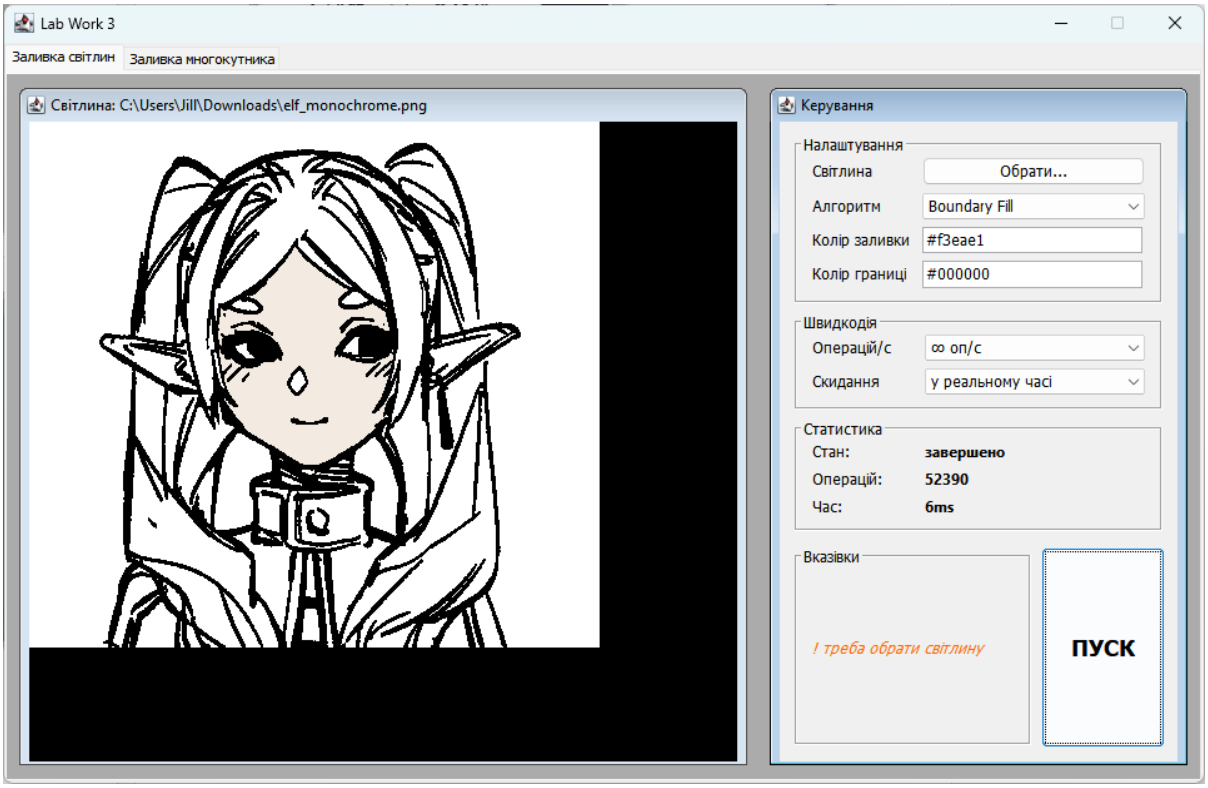


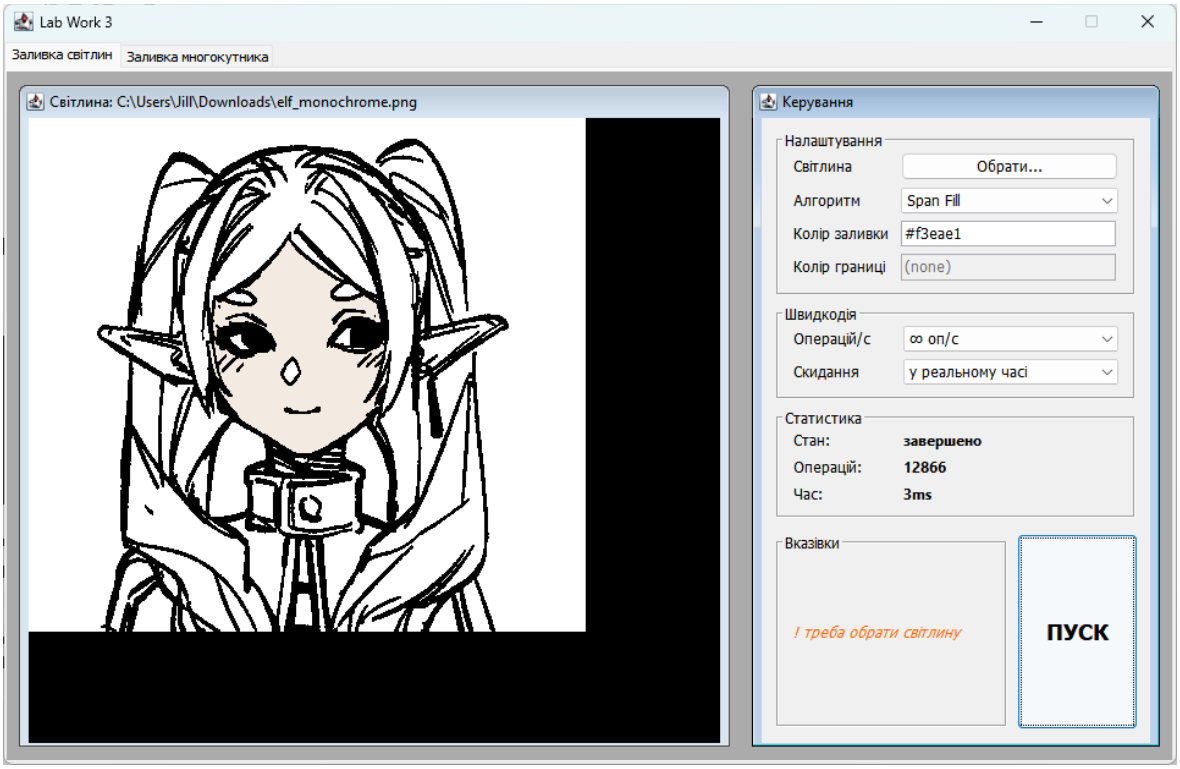


Результати тестування			
Scanline	Flood	Boundary	Span
2ms	50ms	48ms	25ms

Тестування: зображення

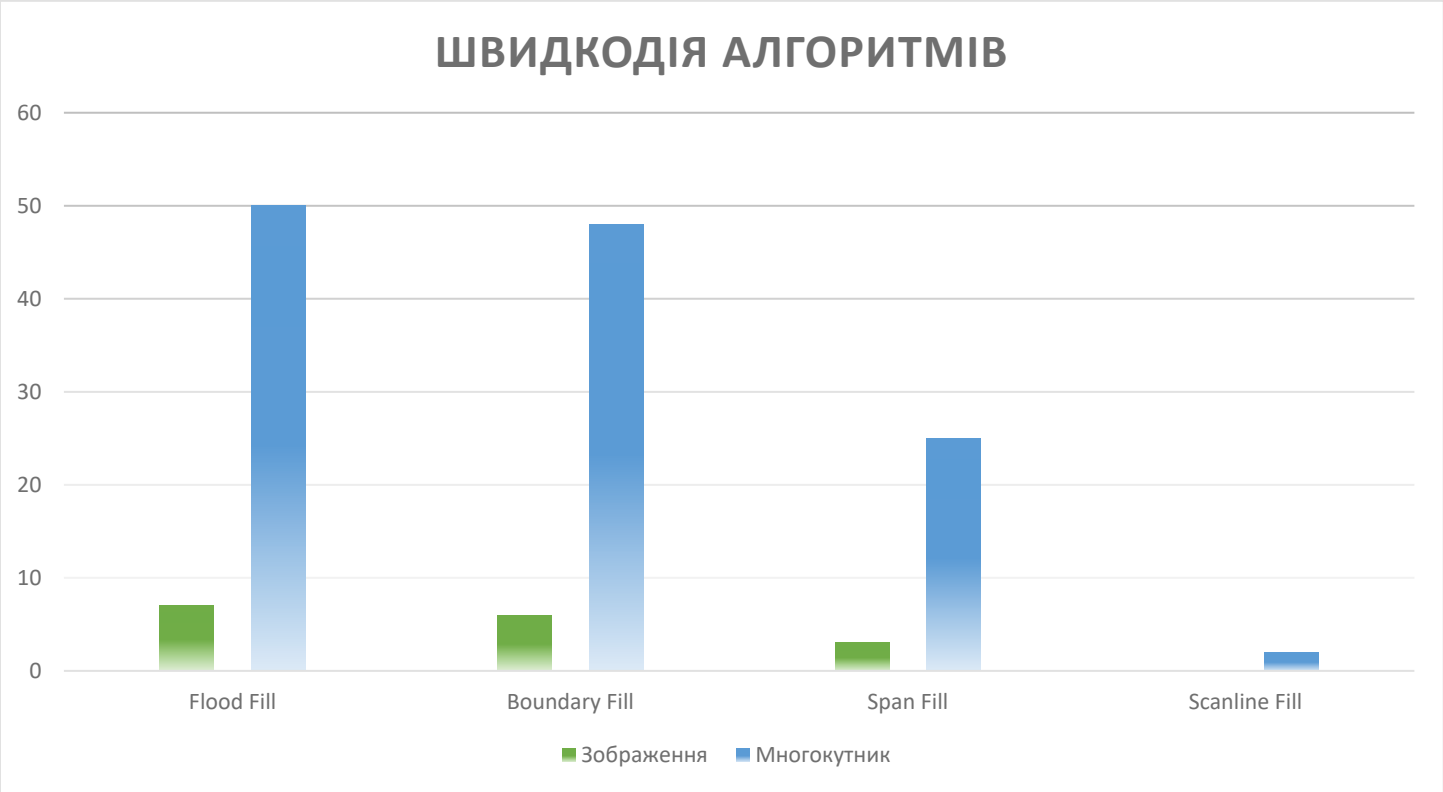






Результати тестування		
Flood	Boundary	Span
7ms	6ms	3ms

Результати



- Алгоритми Flood Fill та Boundary Fill майже не відрізняються за швидкістю.
- Span Fill значно швидший за усі алгоритми, що можуть заливати зображення.
- Scanline Fill швидший за Span Fill більше ніж у 10 разів при заливці многокутника.

Отже, найшвидшим алгоритмом для заливки зображень є Span Fill, але якщо ми заливаємо многокутник та нам відомі координати його вершин, то використання Scanline Fill буде більш доречним через очевидну різницю у швидкості на користь Scanline Fill.