Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Информационных технологий и управления

Кафедра Интеллектуальных информационных технологий

**ОТЧЁТ**

по дисциплине «Графический интерфейс интеллектуальных систем»

Лабораторная работа №4

Тема: Геометрические преобразования

Выполнили: Демидовец Д.В.

Козырев Д.А.

гр. 221703

Проверил: Сальников Д. А.

Минск 2025

**Задание к лабораторной работе**

Разработать графическую программу, выполняющую следующие геометрические преобразования над трехмерным объектом: поворот, скалирование, отображение, перспектива. В программе должно быть предусмотрено считывание координат 3D объекта из текстового файла, обработка клавиатуры и выполнение геометрических преобразований в зависимости от нажатых клавиш. Все преобразования следует производить с использованием матричного аппарата и представления координат в однородных координатах.

**Ход работы**

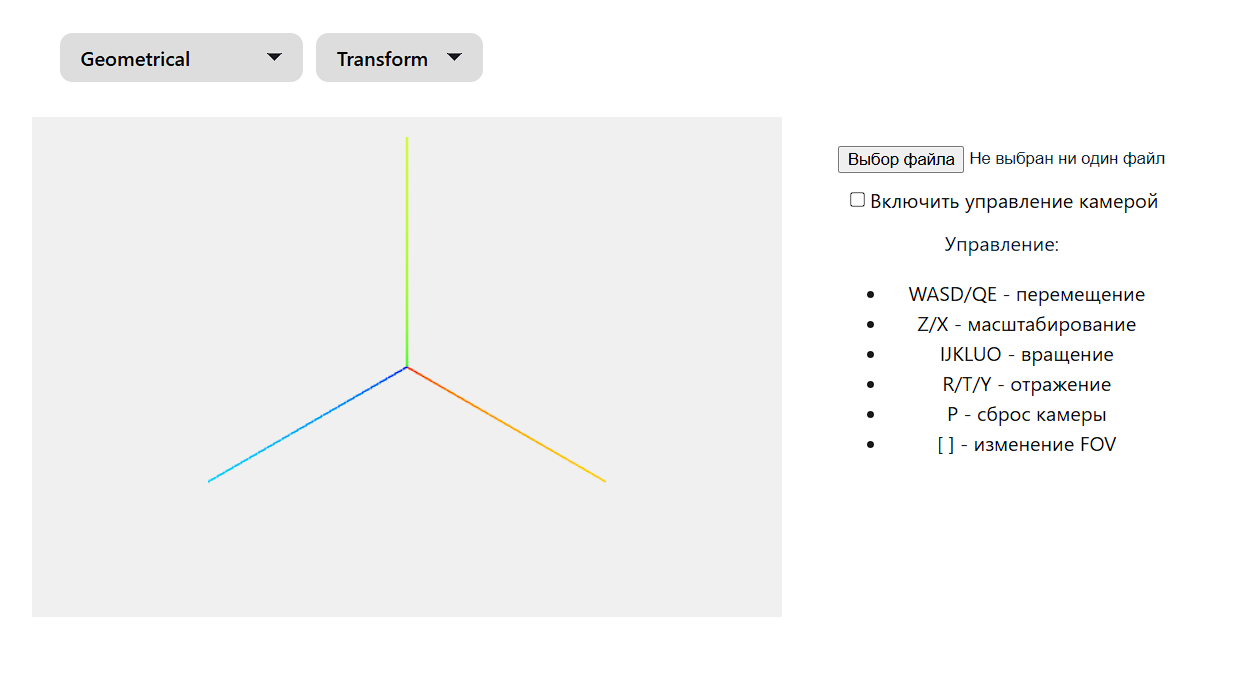
1. Изучение теоретических основ алгоритмов.
2. Реализация алгоритмов преобразований в программной среде.
3. Проверка отображения в редакторе.

**Описание преобразований**

Для выполнения геометрических преобразований использовались матрицы преобразования, где каждая операция (поворот, скалирование, отображение и перспектива) представлена отдельной матрицей. Далее каждая вершина объекта умножалась на соответствующую матрицу.

* Поворот: для поворота объекта использовалась матрица поворота вокруг осей X, Y и Z. Это позволило вращать объект в трехмерном пространстве, изменяя его ориентацию.
* Скалирование: для изменения размеров объекта использовалась матрица масштабирования. Программа позволяла изменять масштаб объекта по осям.
* Отображение: для проекции объекта на экран использовалась матрица проекции. Она преобразует 3D координаты в 2D, позволяя отобразить объект на экране в двумерной системе координат.
* Перспектива: для создания эффекта перспективы использовалась матрица перспективы. Эта матрица искажает координаты объекта в зависимости от его удаленности от камеры, что позволяет имитировать перспективу на экране.

Для обработки клавиш была реализована система, которая отслеживает нажатия клавиш на клавиатуре и в зависимости от нажатой клавиши выполняет соответствующее преобразование.

****

**Листинг кода**

*// Обработка клавиатуры*

  useEffect(() => {

*const* handleKeyDown = (e) => {

      if (!object) *return*;

*const* m = new THREE.Matrix4();

*const* angle = Math.PI / 16;

*const* step = 0.2;

      switch(e.key.toLowerCase()) {

*// Перемещение*

        case 'w': m.makeTranslation(0, step, 0); break;

        case 's': m.makeTranslation(0, -step, 0); break;

        case 'a': m.makeTranslation(-step, 0, 0); break;

        case 'd': m.makeTranslation(step, 0, 0); break;

        case 'q': m.makeTranslation(0, 0, step); break;

        case 'e': m.makeTranslation(0, 0, -step); break;

*// Масштабирование*

        case 'z': m.makeScale(1.1, 1.1, 1.1); break;

        case 'x': m.makeScale(0.9, 0.9, 0.9); break;

*// Поворот*

        case 'i': m.makeRotationX(angle); break;

        case 'k': m.makeRotationX(-angle); break;

        case 'j': m.makeRotationY(angle); break;

        case 'l': m.makeRotationY(-angle); break;

        case 'u': m.makeRotationZ(angle); break;

        case 'o': m.makeRotationZ(-angle); break;

*// Отражение*

        case 'r': m.makeScale(-1, 1, 1); break;

        case 't': m.makeScale(1, -1, 1); break;

        case 'y': m.makeScale(1, 1, -1); break;

*// Перспектива*

        case 'p':

          camera.current.position.set(5, 5, 5);

          camera.current.lookAt(0, 0, 0);

          break;

        case '[':

          camera.current.fov -= 5;

          camera.current.updateProjectionMatrix();

          break;

        case ']':

          camera.current.fov += 5;

          camera.current.updateProjectionMatrix();

          break;

      }

**Вывод**

В ходе выполнения лабораторной работы была дополнена программа для выполнения геометрических преобразований в 3D пространстве с использованием матричных операций. Все преобразования работают корректно, обеспечивая возможность поворота, масштабирования и отображения объектов с учетом перспективы. Важно отметить, что использование матричного аппарата и однородных координат позволяет эффективно и точно управлять преобразованиями.

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Информационных технологий и управления

Кафедра Интеллектуальных информационных технологий

**ОТЧЁТ**

по дисциплине «Графический интерфейс интеллектуальных систем»

Лабораторная работа №5

Тема: Предварительная обработка полигонов

Выполнили: Демидовец Д.В.

Козырев Д.А.

гр. 221703

Проверил: Сальников Д. А.

Минск 2025

Задание

Разработать элементарный графический редактор, реализующий построение полигонов. Реализованная программа должна уметь проверять полигон на выпуклость, находить его внутренние нормали. Программа должна выполнять построения выпуклых оболочек методом обхода Грэхема и методом Джарвиса. Выбор метода задается из пункта меню и должен быть доступен через панель инструментов «Построение полигонов». Графический редактор должен позволять рисовать линии первого порядка (лабораторная работа 2) и определять точки пересечения отрезка со стороной полигона, также программа должна определять принадлежность введенной точки полигону.

Ход работы

1. Изучение теоретических основ алгоритмов обработки и построения полигонов.
2. Реализация алгоритмов в программной среде.
3. Проверка отображение в графическом редакторе.

Описание алгоритмов

*Построение полигонов.* Реализация построения полигонов выполнена с помощью последовательного ввода точек пользователем. Каждая новая точка соединяется с предыдущей линией, а замыкание полигона происходит по команде пользователя.

Для *определения выпуклости* полигона использовался следующий алгоритм:

1. Для каждой тройки последовательных вершин вычисляется векторное произведение.
2. Если все векторные произведения имеют одинаковый знак, полигон считается выпуклым.
3. Если знаки различаются, значит, полигон вогнутый.

*Нормали* вычислялись для каждого ребра полигона по следующему принципу:

1. Находился направляющий вектор ребра.
2. Перпендикулярный вектор к этому ребру (нормаль) вычислялся как (−dy,dx).
3. Направление нормали определялось так, чтобы она указывала внутрь полигона.

Программа реализует два алгоритма построения выпуклой оболочки:

**Метод обхода Грэхема:**

1. Выбирается точка с минимальной y-координатой (если несколько — с минимальной x-координатой).
2. Все остальные точки сортируются по полярному углу относительно этой точки.
3. Последовательно формируется выпуклая оболочка с использованием стека.

**Метод Джарвиса:**

1. Выбирается самая левая (или нижняя) точка.
2. Последовательно находятся точки, образующие наименьший угол с текущей стороной оболочки.
3. Процесс продолжается, пока не будет достигнута исходная точка.

Для нахождения пересечения отрезка со стороной полигона использовался алгоритм проверки пересечения двух отрезков с помощью параметрического уравнения:

P=A+t(B−A), где t рассчитывается так, чтобы удовлетворять условиям принадлежности обоим отрезкам.

Принадлежность точки проверялась методом ***чётности числа пересечений***:

1. Проводится луч вправо от проверяемой точки.
2. Подсчитывается количество пересечений луча со сторонами полигона.
3. Если число пересечений нечётное — точка внутри полигона, если чётное — снаружи.

**Листинг кода**

export function grahamScan(points) {

    if (points.length < 3) *return* points;

*// Шаг 1: Найти опорную точку с минимальными y и x*

*const* pivot = points.reduce((min, p) =>

        p.y < min.y || (p.y === min.y && p.x < min.x) ? p : min

    );

*// Шаг 2: Сортировка по полярному углу и расстоянию*

*const* sorted = [**...**points].sort((a, b) => {

*const* angleA = Math.atan2(a.y - pivot.y, a.x - pivot.x);

*const* angleB = Math.atan2(b.y - pivot.y, b.x - pivot.x);

        if (angleA !== angleB) *return* angleA - angleB;

*return* (a.x - pivot.x) \*\* 2 + (a.y - pivot.y) \*\* 2 -

               ((b.x - pivot.x) \*\* 2 + (b.y - pivot.y) \*\* 2);

    });

*// Шаг 3: Построение выпуклой оболочки*

*const* hull = [sorted[0], sorted[1]];

    for (*let* i = 2; i < sorted.length; i++) {

        while (hull.length > 1) {

*const* a = hull[hull.length - 2];

*const* b = hull[hull.length - 1];

*const* c = sorted[i];

*const* cross = (b.x - a.x) \* (c.y - b.y) - (b.y - a.y) \* (c.x - b.x);

*// Если поворот направо или коллинеарны, удаляем последнюю точку*

            if (cross <= 0) {

                hull.pop();

            } else {

                break;

            }

        }

        hull.push(sorted[i]);

    }

*return* hull;

}

*// Функция построения вогнутого полигона*

export function buildConcavePolygon(points) {

    if (points.length < 3) *return* [];

*return* [**...**points, points[0]]; *// Соединяем последнюю точку с первой*

}

export function lineIntersection(line1, line2) {

*const* x1 = line1.x1, y1 = line1.y1;

*const* x2 = line1.x2, y2 = line1.y2;

*const* x3 = line2.x1, y3 = line2.y1;

*const* x4 = line2.x2, y4 = line2.y2;

*const* denominator = (y4 - y3) \* (x2 - x1) - (x4 - x3) \* (y2 - y1);

    if (denominator === 0) *return* null;

*const* ua = ((x4 - x3) \* (y1 - y3) - (y4 - y3) \* (x1 - x3)) / denominator;

*const* ub = ((x2 - x1) \* (y1 - y3) - (y2 - y1) \* (x1 - x3)) / denominator;

    if (ua >= 0 && ua <= 1 && ub >= 0 && ub <= 1) {

*return* {

            x: x1 + ua \* (x2 - x1),

            y: y1 + ua \* (y2 - y1)

        };

    }

*return* null;

}

*// Алгоритм Джарвиса*

export function jarvisMarch(points) {

    if (points.length < 3) *return* points;

*const* hull = [];

*let* current = points.reduce((min, p) => p.x < min.x ? p : min);

    do {

        hull.push(current);

*let* next = points[0];

        for (*const* p of points) {

            if (p === current) continue;

*const* cross = (next.x - current.x) \* (p.y - current.y) -

                         (next.y - current.y) \* (p.x - current.x);

            if (next === current || cross > 0 ||

                (cross === 0 && distance(current, p) > distance(current, next))) {

                next = p;

            }

        }

        current = next;

    } while (current !== hull[0]);

*return* hull;

}

*// Проверка принадлежности точки полигону*

export function pointInPolygon(point, polygon) {

*let* inside = false;

    for (*let* i = 0, j = polygon.length - 1; i < polygon.length; j = i++) {

*const* xi = polygon[i].x, yi = polygon[i].y;

*const* xj = polygon[j].x, yj = polygon[j].y;

*const* intersect = ((yi > point.y) !== (yj > point.y)) &&

            (point.x < (xj - xi) \* (point.y - yi) / (yj - yi) + xi);

        if (intersect) inside = !inside;

    }

*return* inside;

}

*// Нахождение нормалей*

export function calculateNormals(polygon) {

*const* normals = [];

    for (*let* i = 0; i < polygon.length; i++) {

*const* p1 = polygon[i];

*const* p2 = polygon[(i + 1) % polygon.length];

*const* dx = p2.x - p1.x;

*const* dy = p2.y - p1.y;

*const* normal = { x: -dy, y: dx };

*const* length = Math.sqrt(normal.x\*\*2 + normal.y\*\*2);

        normal.x /= length;

        normal.y /= length;

*const* midPoint = { x: (p1.x + p2.x)/2, y: (p1.y + p2.y)/2 };

*const* testPoint = {

            x: midPoint.x + normal.x,

            y: midPoint.y + normal.y

        };

        if (!pointInPolygon(testPoint, polygon)) {

            normal.x \*= -1;

            normal.y \*= -1;

        }

        normals.push(normal);

    }

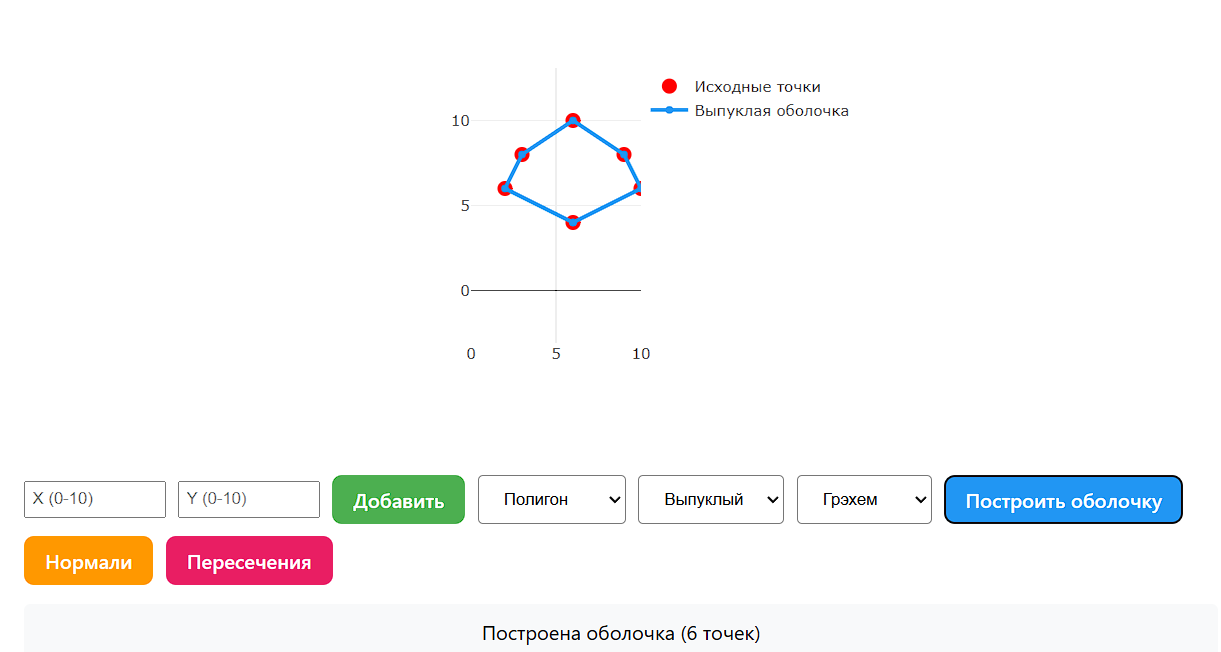
*return* normals;

}

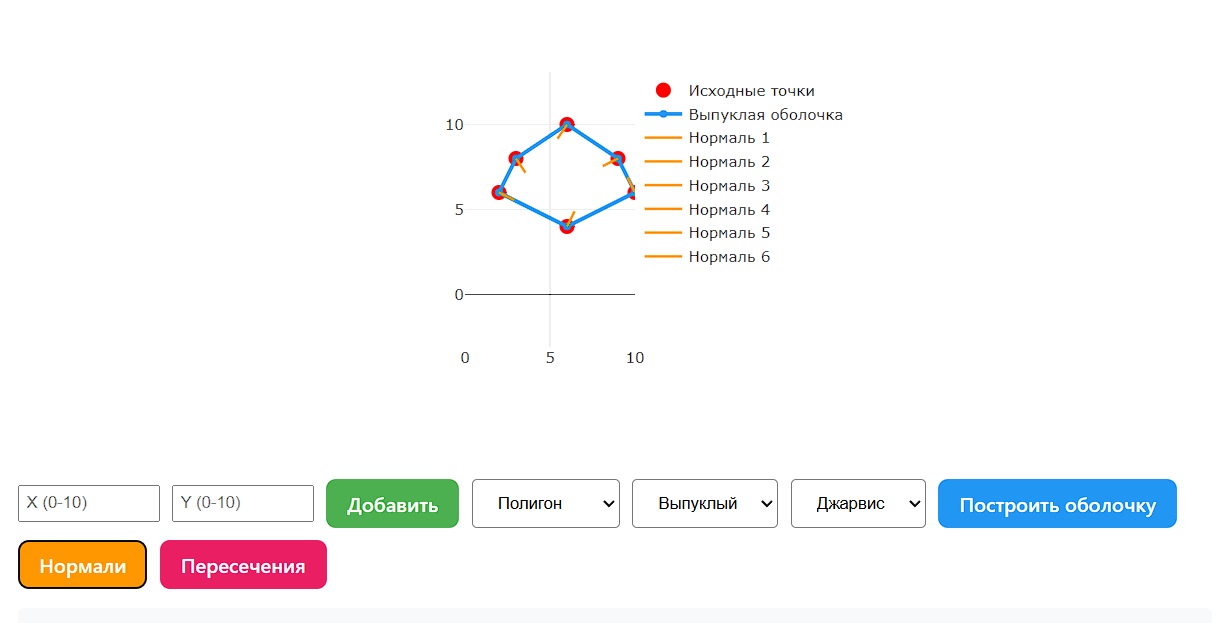
**Результаты работы**

Построение полигонов

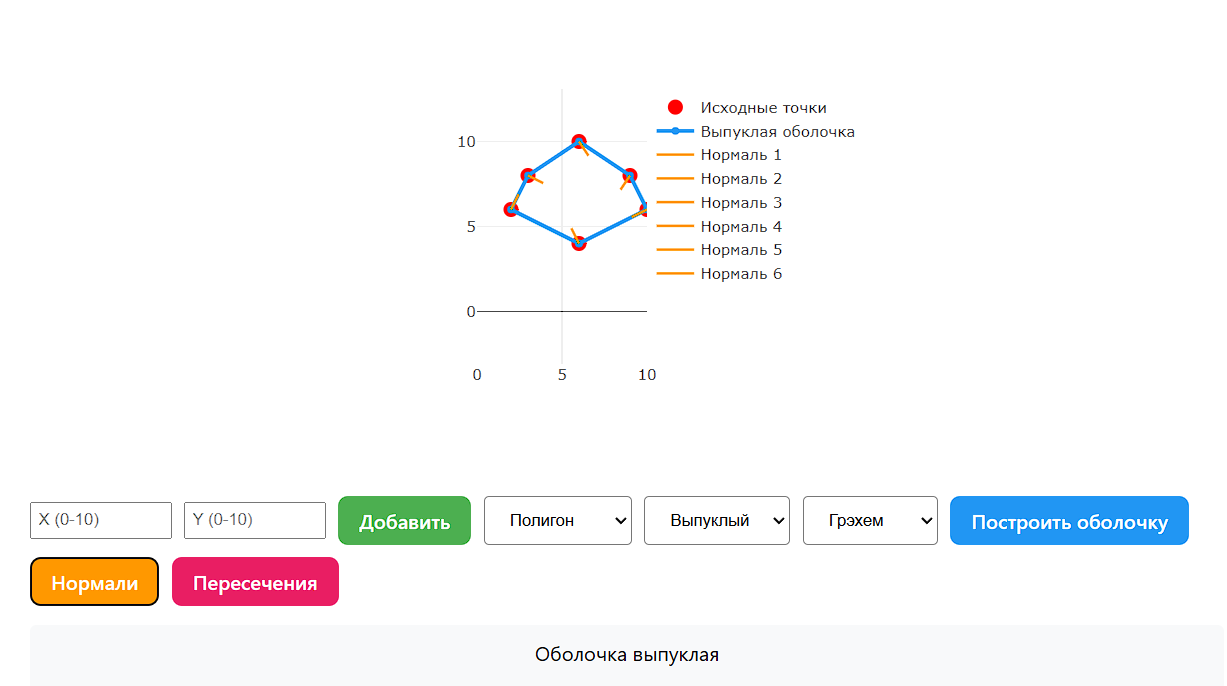
* Метод Грэхема



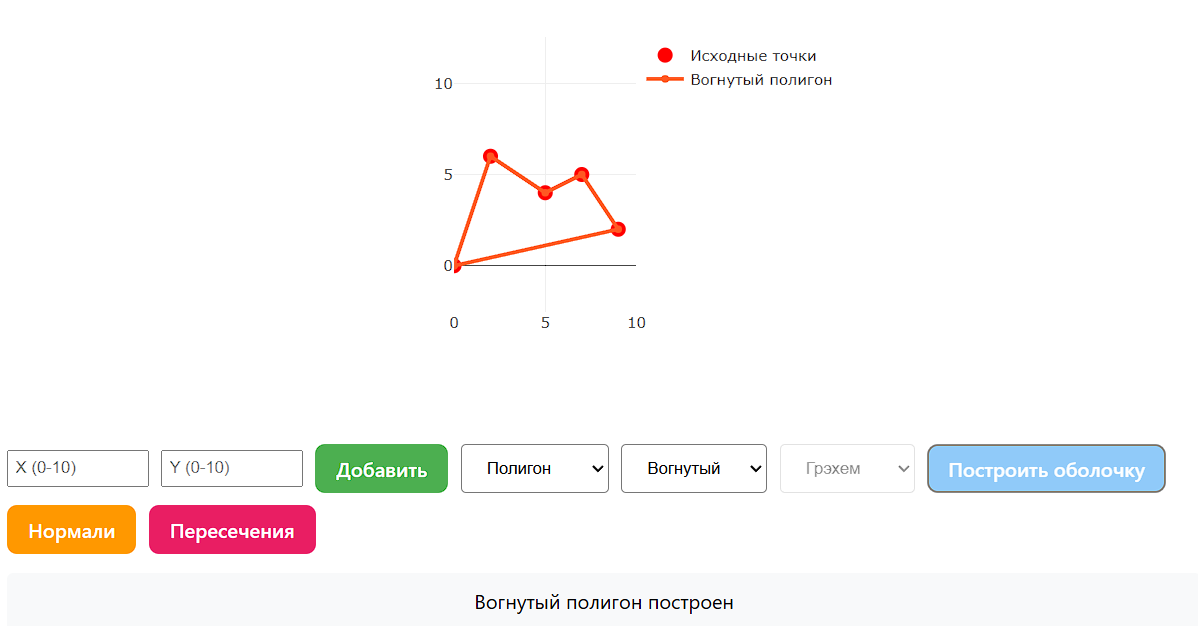
* Метод Джарвиса



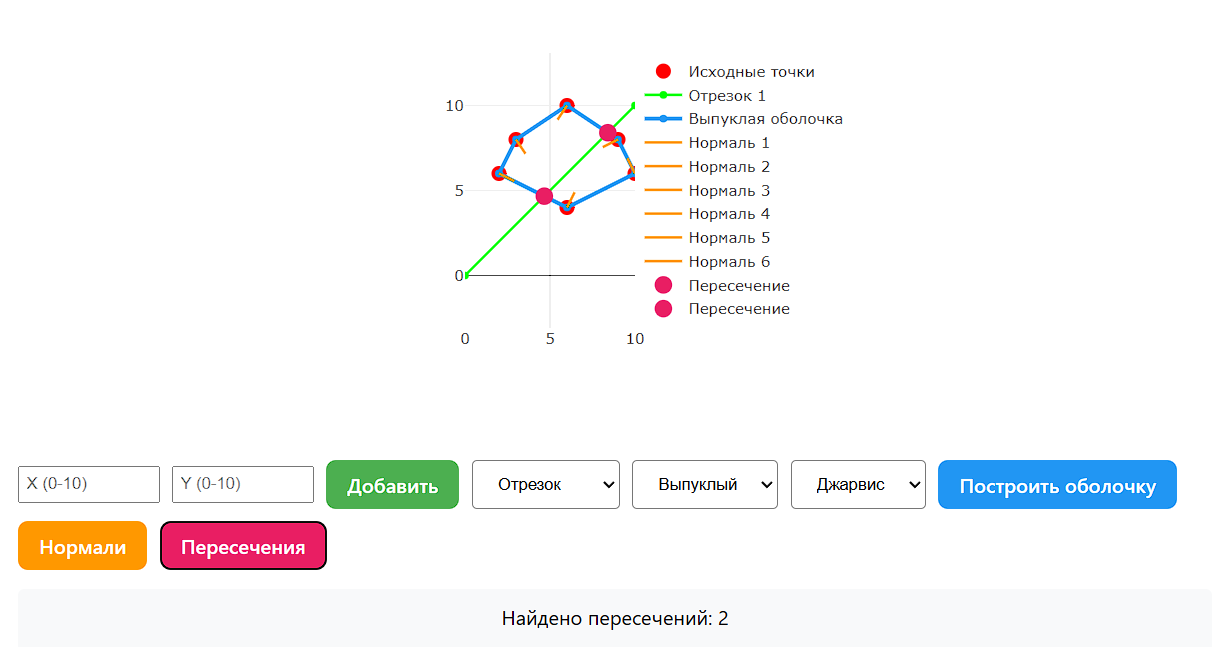
Проверка выпуклости полигона и нахождение внутренних нормалей



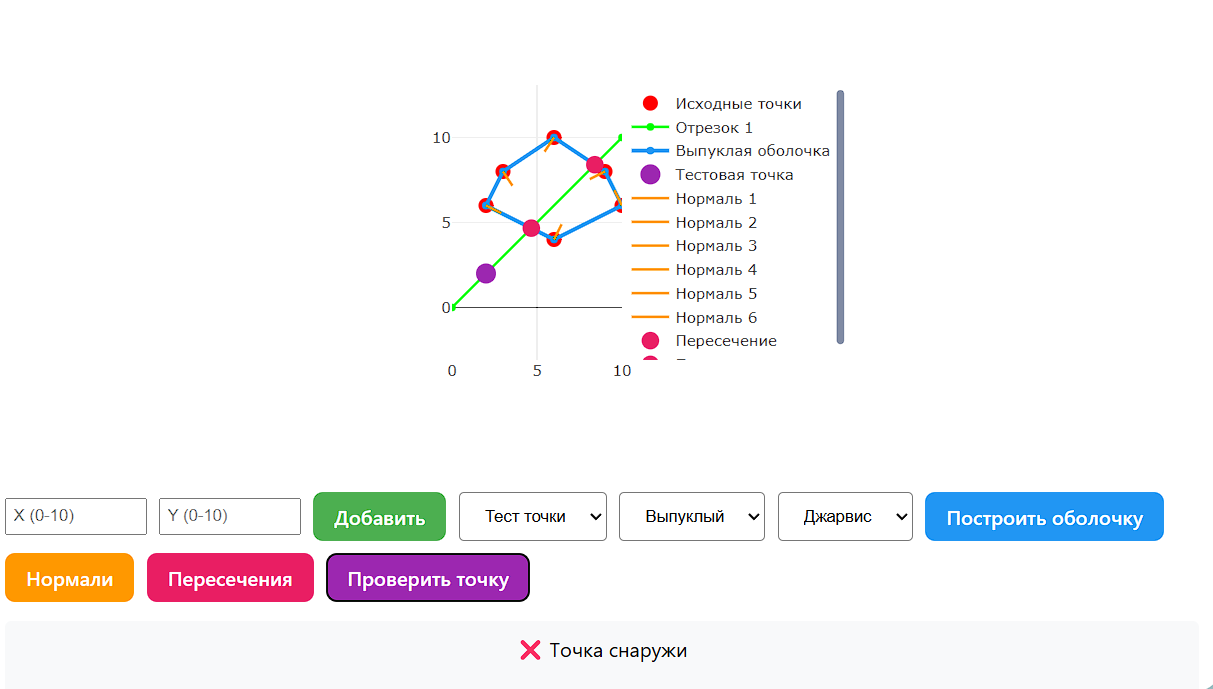
Построение вогнутой оболочки

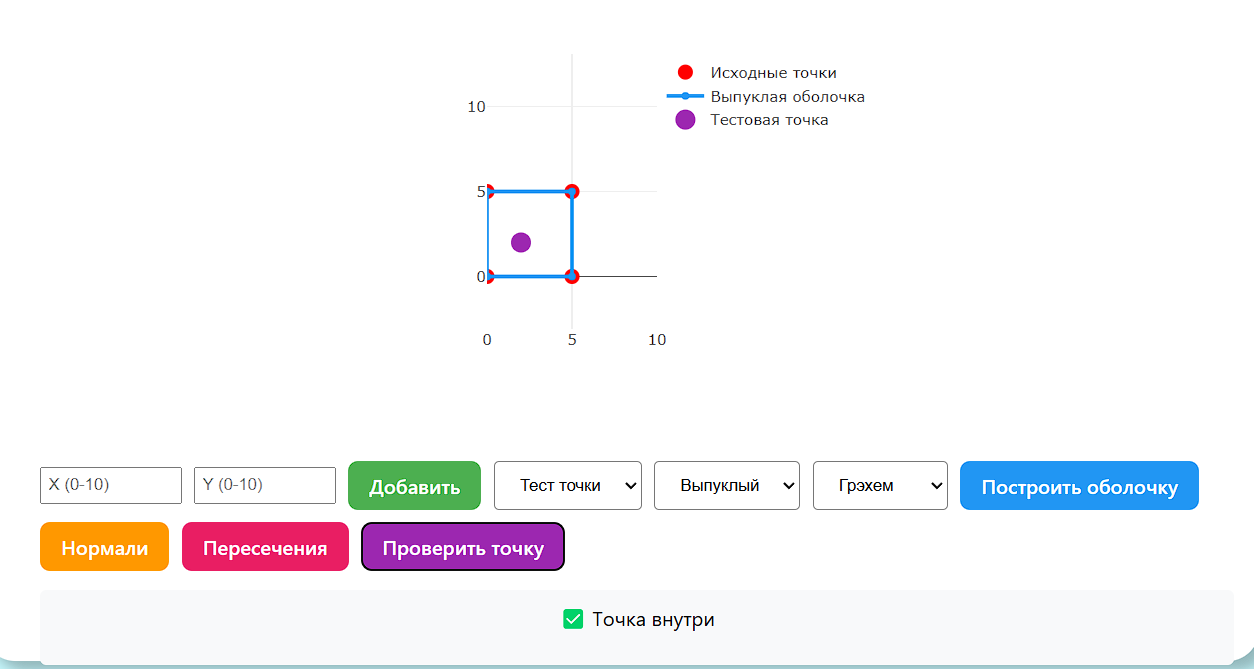


Определение пересечения отрезка со сторонами полигона



Проверку принадлежности точки (2;2) полигону





**Вывод**

В результате выполнения лабораторной работы был разработан графический редактор с возможностью построения и анализа полигонов. Реализованы методы проверки выпуклости, нахождения нормалей, построения выпуклой оболочки, а также алгоритмы определения пересечений и принадлежности точки. Работа позволила углубить знания о геометрических алгоритмах и их применении в компьютерной графике.

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Информационных технологий и управления

Кафедра Интеллектуальных информационных технологий

**ОТЧЁТ**

по дисциплине «Графический интерфейс интеллектуальных систем»

Лабораторная работа №6

Тема: Заполнение полигонов

Выполнили: Демидовец Д.В.

Козырев Д.А.

гр. 221703

Проверил: Сальников Д. А.

Минск 2025

Задание

Разработать элементарный графический редактор, реализующий построение полигонов и их заполнение, используя алгоритм растровой развертки с упорядоченным списком ребер; алгоритм растровой развертки с упорядоченным списком ребер, использующий список активных ребер; простой алгоритм заполнения с затравкой; построчный алгоритм заполнения с затравкой. Выбор алгоритма задается из пункта меню и доступен через панель инструментов «Алгоритмы заполнения полигонов». В редакторе должен быть предусмотрен режим отладки, где отображается пошаговое решение.

Ход работы

1. Изучение теоретических основ алгоритмов заполнения полигонов.
2. Реализация алгоритмов в программной среде.
3. Проверка отображения в графическом редакторе.

Описание алгоритмов

Алгоритм растровой развертки с упорядоченным списком ребер:

1. Создание таблицы рёбер (Edge Table):
   * Все рёбра полигона записываются в список.
   * Каждое ребро хранит:
     + y\_max — максимальную y-координату (где ребро заканчивается),
     + x\_min — x в нижней точке ребра,
     + 1/m — приращение x при переходе к следующей строке (где m — наклон ребра).
   * Рёбра сортируются по y\_min.
2. По сканлинии от y\_min до y\_max:
   * На каждой строке ищем, какие рёбра её пересекают.
   * Вычисляем пересечения с этими рёбрами — получаются x-координаты входа/выхода в полигон.
   * Между парами этих x-ов (внутри многоугольника) заливаем пиксели.

Алгоритм растровой развертки с упорядоченным списком ребер, использующий список активных ребер:

1. Создаём таблицу рёбер (ET), отсортированную по y\_min.
2. Инициализируем AET (Active Edge Table):
   1. На первой сканлинии (y\_min) добавляем в AET все рёбра, которые начинаются на этой строке.
3. Для каждой строки y:
   1. Удаляем из AET рёбра, у которых y\_max == y (они уже не актуальны).
   2. Добавляем новые рёбра из ET, у которых y\_min == y.
   3. Сортируем AET по текущим x.
   4. Пары соседних пересечений x обозначают начало и конец участков, которые нужно залить (пиксели между ними).
   5. Увеличиваем x каждого ребра в AET на его 1/m — то есть готовим на следующую строку.
   6. Переходим на следующую строку (y = y + 1).

Простой алгоритм заполнения с затравкой:

1. Берём координаты точки x, y (затравку).
2. Если пиксель на этой точке:
   1. ещё не закрашен,
   2. и его цвет совпадает с исходным (например, фон), то заливаем (меняем цвет).
3. Повторяем то же самое рекурсивно для соседей:
   1. Вверх (x, y - 1)
   2. Вниз (x, y + 1)
   3. Влево (x - 1, y)
   4. Вправо (x + 1, y)

Комбинированный метод. Построчный алгоритм заполнения с затравкой:

1. Кладём точку-затравку в стек или очередь.
2. Пока в стеке есть точки:
   * Достаём точку x, y.
   * Двигаемся влево и вправо, пока не наткнёмся на границу или уже закрашенную область — заливаем всю эту строку.
   * После этого:
     + Проверяем строку выше (y - 1)
     + Проверяем строку ниже (y + 1)
     + На этих строках ищем новые затравки в пределах залитого участка (то есть если там есть незакрашенные пиксели).
     + Добавляем их в стек.
3. Повторяем, пока стек не опустеет.

**Листинг кода**

  useEffect(() => {

    if (debugMode && fillSteps.length > 0) {

*const* timer = setInterval(() => {

        setCurrentStep(prev => (prev < fillSteps.length - 1 ? prev + 1 : prev));

      }, 100);

*return* () => clearInterval(timer);

    }

  }, [debugMode, fillSteps]);

*const* handleAddPoint = () => {

*const* x = parseFloat(inputCoords.x);

*const* y = parseFloat(inputCoords.y);

    if (isNaN(x) || isNaN(y) || x < 0 || x > 10 || y < 0 || y > 10) {

      setStatus("Ошибка: введите координаты от 0 до 10");

*return*;

    }

*const* newPoint = { x, y };

    switch(mode) {

      case "polygon":

        setOriginalPoints(prev => [**...**prev, newPoint]);

        break;

      case "segment":

        setSegments(prev => [**...**prev, newPoint]);

        break;

      case "test":

        setTestPoint(newPoint);

        break;

      default:

        break;

    }

    setInputCoords({ x: "", y: "" });

    setStatus(`Точка (${x.toFixed(1)}, ${y.toFixed(1)}) добавлена`);

  };

*const* calculateConvexHull = () => {

    if (polygonType === "concave") {

      setHullPoints([**...**originalPoints]);

      setStatus("Вогнутый полигон построен");

*return*;

    }

    if (originalPoints.length < 3) {

      setStatus("Нужно минимум 3 точки!");

*return*;

    }

*const* hull = convexHullMethod === "graham"

      ? grahamScan(originalPoints)

      : jarvisMarch(originalPoints);

    setHullPoints(hull);

    setStatus(`Построена оболочка (${hull.length} точек)`);

  };

*const* checkIntersections = () => {

*const* results = [];

    for (*let* i = 0; i < segments.length; i += 2) {

*const* start = segments[i];

*const* end = segments[i + 1];

      if (!start || !end) continue;

      hullPoints.forEach((p1, j) => {

*const* p2 = hullPoints[(j + 1) % hullPoints.length];

*const* intersect = lineIntersection(

          { x1: start.x, y1: start.y, x2: end.x, y2: end.y },

          { x1: p1.x, y1: p1.y, x2: p2.x, y2: p2.y }

        );

        if (intersect) results.push(intersect);

      });

    }

    setIntersections(results);

    setStatus(`Найдено пересечений: ${results.length}`);

  };

*const* handleFillPolygon = *async* () => {

    if (hullPoints.length < 3) {

      setStatus("Сначала постройте полигон!");

*return*;

    }

    setIsFilling(true);

    setStatus("Начало заполнения...");

*let* seed = hullPoints[0];

    if (testPoint && pointInPolygon(testPoint, hullPoints)) seed = testPoint;

*const* algorithmMap = {

      "ordered-edge-list": () => orderedEdgeListFill(hullPoints),

      "active-edge-list": () => activeEdgeListFill(hullPoints),

      "simple-seed": () => simpleSeedFill(hullPoints, seed),

      "scanline-seed": () => scanlineSeedFill(hullPoints, seed)

    };

    try {

*const* { filledPixels, steps } = algorithmMap[fillAlgorithm]();

      if (debugMode) {

        setFillSteps(steps);

        setCurrentStep(0);

      } else {

        setFillSteps([filledPixels]);

        setCurrentStep(0);

      }

      setStatus(`Заполнено ${filledPixels.length} пикселей`);

    } catch (error) {

      setStatus(`Ошибка: ${error.message}`);

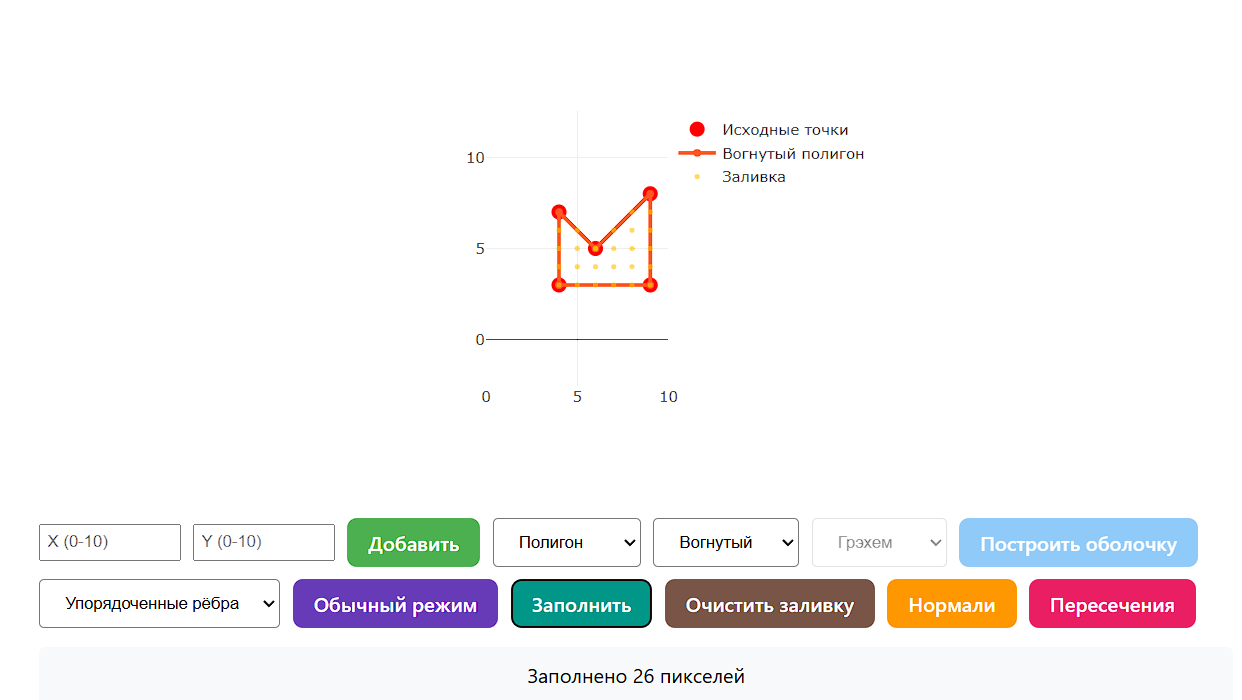
    }

    setIsFilling(false);

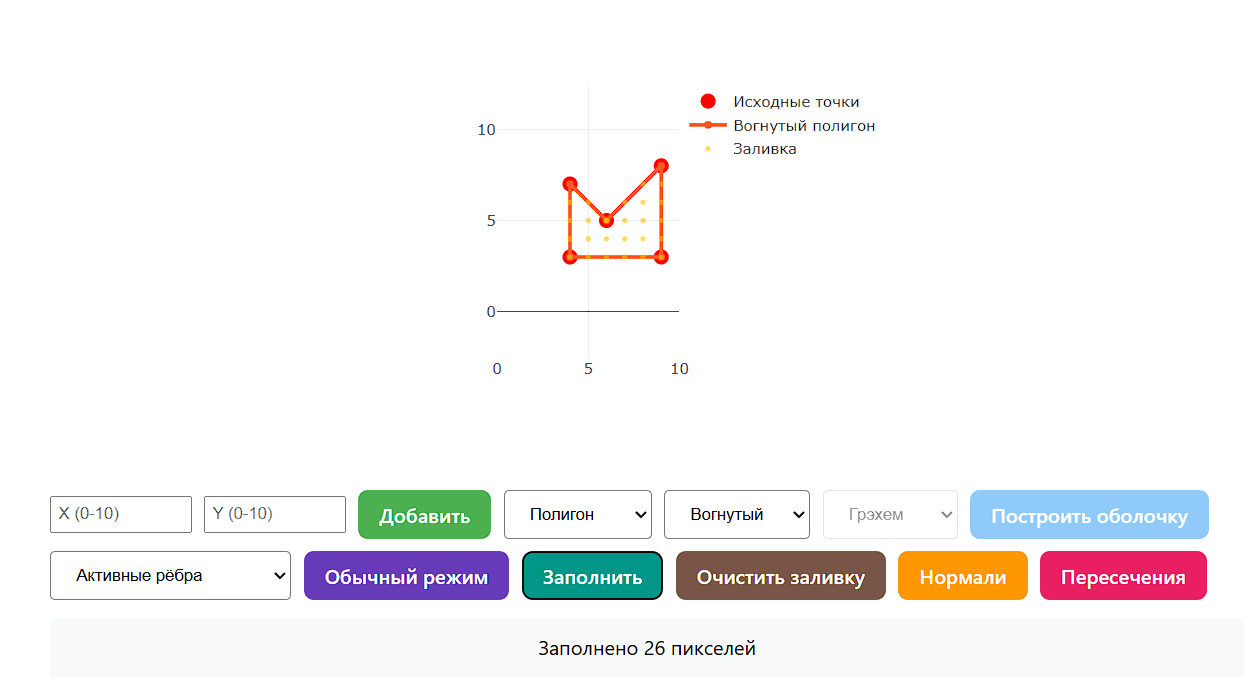
  };

**Результаты работы**

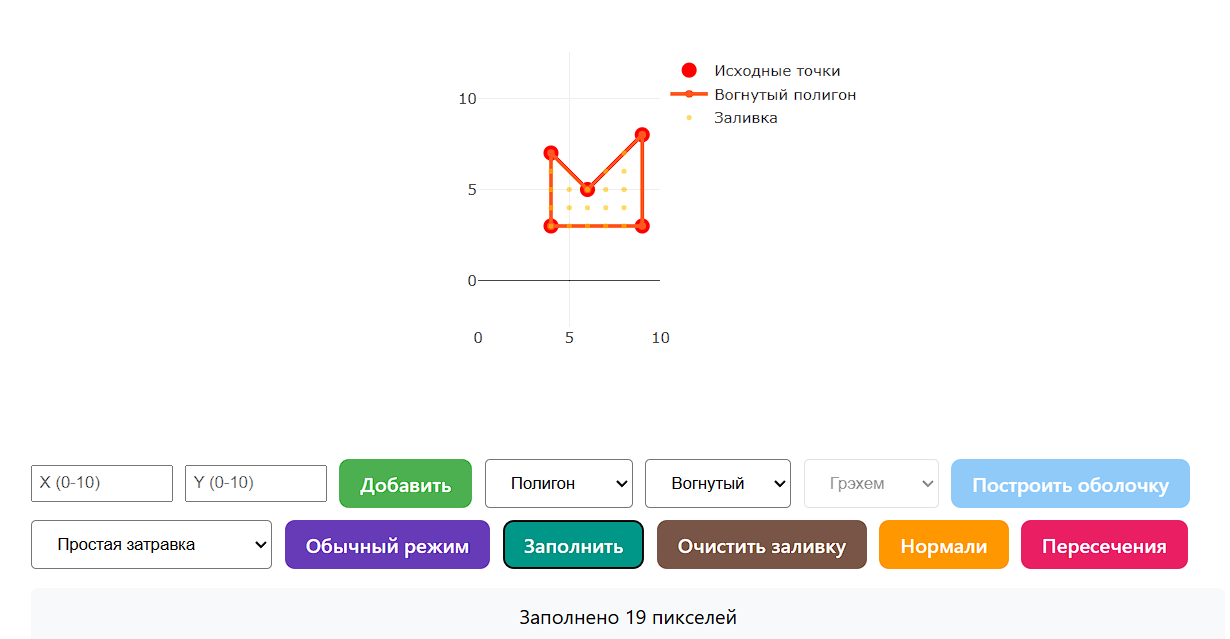
Алгоритм растровой развертки с упорядоченным списком ребер:



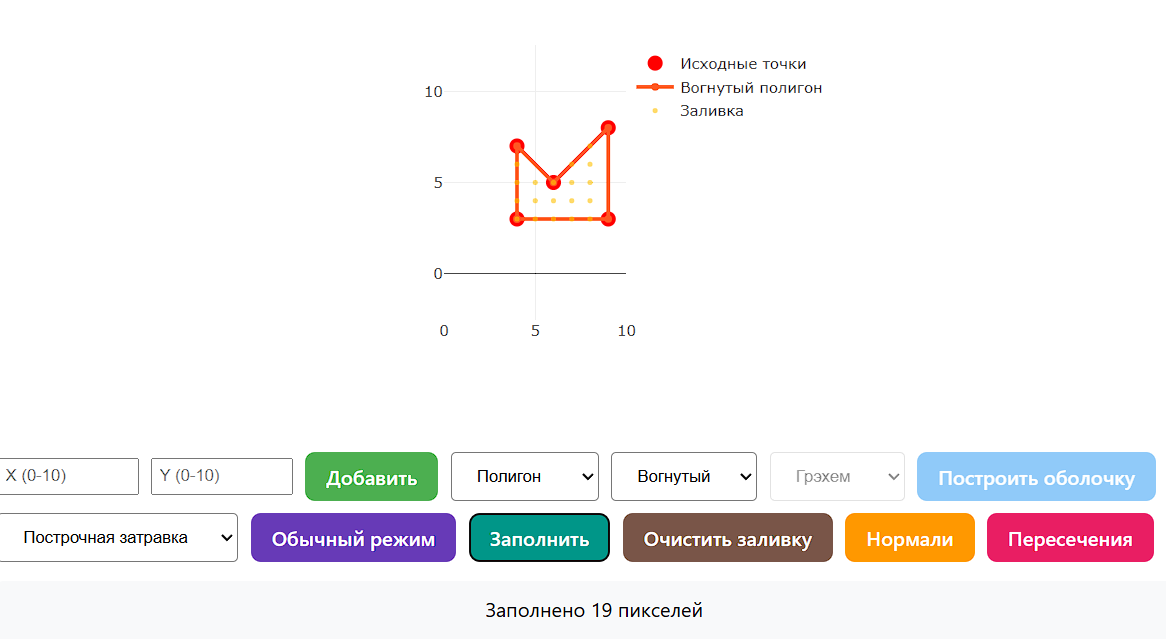
Алгоритм растровой развертки с упорядоченным списком ребер, использующий список активных ребер:



Простой алгоритм заполнения с затравкой:



Комбинированный метод. Построчный алгоритм заполнения с затравкой:



**Вывод:**

В ходе лабораторной работы были изучены и реализованы основные алгоритмы заполнения полигонов: алгоритмы **растровой развертки** (с упорядоченным списком рёбер и активным списком рёбер), а также **алгоритмы с затравкой** (простой рекурсивный и комбинированный построчный). В результате выполнения работы были закреплены навыки работы с графическими примитивами, логикой растровых операций, а также понимание различий между векторным и растровым подходами к построению изображений.