|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ **Информатика и системы управления**

КАФЕДРА **ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

**Отчет**

|  |  |
| --- | --- |
| **по лабораторной работе №** | 6 |

**Название:**

Реализация и исследование алгоритма построчного затравочного заполнения сплошных областей

**Дисциплина:** Компьютерная графика

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ7-43Б |  | 30.04.2020 | С.С. Кононенко |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | А.В. Куров |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2020

**1. Цель работы**

Реализация и исследование алгоритма построчного затравочного заполнения области.

**2. Техническое задание**

Реализовать алгоритм построчного затравочного заполнения сплошной области. Необходимо обеспечить ввод произвольной многоугольной области, содержащей произвольное количество отверстий. Ввод вершин многоугольника производить с помощью мыши, при этом для удобства пользователя должны отображаться ребра, соединяющие вводимые вершины. Предусмотреть ввод горизонтальных и вертикальных ребер. Пользователь должен иметь возможность задания цвета заполнения. Работа программы должна предусматривать два режима: с задержкой и без задержки. Режим с задержкой должен позволить проследить выполняемую последовательность действий. Обеспечить замер времени выполнения алгоритма. Продемонстрировать возможность заполнения с помощью затравочного алгоритма произвольной области, ограниченной замкнутой кривой линией.

**3. Теоретический материал**

Алгоритмы заполнения с затравкой основываются на изложенном далее подходе: в них предполагается, что известен хотя бы один пиксел из внутренней области фигуры, отталкиваясь от которого алгоритм находит и закрашивает все остальные пикселы, лежащие во внутренней области. Для определения внутренней области фигуры используют гранично-определенные области (области, которые определены выделенными пикселями на границе, цвет границы отличается от цвета внутренней области). Такие алгоритмы называются **гранично-заполняющими***.*

Области бывают 4-связными и 8-связными. 4-связная область – область, в которой любой пиксель можно достичь с помощью комбинации движений в 4 направлениях (вверх, вниз, влево, вправо). 8-связные области — в 8 направлениях (к описанным направления добавляются диагональные направления). Алгоритм заполнения 8-связной области может заполнить 4-связную область, но наоборот – нет.

Основная идея любого алгоритма с затравкой заключается в следующем: зная, что затравочный пиксель содержится в области, которую надо закрасить мы проверяем его каждый соседний пиксель, не являющийся граничным или уже закрашеным, и запоминаем его как **затравочный** (то есть принадлежавший внутренней области и подлежащий закраске). Перебрав все пиксели таким образом мы сможем найти и закрасить все пиксели, принадлежащие внутренней области.

Для реализации данного алгоритма можно использовать рекурсивный подход, что будет неэффективно. Для преобразования рекурсивных алгоритмов в итеративные часто используется **стек** (работает по принципу “первый пришел – последний ушел”). Вместо вызова в какой-то момент функции с другим объектом (как это происходит в рекурсии), мы запоминаем этот объект в стеке, чтобы обработать его на одной из следующих итераций.

Самый простой алгоритм заполнения с затравкой перебирает все пиксели: каждый сосед затравочного пикселя попадает в стек затравочных пикселей. Как только рассмотрены все соседи текущего затравочного пикселя, из стека берется следующий затравочный пиксель (однако если он уже был обработан, то происходит просто переход к следующему в стеке пикселю) и так до тех пор, пока стек не будет пустым. Очевидно, что пиксели могут попадать в стек более 1 раза, так как у них имеется 4 соседа, которые, становясь затравочными, добавляют их в стек. В этом и есть главная проблема данного алгоритма: тяжело предположить, как много памяти потребуется для хранения этих затравочных пикселей.

Проблему изложенного алгоритма решает **построчный алгоритм заполнения с затравкой***.* В построчном алгоритме хранится только один затравочный пиксель для каждого непрерывного интервала на сканирующей строке. Теперь при рассмотрении пикселя делаются следующие действия:

1. От текущего затравочного пикселя помечаются все расположенные в непрерывном интервале пикселы справа (идем направо и заполняем **все** подряд пикселы до тех пор, пока не встречается уже помеченный или граничный пиксел). Запоминаем крайний **правый**пиксел.
2. Аналогично поступаем двигаясь в левом направлении. Таким образом мы закрасим весь строчный непрерывный интервал в котором находился затравочный пиксел. На данном шаге также запоминаем крайний **левый** пиксел.
3. Рассматриваем верхнюю и нижнюю строки в интервале [Хлев; Хправ]. Во всех непрерывных интервалах не закрашенных и не граничных пикселей, отмечаем самый правый пиксел (либо с координатой Xправ, если координата Х самого правого больше).

**4. Реализация алгоритма**

def fill(root):

delay = root.modelst.get(root.modelst.curselection()[**0**])

**if** delay == "С задержкой":

delay = True

**else**:

delay = False

color = hex\_to\_dec(root.color)

**while** root.stack:

point = root.stack.pop()

root.image.put(root.color, (point[**0**], point[**1**]))

x, y = point[**0**] + **1**, point[**1**]

**while** root.image.get(x, y) != color:

root.image.put(root.color, (x, y))

x += **1**

rborder = x - **1**

x = point[**0**] - **1**

**while** root.image.get(x, y) != color:

root.image.put(root.color, (x, y))

x -= **1**

lborder = x + **1**

sign = [**1**, -**1**]

**for** i **in** sign:

x = lborder

y = point[**1**] + i

**while** x <= rborder:

is\_exist = False

**while** root.image.get(x, y) != color **and** x <= rborder:

is\_exist = True

x += **1**

**if** is\_exist:

root.stack.extend([[x - **1**, y]])

is\_exist = False

xi = x

**while** root.image.get(x, y) != color **and** x <= rborder:

x += **1**

**if** x == xi:

x += **1**

**if** delay:

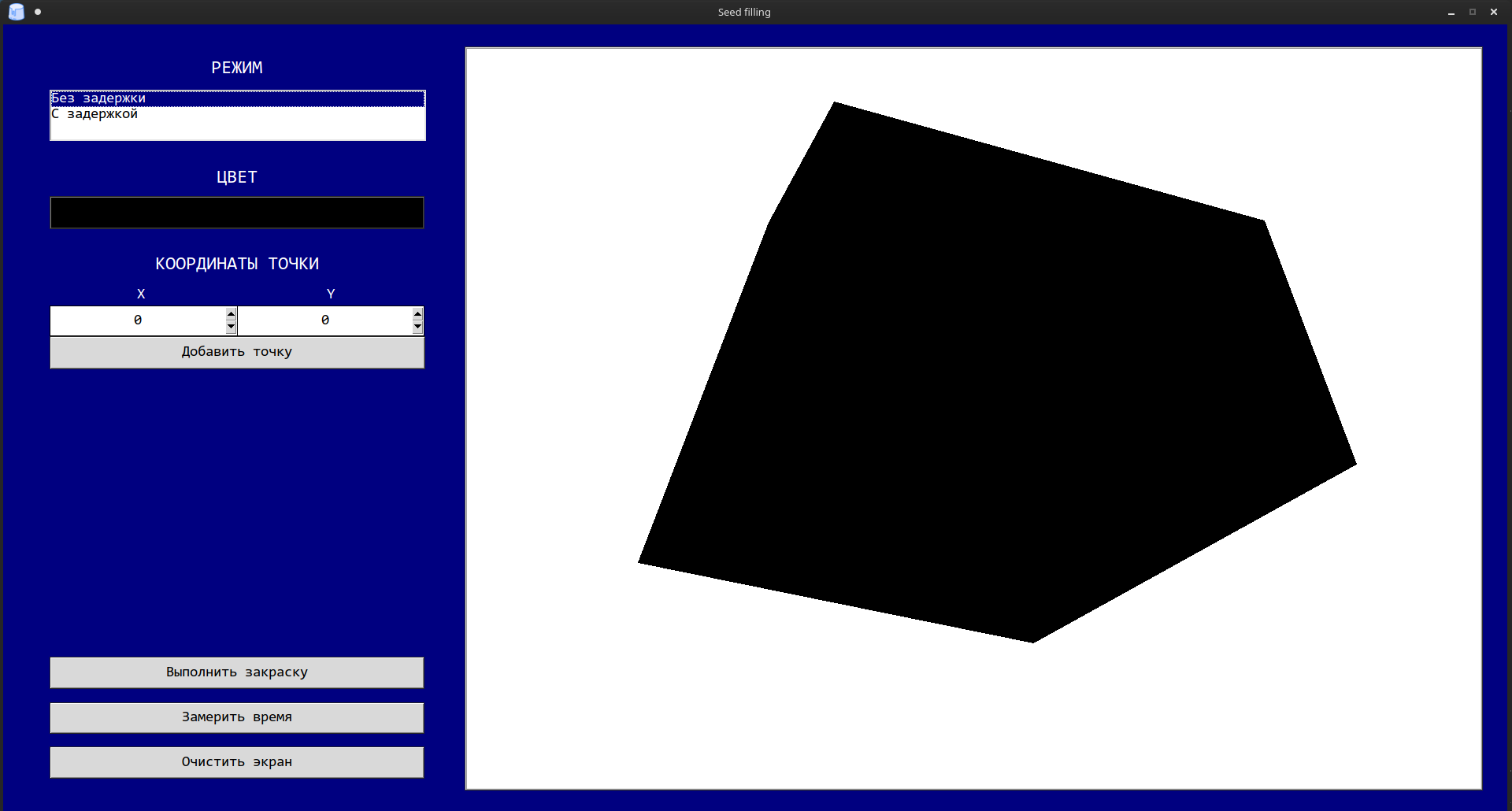
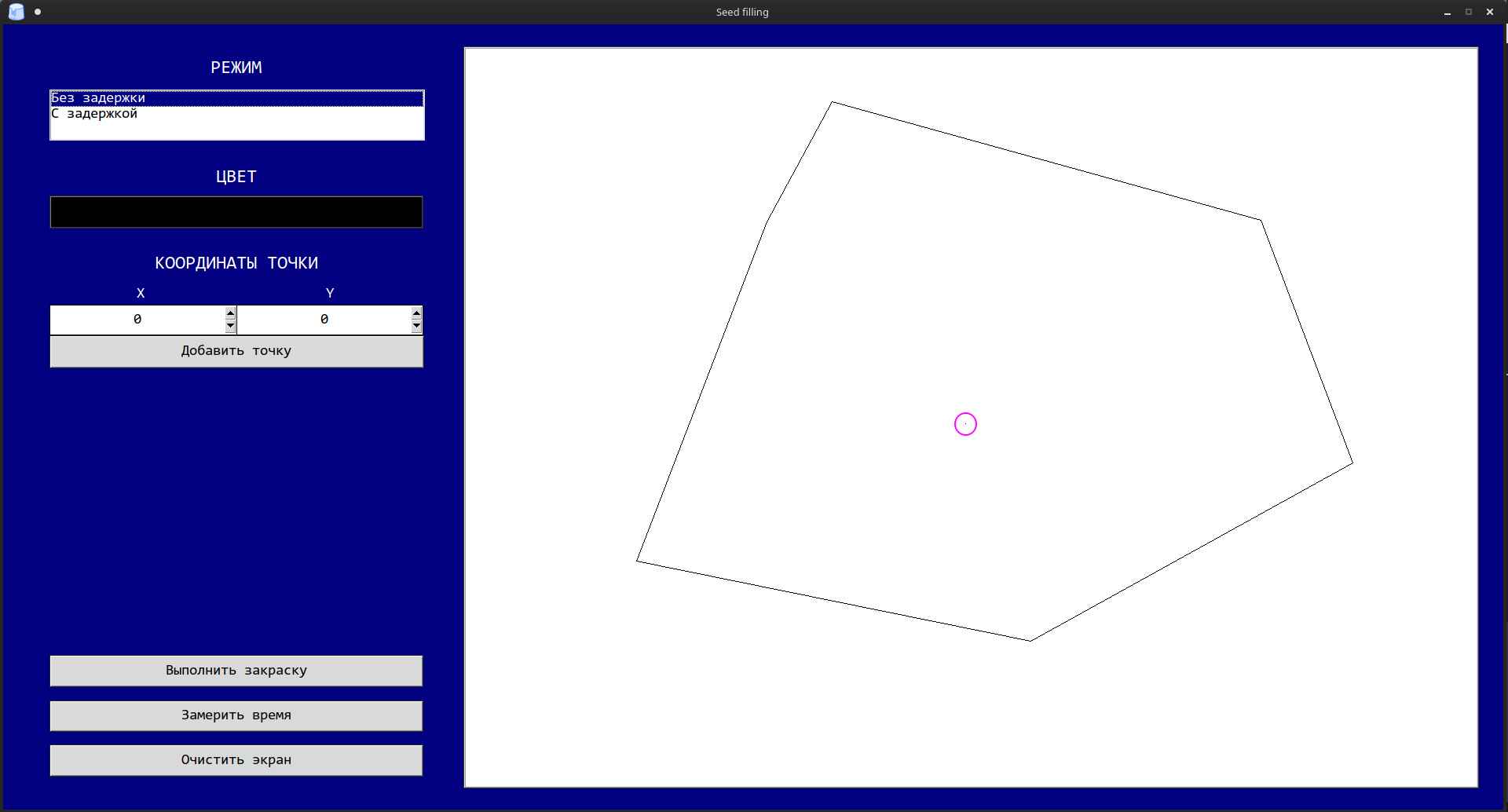
time.sleep(**0.01**)

root.canvas.update()

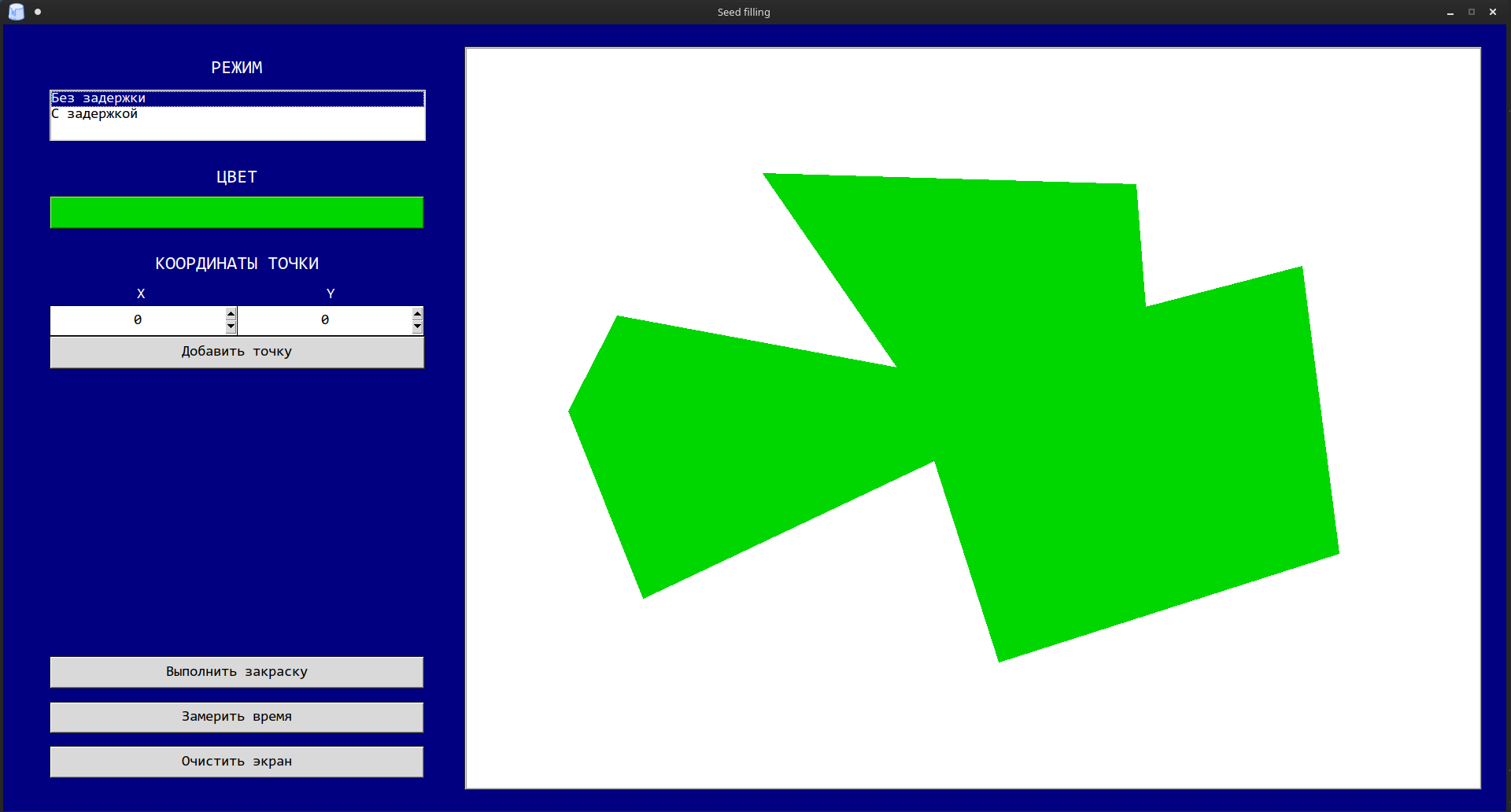
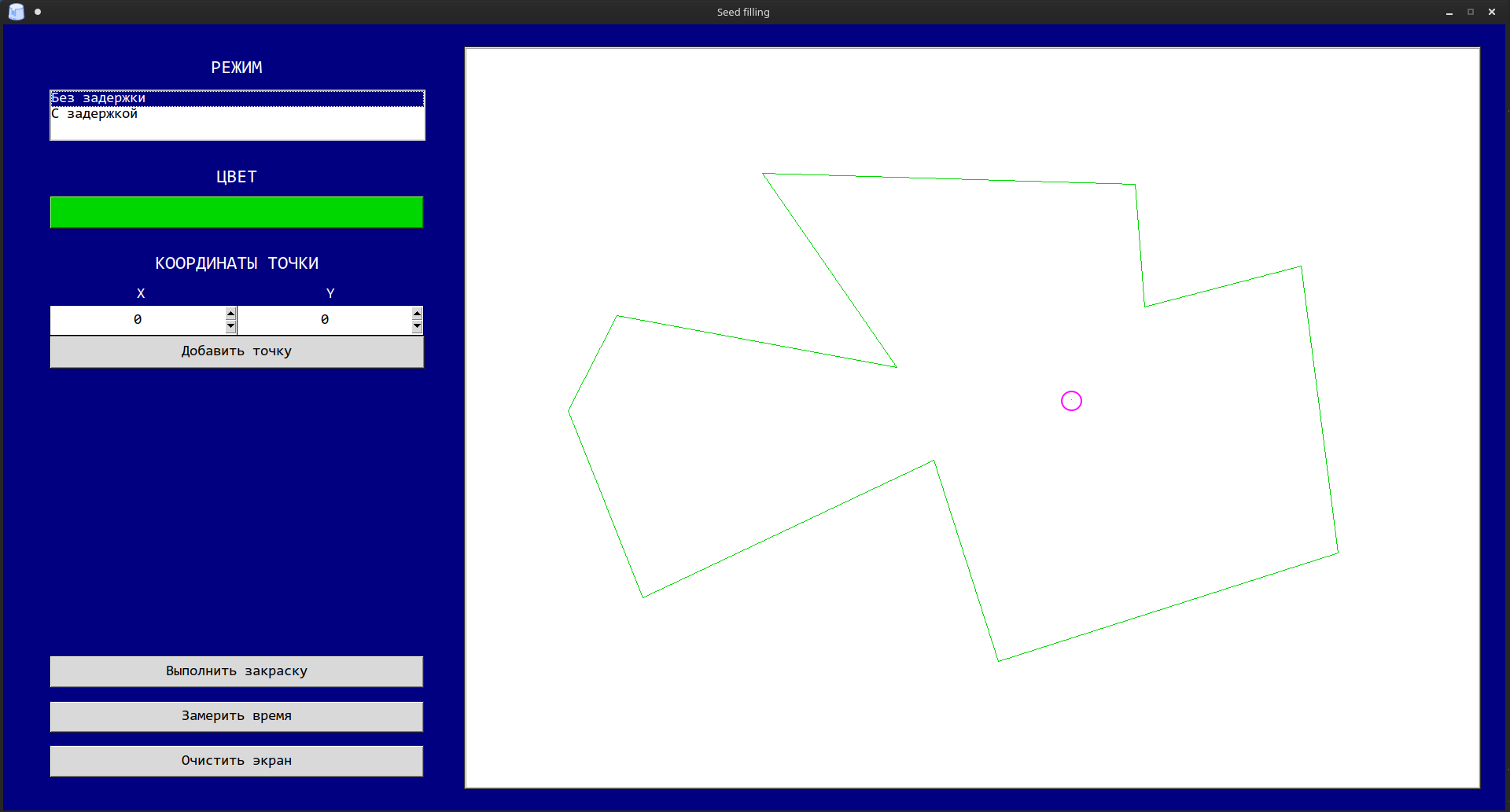
**5. Демонстрация работы программы**

В зоне лиловой окружности будет указано расположение затравочного пикселя.

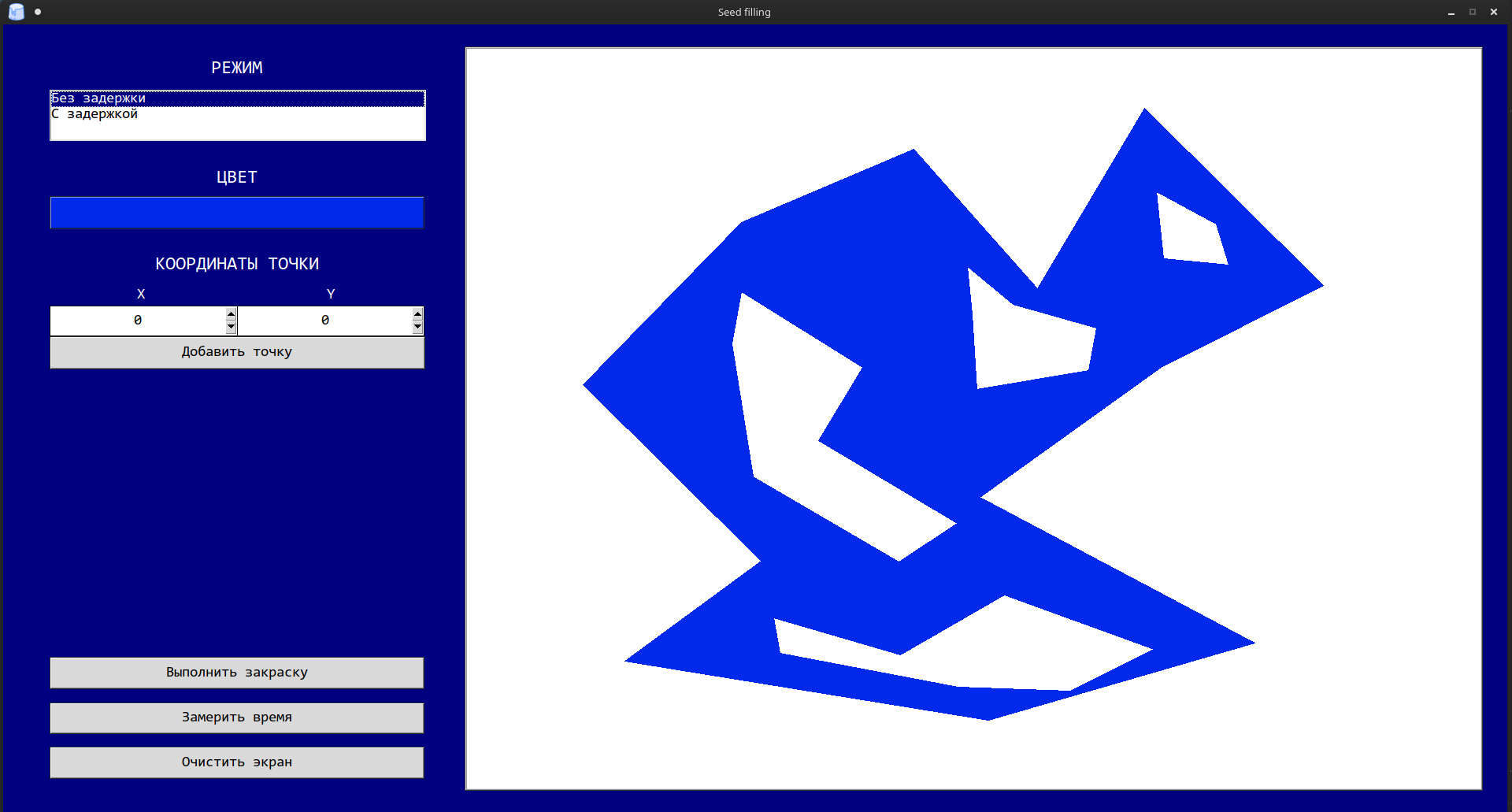
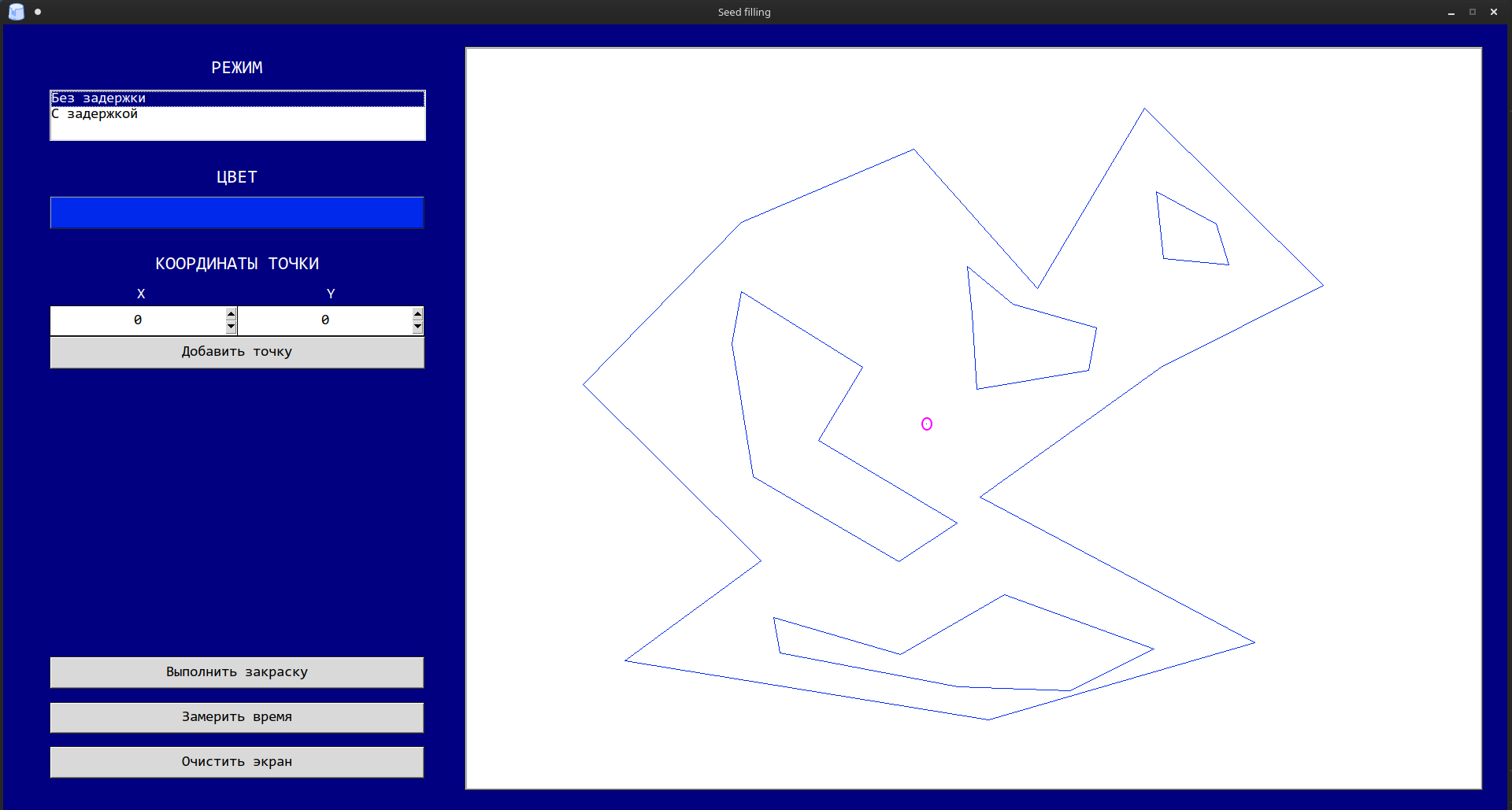
Заполнение произвольного выпуклого многоугольника.

Время, затраченное на закраску: **4.6 с.**

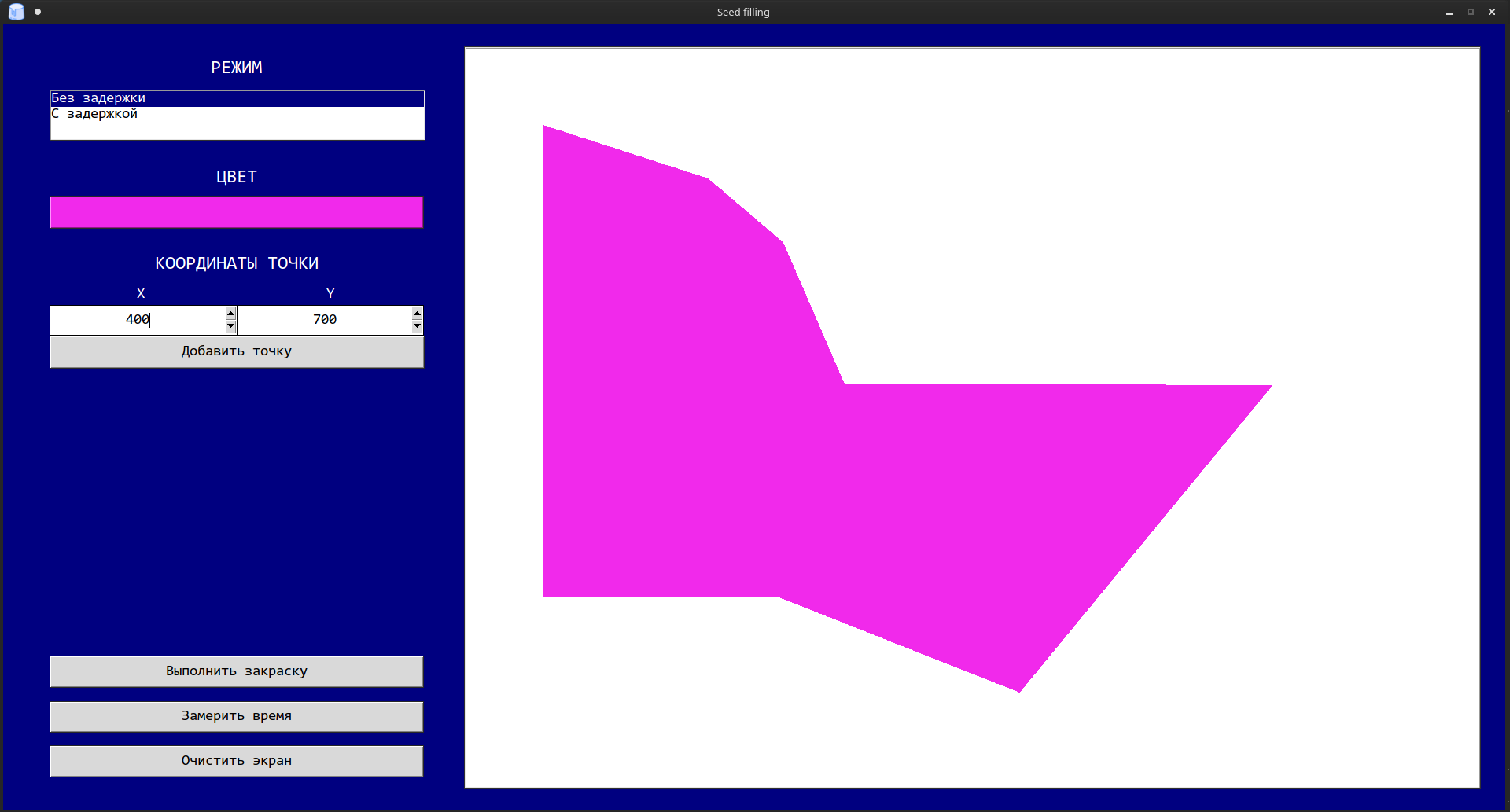
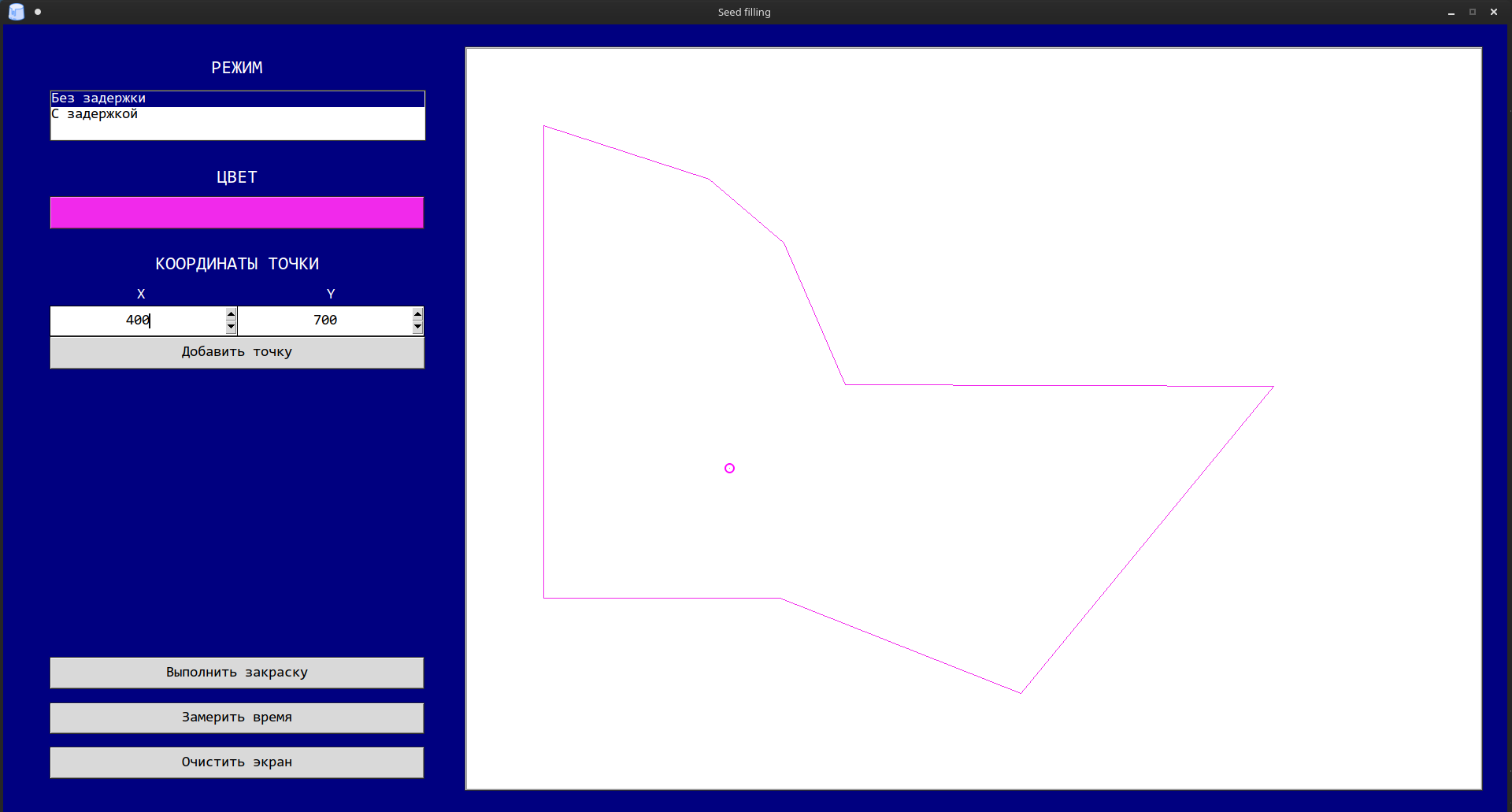
Заполнение произвольного невыпуклого многоугольника.

Время, затраченное на закраску: **3.7 с.**

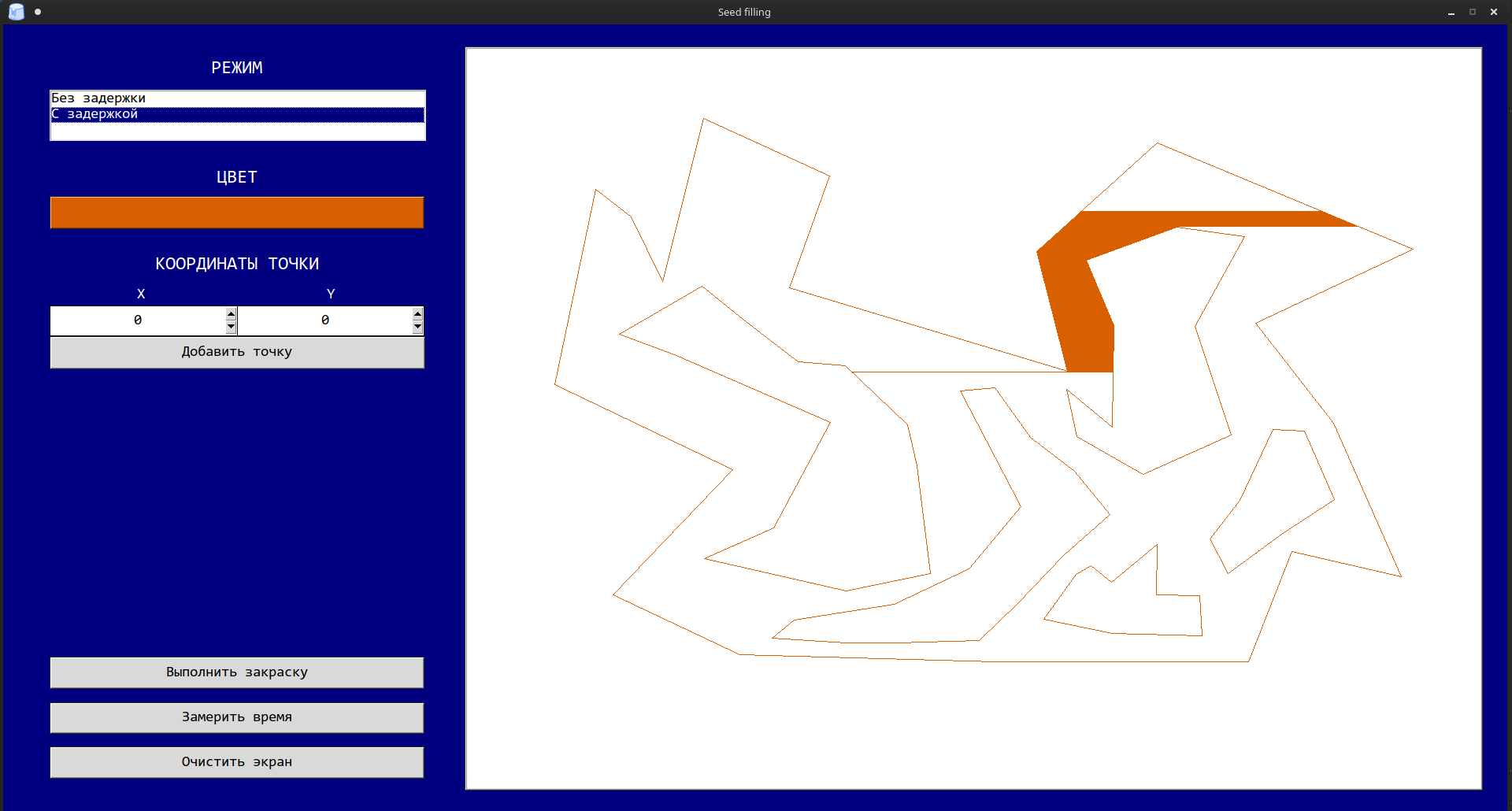
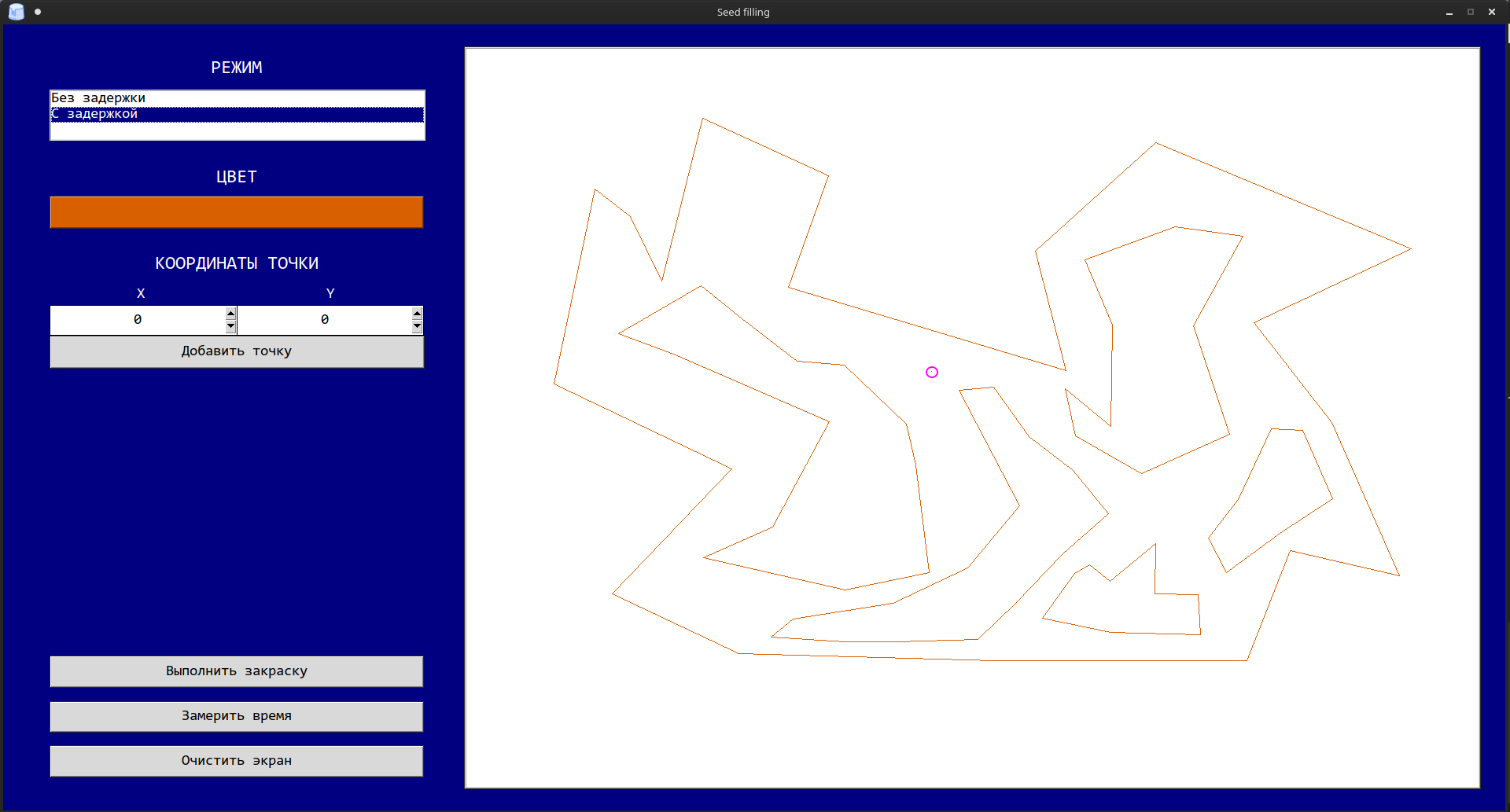
Заполнение произвольного многоугольника с отверстиями.

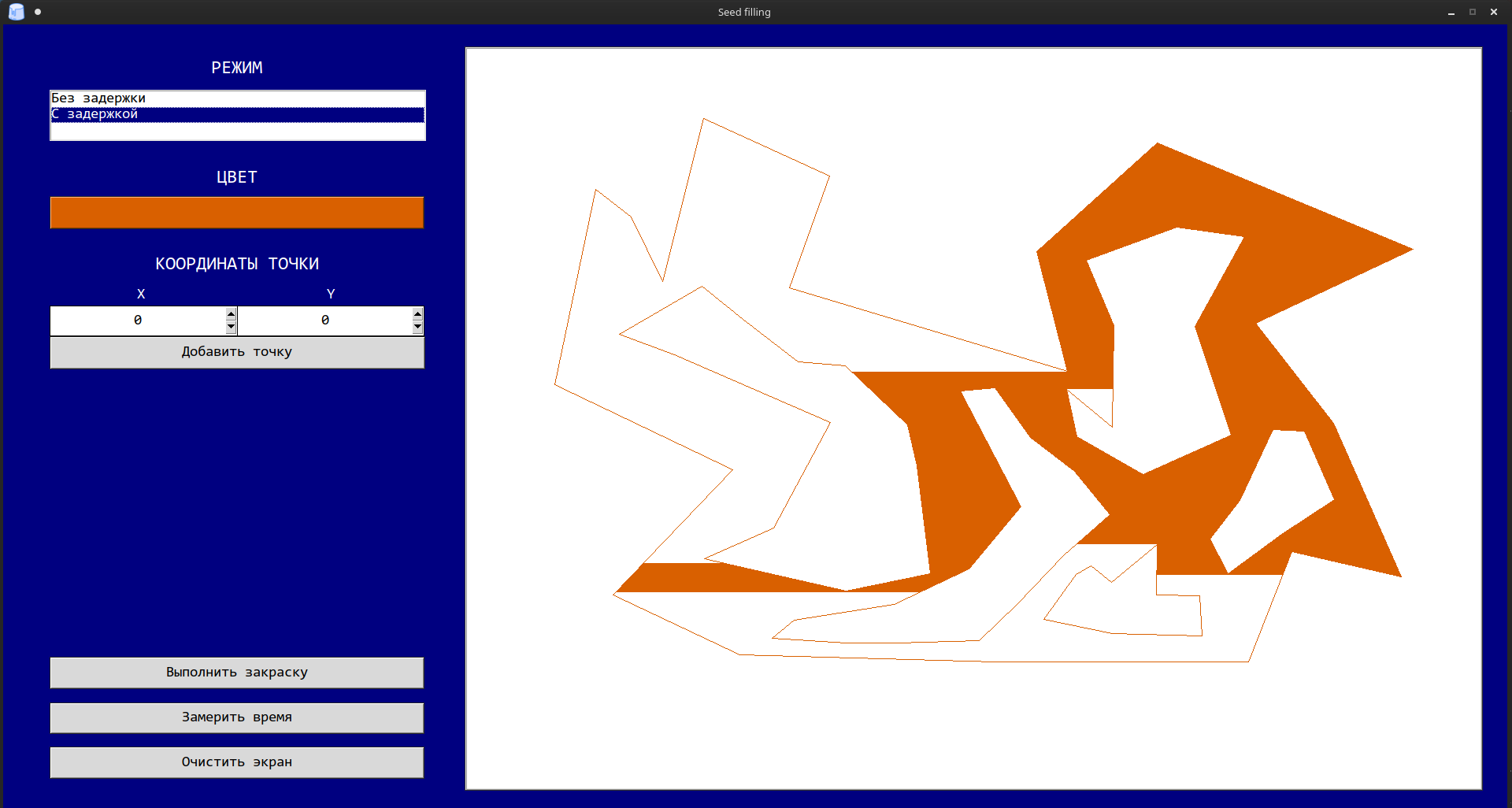
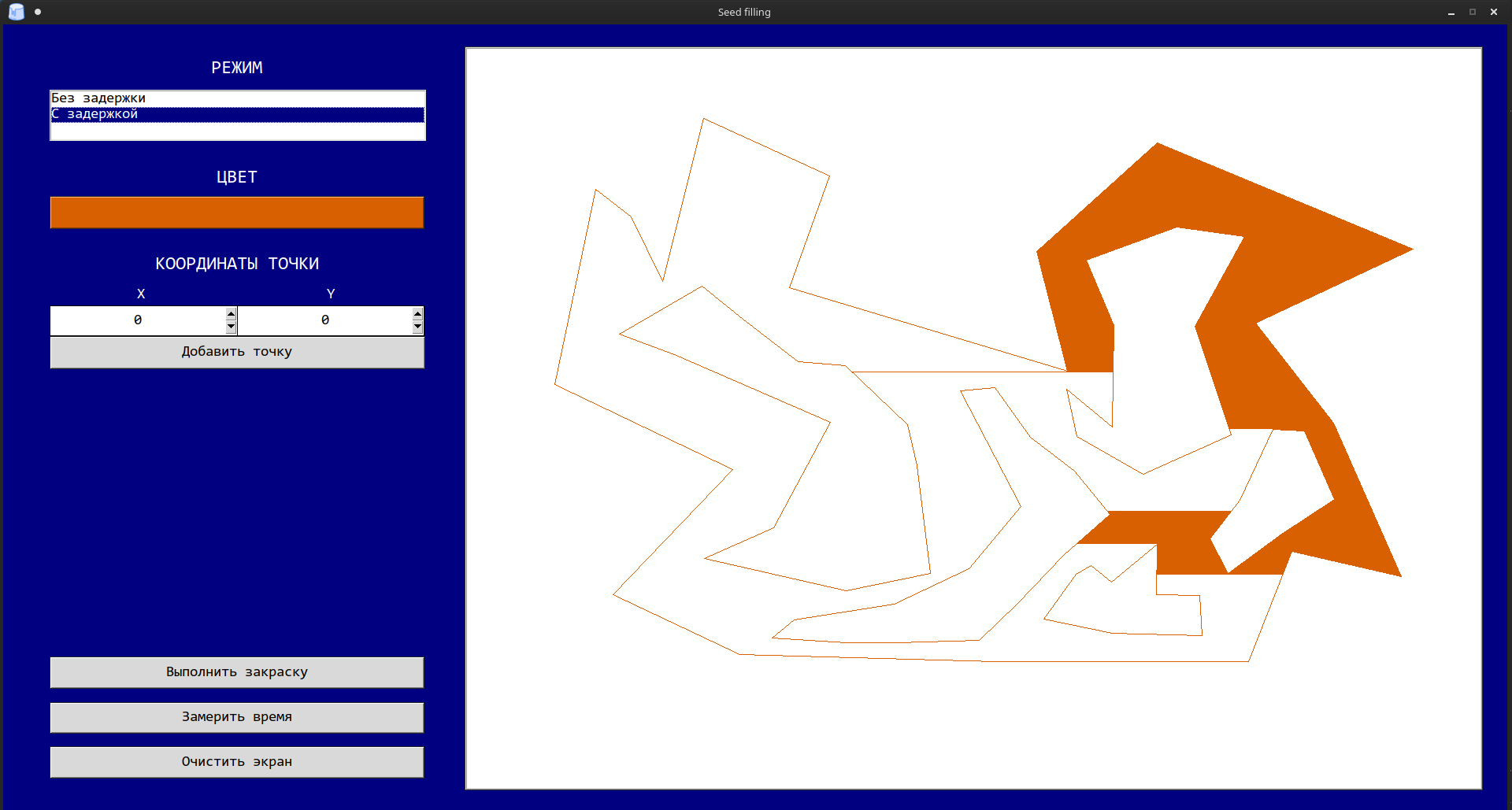
Время, затраченное на закраску: **3.2 с.**

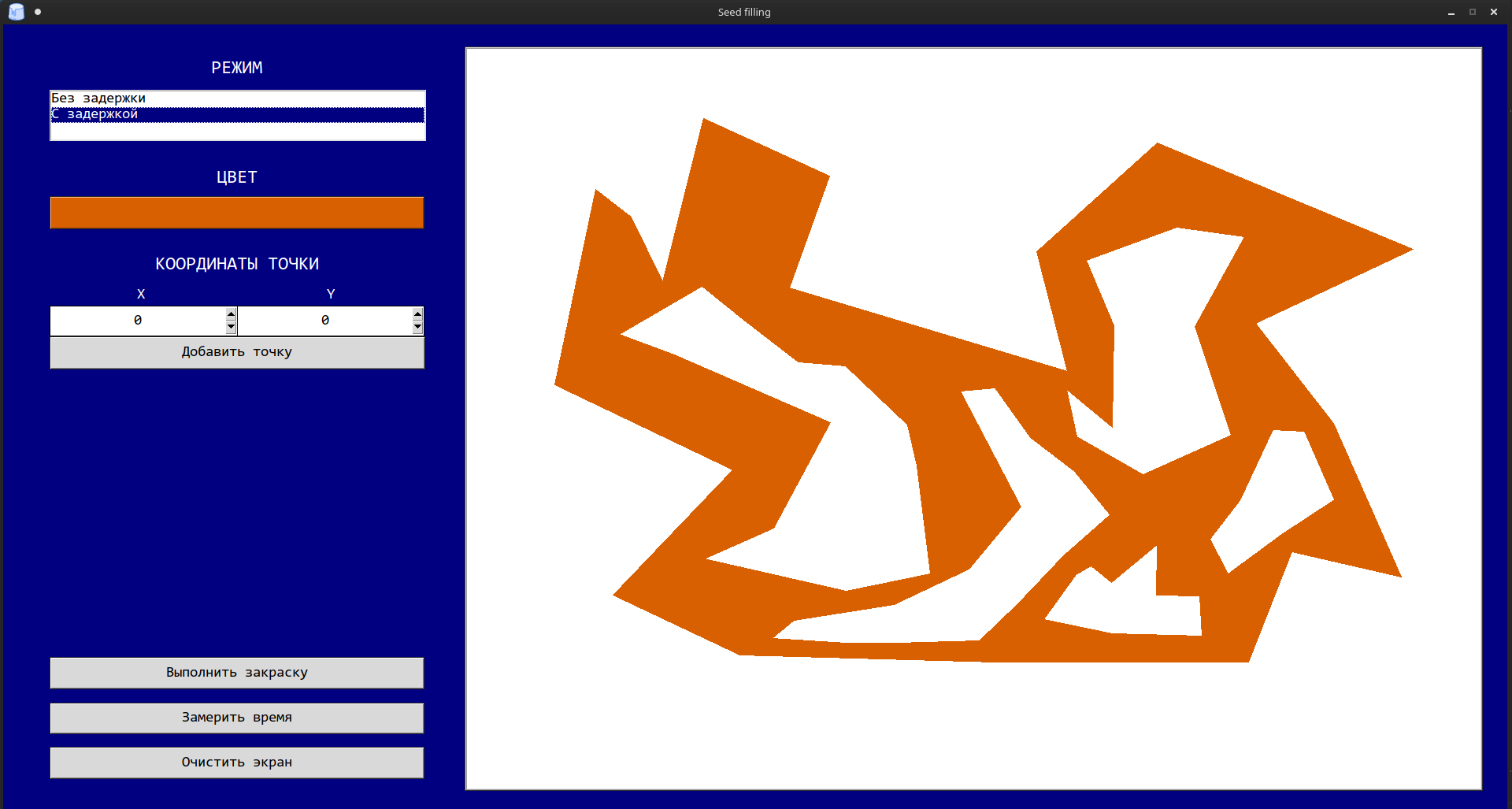
Заполнение многоугольника с горизонтальными и вертикальными ребрами.

Время, затраченное на заркаску: **3.5 с.**

Поэтапное заполнение многоугольника (режим работы с задержкой).







Заполнение области, ограниченной замкнутой кривой линией.

