Daniel Victor Ferreira da Silva
2019006523
Link para notebook: https://github.com/Cogictor/Galeria-de-Arte (https://github.com/Cogictor/Galeria-de-Arte)

Problema da Galeria de Arte

A solução do problema da Galeria de Arte foi implementada baseada no algoritmo Ear Clipping para triangulação de polígonos, e o algoritmo para 3-coloração do gráfico.

O primeiro passo para a solução do problema é a triangulação do polígono usando o método Ear-Clipping. Inicialmente o algoritmo verifica os vértices que formam orelhas, ou seja, de pi-1->pi->pi+1 há um deslocamento para a direita sendo que o triângulo formado pelos três pontos não contém nenhum outro ponto do polígono. A partr dessa primeira verificação, retira-se cada orelha do polígono verificandose os vértices vizinhos se tornaram novas orelhas, enquanto o polígono tem mais de três vértices.

Dado um grafo cujos vértices são os triângulos da triangulação e arestas são as arestas compartilhadas entre os triângulos, podemos colorir esse grafo, que é uma árvore, com uma busca em profundidade. Partindo de um nó inicial onde atribui-se uma cor diferente a cada ponto, caminha-se pelo gráfico indo para o próximo triângulo de aresta comum, atribuindo ao ponto fora da aresta a cor restante. Após caminhar por todos os nós do grafo, o menor conjunto de pontos de mesma cor é uma atribuição possível de posições das câmeras na galeria de arte.

Detalhes e descrição da implementação

Assume-se que como entrada tem-se um conjunto de n pontos (x,y) representando um polígono, de modo que os pontos etão em ordem crescente de ângulo no sentido ant-horário em relação ao primeiro ponto.

Para a primeira etapa da triangulação, foram implementadas as funções prod_vet, vetor, direcao e inTriangulo para retornar se um ponto está contido no triângulo formado por outros três pontos. já para reconhecer orelhas no polígono, a função check testa a direção do deslocamento de um elemento no cojunto, e então se o triângulo formado pelo ponto e seus vizinhos não contém nenhum outro para determinar se é orelha; enquanto find_ear encontra as orelhas no conjunto inicial de pontos.
A função triangulação é a função principal do algoritmo, onde enquanto o conjunto tem mais de três pontos, remove-se o ponto com uma função remover, armana-se a diagonal interna eo triângulo reconhecidos com a remoção do ponto, e armazena-se um gráfico ilustrando o polígono com o ponto removido e diagonal interna encontrada a cada etapa. Assim, a função remover remove o ponto do conjunto e testa se os vizinhos do ponto original agora formam orelhas com check, atualizando esses conjuntos.
Dessa forma, temos como saída o conjunto de triângulos, as diagonais internas, e um gráfico mostrando interativamente o processo de triangulação do polígono.

Já para a coloração do grafo, primeiramente foi decidido representar o grafo como uma lista encadeada, com o índice do triângulo apontando para uma lista com índice dos triângulos que compartilha aresta, essa grafo sendo gerado pela função gerar_grafo usando a lista de triângulos e diagonais internas. Já a função caminha, caminha recursivamente no grafo em profundidade enquanto associa cores aos pontos. Assim, dada a coloração do grafo, listar_cores separa os pontos em conjuntos para cada cor, usadas para gerar o gráfico ilustrando a coloração no polígono.

Biblioteca para visualização

In [1]:

```
import holoviews as hv
from holoviews import dim, opts
from bokeh.plotting import show

hv.extension('bokeh')
```

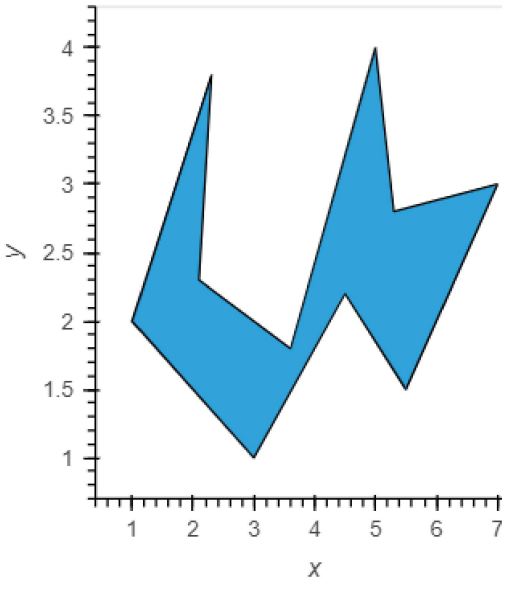
Entrada

Exemplos modelo para a entrada, que também foram usadas para teste:

In [2]:

```
p0 = (1,2)
p1 = (3,1)
p2 = (4.5,2.2)
p3 = (5.5,1.5)
p4 = (7,3)
p5 = (5.3,2.8)
p6 = (5,4)
p7 = (3.6,1.8)
p8 = (2.1,2.3)
p9 = (2.3,3.8)
conjunto = [p0,p1,p2,p3,p4,p5,p6,p7,p8,p9]
pplot=conjunto.copy()
hv.Polygons(conjunto)
```

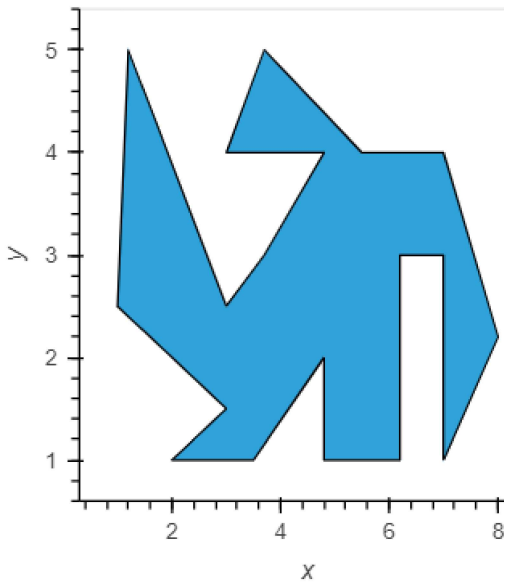
Out[2]:



In [3]:

```
p0 = (2,1)
p1 = (3.5,1)
p2 = (4.8,2)
p3 = (4.8,1)
p4 = (6.2,1)
p5 = (6.2,3)
p6 = (7,3)
p7 = (7,1)
p8 = (8,2.2)
p9 = (7,4)
p10 = (5.5,4)
p11 = (3.7,5)
p12 = (3,4)
p13 = (4.8,4)
p14 = (3.7,3)
p15 = (3,2.5)
p16 = (1.2,5)
p17 = (1,2.5)
p18 = (3,1.5)
conjunto = [p18,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17]
pplot=conjunto.copy()
hv.Polygons(conjunto)
```

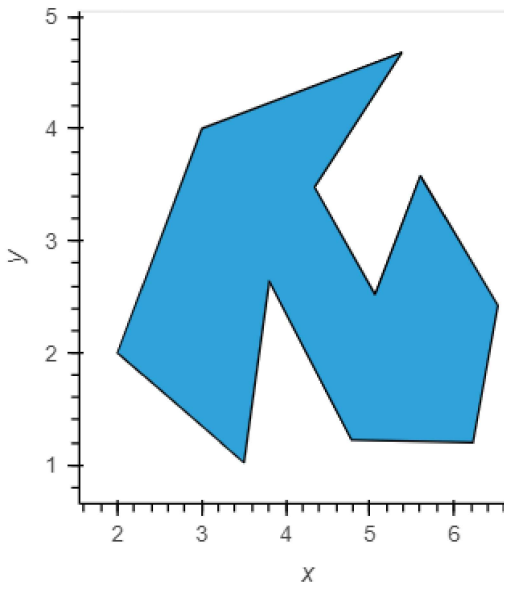
Out[3]:



In [4]:

```
points4 = [(3,4),
(2,2),
(3.5025,1.02125),
(3.8025,2.64125),
(4.7825,1.22125),
(6.2225,1.20125),
(6.5225,2.42125),
(5.6025,3.58125),
(5.0625,2.52125),
(4.3425,3.48125),
(5.3825,4.68125)]
conjunto = points4.copy()
pplot =conjunto.copy()
hv.Polygons(conjunto)
```

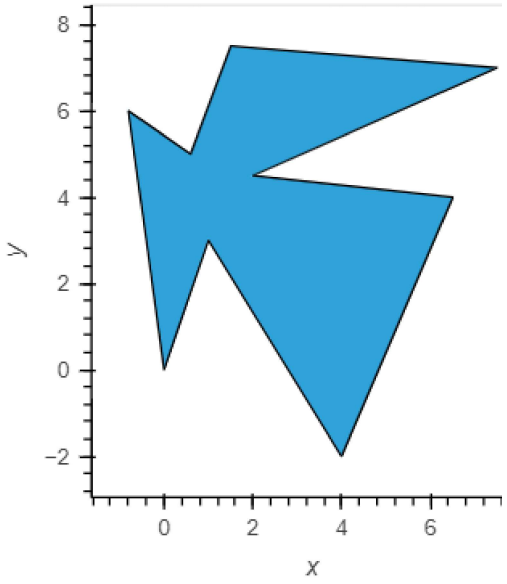
Out[4]:



In [5]:

```
points2 = [(0,0), (1,3), (4,-2),(6.5,4),(2,4.5),(7.5,7),(1.5,7.5),(0.6,5),(-0.8,6)]
conjunto = points2.copy()
pplot =conjunto.copy()
hv.Polygons(conjunto)
```

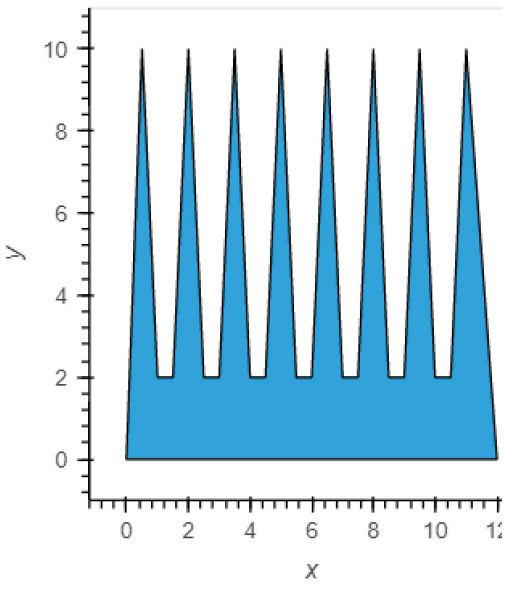
Out[5]:



In [6]:

```
points = [(0, 0),
 (0.5, 10),
 (1.0, 2),
 (1.5, 2),
 (2.0, 10),
 (2.5, 2),
 (3.0, 2),
 (3.5, 10),
 (4.0, 2),
 (4.5, 2),
 (5.0, 10),
 (5.5, 2),
 (6.0, 2),
 (6.5, 10),
 (7.0, 2),
 (7.5, 2),
 (8.0, 10),
 (8.5, 2),
 (9.0, 2),
 (9.5, 10),
 (10.0, 2),
 (10.5, 2),
 (11.0, 10),
 (12, 0)]
conjunto = points[::-1].copy()
pplot =conjunto.copy()
hv.Polygons(conjunto)
```

Out[6]:



Triangulação

In [7]:

```
def prod_vet(v1,v2):
    return v1[0]*v2[1]-v2[0]*v1[1]

def vetor(p0,p1):
    return (p1[0]-p0[0],p1[1]-p0[1])

def direcao(p0,p1,p2):
    return prod_vet(vetor(p0,p1),vetor(p2,p0))

def intriangulo(s,p0,p1,p2):
    # Retorna se o ponto s está contido no triângulo formao pelos outros 3 pontos.
    # O ponto s sobre uma aresta do triângulo é considerado dentro do triângulo
    d1 = direcao(p0,p1,s)
    d2 = direcao(p1,p2,s)
    d3 = direcao(p2,p0,s)
    if ((d1 <= 0) and (d2 <= 0) and (d3 <=0)):
        return True
    elif ((d1 >= 0) and (d2 >= 0) and (d3 >= 0)):
        return True
    else:
        return False
```

In [8]:

```
# Exemplo da execução da função Intriagulo
p0 = (0,0)
p1 = (1,1)
p2 = (0,1)
p = (0.5,0.5)
intriangulo(p,p1,p2,p0)
```

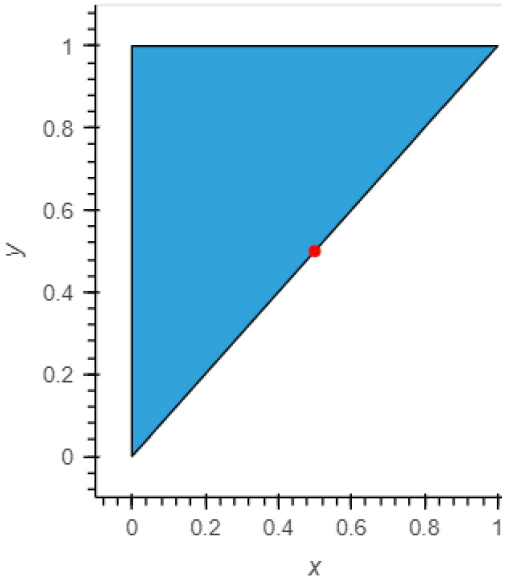
Out[8]:

```
True
```

In [9]:

```
# Representação gráfica do exemplo anterior
pol = hv.Polygons([p1,p2,p0])
pol = pol*hv.Points(p)
pol.opts({'Points':{'color':'red','size':5}})
```

Out[9]:



In [10]:

```
def check(i,conjunto):
    # Retorna True se o i-ésimo elemento de conjunto não for orelha e False se for orelha
    pontos = conjunto+conjunto[0:2]
    sz = len(pontos)
    first = 0
    if i==sz-3:
        first = 2
    elif i==sz-4:
        first = 1
    dentro = True
    if direcao(pontos[i],pontos[i+1],pontos[i+2])<0:
        dentro = False
        for j in (pontos[first:i]+pontos[i+3:-2]):
            if intriangulo(j,pontos[i],pontos[i+1],pontos[i+2]):
                dentro = True
                break
    return dentro
```

In [11]:

```
def find_ear(conjunto):
    # Retorna o índice das orelhas encontradas no conjunto de pontos
    ear = []
    sz = len(conjunto)
    first = 0
    for i in range(sz):
        if not check(i,conjunto):
            if i+1==sz:
                ear.append(0)
            else:
                ear.append((i+1))
    return ear
```

In [12]:

```
ear  = find_ear(conjunto)
ear
```

Out[12]:

```
[1, 4, 7, 10, 13, 16, 19, 22]
```

In [13]:

```
def remover(ind_ear,conjunto,ear):
    # Retorna um conjunto sem a orelha de índice recebido na entrada
    # e atualiza as orelhas detectadas no conjunto
    p_ear = conjunto[ind_ear]
    conjunto.remove(p_ear)
    ant_ant =ind_ear-2
    ant =ind_ear-1
    if ind_ear==0:
        ant_ant =len(conjunto)-2
        ant = len(conjunto)-1
    elif ind_ear==1:
        ant_ant =len(conjunto)-1
        ant = 0
    if not check(ant_ant,conjunto):
        if ant not in ear:
            ear = [ant]+ear
    if not check(ant,conjunto):
        if ind_ear not in ear:
            ear = [ind_ear]+ear
    else:
        if ind_ear in ear:
            ear = ear[1:]
    return conjunto,ear
```

In [14]:

```
def triangulacao(conjunto,ear,pplot):
    # Execução central do algoritmo Ear Cilpping
    # Recede um conjunto de pontos, as orelhas detectadas no conjunto, e a
    # sequência de pontos para gráficos
    # Retorna a triangulação do poligono, as diagonais internas, e uma lista
    # com o representação gráfica do poligono em cada etapa do algoritmo
    triangulos = []
    diag_internas = []
    x=0
    plots = []
    p = []
    ppo = hv.Polygons(pplot)*hv.Points([]).opts({'Points':{'color':'red','size':7,'marker':
    #plots.append(ppo)
    holom=hv.HoloMap(kdims="Step")
    holom[0] = ppo
    ind = 1
    while (len(conjunto) > 3):
        ind_ear = ear[0]
        if ind_ear==0:
            p0_,p1_,p2_=conjunto[len(conjunto)-1],conjunto[0],conjunto[1]
        else:
            [p0_,p1_,p2_]=conjunto[ind_ear-1:ind_ear+2]
        ear = ear[1:]
        for i,v in enumerate(ear):
            if v>ind_ear:
                ear[i] = v-1
        conjunto,ear = remover(ind_ear,conjunto,ear)
        triangulos.append((p0_,p1_,p2_))
        diag_internas.append((p0_,p2_))
        for i,e in enumerate(pplot):
            if e==p0_:
                pplot=pplot[:i+1]+[p2_]+[p0_]+pplot[i+1:]
                break
        p.append(p1_)
        ppo = hv.Polygons(pplot)*hv.Points(p).opts({'Points':{'color':'red','size':7,'marke
        holom[ind] = ppo
        ind += 1
    triangulos+=[tuple(conjunto)]
    return triangulos,diag_internas,holom
```

In [15]:

```
triangulos,diag_internas,plots = triangulacao(conjunto,ear,pplot)
triangulos
```

Out[15]:

```
[((12, 0), (11.0, 10), (10.5, 2)),
 ((12, 0), (10.5, 2), (10.0, 2)),
 ((0, 0), (12, 0), (10.0, 2)),
 ((10.0, 2), (9.5, 10), (9.0, 2)),
 ((0, 0), (10.0, 2), (9.0, 2)),
 ((0, 0), (9.0, 2), (8.5, 2)),
 ((8.5, 2), (8.0, 10), (7.5, 2)),
 ((0, 0), (8.5, 2), (7.5, 2)),
 ((0, 0), (7.5, 2), (7.0, 2)),
 ((7.0, 2), (6.5, 10), (6.0, 2)),
 ((0, 0), (7.0, 2), (6.0, 2)),
 ((0, 0), (6.0, 2), (5.5, 2)),
 ((5.5, 2), (5.0, 10), (4.5, 2)),
 ((0, 0), (5.5, 2), (4.5, 2)),
 ((0, 0), (4.5, 2), (4.0, 2)),
 ((4.0, 2), (3.5, 10), (3.0, 2)),
 ((0, 0), (4.0, 2), (3.0, 2)),
 ((0, 0), (3.0, 2), (2.5, 2)),
 ((2.5, 2), (2.0, 10), (1.5, 2)),
 ((0, 0), (2.5, 2), (1.5, 2)),
 ((0, 0), (1.5, 2), (1.0, 2)),
 ((1.0, 2), (0.5, 10), (0, 0))]
```

In [16]:

```
diag_internas
```

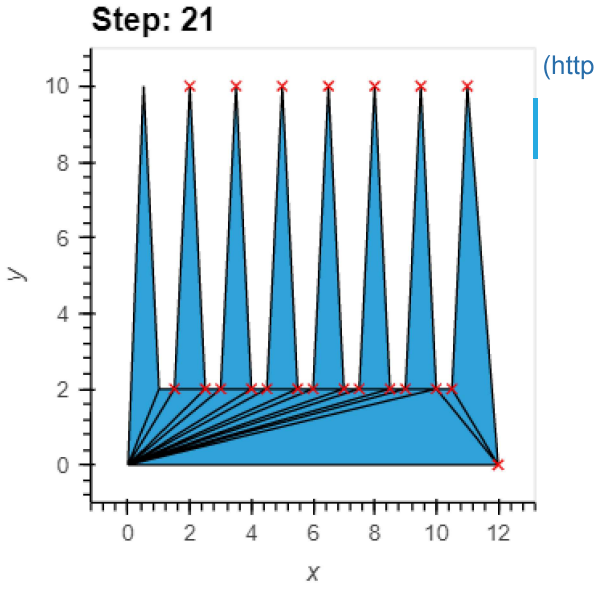
Out[16]:

```
[((12, 0), (10.5, 2)),
 ((12, 0), (10.0, 2)),
 ((0, 0), (10.0, 2)),
 ((10.0, 2), (9.0, 2)),
 ((0, 0), (9.0, 2)),
 ((0, 0), (8.5, 2)),
 ((8.5, 2), (7.5, 2)),
 ((0, 0), (7.5, 2)),
 ((0, 0), (7.0, 2)),
 ((7.0, 2), (6.0, 2)),
 ((0, 0), (6.0, 2)),
 ((0, 0), (5.5, 2)),
 ((5.5, 2), (4.5, 2)),
 ((0, 0), (4.5, 2)),
 ((0, 0), (4.0, 2)),
 ((4.0, 2), (3.0, 2)),
 ((0, 0), (3.0, 2)),
 ((0, 0), (2.5, 2)),
 ((2.5, 2), (1.5, 2)),
 ((0, 0), (1.5, 2)),
 ((0, 0), (1.0, 2))]
```

In [17]:

```
plots
```

Out[17]:



(http

Coloração

In [18]:

```
def gerar_grafo(triangulos,diag_internas):
    # Retorna um grafo como lista encadeada, com os índices dos triangulos como vértices do
    # arestas compartilhadas pelos triângulos, diagonais inernas do poligono, como arestas
    grafo = dict()
    for indt,tri in  enumerate(triangulos):
        for indd,diag in enumerate(diag_internas):
            if indt!=indd and diag[0] in tri and diag[1] in tri:
                try:
                    grafo[indt].append(indd)
                except:
                    grafo[indt] = [indd]
                try:
                    grafo[indd].append(indt)
                except:
                    grafo[indd] = [indt]
    return grafo
```

In [19]:

```
grafo = gerar_grafo(triangulos,diag_internas)
grafo
```

Out[19]:

```
{1: [0, 2],
 0: [1],
 2: [1, 4],
 4: [2, 3, 5],
 3: [4],
 5: [4, 7],
 7: [5, 6, 8],
 6: [7],
 8: [7, 10],
 10: [8, 9, 11],
 9: [10],
 11: [10, 13],
 13: [11, 12, 14],
 12: [13],
 14: [13, 16],
 16: [14, 15, 17],
 15: [16],
 17: [16, 19],
 19: [17, 18, 20],
 18: [19],
 20: [19, 21],
 21: [20]}
```

In [20]:

```
def Caminha(indatual,already,grafo,triangulos,cores):
    # Procedimento recursivo que caminha no grafo para cada triangulo através das arestas e
    # atribuindo cores aos pontos que ainda não possuem
    soma = 0
    for ponto in triangulos[indatual]:
        if ponto not in cores.keys():
            p = ponto
        else:
            soma+=cores[ponto]
    if soma==3:
        cores[p] = 3
    elif soma == 4:
        cores[p] = 2
    elif soma == 5:
        cores[p] = 1
    already.append(indatual)
    listatri = grafo[indatual]
    for i in listatri:
        if i not in already:
            Caminha(i,already,grafo,triangulos,cores)
    return
```

In [21]:

```
indatual = 0
cores = dict()
cores[triangulos[0][0]] = 1
cores[triangulos[0][1]] = 2
cores[triangulos[0][2]] = 3
already = []
Caminha(0,already,grafo,triangulos,cores)
```

In [22]:

```
def listar_cores(cores):
    # Agrupa os pontos de mesma cor e retorna as listas de pontos de mesma cor
    p_1 = []
    p_2 = []
    p_3 = []
    for p,c in cores.items():
        if c==1:
            p_1.append(p)
        if c==2:
            p_2.append(p)
        if c==3:
            p_3.append(p)
    return p_1,p_2,p_3
```

In [23]:

```
p_1,p_2,p_3 = listar_cores(cores)
p_1,p_2,p_3
```

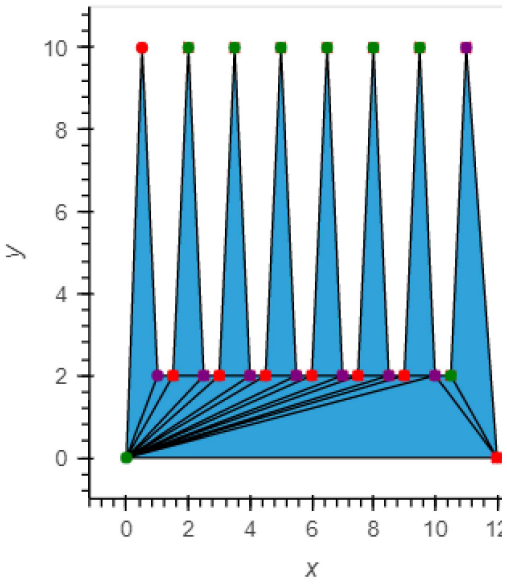
Out[23]:

```
([(12, 0),
  (9.0, 2),
  (7.5, 2),
  (6.0, 2),
  (4.5, 2),
  (3.0, 2),
  (1.5, 2),
  (0.5, 10)],
 [(11.0, 10),
  (10.0, 2),
  (8.5, 2),
  (7.0, 2),
  (5.5, 2),
  (4.0, 2),
  (2.5, 2),
  (1.0, 2)],
 [(10.5, 2),
  (0, 0),
  (9.5, 10),
  (8.0, 10),
  (6.5, 10),
  (5.0, 10),
  (3.5, 10),
  (2.0, 10)])
```

In [24]:

```
r_ = hv.Points(p_1).opts({'Points':{'color':'red','size':5}})
p_ = hv.Points(p_2).opts({'Points':{'color':'purple','size':5}})
g_ = hv.Points(p_3).opts({'Points':{'color':'green','size':5}})
pol_po = plots[len(triangulos)-1]*r_*g_*p_
pol_po
```

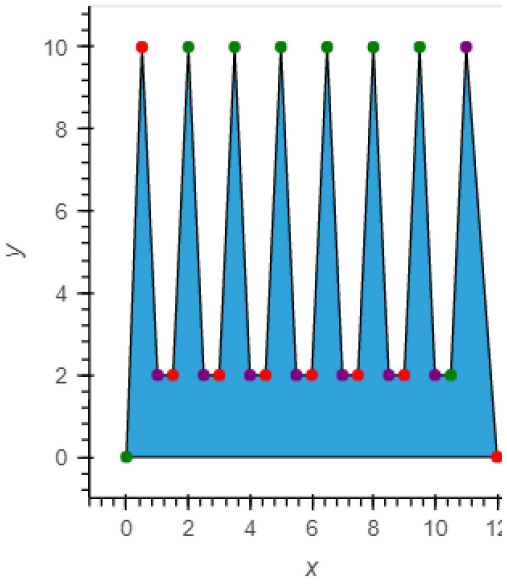
Out[24]:



In [25]:

```
po = plots[0]*r_*g_*p_
po
```

Out[25]:



In [26]:

```
sz1 = len(p_1)
sz2 = len(p_2)
sz3 = len(p_3)

if sz1 <= sz2 and sz1 <= sz3:
    print("São necessários",sz1,"câmeras para monitorar toda a galeria de arte, possíveis p
elif sz2 <= sz1 and sz2 <= sz3:
    print("São necessários",sz2,"câmeras para monitorar toda a galeria de arte, possíveis p
else:
    print("São necessários",sz3,"câmeras para monitorar toda a galeria de arte, possíveis p
```

```
São necessários 8 câmeras para monitorar toda a galeria de arte, possíveis p
ontos para instalá-las são [(12, 0), (9.0, 2), (7.5, 2), (6.0, 2), (4.5, 2),
(3.0, 2), (1.5, 2), (0.5, 10)]
```