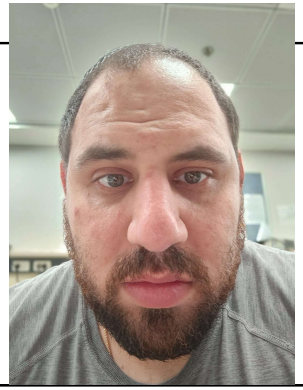
Course: Image Processing 31651
Assignment #11
Synthetic Image Creation (Part 1)-version 2

	ID (4 last digits)	Shorten Name	Photo of the student
Student #1	**1950**	shienfeld	
Student #2	**2210**	pony	
Student #3	**7939**	akimov	

Part 1 of Introduction to 11.1 – apply "checkValidation" function

```
13  bool checkValidation(s2dPoint p1, s2dPoint p2, unsigned char image[][NUMBER_OF_COLUMNS]) {
14      int top = max(p1.Y, p2.Y);
15      int bottom = min(p1.Y, p2.Y);
16      int left = min(p1.X, p2.X);
17      int right = max(p1.X, p2.X);
18      if ((0 > p1.X || NUMBER_OF_COLUMNS < p1.X) || (0 > p1.Y || NUMBER_OF_ROWS < p1.Y)) {
19          printf("Out of boundaries\n");
20          return false;
21      }
22      //printf("Place in boundaries\n");
23      for (int row = top; row < bottom; row++) {
24          for (int col = left; col < right; col++) {
25              if (image[row][col] != 255) {
26                  printf("This place is occupied\n");
27                  return false;
28              }
29          }
30      }
31      return true;
32  }
```

The checkValidation function is designed to verify whether a given rectangle, defined by two points (p1 and p2), lies within the boundaries of an image and that the region is not already occupied.

In the next slide a more thorough explanation about that function will be presented

Part 2 of Introduction to 11.1 – apply "checkValidation" function

In the next few slides we will present the "checkValidation" function a bit deeper

```
17  bool checkValidation(s2dPoint p1, s2dPoint p2, unsigned char image[][NUMBER_OF_COLUMNS])
18  {
```

First step- Parameters of the function:

s2dPoint p1 → is the first point defining one corner of the rectangle.

s2dPoint p2 → is the second point defining the opposite corner of the rectangle.

unsigned char image[][NUMBER_OF_COLUMNS] → is A 2D array representing the image.

The function returns a Boolean output : true if the rectangle is valid (within boundaries and not occupied), otherwise false.

```
return true;
```

Second step - define Rectangle Boundaries:

```
int top = max(p1.Y, p2.Y);
int bottom = min(p1.Y, p2.Y);
int left = min(p1.X, p2.X);
int right = max(p1.X, p2.X);
```

top: The maximum Y-coordinate between p1 and p2.
bottom: The minimum Y-coordinate between p1 and p2.
left: The minimum X-coordinate between p1 and p2.
right: The maximum X-coordinate between p1 and p2.

Part 3 of Introduction to 11.1 – apply “checkValidation” function

third step – check if a defined point is in or out of bounds:

```
23      if ((0 > p1.X || NUMBER_OF_COLUMNS < p1.X) || (0 > p1.Y || NUMBER_OF_ROWS < p1.Y))
24      {
25          printf("Out of boundaries\n");
26          return false;
27      }
```

This part checks if either point p1 is outside the image boundaries. If p1 is out of bounds, it prints "Out of boundaries" and returns false. If within bounds, it prints "Place in boundaries".

fourth step – checking occupancy of pixels within the rectangle:

```
29      for (int row = top; row < bottom; row++)
30      {
31          for (int col = left; col < right; col++)
32          {
33              if (image[row][col] != 255)
34              {
35                  printf("This place is occupied\n");
36                  return false;
37              }
38          }
39      }
40      return true;
41  }
```

This loop iterates over the region defined by the rectangle:
Outer loop: Iterates from top to bottom (Y-coordinates).
Inner loop: Iterates from left to right (X-coordinates).
It checks if any pixel within the rectangle is not equal to 255 (assuming 255 represents an unoccupied pixel in a grayscale image).
If it finds an occupied pixel, it prints "This place is occupied" and returns false.

Part 4 of Introduction to 11.1 – apply “s2dpoint” struct (1/2)


```
136  struct s2dPoint
137  {
138      int X, Y;
139
140      // Constructor with default values and validation
141      s2dPoint(int x = 0, int y = 0) : X(x), Y(y)
142      {
143          validate();
144      }
145
146      // Validation function
147      void validate()
148      {
149
150          if (X < 0 || X > NUMBER_OF_COLUMNS)
151          {
152
153
154              if (X < 0)
155                  X = 0;
156              if (X > NUMBER_OF_COLUMNS)
157                  X = NUMBER_OF_COLUMNS;
158
159          }
```

```
160          if (Y < 0 || Y > NUMBER_OF_ROWS)
161          {
162
163
164              if (Y < 0)
165                  Y = 0;
166              if (Y > NUMBER_OF_ROWS)
167                  Y = NUMBER_OF_ROWS;
168
169          }
170
171      }
```

These are the coordinates of the point. X represents the horizontal coordinate, and Y represents the vertical coordinate.

This is a constructor for the struct. It initializes the member variables X and Y with the values provided (defaulting to 0 if no values are given). After initialization, it calls the validate method to ensure the coordinates are within the allowed range.

The **validate** function ensures that the coordinates X and Y stay within the bounds defined by NUMBER_OF_COLUMNS and NUMBER_OF_ROWS.
If X is less than 0, it is set to 0. If X is greater than NUMBER_OF_COLUMNS, it is set to NUMBER_OF_COLUMNS.
The same logic applies to Y with respect to NUMBER_OF_ROWS.

Pay attention that the s2dPoint struct is defined at the header file to represent a 2D point with X and Y coordinates.

Part 4 of Introduction to 11.1 – apply "s2dpoint" struct (2/2)

```
173     // Example method to display the point
174     void display() const
175     {
176         std::cout << "Point (" << X << ", " << Y << ")" << std::endl;
177     }
178
179     // Setters with validation
180     void setX(int x)
181     {
182         X = x;
183         validate();
184     }
185
186     void setY(int y)
187     {
188         Y = y;
189         validate();
190     }
191
192     void setXY(int x, int y)
193     {
194         X = x;
195         Y = y;
196         validate();
197     }
198 };
```

The display function prints the coordinates of the point to the standard output. The const keyword indicates that this function does not modify the member variables of the struct.

Setters are functions that allow you to change the values of member variables while ensuring they remain valid.

setX(int x): Sets the X coordinate and validates it.
setY(int y): Sets the Y coordinate and validates it.
setXY(int x, int y): Sets both X and Y coordinates and validates them.

Each part of the s2dpoint struct - ensures that the point's coordinates are always valid and provides a way to manipulate and display the point's data.

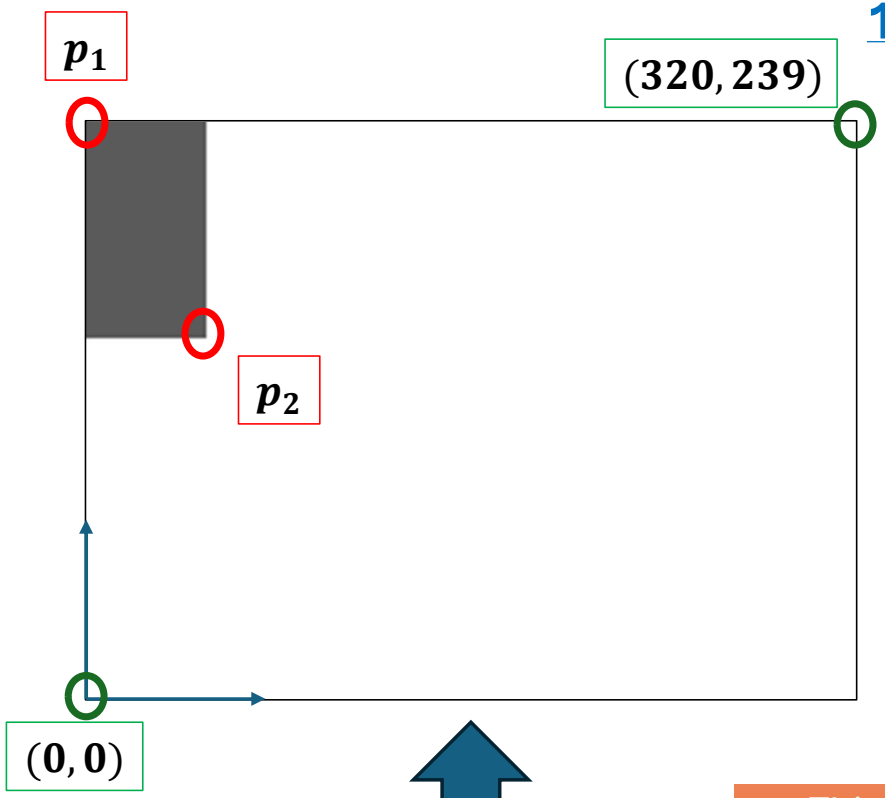
11.1 Code of the function “AddGrayRectangle”

Pay attention that in the code that For each pixel, blend the new gray level using the formula:
image[row][col] = static_cast<unsigned char>(transparency * (image[row][col] / 255.0) + (255 - transparency) * (grayLevel / 255.0));

```
46  // Function Implementation;
47  void AddGrayRectangle(unsigned char image[][NUMBER_OF_COLUMNS], s2dPoint A,
48                        s2dPoint B1, unsigned char transparency, unsigned char grayLevel) {
49      // Ensure coordinates are within bounds and place is not occupied
50      if (!checkValidation(A, B1, image)) {
51          return;
52      }
53
54      int top = max(A.Y, B1.Y);
55      int bottom = min(A.Y, B1.Y);
56      int left = min(A.X, B1.X);
57      int right = max(A.X, B1.X);
58
59      // Apply blending technique to the region of the rectangle
60      for (int row = max(bottom, 0); row < min(top, NUMBER_OF_ROWS); row++) {
61          for (int col = max(left, 0); col < min(right, NUMBER_OF_COLUMNS); col++) {
62              image[row][col] = static_cast<unsigned char>(transparency * (image[row][col] / 255.0)
63                  + (255 - transparency) * (grayLevel / 255.0));
64          }
65      }
66  }
```

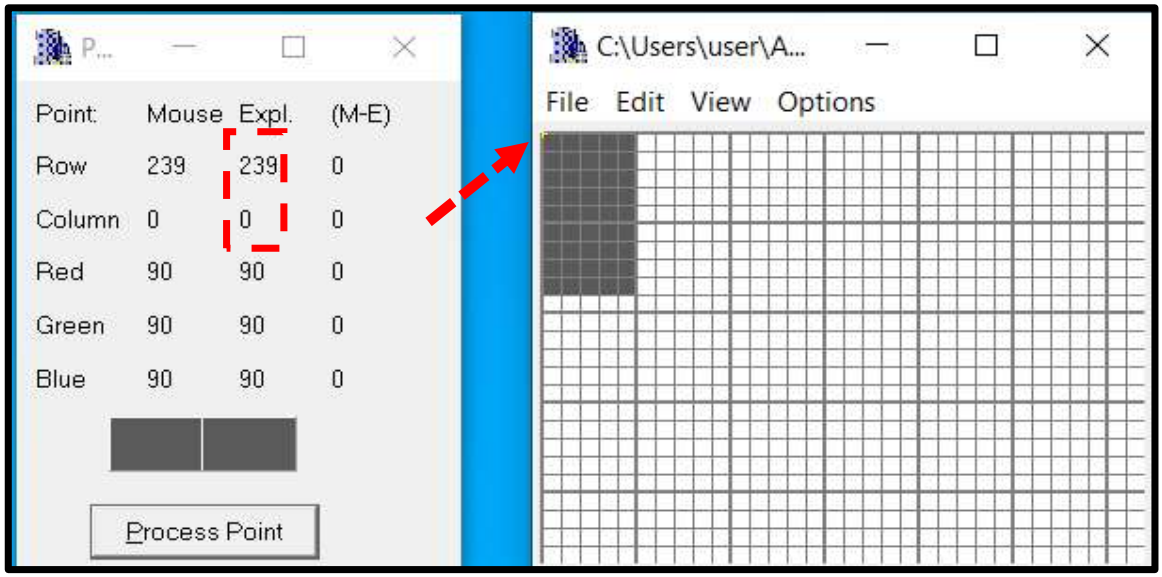
The AddGrayRectangle function ensures that the specified rectangle is within valid boundaries and not occupied before adding it to the image. It calculates the top, bottom, left, and right boundaries of the rectangle and then applies a blending technique to each pixel within these boundaries to incorporate the rectangle with the specified gray level and transparency.

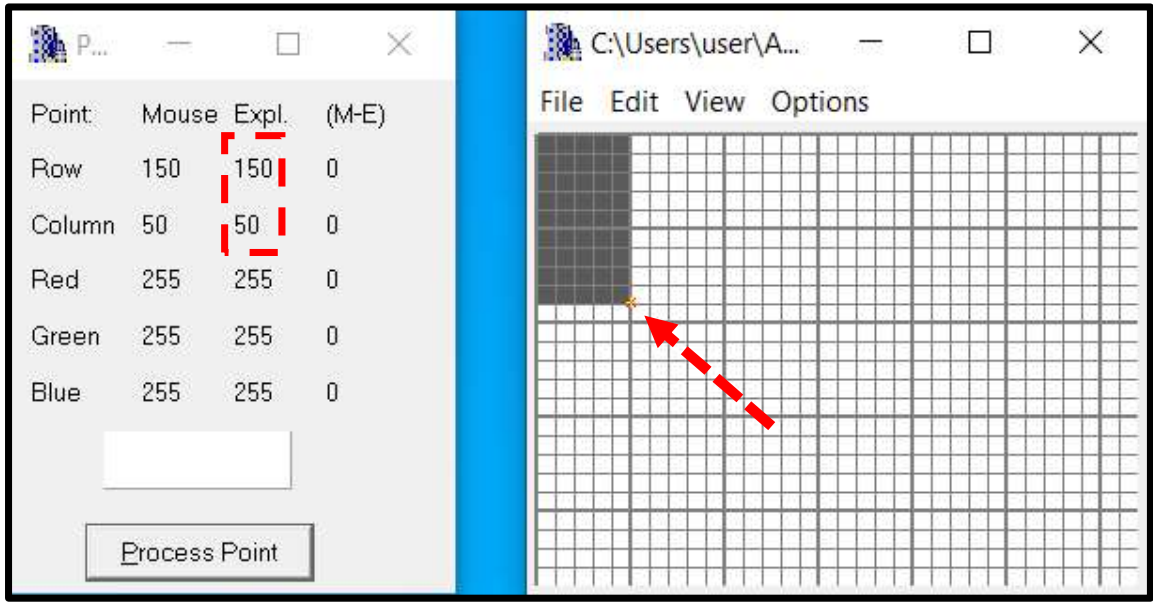
11.2 List of situations tested – part 1



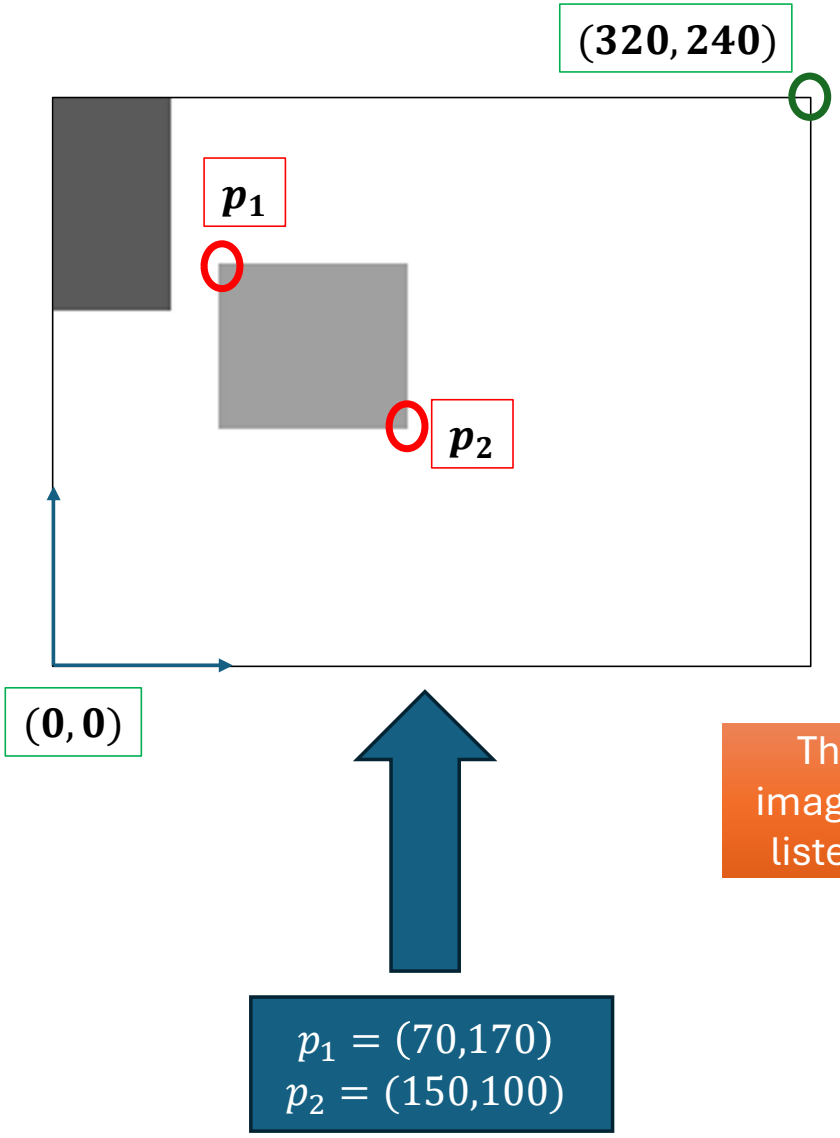
$p_1 = (0,240)$
$p_2 = (50,150)$

This test image will be listed as #1



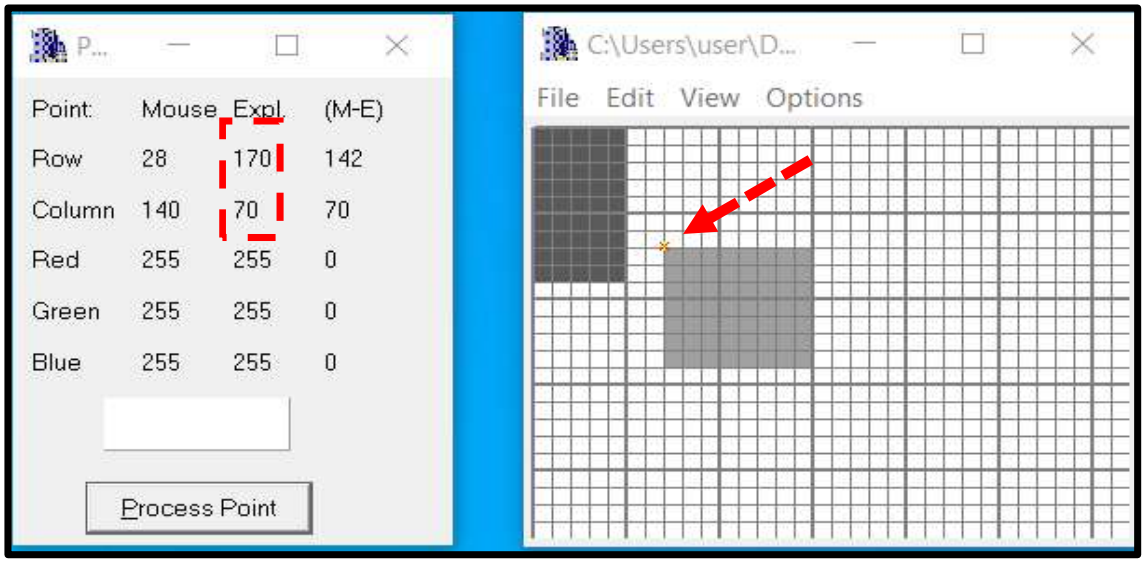


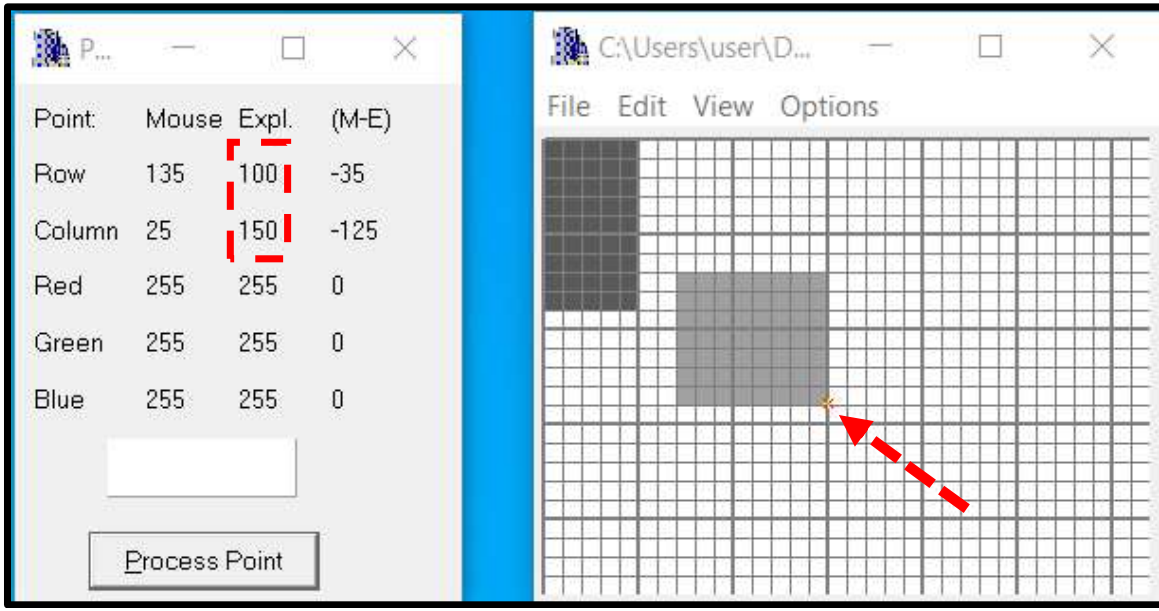
11.2 List of situations tested – part 2



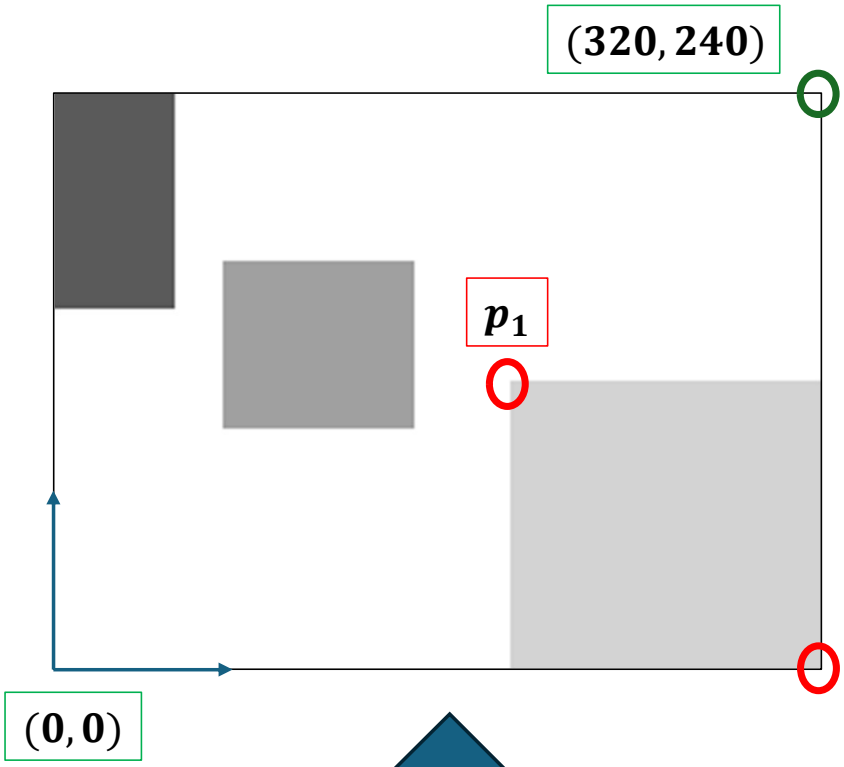
$p_1 = (70,170)$
$p_2 = (150,100)$

This test image will be listed as #2



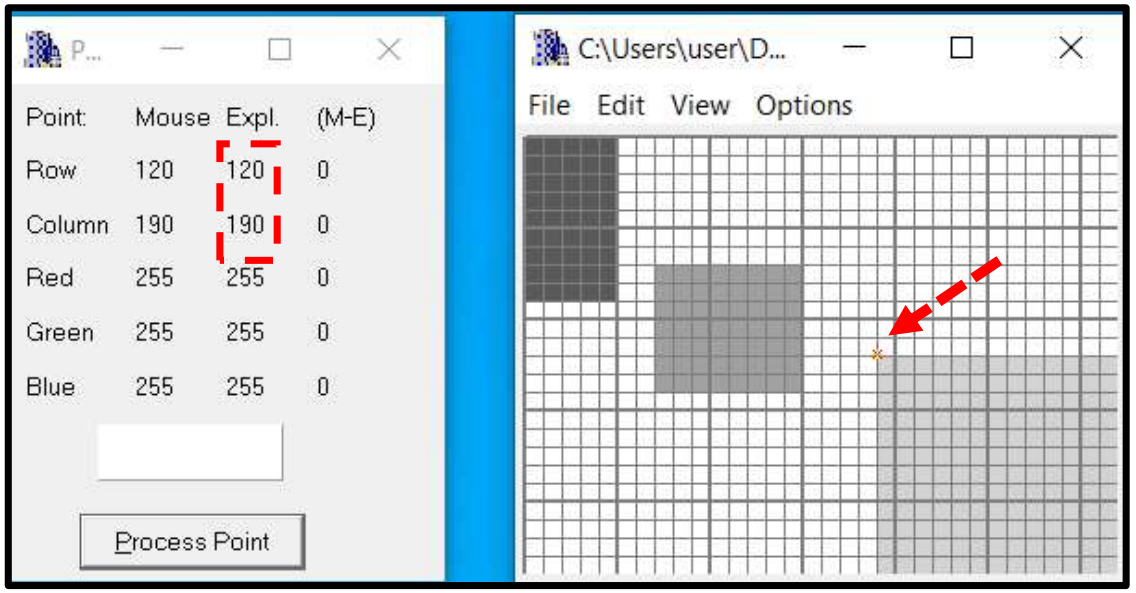


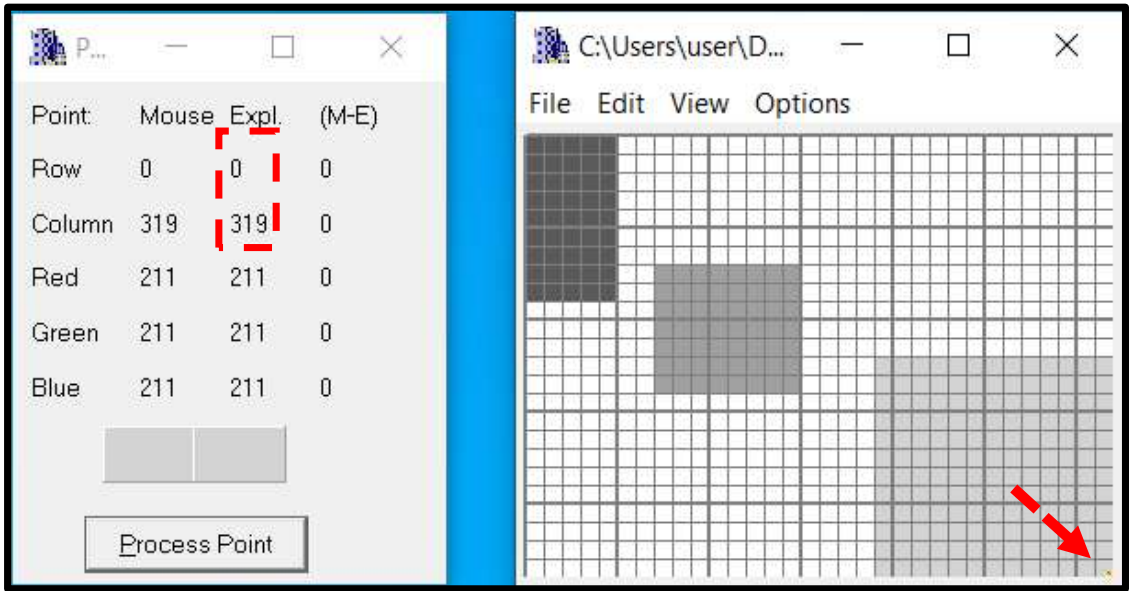
11.2 List of situations tested – part 3



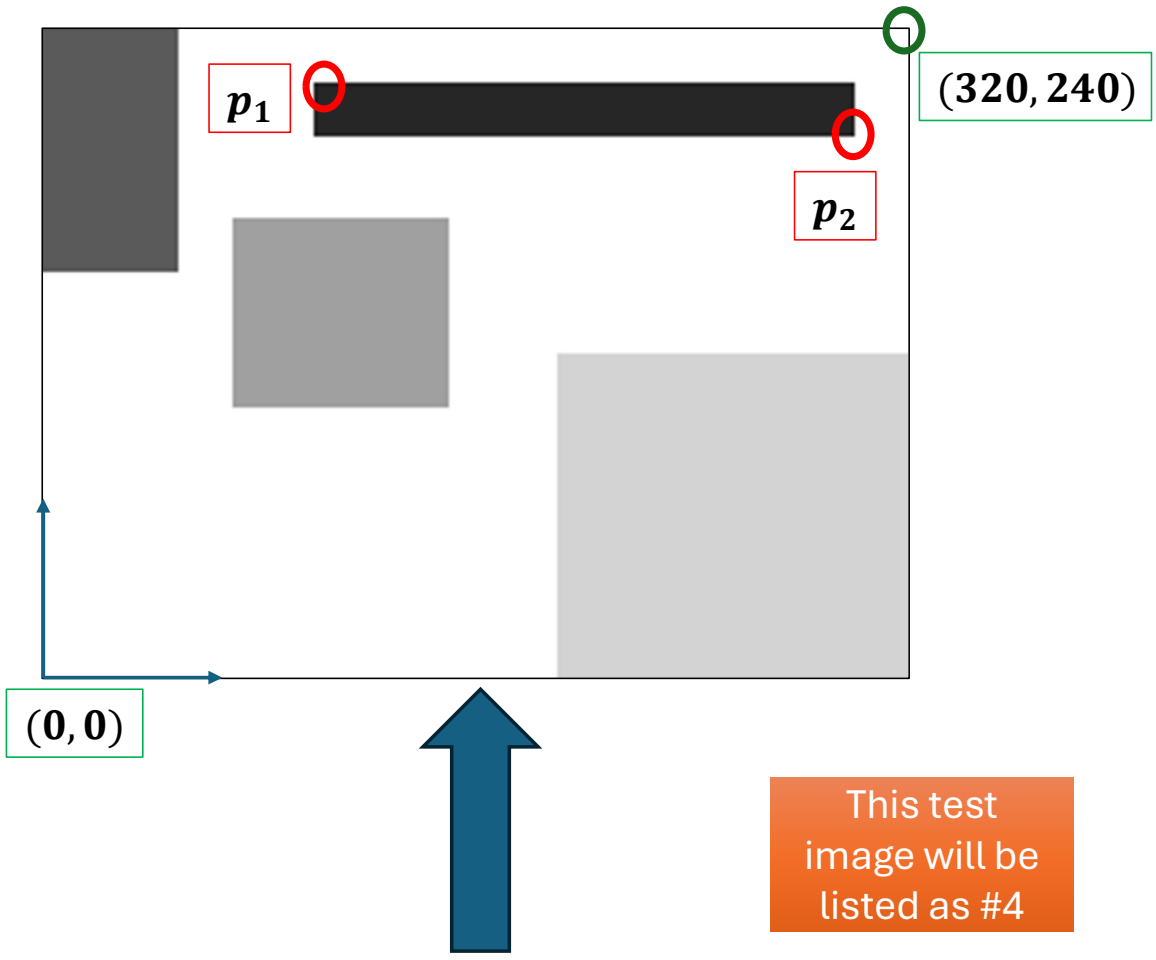
$p_1 = (190,120)$
$p_2 = (320,0)$

This test image will be listed as #3



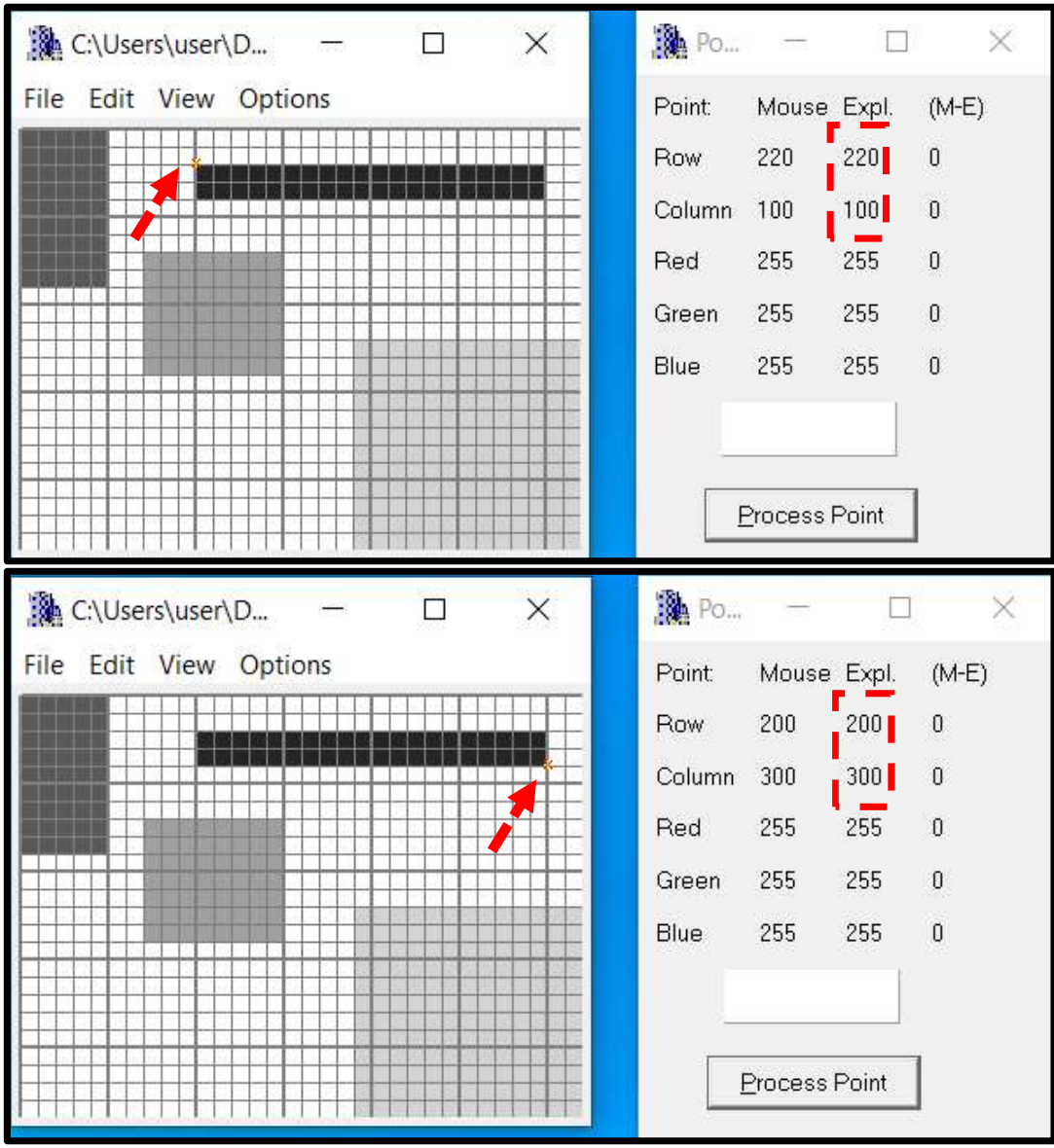


11.2 List of situations tested – part 4

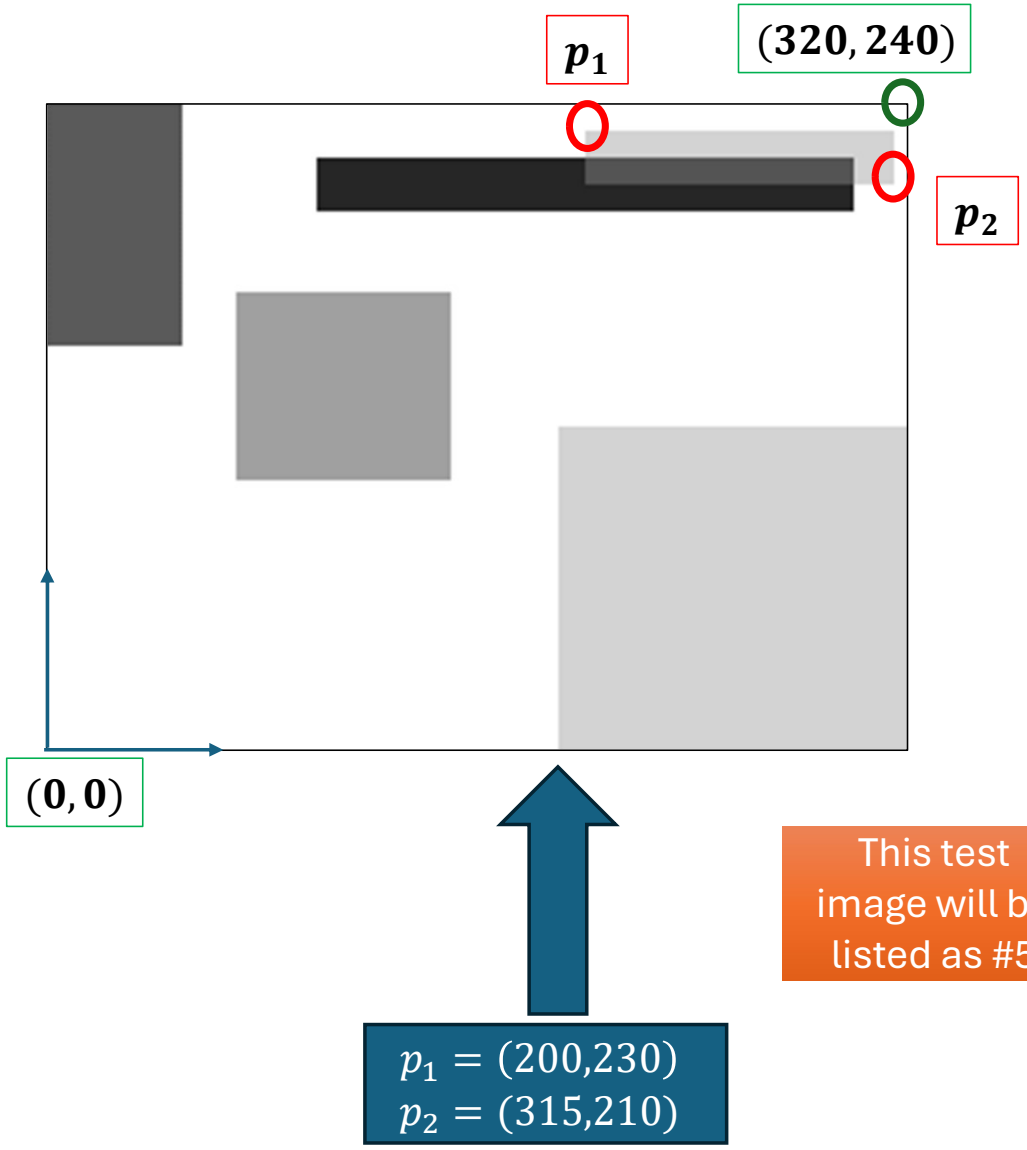


$p_1 = (100,220)$
$p_2 = (300,200)$

This test image will be listed as #4

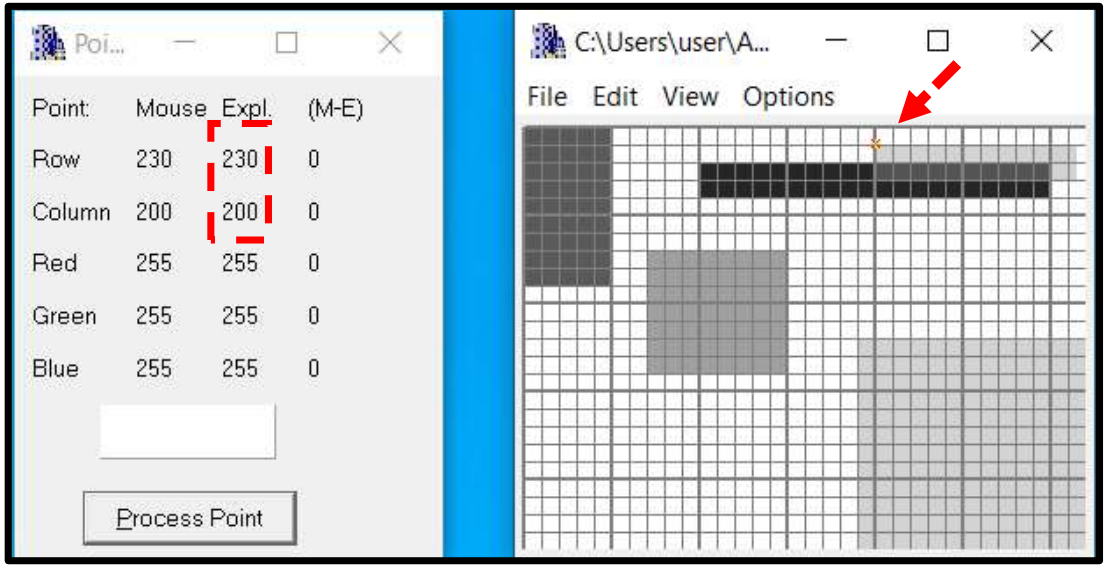


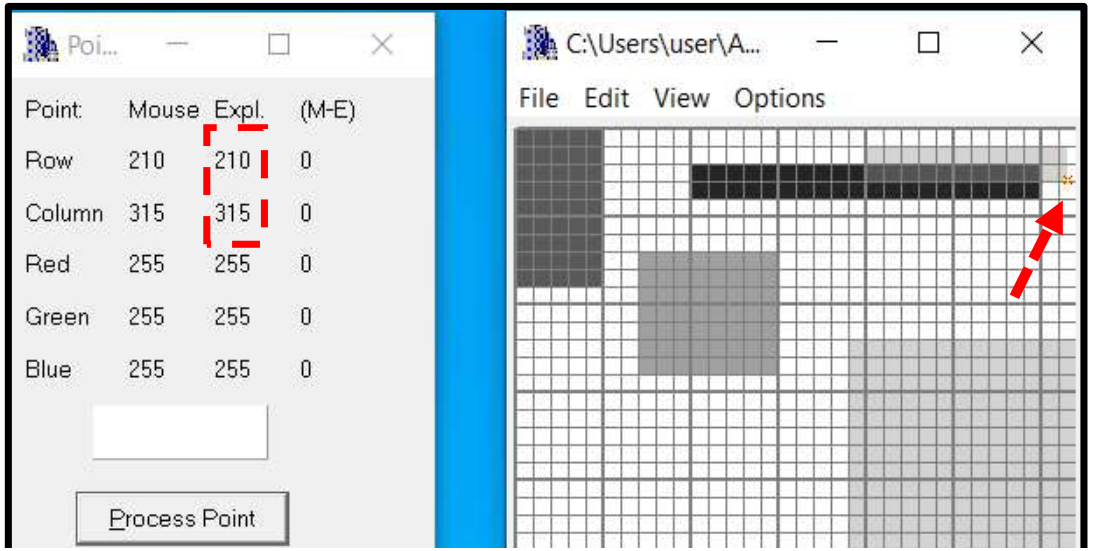
11.2 List of situations tested – part 5



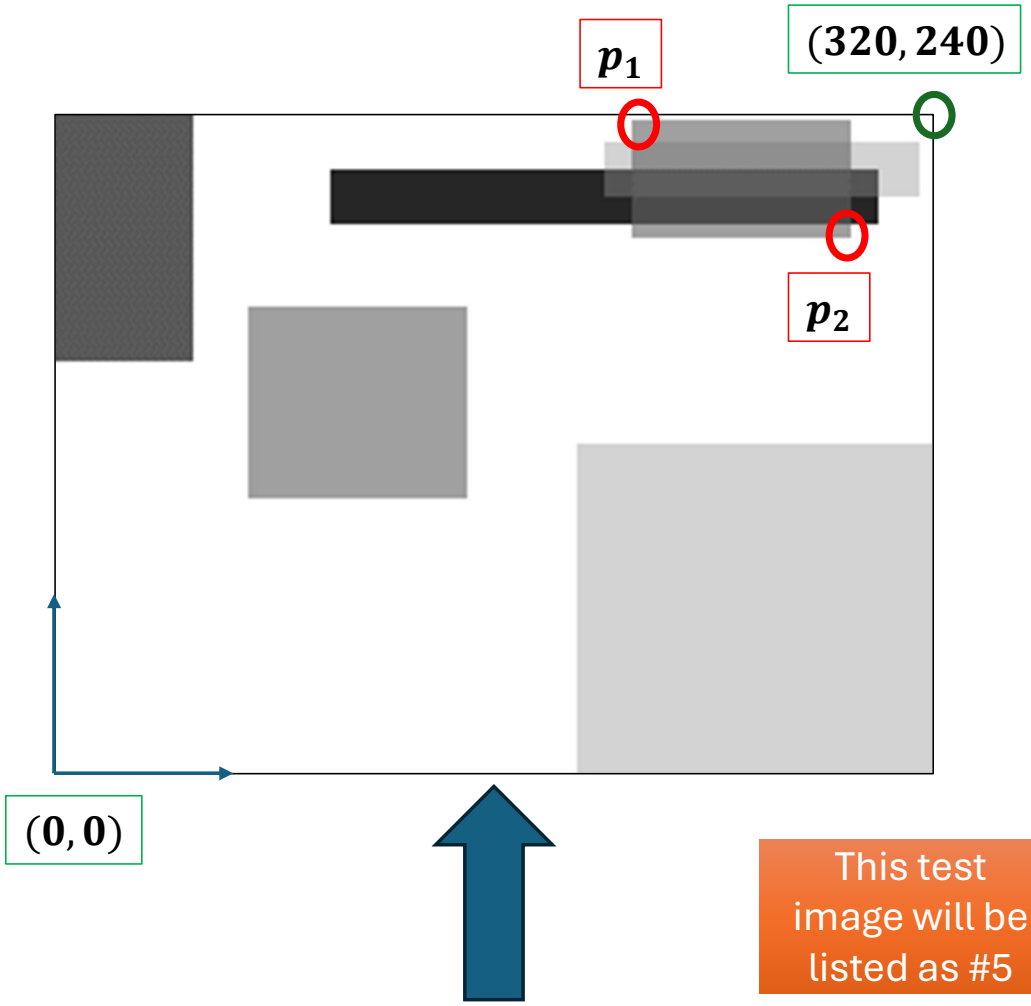
This test image will be listed as #5

$p_1 = (200,230)$
$p_2 = (315,210)$



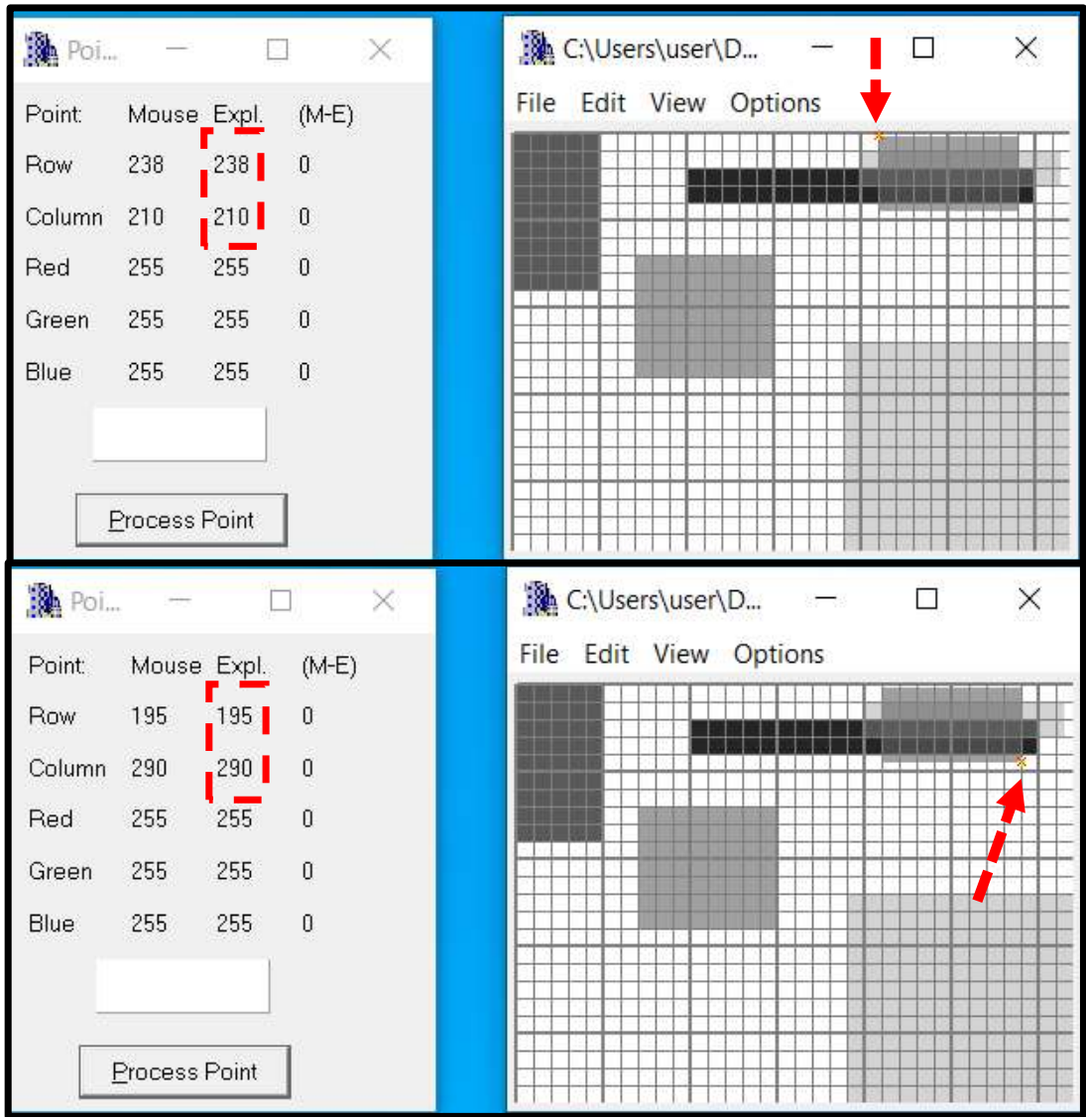


11.2 List of situations tested – part 6

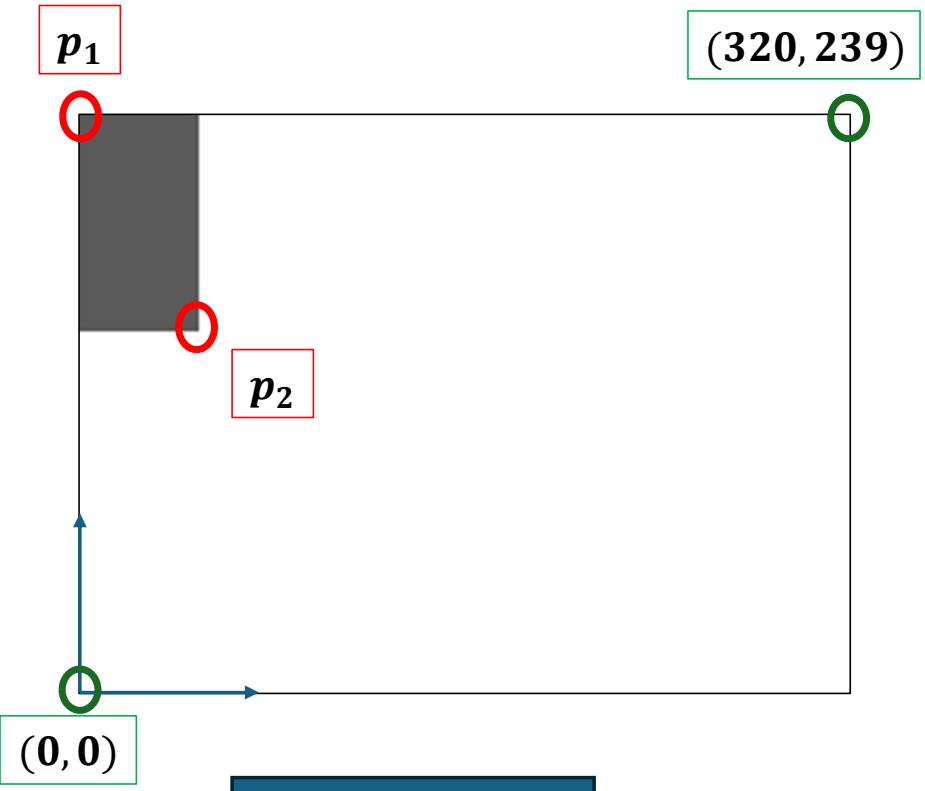


$p_1 = (210,238)$
$p_2 = (290,195)$

This test image will be listed as #5

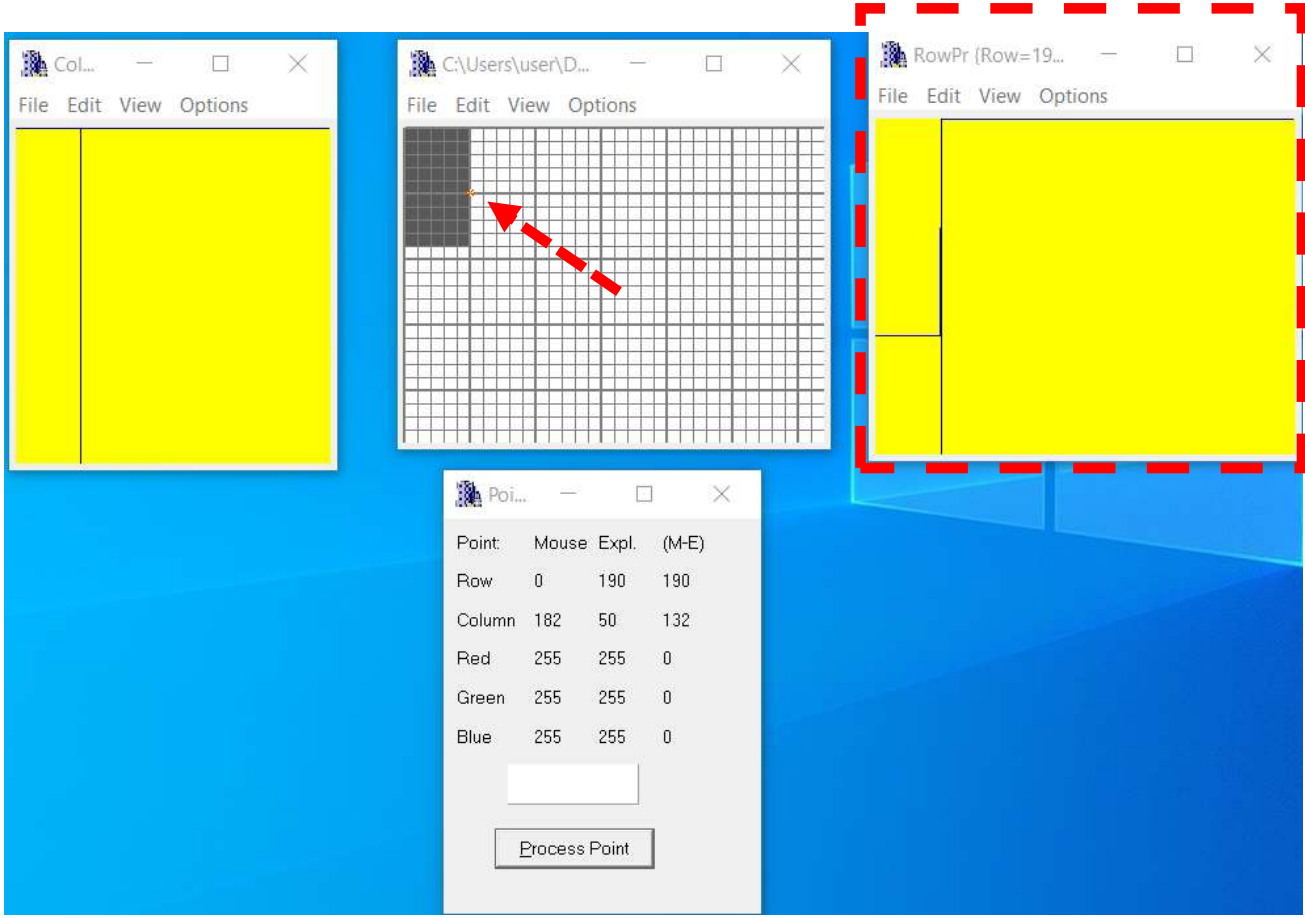


11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 1

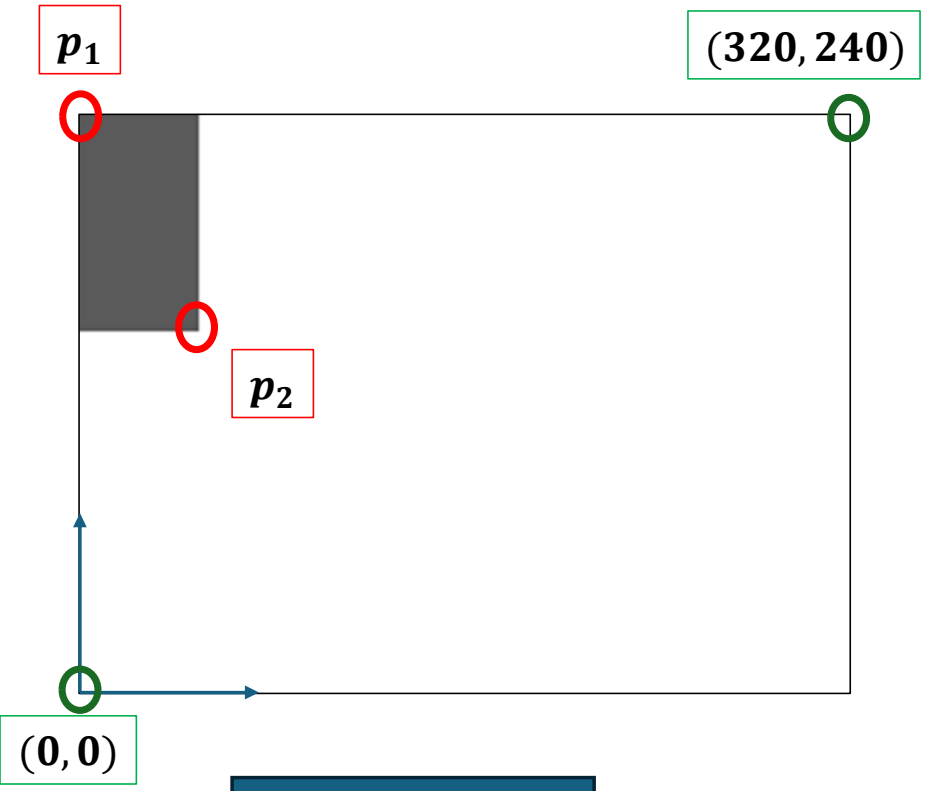


$p_1 = (0,239)$
$p_2 = (50,150)$

Regarding **test image #1** – we look at the border of the rectangle and can see a clear step (like a step function) in the rows view.

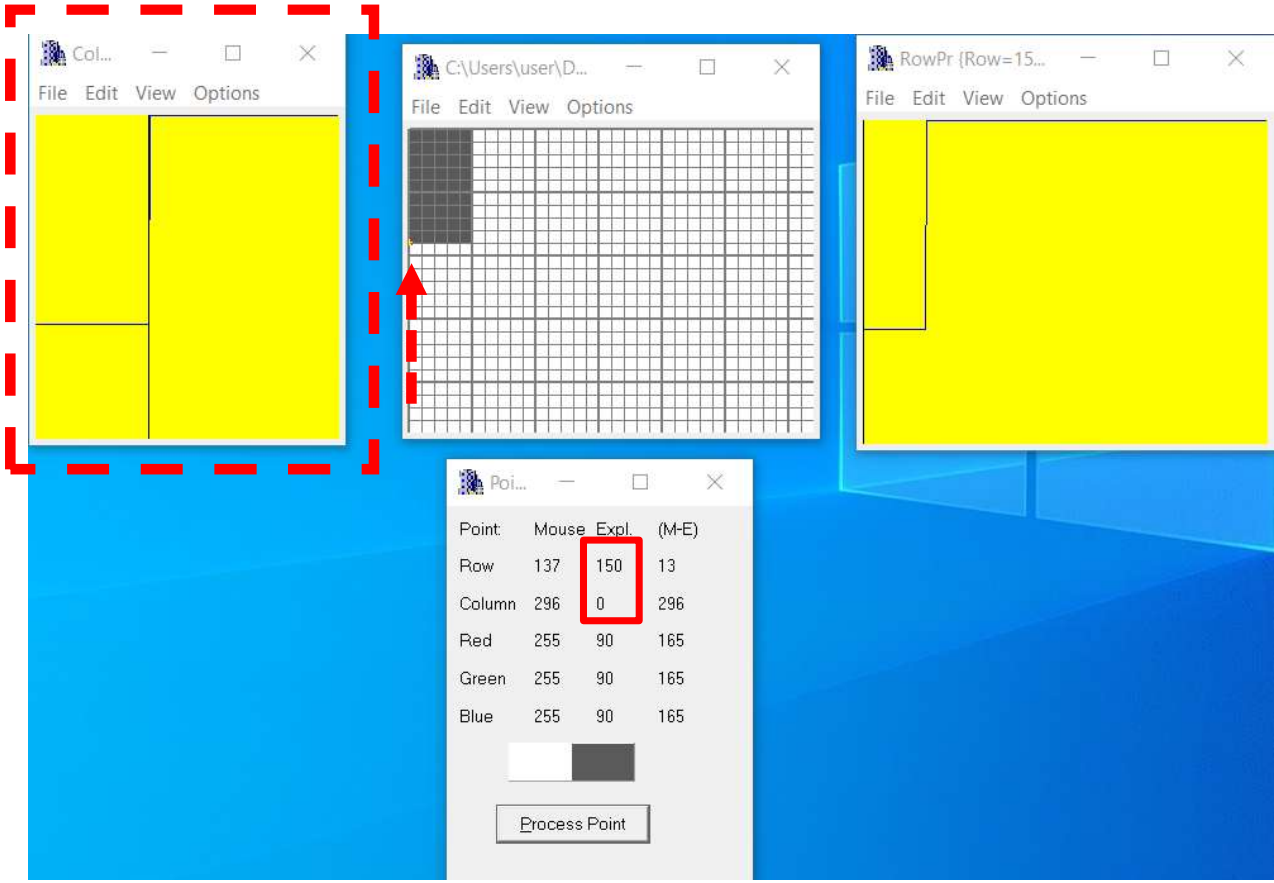


11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 2

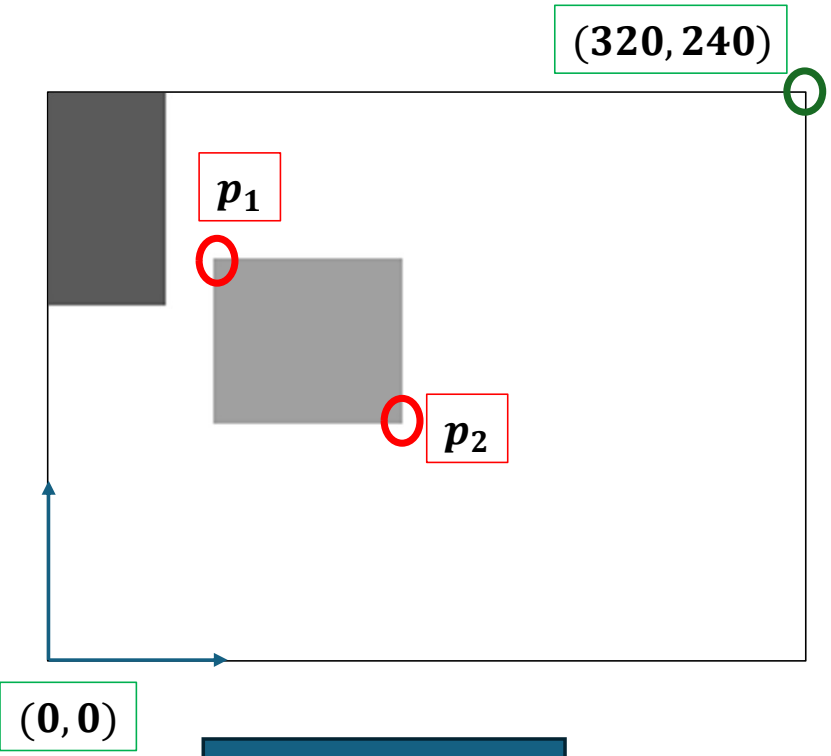


$p_1 = (0,239)$
$p_2 = (50,150)$

Regarding **test image #1** – we look at the border of the rectangle and can see a clear step (like a step function) in the columns view.

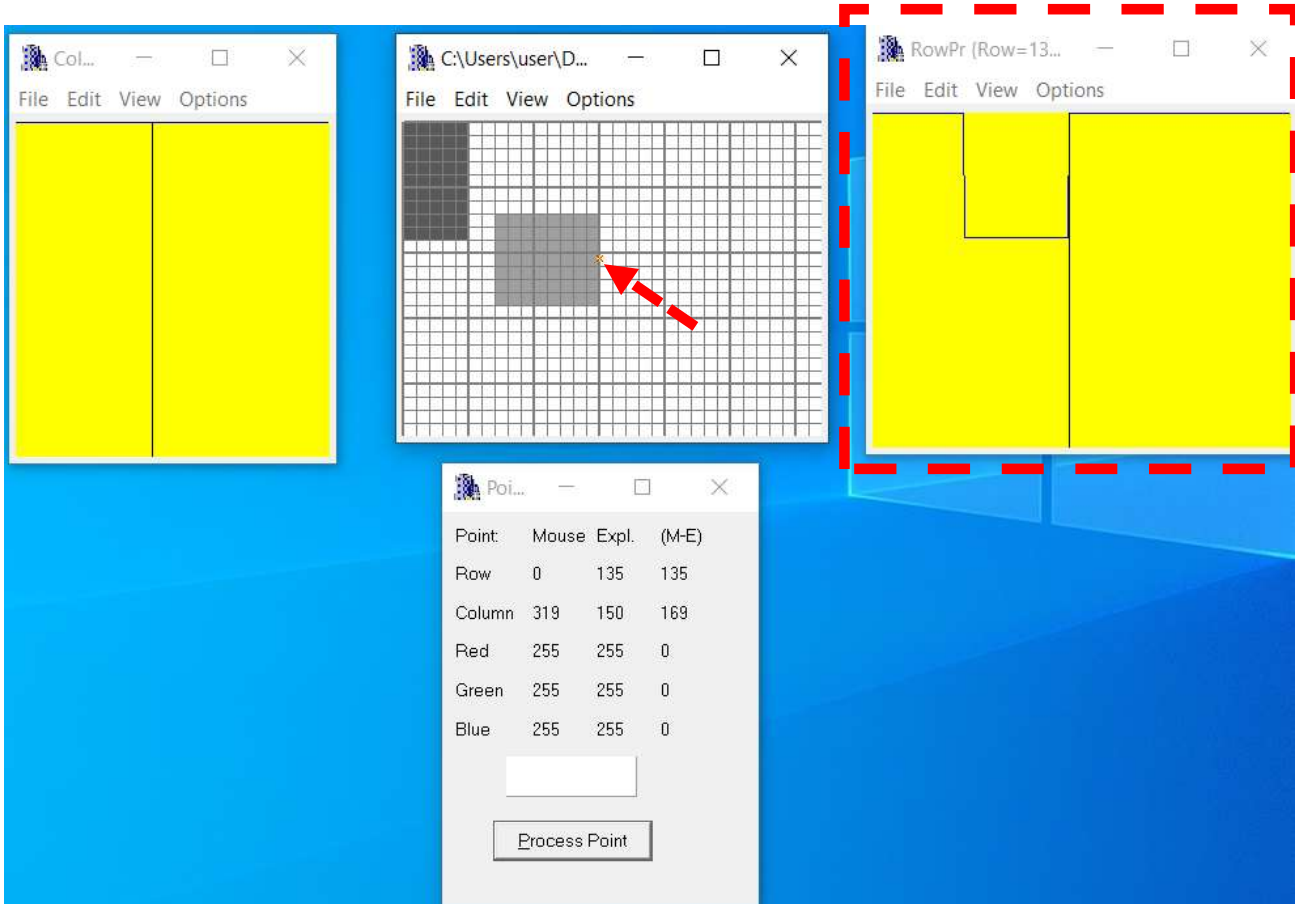


11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 3

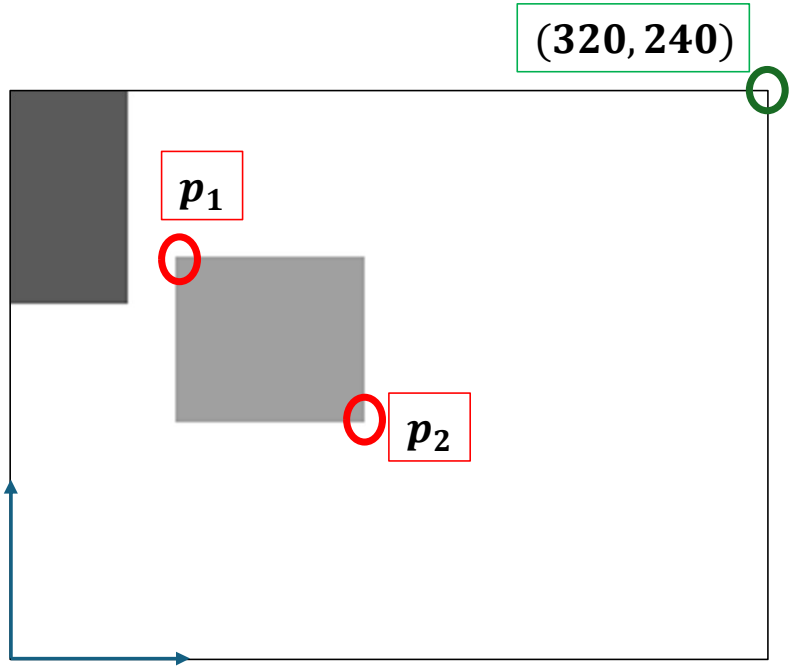


$p_1 = (70,170)$
$p_2 = (150,100)$

Regarding **test image #2** – we look at the border of the rectangle and can see a clear hole (like a potential hole) in the rows view. The hole is between the borders of the second rectangle

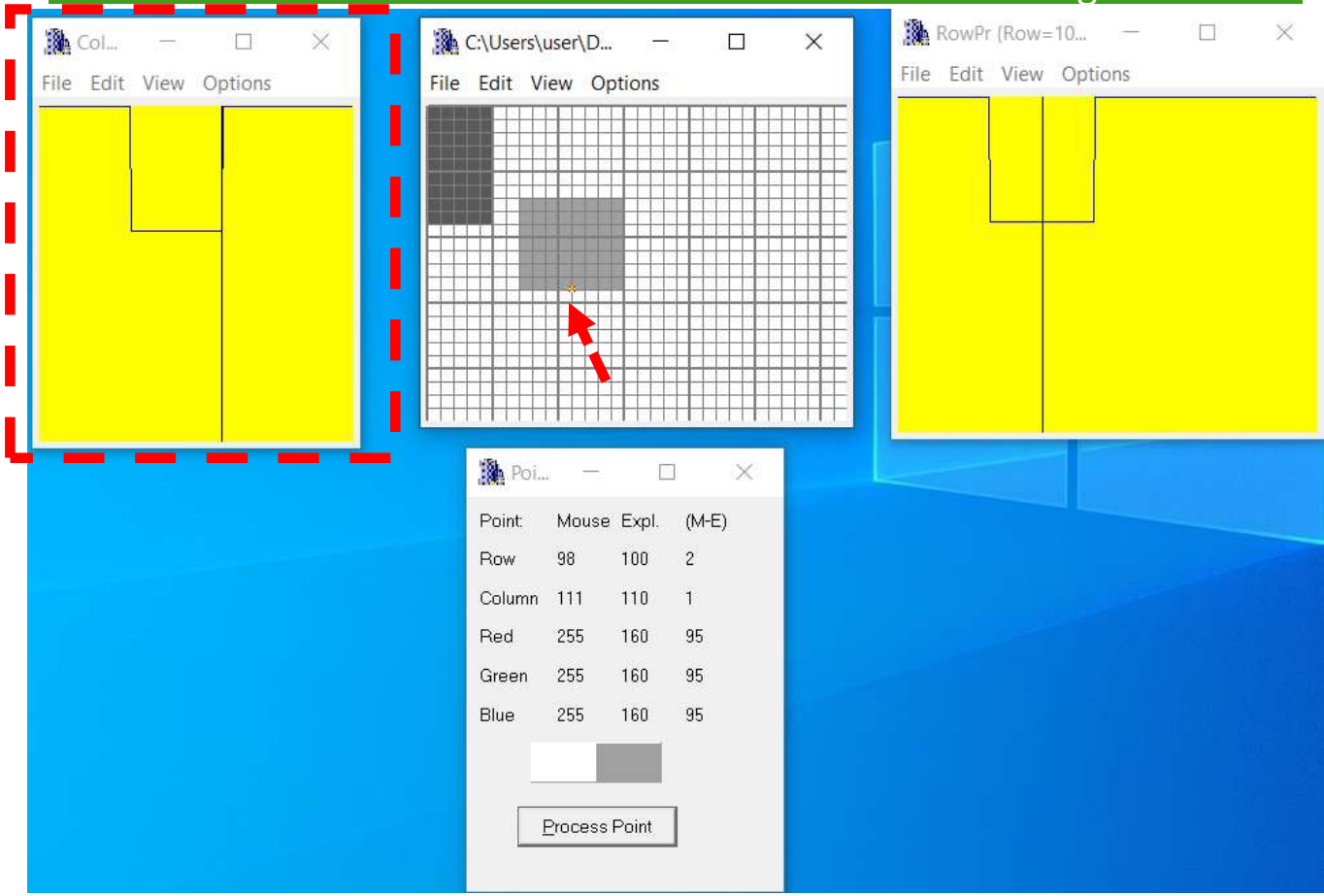


11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 4

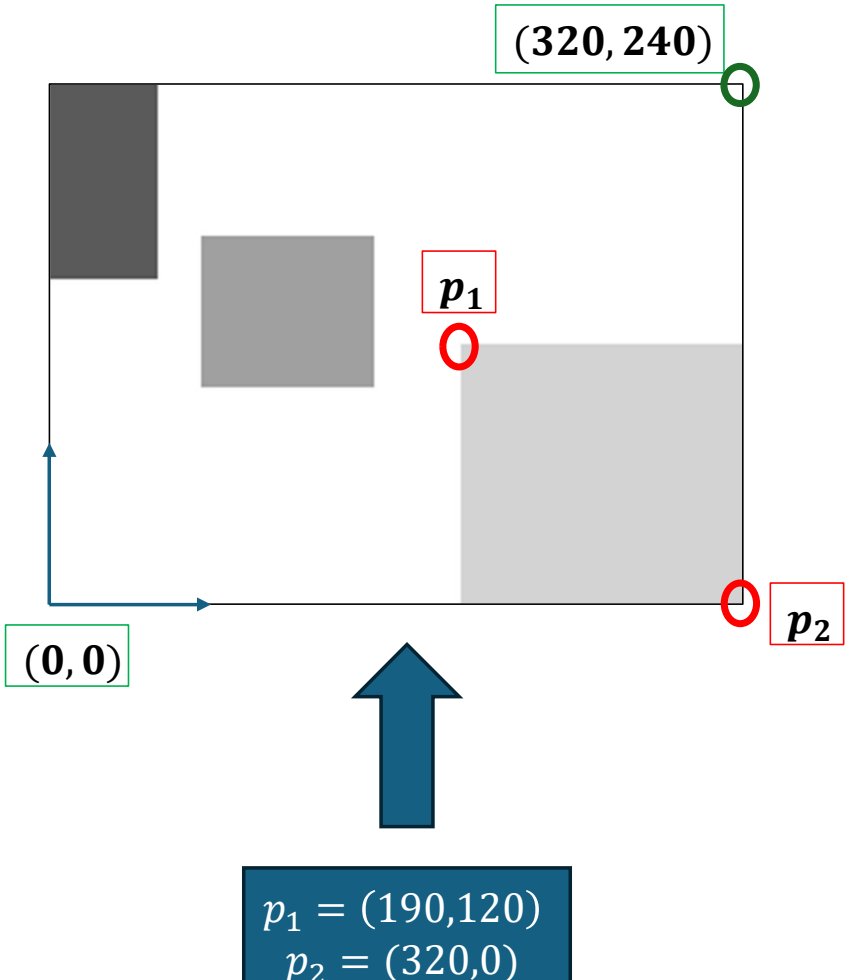


$p_1 = (70,170)$
$p_2 = (150,100)$

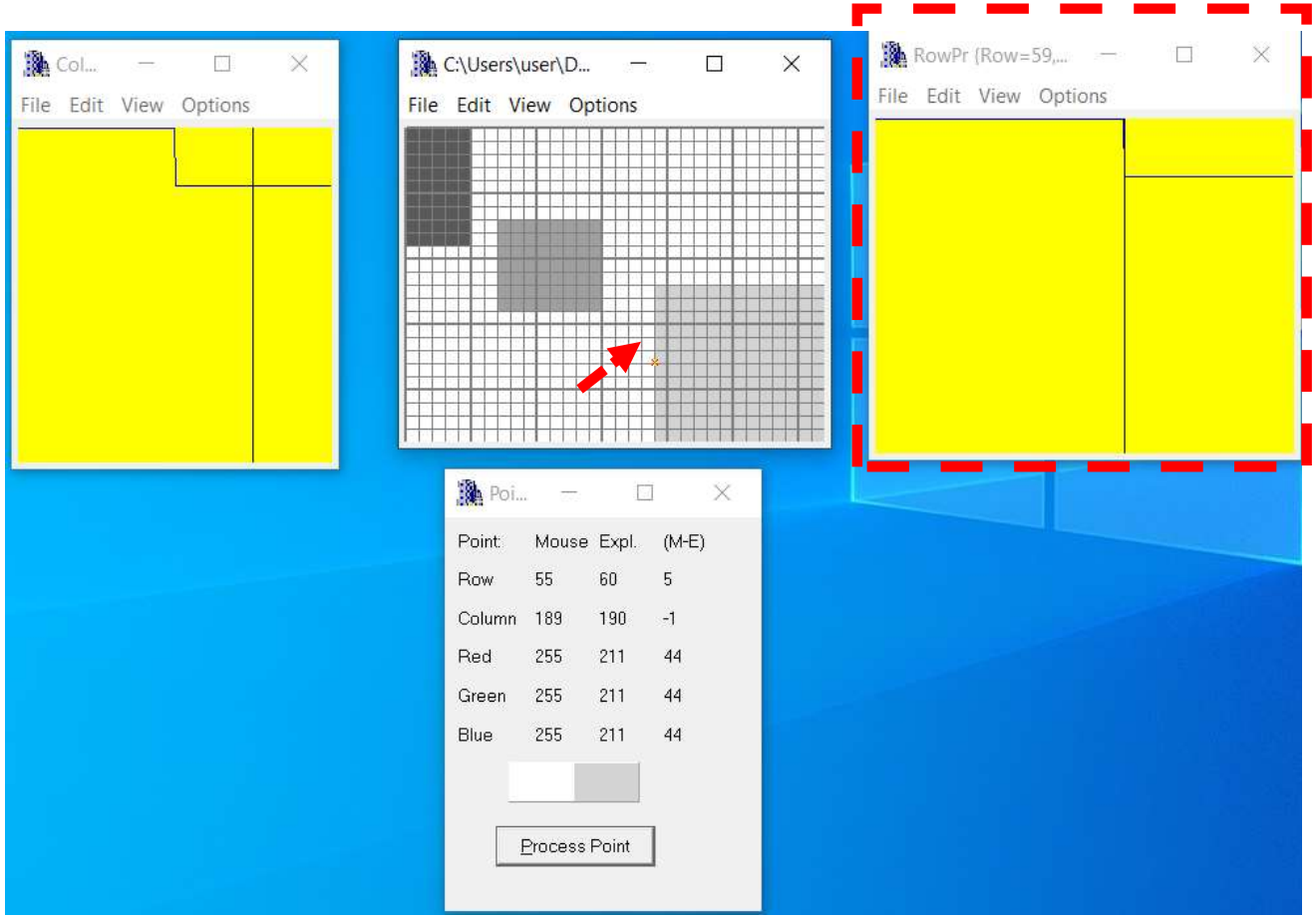
Regarding **test image #2** – we look at the border of the rectangle and can see a clear hole (like a potential hole) in the columns view. The hole is between the borders of the second rectangle



11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 5

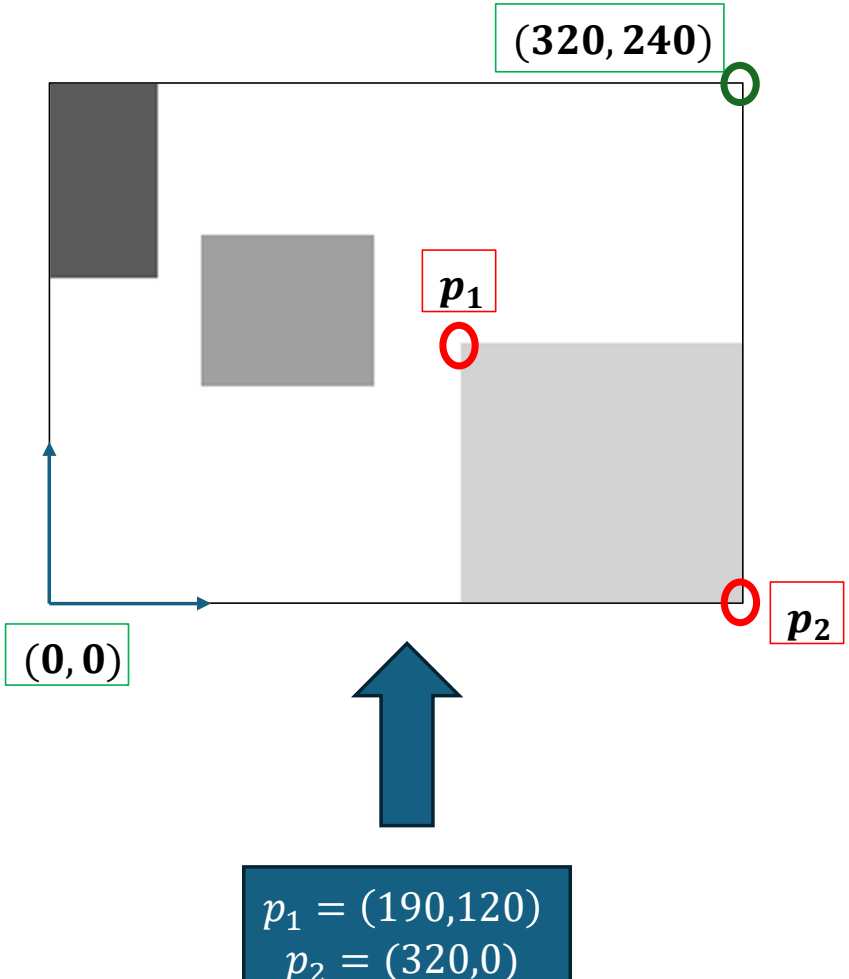


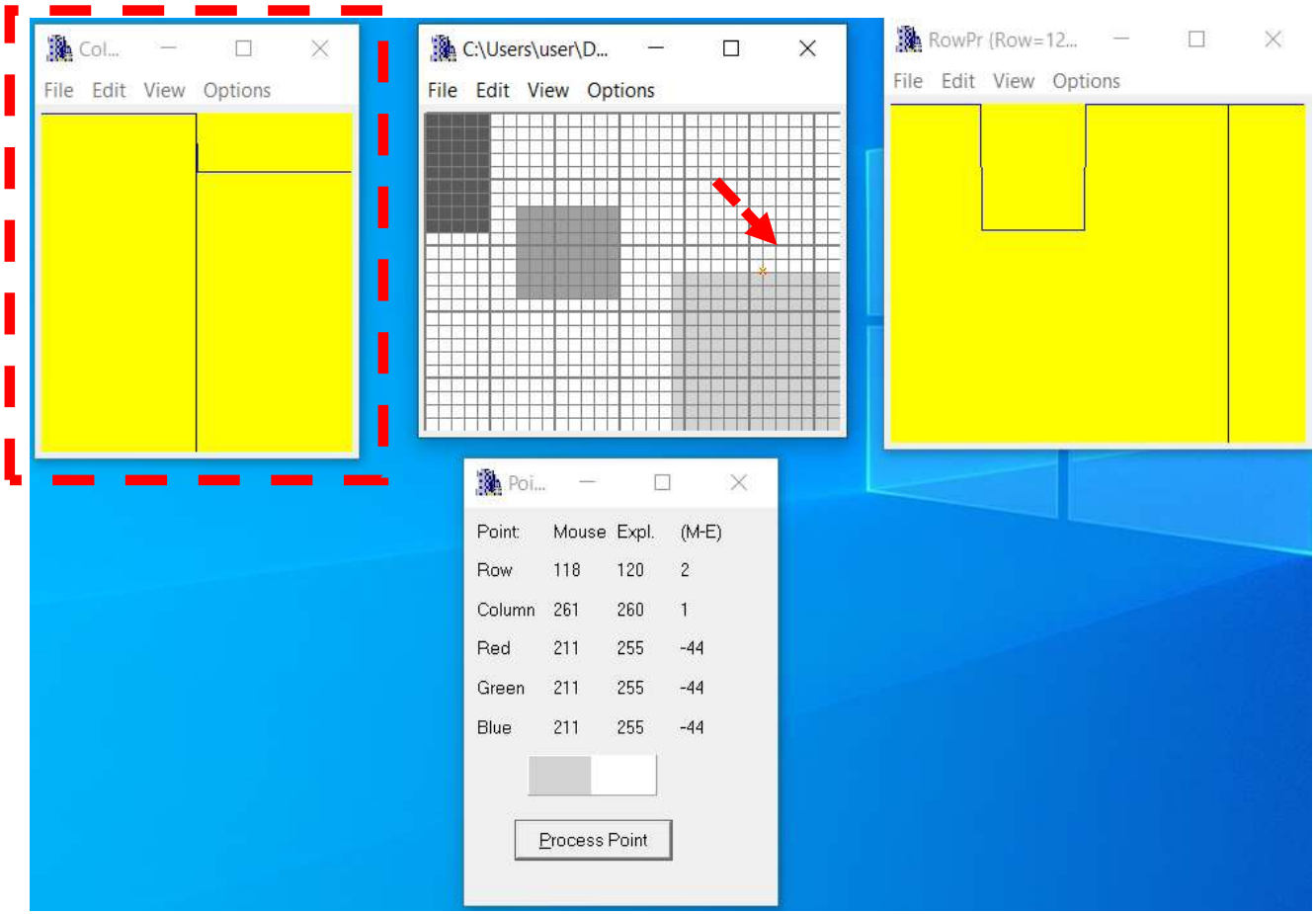
Regarding **test image #3** – we look at the border of the rectangle and can see a step small in the rows view. The step is between the borders of the third rectangle



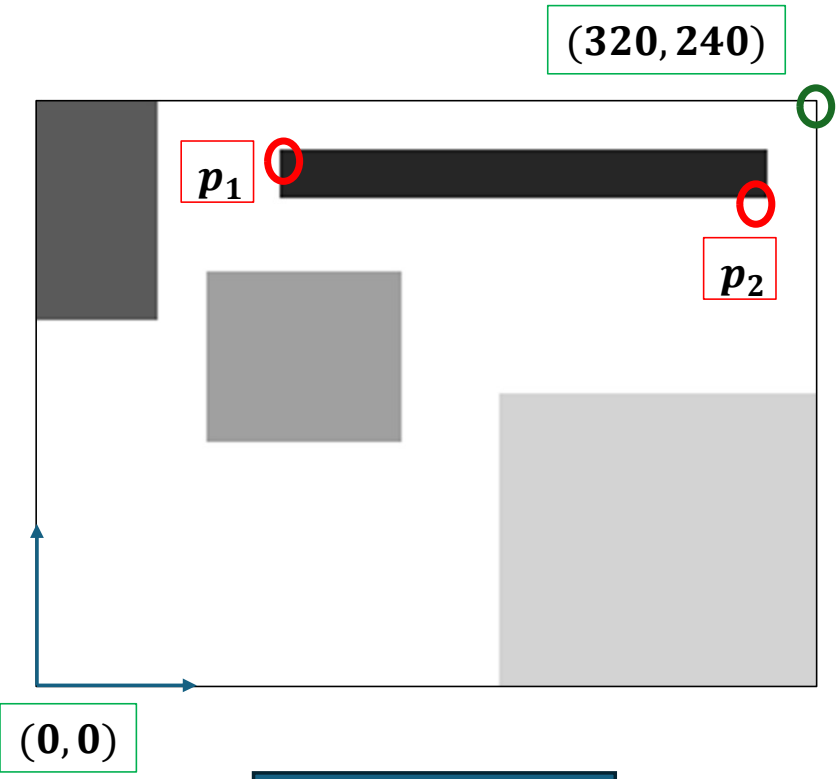
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 6

Regarding **test image #3** – we look at the border of the rectangle and can see a step small in the columns view. The step is between the borders of the third rectangle

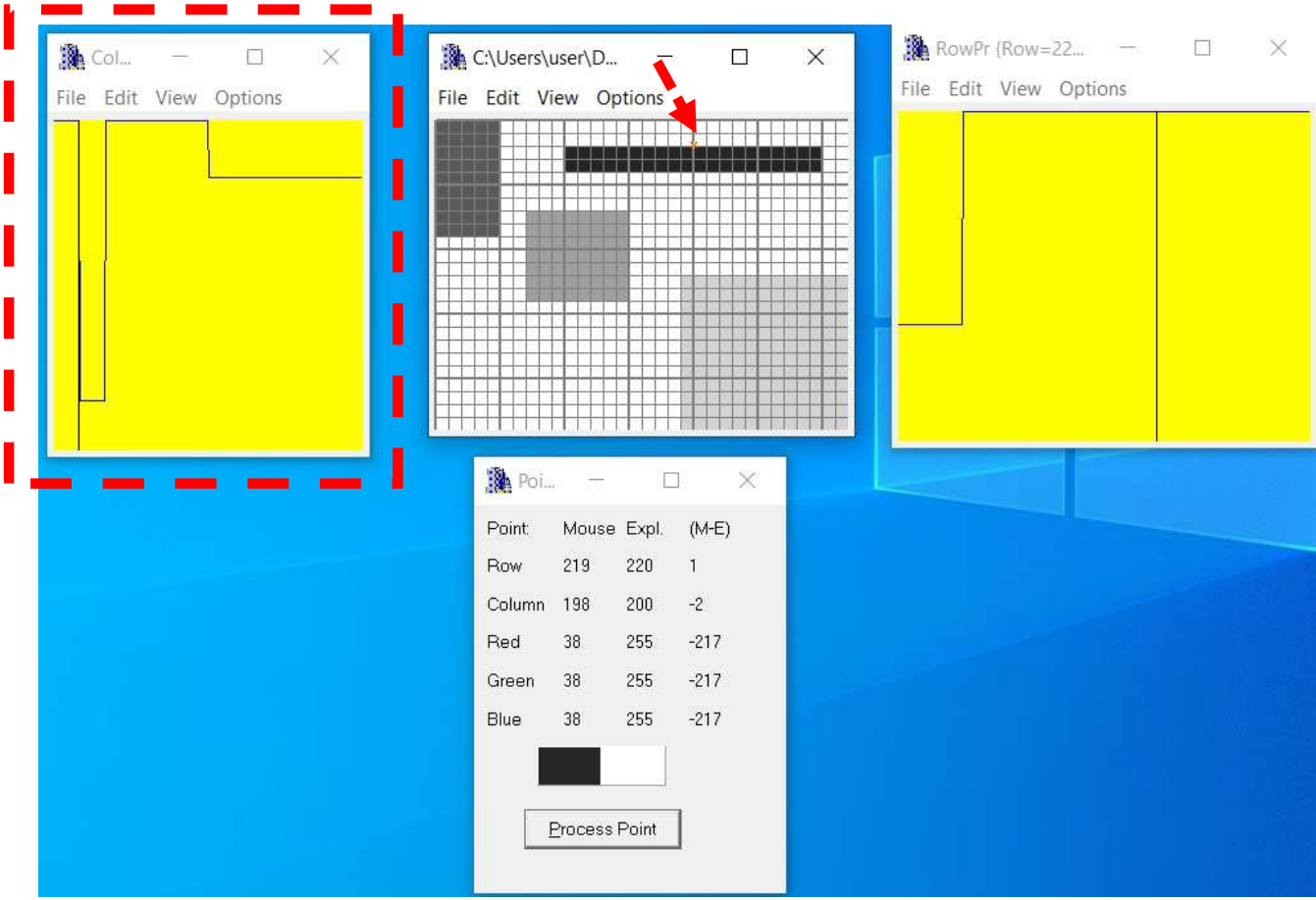




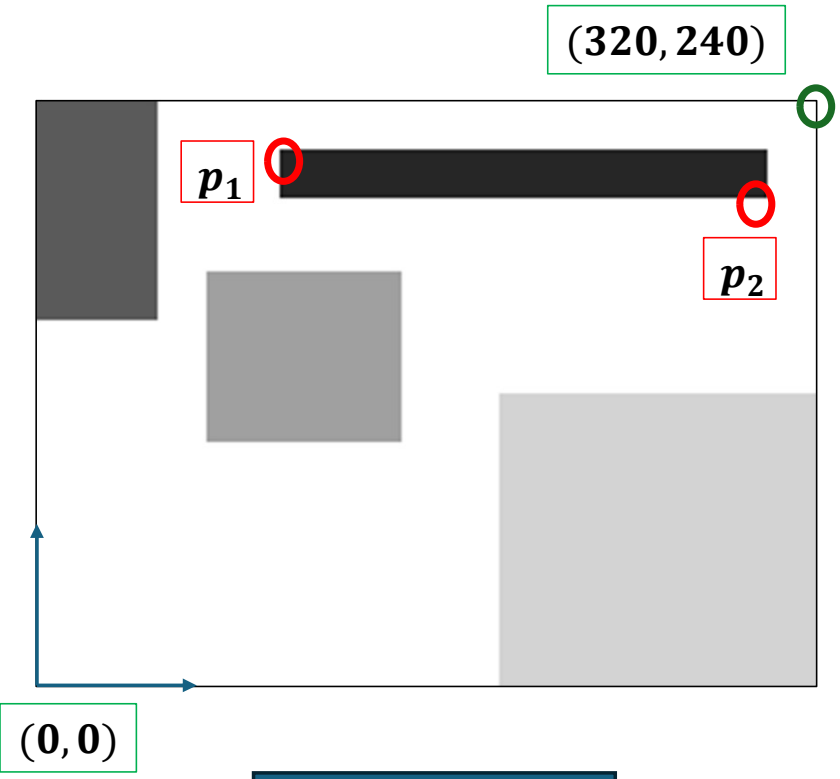
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 7



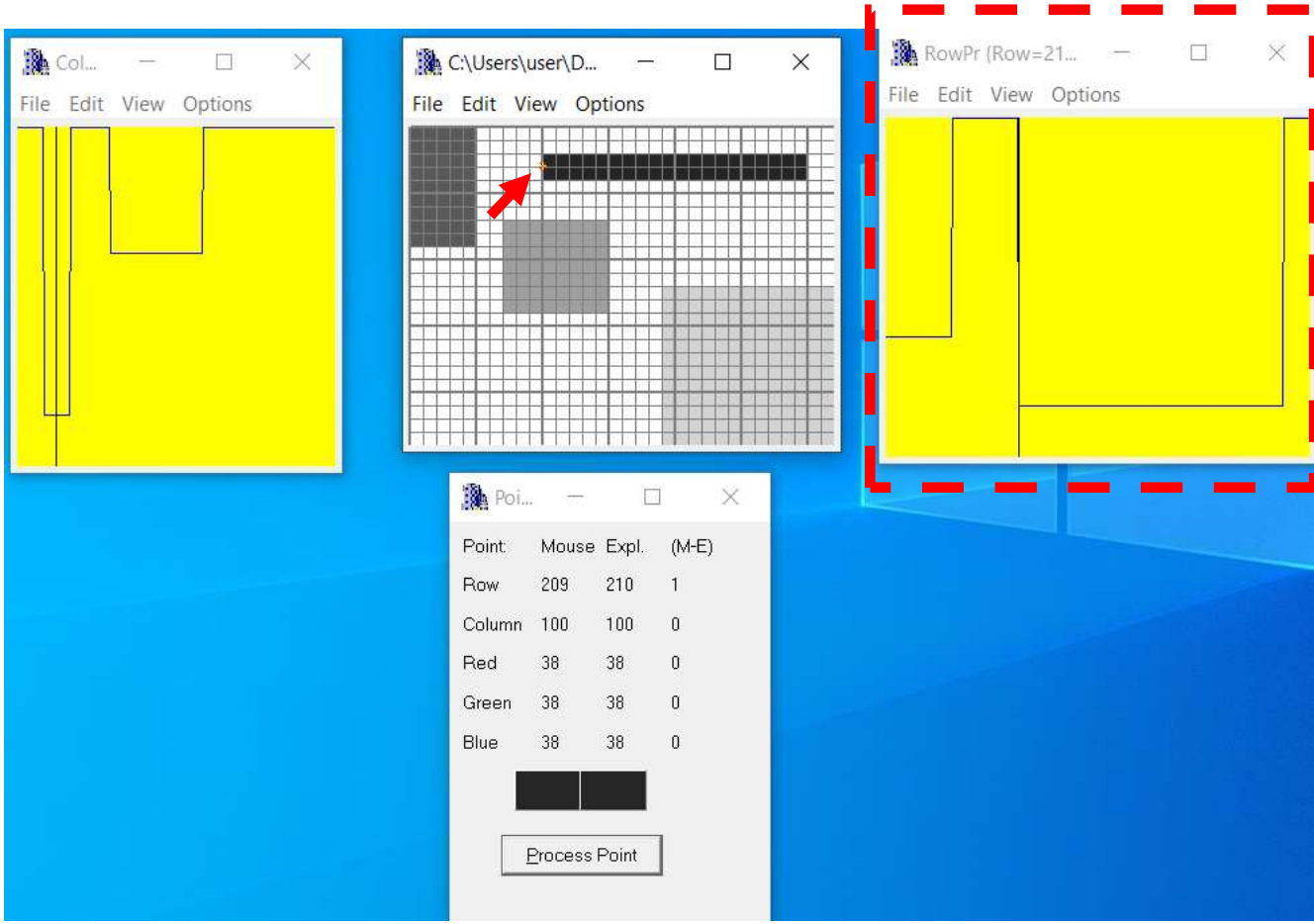
$p_1 = (100,220)$
$p_2 = (300,200)$



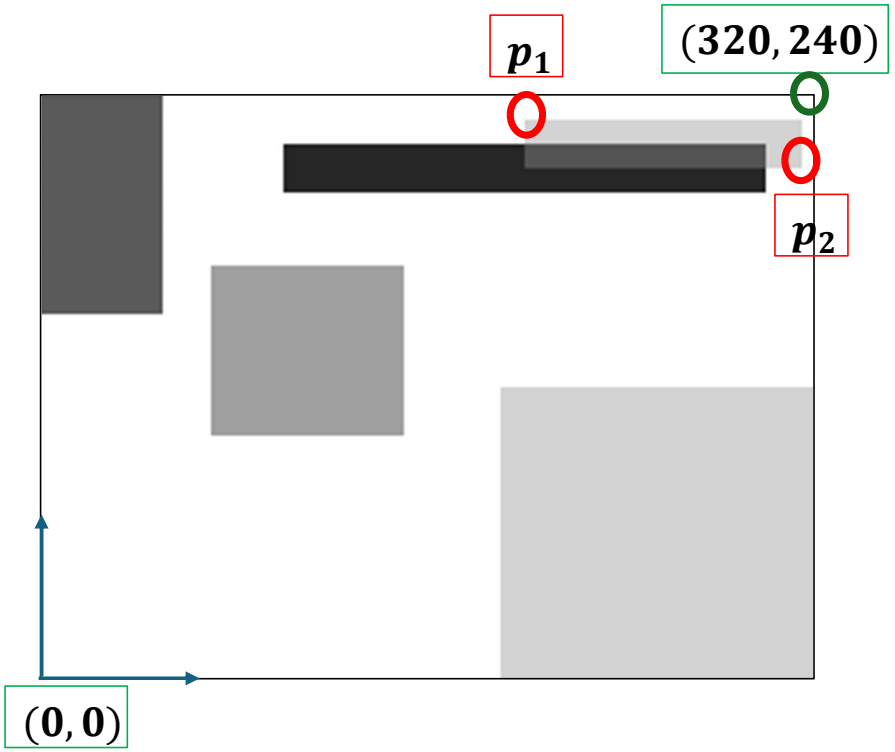
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 8



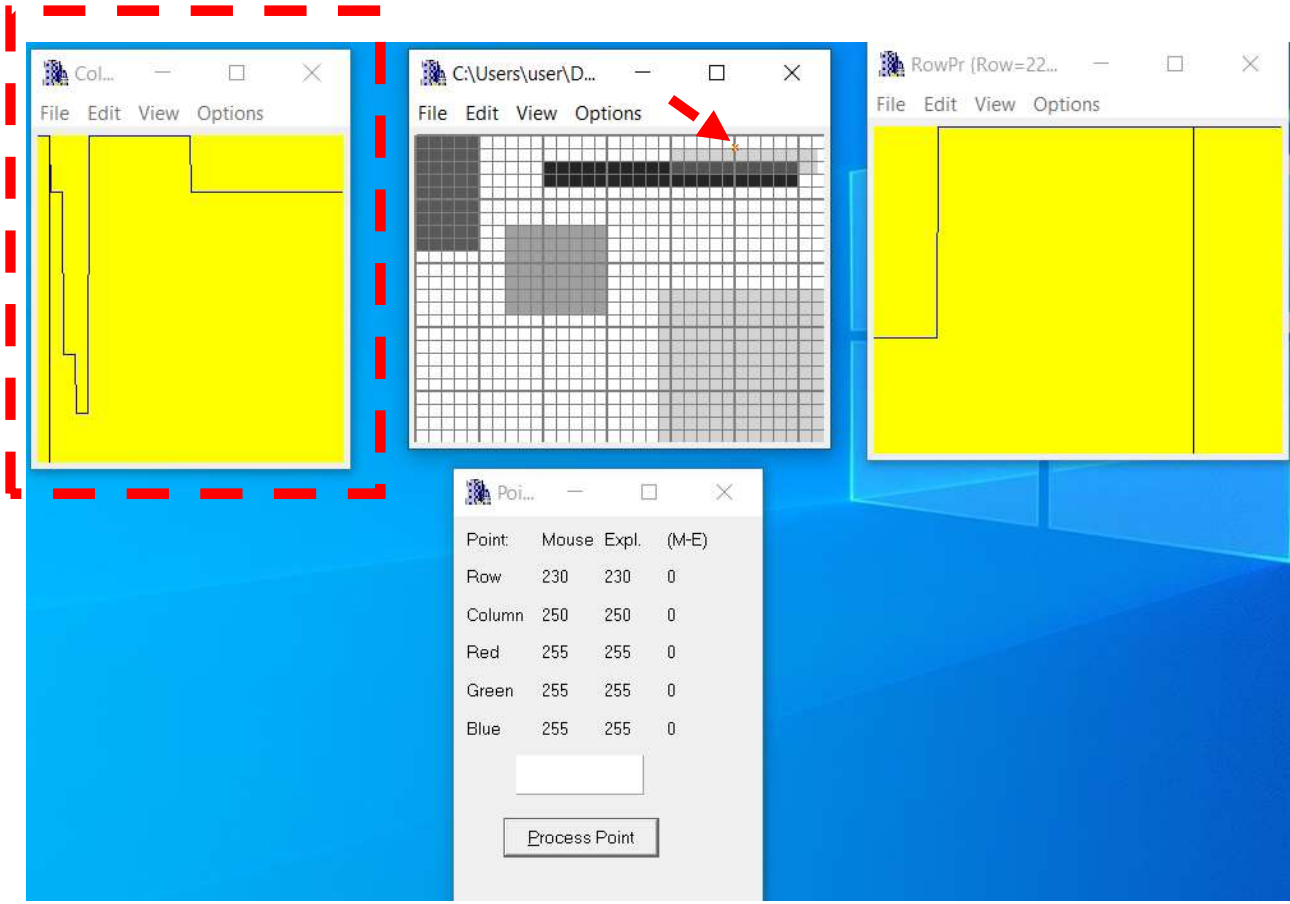
$p_1 = (100,220)$
$p_2 = (300,200)$



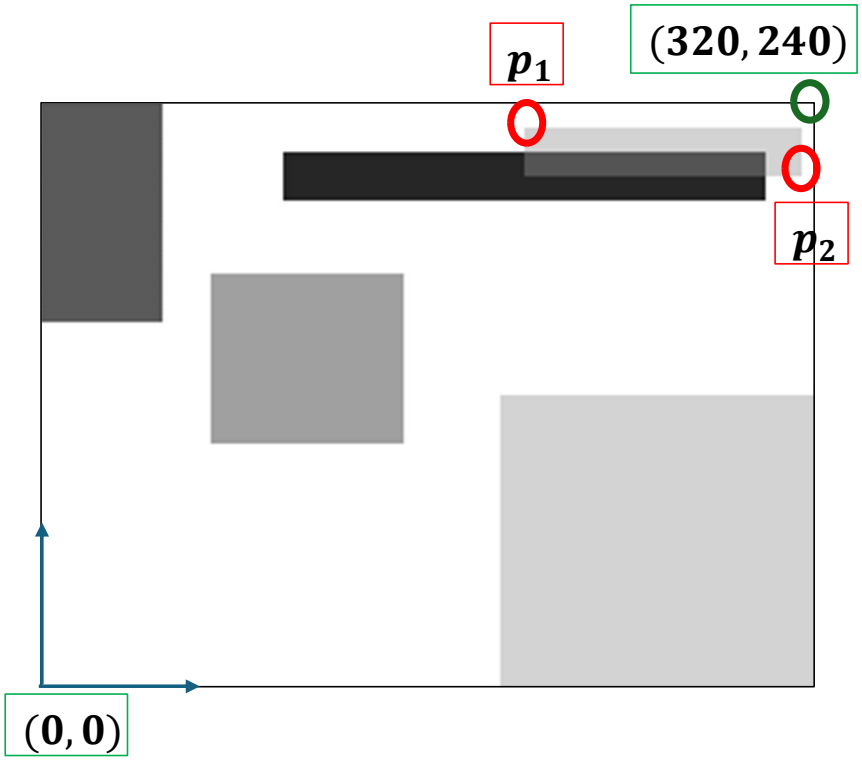
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 9



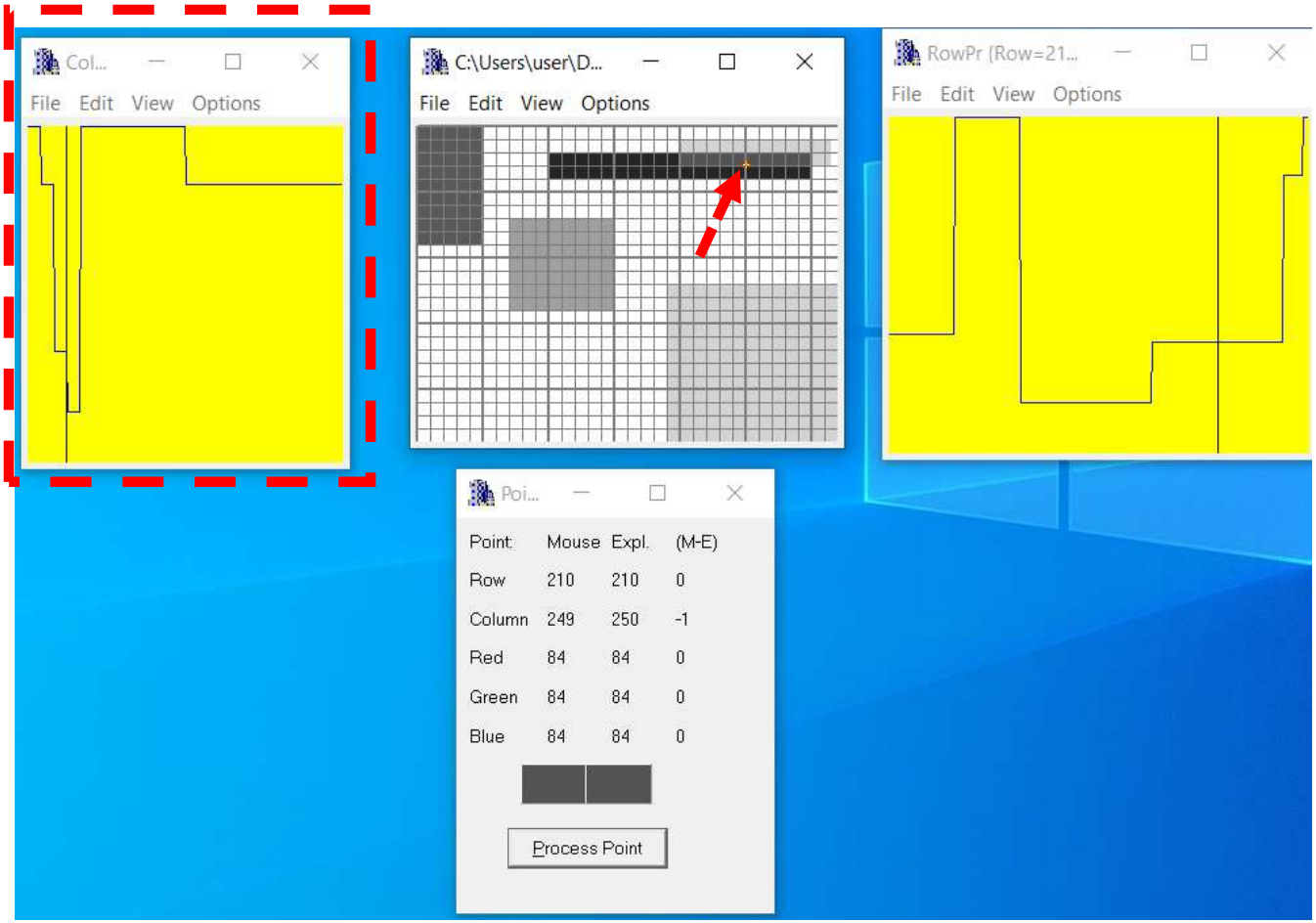
$p_1 = (200,230)$
$p_2 = (315,210)$



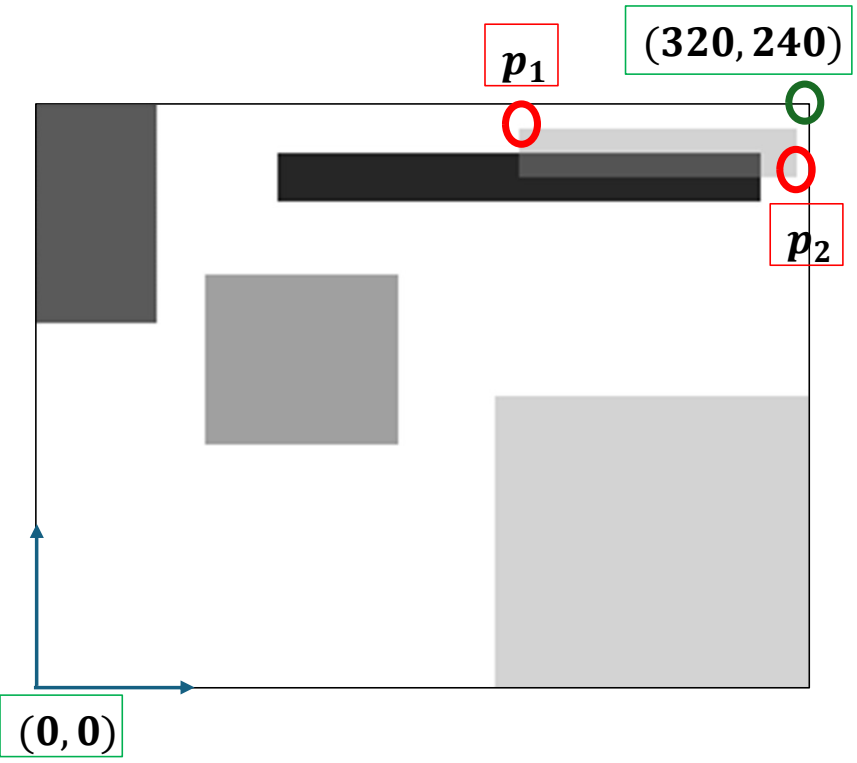
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 10



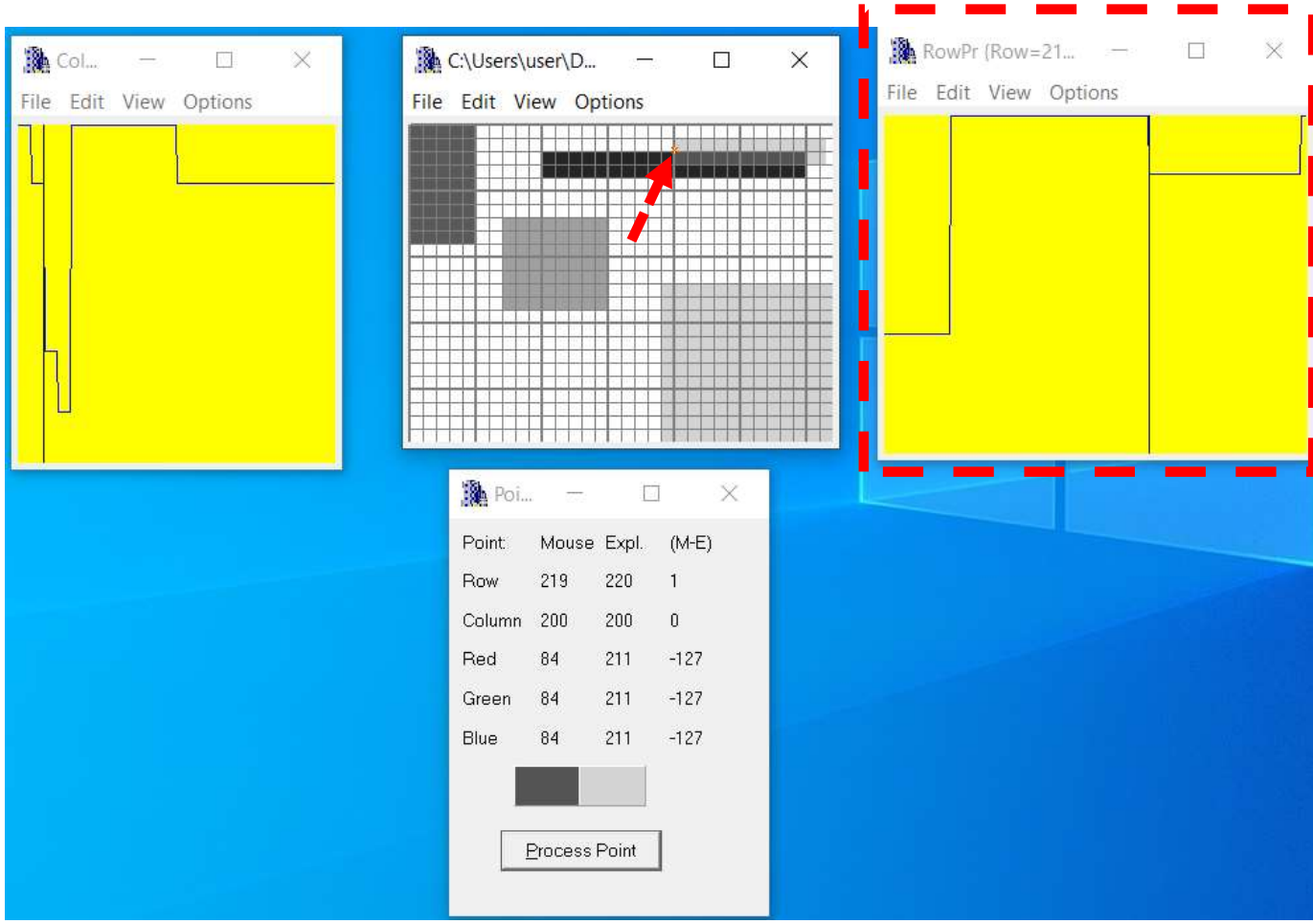
$p_1 = (200,230)$
$p_2 = (315,210)$



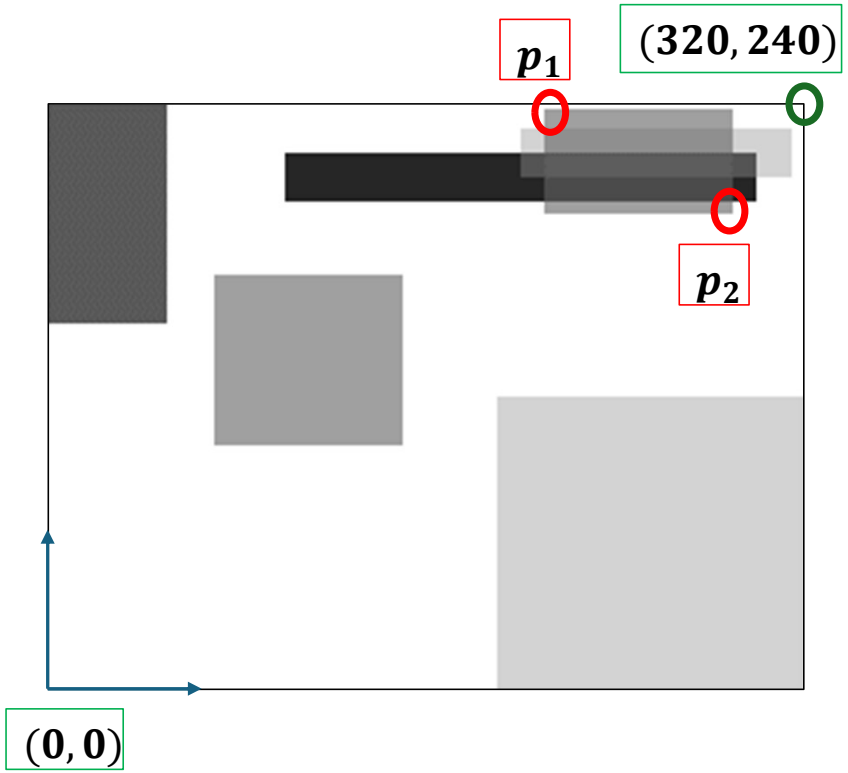
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 11



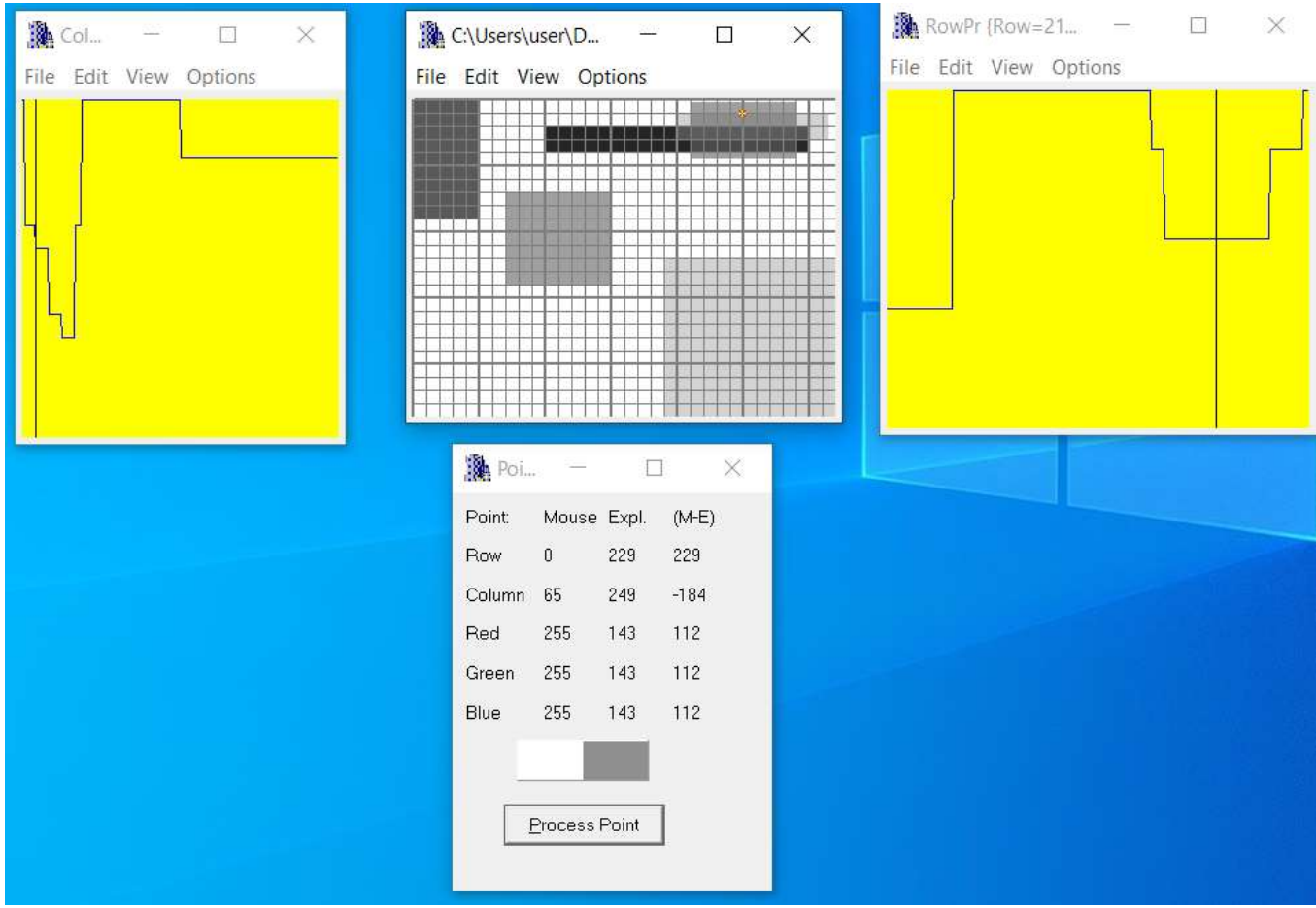
$p_1 = (200,230)$
$p_2 = (315,210)$



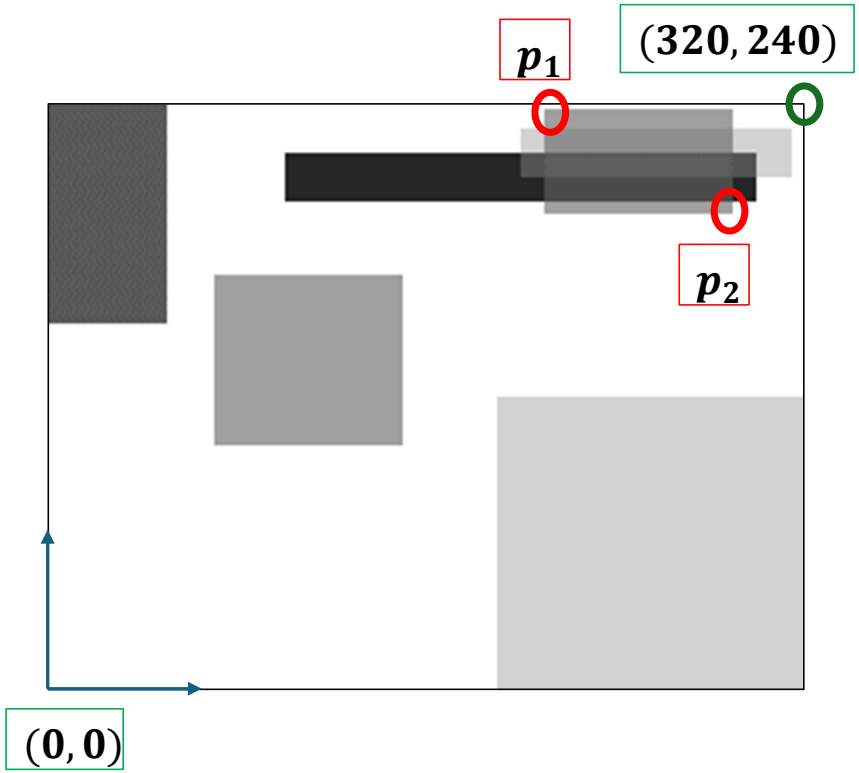
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 12



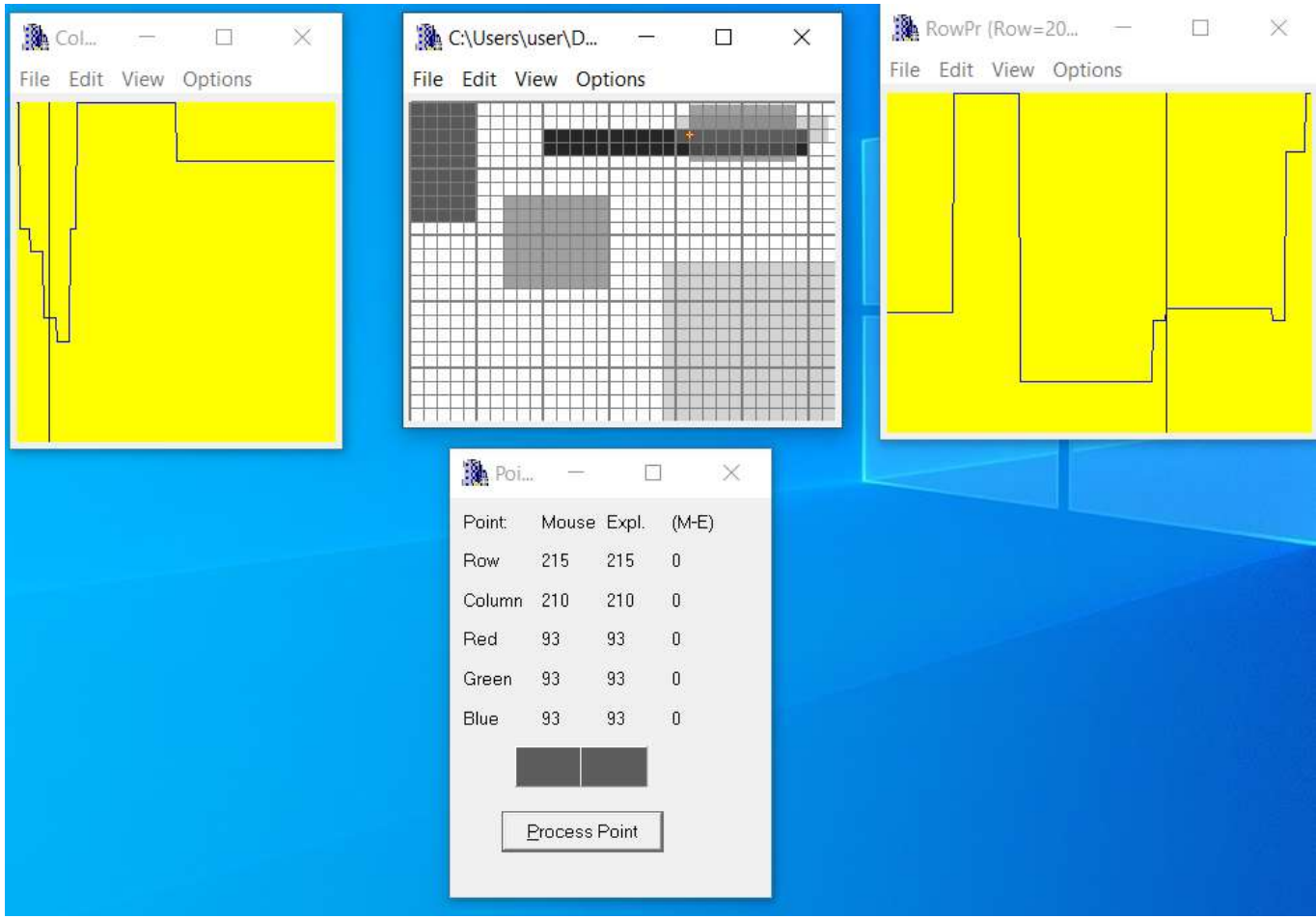
$p_1 = (210,238)$
$p_2 = (290,195)$



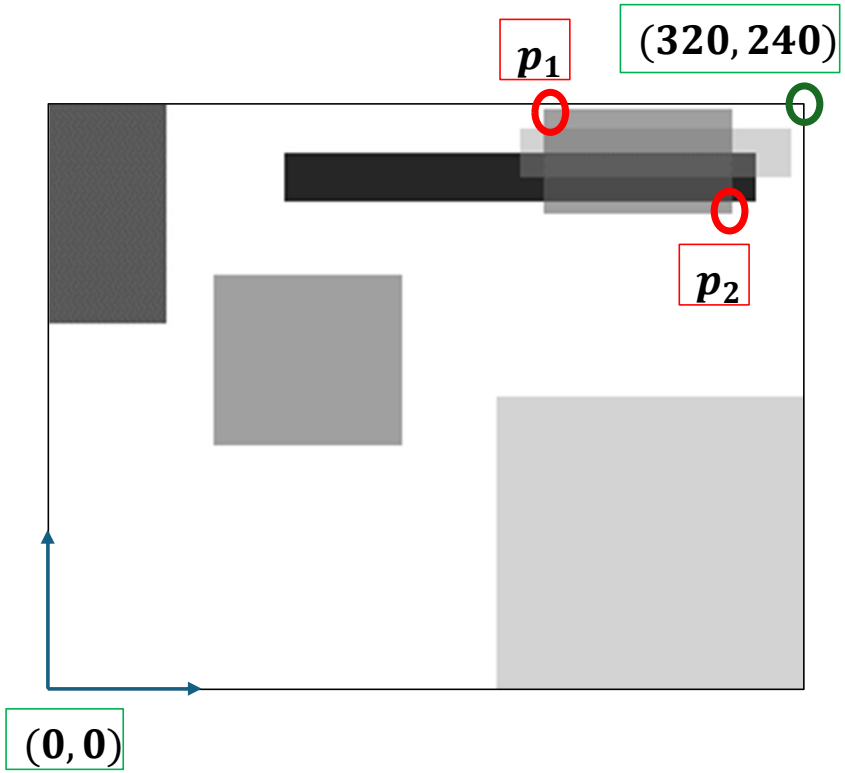
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 13



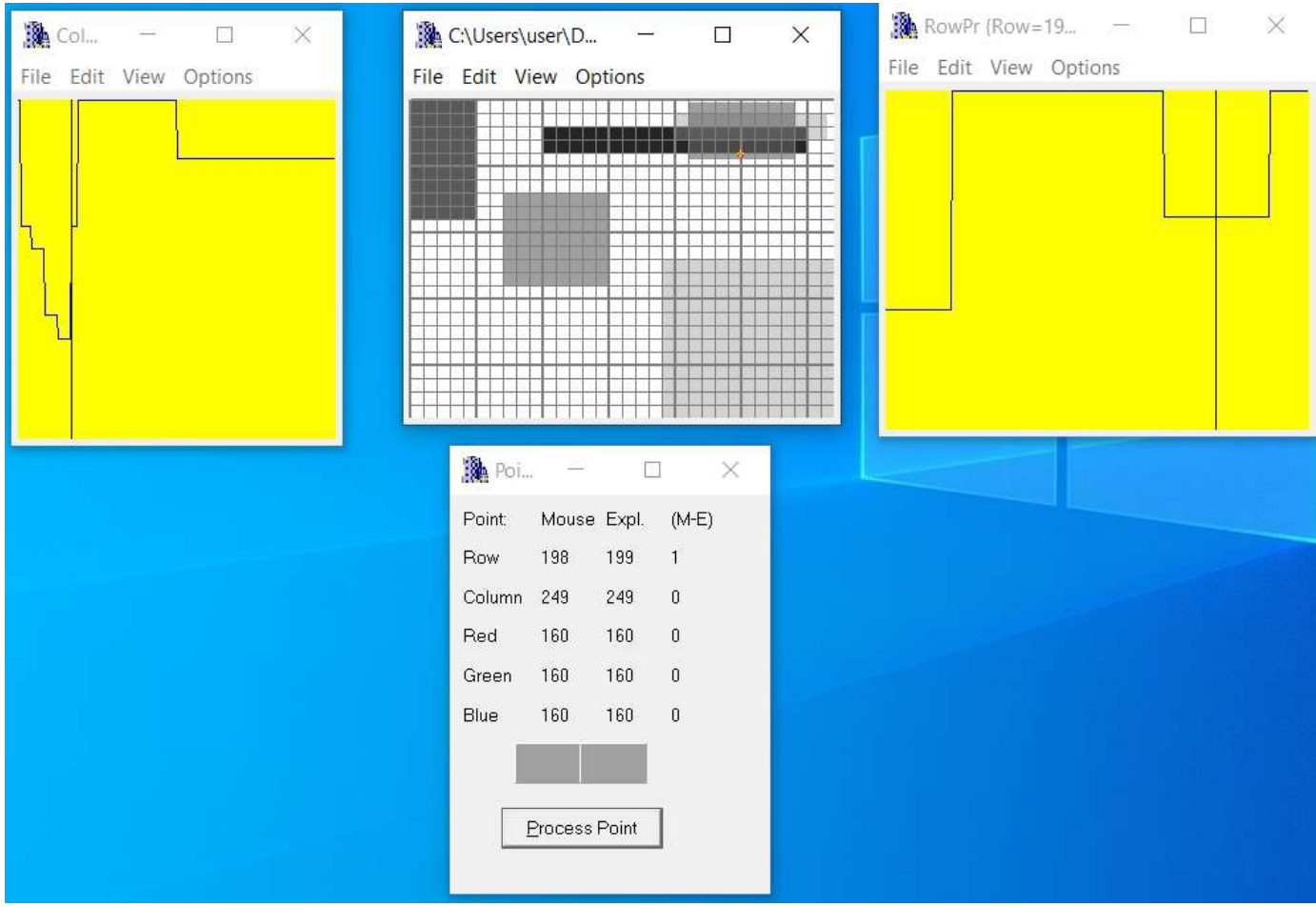
$p_1 = (210,238)$
$p_2 = (290,195)$



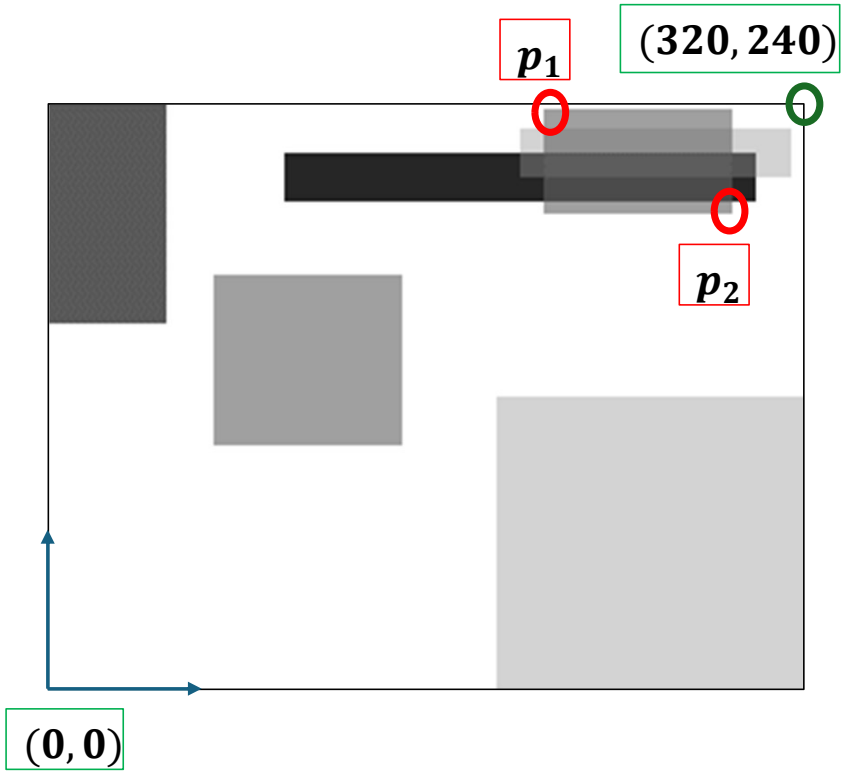
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 14



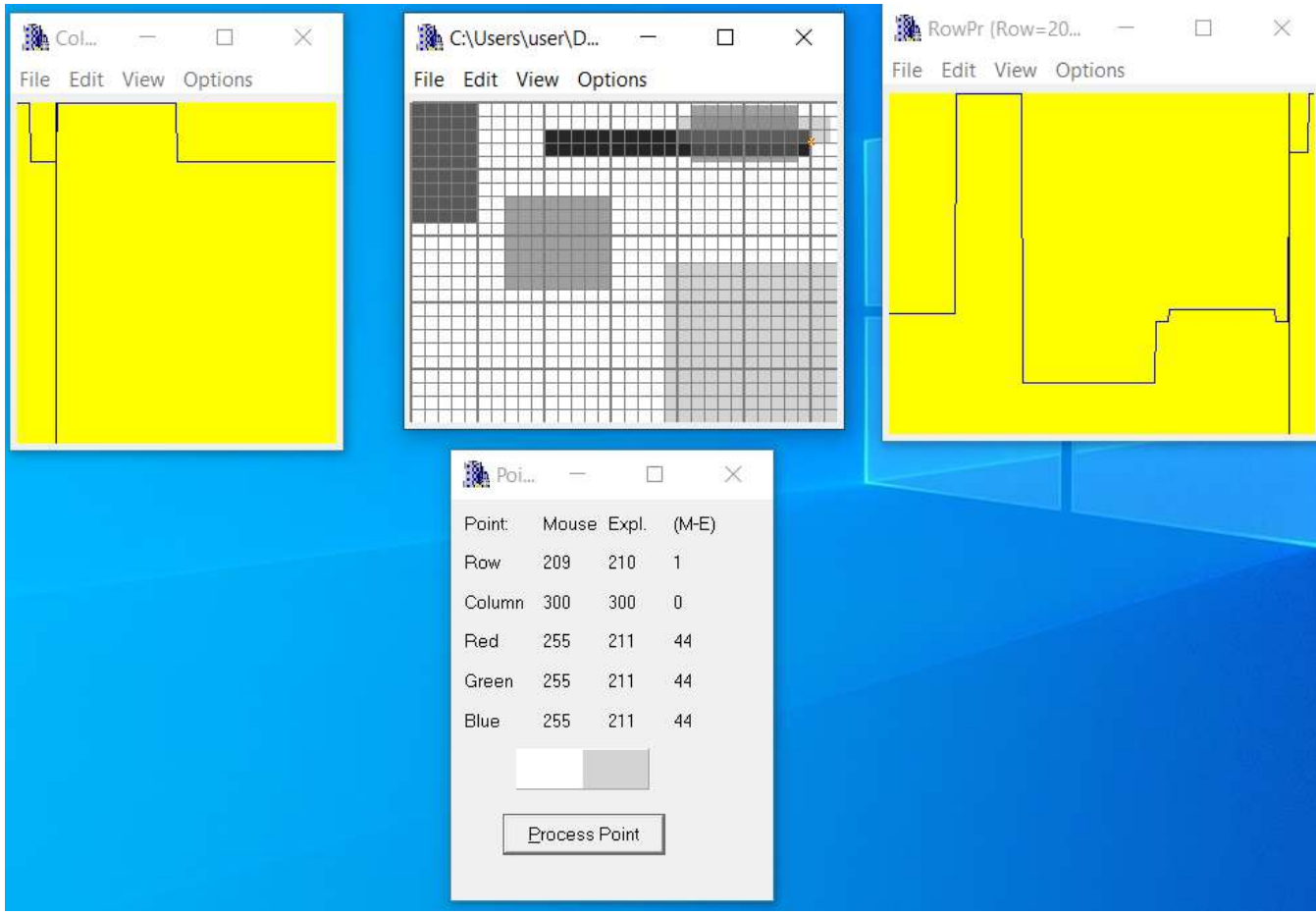
$p_1 = (210,238)$
$p_2 = (290,195)$



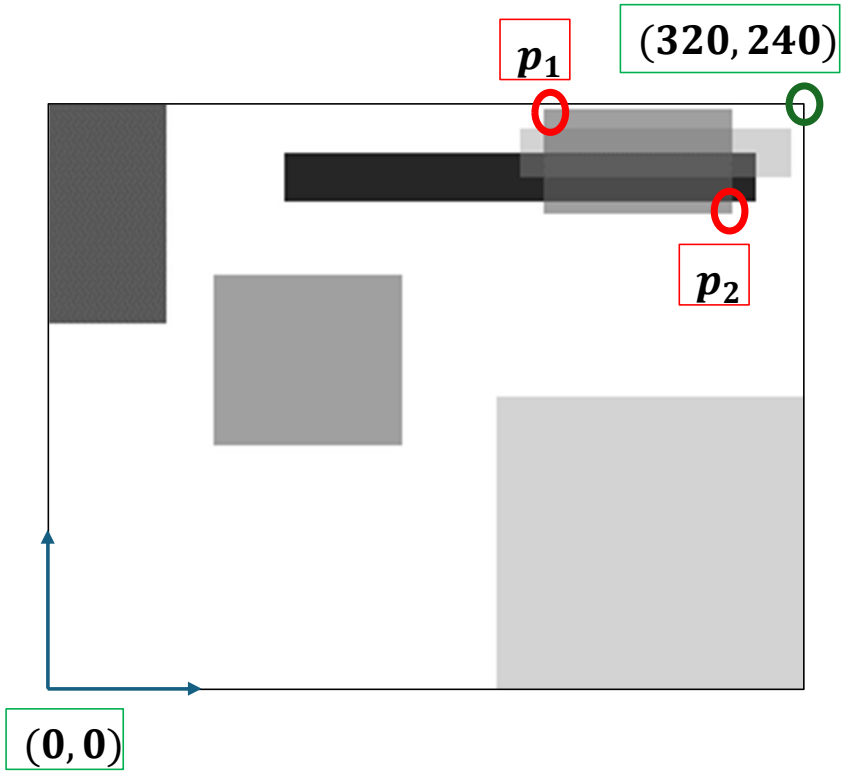
11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 15



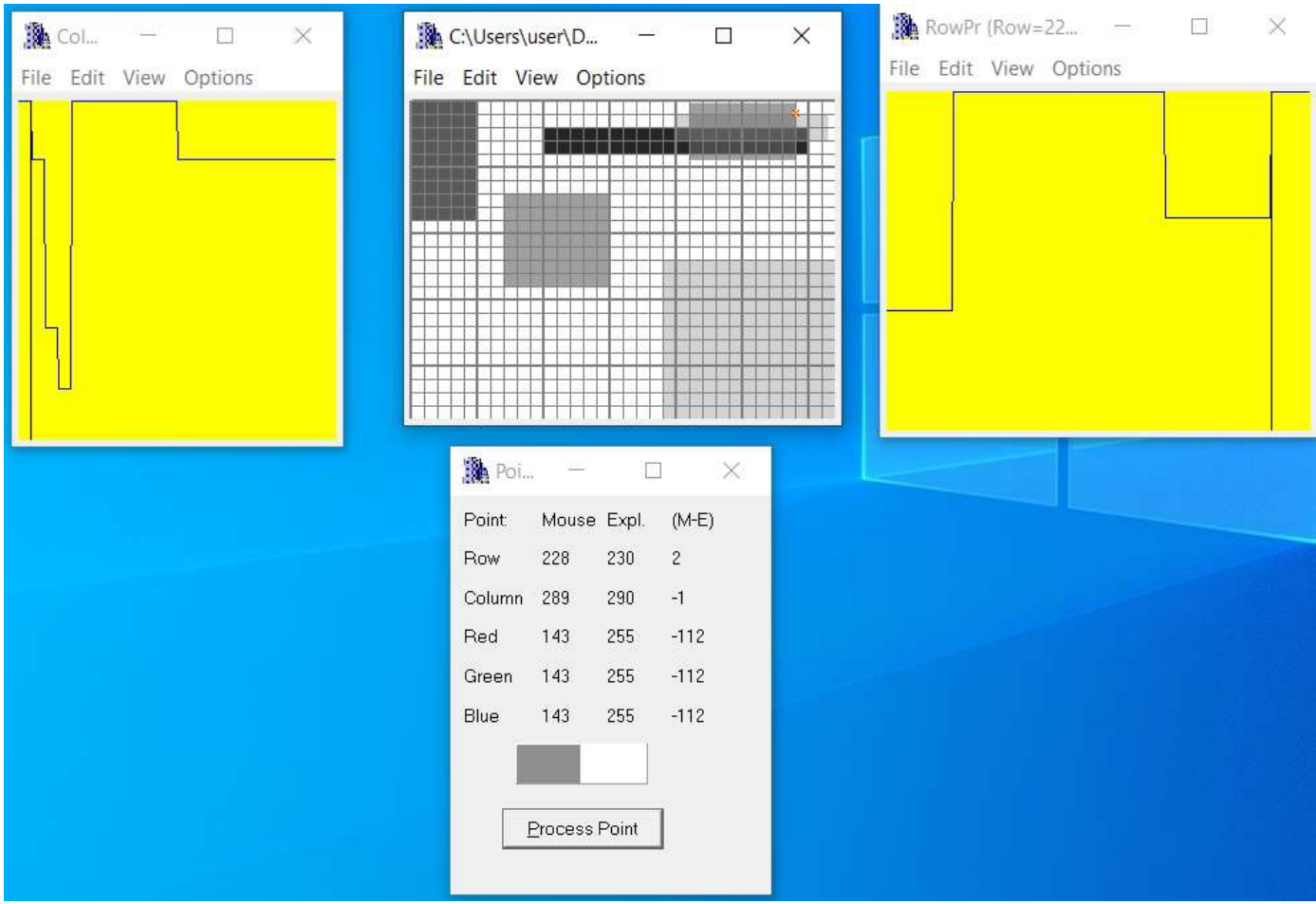
$p_1 = (210,238)$
$p_2 = (290,195)$



11.3 For the specific test images (see 11.2), put relevant profiles and refer relevant lines of code to prove that specific test having specific rectangle were created as required – part 16



$p_1 = (210,238)$
$p_2 = (290,195)$



11.4 Code of the "main" function and set of intermediate images-part 1

```
58  int main() {
59      // Initialize gray image background to white (=255)
60      for (int row = 0; row < NUMBER_OF_ROWS; row++) {
61          for (int col = 0; col < NUMBER_OF_COLUMNS; col++) {
62              img[row][col] = 255;
63          }
64      }
65
66      // Define points for rectangles
67      const int numRectangles = 6;
68      s2dPoint points[numRectangles][2] = {
69      {{0, 240}, {50, 150}},      // Stand-alone
70      {{70, 170}, {150, 100}},    // Stand-alone
71      {{190, 120}, {320, 0}},    // Stand-alone
72      {{100, 200}, {300, 220}},    // overlap
73      {{200, 230}, {315, 210}},    // overlap
74      {{210, 238}, {290, 195}},    // overlap
75
76      };
77      unsigned char transparencies[numRectangles] = { 50, 100, 150, 200, 150, 100 };
78      unsigned char grayLevels[numRectangles] = { 50, 100, 150, 200, 150, 100 };
79
80      // Add rectangles to the image and save each step
81      for (int i = 0; i < numRectangles; i++) {
82          AddGrayRectangle(img, points[i][0], points[i][1], transparencies[i], grayLevels[i]);
83          // Save the image after each rectangle is added
84          char filename[20];
85          sprintf_s(filename, "grayImg_step%d.bmp", i + 1);
86          StoreGrayImageAsGrayBmpFile(img, filename);
87      }
```

This nested loop initializes a 2D array **img** representing the image. Each pixel in the image is set to 255, which represents **white** in a grayscale image. The loop iterates over all rows and columns of the image, ensuring the entire background is white.

points is a 2D array of s2dPoint structures, where each rectangle is defined by two points (top-left and bottom-right corners).

This arrays stores the transparencies and gray levels for each rectangle.

This for loop save an image in the folder of the project after every iteration – in order to show a step-by-step creation of the rectangles

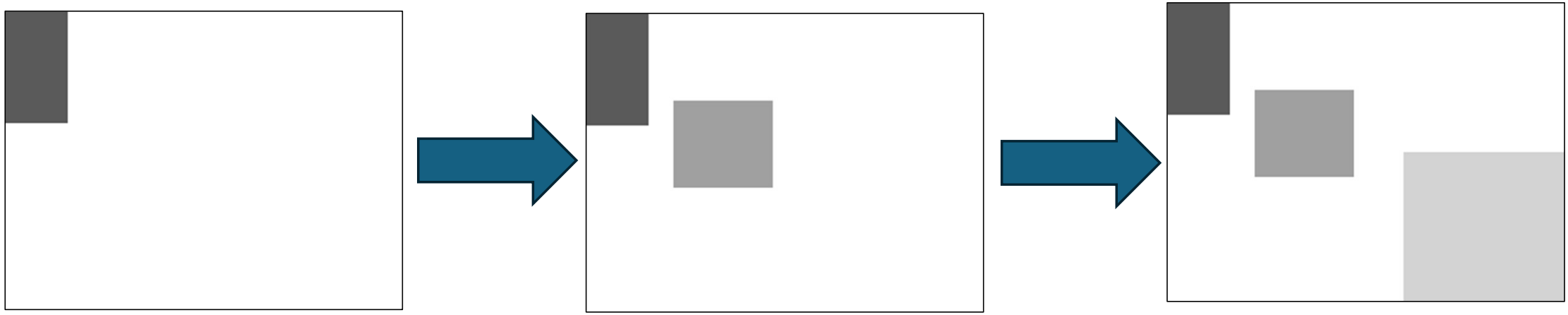
11.4 Code of the “main” function and set of intermediate images-part 2

```
89      // Save the final image
90      StoreGrayImageAsGrayBmpFile(img, "grayImg_final.bmp");
91
92      // Wait for user to press a key
93      WaitForUserPressKey();
94
95      return 0;
96  }
```

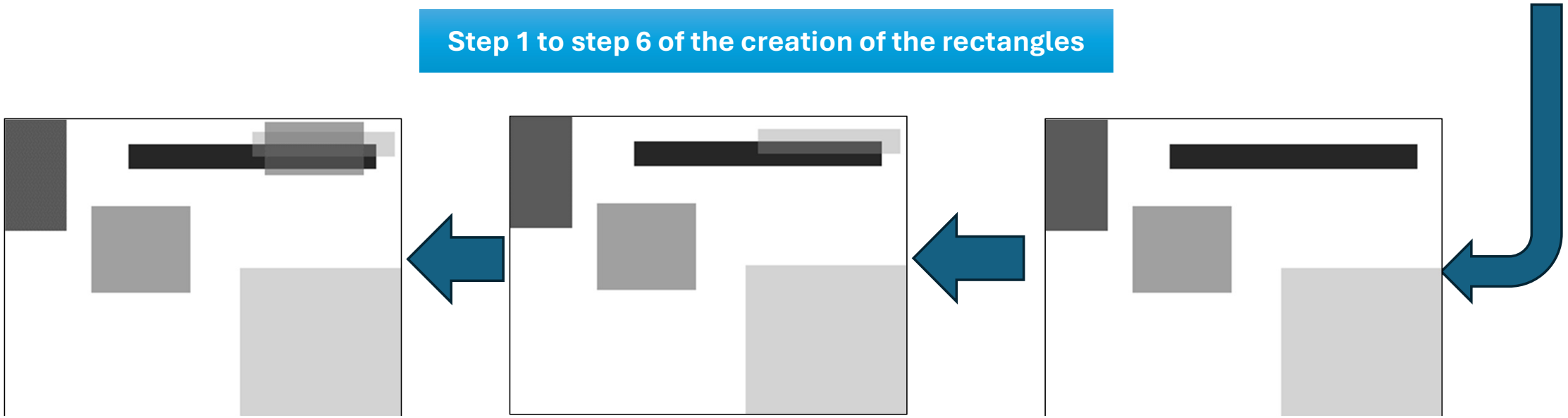
Calls the function StoreGrayImageAsGrayBmpFile to save the image img as a BMP file named "grayImg11.bmp".

11.4 Code of the “main” function and set of intermediate images-part 3

series of intermediate images created in the main step by step with proper explanations



Step 1 to step 6 of the creation of the rectangles



11.5 what did we learned?

We learned how to create a base image.

We learned how to draw on an existing image.

We learned about the use of transparency and gray level of an image

We learned to find the coordinate (0,0) on the screen

We learned to create a rectangle of gray levels

We learned to draw several rectangles in one drawing in different places in the picture