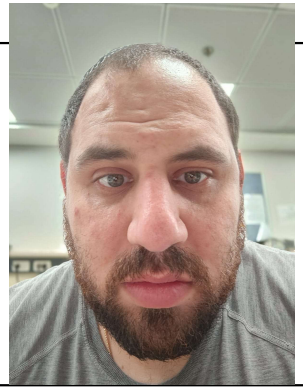
Course: Image Processing 31651
Assignment #11
Synthetic Image Creation (Part 1) – version 3

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

Assignment #11: Create Function AddGrayRectangle

Numerical values of: NUMBER_OF_ROWS and NUMBER_OF COLUMNS
Defined in the file ImProcInPlainC.h

#	Description
1	Define structure: **s2dPoint** {int X, int Y}; Point {0,0} is LEFT BOTTOM point Create function: void AddGrayRectangle(unsigned char **image**[][NUMBER_OF COLUMNS], s2dPoint A, s2dPoint B, unsigned char transparency, unsigned char grayLevel,) Function must **BLEND** rectangle having gray level "grayLevel" with the preliminary created gray **image** with sizes NUMBER_OF_ROWS and NUMBER_OF COLUMNS. Resulted image is created by using blending technique **for each pixel IN THE REGION OF the RECTANGLE defined by 2dPoints A and B** **image**[row,col] = transparency***image**[row,col]/255.0 + (255-transparency)*grayLevel/255.0 (0 – non-transparent rectangle, 255 – absolutely transparent rectangle At any combination of the function parameters, the function must not address pixels with values row and column that are out of ranges of the image. In case user set wrong parameters, SMART solution how to cope this situation must be found (Smart solution: Google "clipping in 2D graphics")
2	Prove by creating a set of simple images that your function works as described and is reliable enough to be used in the future assignments. Proof must be on EE Level : by using numbers, calculations and their validation.
3	By using **above function ONLY**, create gray BMP file "grayImage11.bmp" that has white background and contains at least 6 different gray rectangles: (three partially overlapped rectangles and three stand-alone rectangles)

11.1 - Code of the function “AddGrayRectangle” – part 1


```
14  void AddGrayRectangle(unsigned char image[][NUMBER_OF_COLUMNS], s2dPoint A, s2dPoint B1, unsigned char transparency, unsigned char grayLevel) {
15
16      int top = max(A.Y, B1.Y);
17      int bottom = min(A.Y, B1.Y);
18      int left = min(A.X, B1.X);
19      int right = max(A.X, B1.X);
20
21      // Apply blending technique to the region of the rectangle
22      for (int row = max(bottom, 0); row < min(top, NUMBER_OF_ROWS); row++) {
23          for (int col = max(left, 0); col < min(right, NUMBER_OF_COLUMNS); col++) {
24              image[row][col] = static_cast<unsigned char>(transparency * (image[row][col] / 255.0)
25                  + (255 - transparency) * (grayLevel / 255.0));
26          }
27      }
28  }
```

image[row,col] = transparency*i**mage**[row,col]/255.0 + (255-transparency)*grayLevel/255.0

Although the task did not demand presenting the s2dpoint struct – we decided to present it in part 2 of 11.1 – see next slide.

11.1 - Code of the function “AddGrayRectangle” – part 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      }
172  };
```

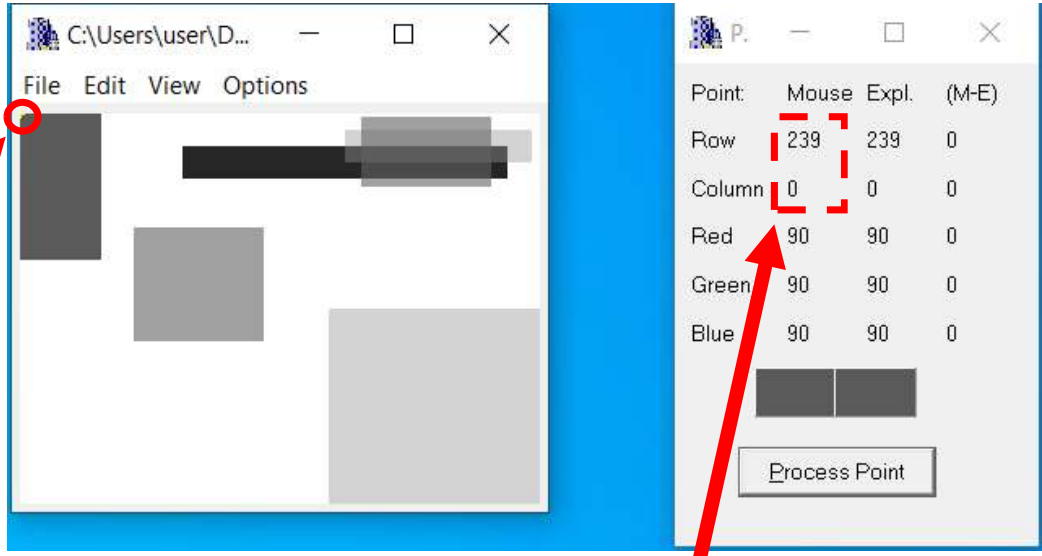
Here is the presentation of the s2dpoint struct – it includes the mechanism for prevent any clipping problem. The use of it will be demonstrated in the next slides.
Another important notation : we added the s2dpoint to the **header** file as an object-oriented approach.

11.2 - List of situations tested – part 1

```
s2dPoint points[numRectangles][2] = {
{{0, 250}, {50, 150}},      // Stand-alone
```

In order to test clipping from the top (row direction) -we changed the value of the first rectangle from 239 to 250 – increase by 11 – to test the clipping. We expect the rectangle will be drawn and the point will be the at B1 max value which is 239!

The point we test



The code for preventing clipping in the y direction (located at the header file)

```
if (Y < 0 || Y > NUMBER_OF_ROWS)
{


    if (Y < 0)
        Y = 0;
    if (Y > NUMBER_OF_ROWS)
        Y = NUMBER_OF_ROWS;

}
```

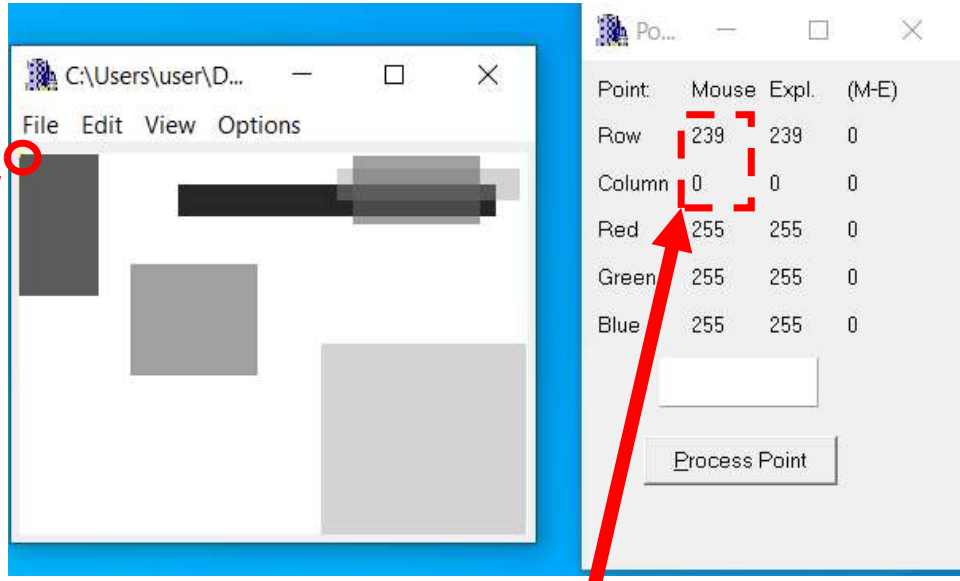
As expected!

11.2 - List of situations tested – part 2

```
s2dPoint points[numRectangles][2] = {
{{-20, 239}, {50, 150}},      // Stand-alone
```

In order to test clipping from the left (column direction) - we changed the value of the first rectangle from 0 to -20 – decreasing by 20 – to test the clipping. We expect the rectangle will be drawn and the point will be the at A min value which is 0!

The point we test



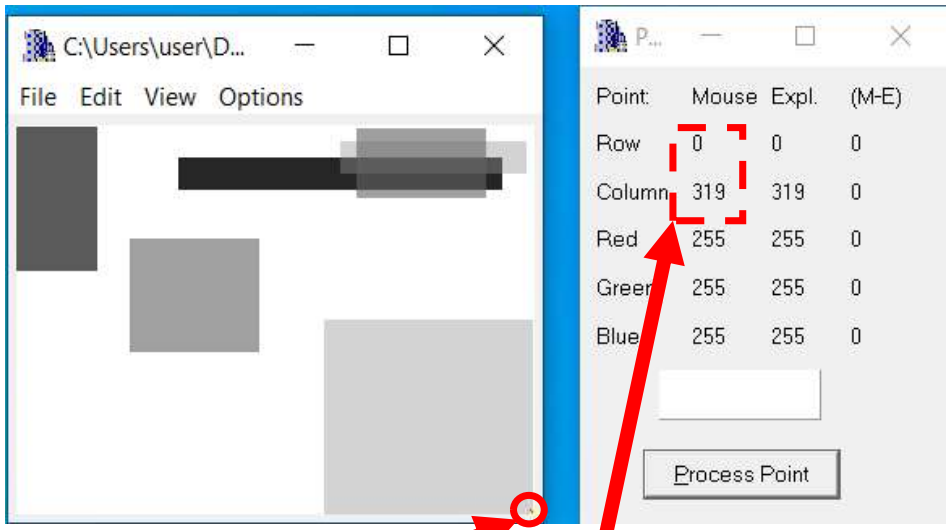
As expected!

The code for preventing clipping in the x direction (located at the header file)

```
if (X < 0 || X > NUMBER_OF_COLUMNS)
{


    if (X < 0)
        X = 0;
    if (X > NUMBER_OF_COLUMNS)
        X = NUMBER_OF_COLUMNS;

}
```

11.2 - List of situations tested – part 3

```
{{190, 120}, {319, -10}},    // Stand-alone
```



In order to test clipping from the bottom (row direction) - we changed the value of the third rectangle from 0 to -10 – decreasing by 10 – to test the clipping. We expect the rectangle will be drawn and the point will be the at B1 min value which is 0!

The code for preventing clipping in the x direction (located at the header file)

```
if (X < 0 || X > NUMBER_OF_COLUMNS)
{


    if (X < 0)
        X = 0;
    if (X > NUMBER_OF_COLUMNS)
        X = NUMBER_OF_COLUMNS;

}
```

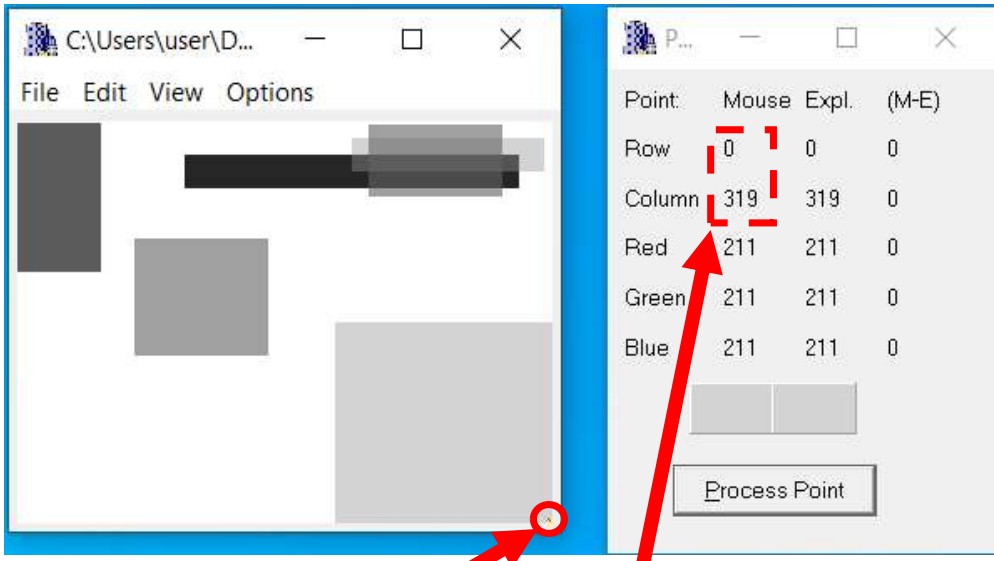
The point we test

As expected!

11.2 - List of situations tested – part 4


```
{{190, 120}, {340, 0}},    // Stand-alone
```

In order to test clipping from the right (column direction) - we changed the value of the third rectangle from 319 to 340 – decreasing by 21 – to test the clipping. We expect the rectangle will be drawn and the point will be the at A min value which is 319!



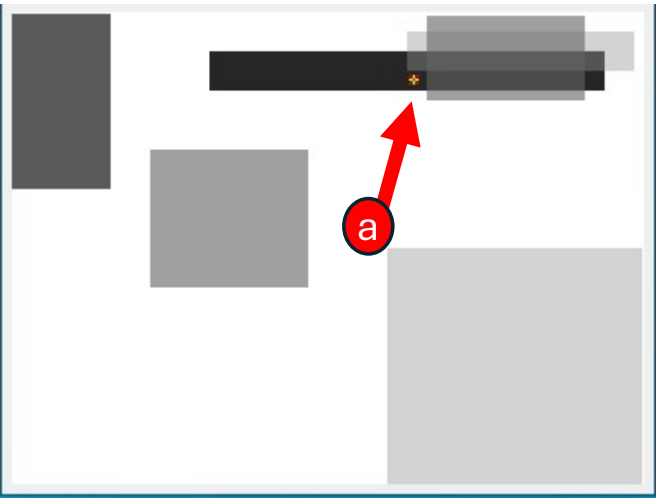
The point we test

As expected!

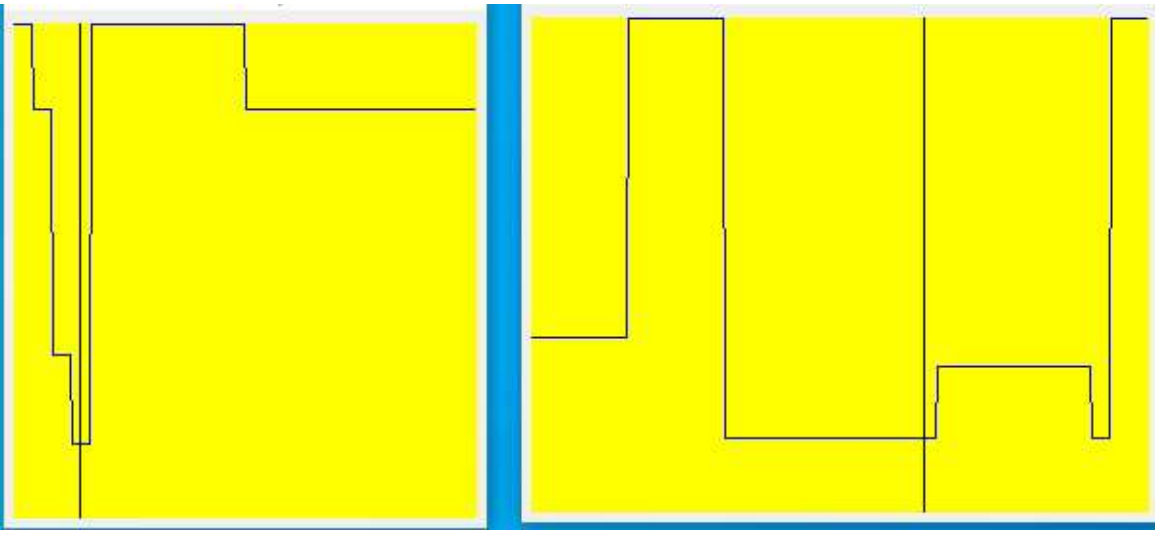
The code for preventing clipping in the y direction (located at the header file)

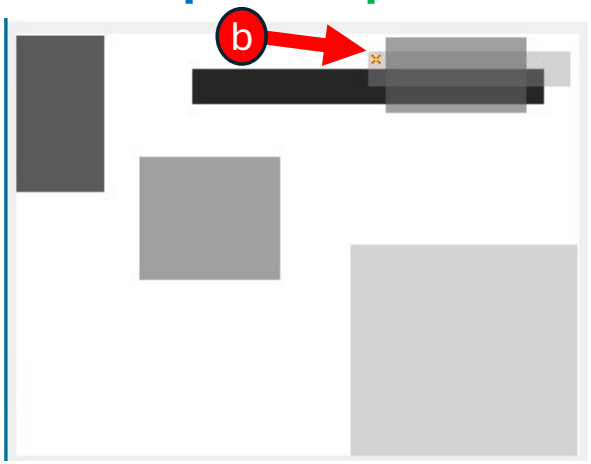
```
if (Y < 0 || Y > NUMBER_OF_ROWS)
{


    if (Y < 0)
        Y = 0;
    if (Y > NUMBER_OF_ROWS)
        Y = NUMBER_OF_ROWS;

}
```

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

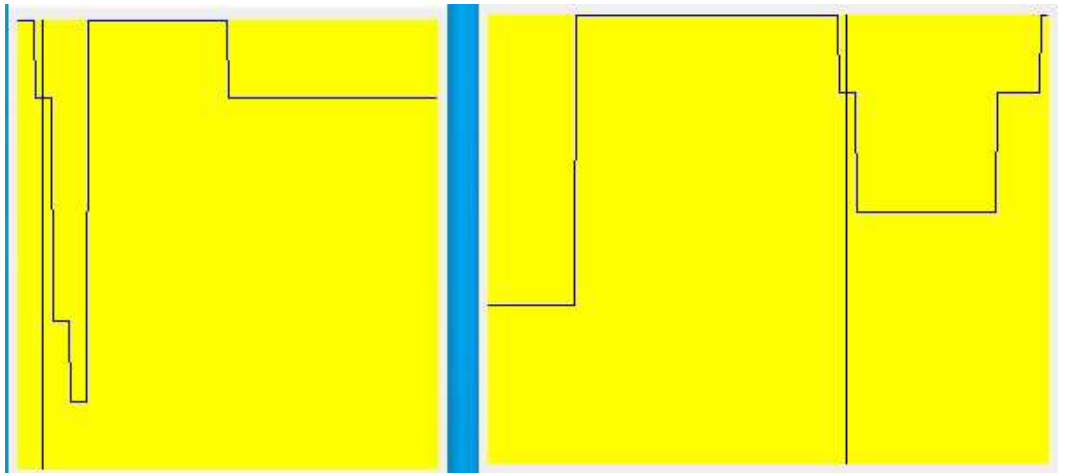


Point:	Mouse	Expl.	(M-E)
Row	205	205	0
Column	203	203	0
Red	38	38	0
Green	38	38	0
Blue	38	38	0





Point:	Mouse	Expl.	(M-E)
Row	225	225	0
Column	204	204	0
Red	211	211	0
Green	211	211	0
Blue	211	211	0



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

image[row,col] = transparency***image**[row,col]/255.0 + (255-transparency)*grayLevel/255.0

For point b : $transparency = 150$
$graylevel = 150$

$$image_b = 150 * \frac{255}{255} + (255 - 150) * \left(\frac{150}{255}\right) = 211.7$$

Real results

Point:	Mouse	Expl.	(M-E)
Row	225	225	0
Column	204	204	0
Red	211	211	0
Green	211	211	0
Blue	211	211	0

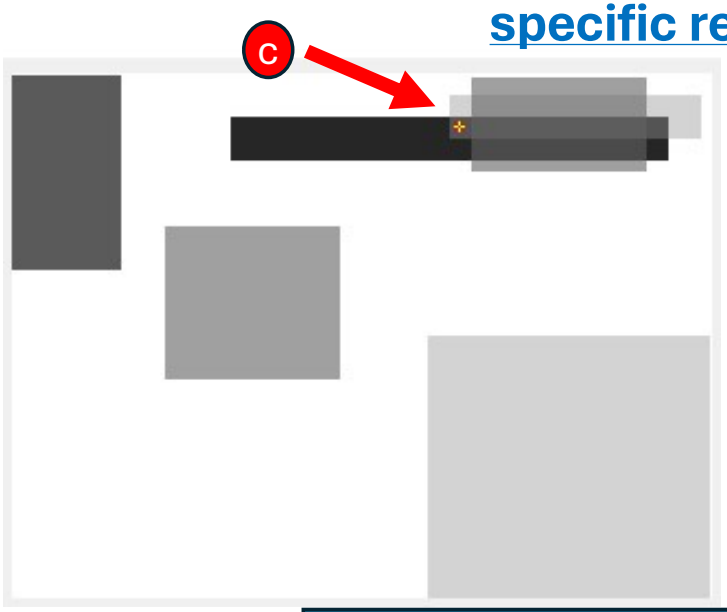
For point a : $transparency = 20$
$graylevel = 20$

$$image_a = 20 * \frac{255}{255} + (255 - 20) * \left(\frac{20}{255}\right) = 38.4$$

Point:	Mouse	Expl.	(M-E)
Row	205	205	0
Column	203	203	0
Red	38	38	0
Green	38	38	0
Blue	38	38	0

Real results

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



Point:	Mouse	Expl.	(M-E)
Row	215	215	0
Column	204	204	0
Red	84	84	0
Green	84	84	0
Blue	84	84	0

Regarding point c : the result has to be the substitution between 'a' and 'b' -> **211 –38=173**

But the real result is 84 as it can be seen above – which is wrong.

we tried an experimental formula (made by us) which is :

image[row,col]=$\frac{1}{N}\sum_{i=1}^{N} transparency(i) + \sum_{i=1}^{N}(255 - graylevel(i) * \left(\frac{graylevel(i)}{255}\right)$ that gave us the approximate result.

When we discussed it with the lecturer we were advice to change the order of the rectangles to receive the correct value. After doing so we got the same result which indicate that we have a mathematical problem which we can't solve. After consulting with the lecturer , we were permitted to use it with the mathematical error and to submit it as is.

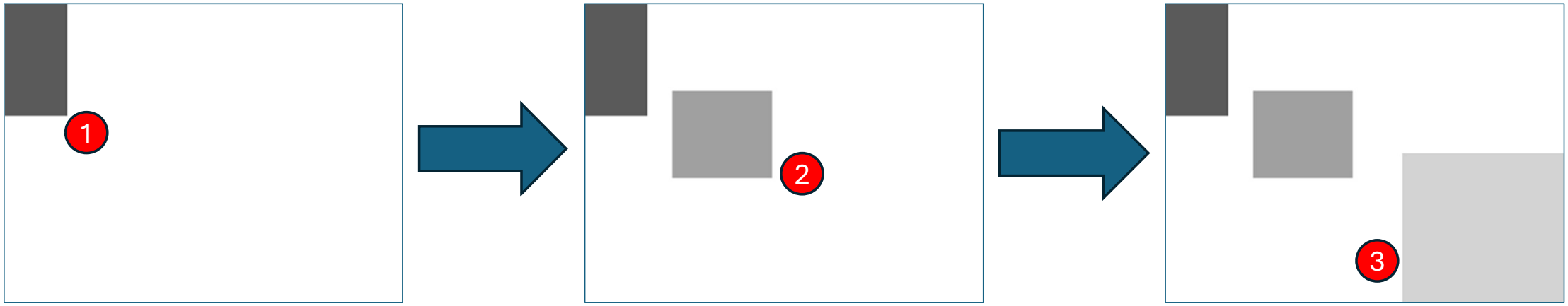
11.4 - Code of the “main” function and set of intermediate images - part 1


```
33  int main() {
34      // Initialize gray image background to white (=255)
35      for (int row = 0; row < NUMBER_OF_ROWS; row++) {
36          for (int col = 0; col < NUMBER_OF_COLUMNS; col++) {
37              img[row][col] = 255;
38          }
39      }
40
41      // Define points for rectangles
42      const int numRectangles = 6;
43      s2dPoint points[numRectangles][2] = {
44      {{0, 239}, {50, 150}},      // Stand-alone
45      {{70, 170}, {150, 100}},    // Stand-alone
46      {{190, 120}, {319, 0}},    // Stand-alone
47      {{100, 220}, {300, 200}},    // overlap
48      {{200, 230}, {315, 210}},    // overlap
49      {{210, 238}, {290, 195}},    // overlap
50
51      };
52      unsigned char transparencies[numRectangles] = { 50, 100, 150, 20, 150, 100 };
53      unsigned char grayLevels[numRectangles] = { 50, 100, 150, 20, 150, 100 };
54
55      //// Add rectangles to the image
56      for (int i = 0; i < numRectangles; i++) {
57          AddGrayRectangle(img, points[i][0], points[i][1], transparencies[i], grayLevels[i]);
58      }
59      StoreGrayImageAsGrayBmpFile(img, "grayImg_11.bmp");
60      return 0;
61  }
```

In addition – as mentioned before – the s2dpoint struct & the validation process to prevent clipping in located at the header file

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

The set of the intermediate images – every new rectangle (6 total) will be numbered and its A and B1 coordinates & their transparencies and gray levels – will be written next to the new image



$A = \{0, 239\}$
$B_1 = \{50,150\}$
$transparancy = 50$
$graylevel = 50$

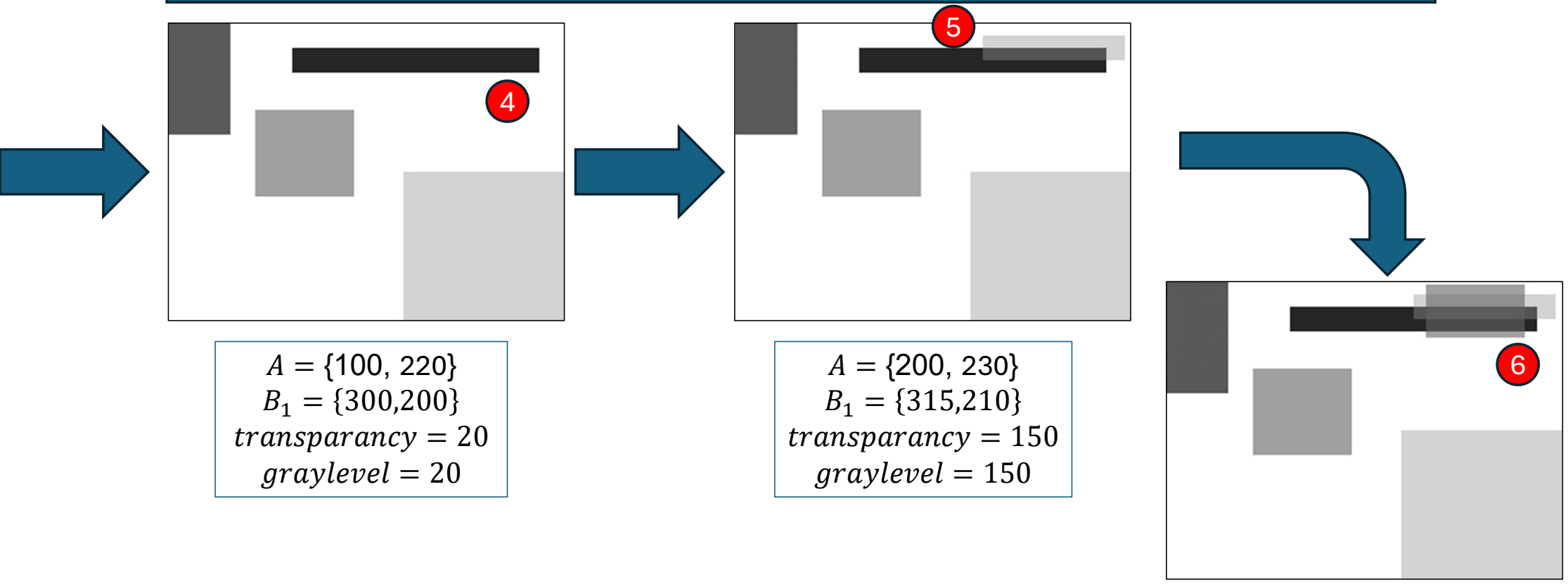
$A = \{70, 170\}$
$B_1 = \{150,100\}$
$transparancy = 100$
$graylevel = 100$

$A = \{190, 120\}$
$B_1 = \{319,0\}$
$transparancy = 150$
$graylevel = 150$

Continue in the next slide

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

The set of the intermediate images – every new rectangle (6 total) will be numbered and its A and B1 coordinates & their transparencies and gray levels – will be written next to the new image



$A = \{100, 220\}$
$B_1 = \{300,200\}$
$transparancy = 20$
$graylevel = 20$

$A = \{200, 230\}$
$B_1 = \{315,210\}$
$transparancy = 150$
$graylevel = 150$

$A = \{210, 238\}$
$B_1 = \{290,195\}$
$transparancy = 100$
$graylevel = 100$

11.4 - Code of the “main” function and set of intermediate images - part 4

The final form – all 6 rectangles – is:

11.5 – what did we learned?

1. We learned how to use the bmp viewer
2. How overlapping creates different shades from the original and how its calculated
3. How to avoid clipping using smart solutions
4. We learned to create a grayscale image in the location and size we wanted, and changed the values of graylevel and transparency to see how it affects the grayscale.