0.1 IP Miniproject - Rasmus Christiansen - Student no. 20144790

In this miniproject I present two algorithms for scaling an image. One uses forward mapping, and the other uses backward mapping. I have chosen to present two algorithms instead of one because they both follow the same mathematical principle and it provides a good point for discussing why you would use one over the other.

The first algorithm is as follows:

```
output_image_height = input_image_height * Sy
output_image_width = input_image_width * Sx
for (y = 0; y < input_image_height; ++y)
{
    for (x = 0; x < input_image_width; ++x)
    {
        output_image(y*Sy, x*Sx) = input_image(y, x)
    }
}
```

When you look at this algorithm it becomes clear why this is called forward mapping. The algorithm goes through each pixel in the input image and maps it to the most appropriate output; input -> output. The downside of this is that it leaves black spots, because the algorithm goes through the input image rather than the output. This can work in other image processing algorithms where the input and output have a 1:1 relationship, but for a scaling algorithm this is never the case. Therefore, it is better to use backward mapping, which is the second algorithm i use:

```
output_image_height = input_image_height * Sy
output_image_width = input_image_width * Sx
for (y = 0; y < output_image_height; ++y)
{
    for (x = 0; x < output_image_width; ++x)
    {
        output_image(y, x) = input_image(1 / Sy * y, 1 / Sx * x)
    }
}
```

Contrary to the first algorithm, this one goes through each pixel in the output image, and then maps the most appropriate pixel *to it*. This is why it is called backward mapping. If the forward mapping algorithm pushes data, the backward mapping pulls it instead; output <- input.

0.1.1 Code overview

Entirety of code

```
#include "opencv2/opencv.hpp"
#include <iostream>

using namespace cv;
using namespace std;

int roundNum(float num) {

    if (num - floor(num) >= 0.5)
        num = ceil(num);
    else
        num = floor(num);

    return num;
}

//Simple scaling algorithm using forward mapping
Mat simpleScaling(Mat src, float Sx, float Sy)
{

    if (Sx <= 0 || Sy <= 0)
    {
        cout << "Negative scaling is not possible";
        return src;
    }

    //Create a Mat object with scaled dimensions, greyscale,
    completely black.
    Mat output(src.rows*Sy,src.cols*Sx, CV_8U, Scalar(0, 0, 0));

    for (int y = 0; y < src.rows; ++y)
    {
        for (int x = 0; x < src.cols; ++x)
        {
            output.at<uchar>(y*Sy, x*Sx) = src.at<uchar>(y, x);
        }
    }

    return output;
}

Mat backScaling(Mat src, float Sx, float Sy)
{
    if (Sx <= 0 || Sy <= 0)
    {
        cout << "Negative scaling is not possible";
        return src;
    }

```

```
49      Mat output(src.rows*Sy, src.cols*Sx, CV_8U, Scalar(0, 0, 0))
        ;
50      int tmpy, tmpx;
51
52      for (int y = 0; y < output.rows; ++y)
53      {
54          for (int x = 0; x < output.cols; ++x)
55          {
56              tmpy = roundNum(1 / Sy*y);
57              tmpx = roundNum(1 / Sx*x);
58
59              if (tmpy >= src.rows)
60                  tmpy = src.rows - 1;
61
62              if (tmpx >= src.cols)
63                  tmpx = src.cols - 1;
64
65              output.at<uchar>(y, x) = src.at<uchar>(tmpy, tmpx);
66          }
67      }
68      return output;
69  }
70
71  int main(int, char)
72  {
73      Mat image = imread("input.jpg");
74      cvtColor(image, image, CV_BGR2GRAY);
75
76      float Sx, Sy;
77      cout << "Enter horizontal scalar: "; cin >> Sx; cout << endl
        ;
78      cout << "Enter vertical scalar: "; cin >> Sy; cout << endl;
79
80      Mat image2 = simpleScaling(image, Sx, Sy);
81      Mat image3 = backScaling(image, Sx, Sy);
82      imshow("IMAGE", image);
83      imshow("IMAGE2", image2);
84      imshow("IMAGE3", image3);
85      waitKey(0);
86
87      return 0;
88  }
```

Explanation of code

Besides the main function, the code contains two functions that each contain the different scaling algorithms, and a third functions which is a basic number rounding function. The algorithm functions take an image and two floats - the scalars - as arguments.

Both scaling functions start with this if-statement:

```
1  if (Sx <= 0 || Sy <= 0)
```

```
{
    cout << "Negative scaling/scaling to 0 is not possible";
    return src;
}
```

This is simple error prevention; while it could be argued that "negative" scaling could be achieved by flipping the image on both axes, that is not a feature of this software. A scalar with the value of "0" would result in an image width or height of 0 which is also no good.

The rest of the forward mapping function is as follows:

```
//Create a Mat object with scaled dimensions, greyscale,
   completely black.
Mat output(src.rows*Sy,src.cols*Sx, CV_8U, Scalar(0, 0, 0));

for (int y = 0; y < src.rows; ++y)
{
    for (int x = 0; x < src.cols; ++x)
    {
        output.at<uchar>(y*Sy, x*Sx) = src.at<uchar>(y, x);
    }
}
return output;
```

So, for each pixel in the input image, it takes the most appropriate pixel of the output image, and makes it equal to the current pixel.

From executing bits of code, I found that when a floating point number is interpreted as an integer, it is simply rounded down. I wanted something a bit more precise than that, so I created this function:

```
int roundNum(float num) {

    if (num - floor(num) >= 0.5)
        num = ceil(num);
    else
        num = floor(num);

    return num;
}
```

It simply rounds a floating point number to it's nearest integer. I found that this resulted in pixels potentially being mapped out of bounds of the output image. So besides the rounding and the backward mapping, there is another slight change to the other function. If a pixel has a position outside the array, it is simply mapped to the outmost position within the array. Here is the code:

```
Mat output(src.rows*Sy, src.cols*Sx, CV_8U, Scalar(0, 0, 0));
int tmpy, tmpx;

```

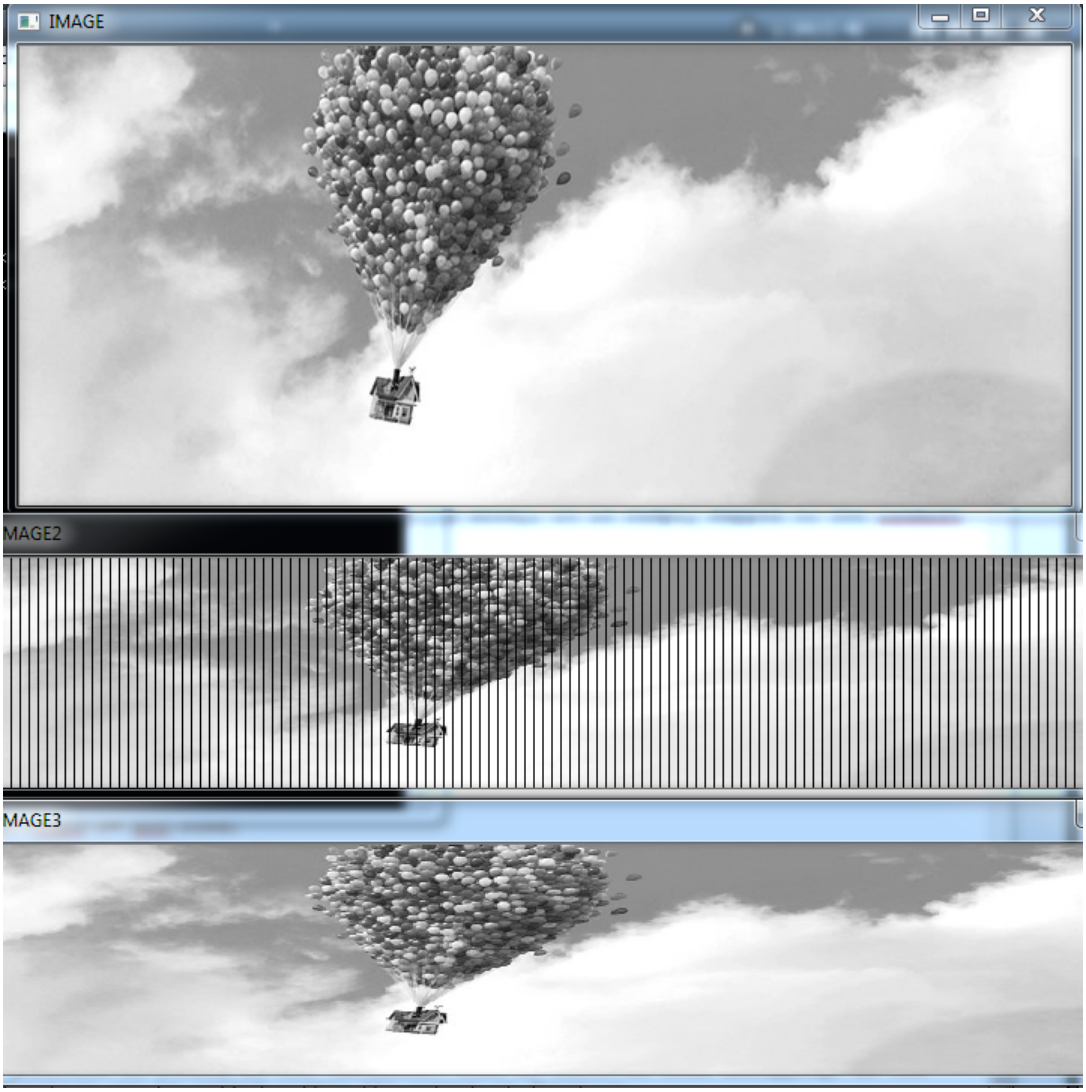
```
for (int y = 0; y < output.rows; ++y)
{
    for (int x = 0; x < output.cols; ++x)
    {
        tmpy = roundNum(1 / Sy*y);
        tmpx = roundNum(1 / Sx*x);

        if (tmpy >= src.rows)
            tmpy = src.rows - 1;

        if (tmpx >= src.cols)
            tmpx = src.cols - 1;
        output.at<uchar>(y, x) = src.at<uchar>(tmpy, tmpx);
    }
}
return output;
```

The pixel position is stored in two variables. This is leftover from debugging, as the roundNum function can be called within the output.at function.

Finally i will show the effects of this software in action. I put in 1.2 and 0.5 as my Sx and Sy scalars, and this is the result:



The top image is the original, the one in the middle is scaled with forward mapping and the bottom image is scaled with backward mapping. The image scaled with forward mapping has some noticeable black vertical lines which are not present on the bottom image.