Image filters on BMP-files

Group 3: Hampus Berg, Isak Nilsson and Thea Arvidsson

Project in Program Design and Data Structures, spring 2021

Contents

1 Introduction

Image processing is a very common thing now when smartphones are widely used all over the world. Many social media apps like Tik Tok, Instagram and Snapchat let the user apply different filters to their images or videos with just a tap on the screen. The process for the user seems trivial, just choose a filter and in the blink of an eye your picture is now in grey-scale or tap again and you have change faces with your dog. But under the hood there are usually millions of pixels being processed even for the simplest filters.

We chose this project to get a better understanding of image processing and how filters work. We decided to use a simple image file format, BMP, so that we easily could manipulate the pixels. Even tough the structure of a BMP file is simple; it was not an easy task to get to the point where we could change the value of the pixels. The biggest problem was to convert the BMP file into an 2D array. With the 2D array it was easy to fetch specific pixels and manipulate them in different ways.

1.1 Background

For this project we have chosen to work with bitmap image file or BMP for short. BMP uses a rather simple structure to store images, and there were already some modules at hand to extract the information we needed from the file, that is the pixel array. Beyond the pixel array there is a header that store some general information about the image and a DIB header. We don't have to worry about the header's thanks to Codec.BMP [1] which allow us to extract only the pixel array.

Each pixel is represented as four 8-bit values called RGBA; Red, Green, Blue and Alpha, where alpha represent the transparency. An 8-bit value can have a decimal value from 0 – 255. Which means, for example, that the red value is 0, then there is no red in the pixel and if it is 255 there is maximum of red in the pixel. If the value for alpha is 0 the pixel is transparent and if it is 255 it is not transparent, see figure 1.

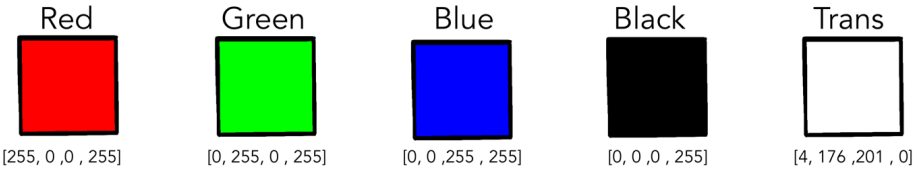

Figure 1: From left to right, a red pixel, a green pixel, a blue pixel, a black pixel and a transparent pixel.

1.2 Summary

The program lets the user load a BMP image file and apply a filter to it. The filters available are:

- Grey-scale
- Box blur
- Brighten
- Flip X
- Flip Y
- Flip XY
- Rotate 90 Right
- Rotate 90 Left
- Rotate 180

2 Use Cases

This section will deal with what modules are needed for the program, and examples on how to run the program.

2.1 Modules

In order for the program to compile two modules needs to be downloaded bmp [1] and HUnit. This will be done in the terminal and will make it possible to work with BMP-files and preform tests.

2.2 Example of running the program

To start the program load the bmp.hs file in ghci. To run the main function, type main in the terminal. The program will now prompt you with a request to specify which BMP file you would like to work with. The second prompt ask what type of filter to use. There are two categorise with filters. The first one is called filter, these filter changes the values for each pixel. While the second one is called rotate/flip, these changes the position of the pixels. After you have chosen which type of filter to use you will be prompted with a list of filter to chose from. After you have made your decision, the image will bee processed and is ready to be saved. You will be asked to input a name for the new image, don't forget to end it with '.bmp'.

Example:
> main
> Enter a file path to a BMP file to work with: sunflower.bmp
> Do you want to add a filter or rotate/flip the image?
> 1. Filter
> 2. Rotate/Flip
> 1
> What filter do you want to apply to the image?
> 1. Grey scale
> 2. Box blur
> 3. Brighten
> 1
> What name do you want for your new picture? Don't forget to end it with .bmp
> sunflowergrey.bmp
> main
See figure 2.

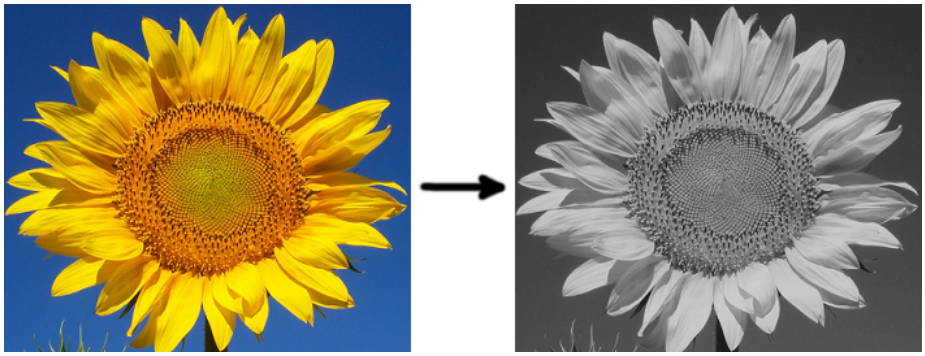

Figure 2: From left to right, original image sunflower.bmp, new image sunflowergrey.bmp with grey scale filter applied.

3 Program Documentation

3.1 Overview

The program is divided in to three main parts, main, loadFileFilter/loadFilePixelMover and saveFile, see figure 3. The 'main' function gets the image from the user, loadFileFilter/loadFilePixelMover loads the image, converts the data and applies a filter and saveFile saves a new image.

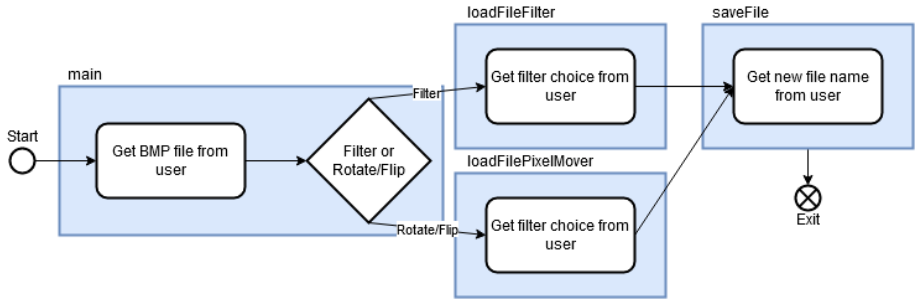

Figure 3: A flowchart over the main functions of the program.

3.2 Abstract data type

To build an pixel array we have used Data.Array [2]. The function of Array's interface we have used are:

```
array :: Ix i => (i, i) -> [(i, e)] -> Array i e
(!) :: Ix i => Array i e -> i -> e
bounds :: Array i e -> (i, i)
elems :: Array i e -> [e]
```

3.3 Data type

```
type Picture = [Word8]
```

The data type Picture represents a photo as a list of Word8, the length of Picture must be equal to or bigger than four and must have an increment of four. In short the entire photo will be stored inside Picture.

```
type Pixel = [Word8]
```

Pixel is also a list of Word8, the difference from Picture is that in Pixel the length of Pixel must be equal to four. Inside this data type is where the RGB value is stored, for example a Pixel would look like this [R, G, B, A].

```
type PixelBox = [Word8]
```

PixelBox represents all of the pixels surrounding a specific pixel represented as a list of Word8, the length of PixelBox must be equal to or bigger than 32. PixelBox is used to easily access all pixels surrounding a pixel.

```
type PixelPosition = (Int,Int)
```

PixelPosition represents the index of a given pixel in an array as a double of Int's. Both of these values must be larger than one because the array's index that store all pixels start at (1,1). This data type is also necessary to be able to find and manipulate surrounding pixels.

```
type Dimensions = (Int, Int)
```

This data type represents the width and height of an image as a double where both values are an Int. These values must also be greater than one as an image can't have negative dimensions as it then would not exist. This data type is very helpful for when we want to for instance rotate or flip an image.

```
type Filter = String
```

Filter represents the name of which filter the user wants to use as a String. The Filter must exist in either filterList or pixelMoverList as an option.

3.4 Algorithms and Functions

3.4.1 filterList

```
filterList :: [(Filter, PixelPosition -> Dimensions -> Array Dimensions
    Picture -> Pixel)]
filterList = [ ("1", greyScale)
             , ("2", boxBlur)
             , ("3", brighten)
             ]
```

This function associates elements inside a list with key value pairs. In short, each element inside this list is assigned a value and each element represents a different filter which can be applied to a single pixel. For example, 1 is assigned as greyScale's key value and 2 is assigned to boxBlur.

3.4.2 pixelMoverList

```
pixelMoverList :: [(Filter, Dimensions -> Array Dimensions Picture ->
    Array Dimensions Picture)]
pixelMoverList = [ ("1", flipX)
```

```
    , ("2", flipY)
    , ("3", flipXY)
    , ("4", rotate90R)
    , ("5", rotate90L)
    , ("6", rotate180)
    ]
```

This function is basically the same as filterList, the only difference being that this list represents all the filters which will change the positions of the pixels rather than manipulating the pixel itself.

3.4.3 main

```
main :: IO ()
main = do
    putStr "Enter a file path to a BMP file to work with: "
    file <- getLine
    putStrLn "Do you want to add a filter or rotate/flip the image?"
    putStrLn "1. Filter\n2. Rotate/Flip"
    choice <- getLine
    if choice == "1"
      then loadFileFilter file
      else if choice == "2"
           then loadFilePixelMover file
           else error "Choose between 1 or 2."
```

The program starts when the main function is called which functions as a menu. Different inputs will result in different paths. First thing that happens after main is called is that the user will be asked to enter a file path to the BMP-file the user would like to work with. After entering the file path to said image the user will be asked which type of filter to apply to the image, either filter or rotation/flip. Once the type has been chosen the program will call loadFile-Filter or loadFilePixelMover, depending on the choice and with file path as the argument.

3.4.4 loadFileFilter

```
loadFileFilter :: FilePath -> IO ()
loadFileFilter file = do
    Right bmp <- readBMP file
    putStrLn "What filter do you want to apply to the image?"
    putStrLn "1. Grey scale\n2. Box blur\n3. Brighten"
    filter <- getLine
    let dimensions = bmpDimensions bmp
        rgba       = unpackBMPToRGBA32 bmp
        unpacked   = BS.unpack rgba
```

```
        pArray      = pixelArray dimensions unpacked
        newArray'   = newArray filter dimensions pArray
        newBMP      = concat $ arrayToPicture dimensions newArray'
        newRGBA     = BS.pack newBMP
    saveFile newRGBA dimensions
```

loadFileFilter takes the file path selected from main as an argument. The function first reads the BMP-file and then gives the user a choice: "What filter do you want to apply to the image?", different alternatives are shown: "1. greyScale, 2. boxBlur, 3. brighten". The user selects one by writing the corresponding number. Depending on which number were selected different functions will be called. for example, if 1 were selected the function greyScale will be called.

The image dimensions are then stored in a variable, 'dimensions'. The image is the unpacked two times, one to get the ByteString data and the second one to get the Word8 data. The list with Word8, 'unpacked', is then added to a 2 dimensional array with the function pixelArray. A new array is created from the other one with the function newArray. It is also here the filter gets applied. A new Word8 list, 'newBMP', is made with the arrayToPixel function from the new array. The new Word8 list is then converted into a ByteString, 'newRGBA', that could be stored to a BMP-file.When the filter has been applied and a new BytesTring have been made, saveFile will be called with the ByteSTring, newRGBA, and the dimensions for the image.

3.4.5 loadFilePixelMover

```
loadFilePixelMover :: FilePath -> IO ()
loadFilePixelMover file = do
    Right bmp <- readBMP file
    putStrLn "What rotate/flip do you want to apply to the image?"
    putStrLn "1. Flip X\n2. Flip Y\n3. Flip XY\n4. Rotate 90 Right\n5.
        Rotate 90 Left\n6. Rotate 180"
    filter <- getLine
    let (Just filter') = lookup filter pixelMoverList
        dimensions     = bmpDimensions bmp
        rgba           = unpackBMPToRGBA32 bmp
        unpacked       = BS.unpack rgba
        pArray         = pixelArray dimensions unpacked
        newArray'      = filter' dimensions pArray
        newDimensions  = bounds newArray'
        newBMP         = concat $ arrayToPicture (snd newDimensions)
            newArray'
        newRGBA        = BS.pack newBMP
    saveFile newRGBA (snd newDimensions)
```

loadFilePixelMover takes the file path selected from main as an argument. The functions starts by reading the BMP-file and then gives the user a choice: "What rotate/flip do you want to apply to the image?", different alternatives are shown:

"1. Flip X 2. Flip Y 3. Flip XY 4. Rotate 90 Right 5. Rotate 90 Left 6. Rotate 180". The user selects one by writing the corresponding number. Depending on which number were selected different functions will be called. for example, if 1 were selected the function flipX will be called.

The rest of the function is the same as loadFileFilter except for the way the filter is applied. Instead of calling newArray to build the new array the filter function take care of that. Another difference is that the dimensions is collected from the new array, this is because after a rotation of the image the height and width changes if the image is not a square.

3.4.6 pixelArray

```
pixelArray :: Dimensions -> Picture -> Array Dimensions Picture
pixelArray (x,y) pl = array ((1,1),(x,y)) [((i,j), take 4 (drop
    (4*((j*x-(x-i))-1)) pl)) | j <- [1..y], i <- [1..x]]
```

The purpose of pixelArray is to transform a list of Word8 into a two-dimensions array. The function takes the dimensions of the image and a list of Word8 elements as arguments. pixelArray makes an index from (1,1) to (x,y) and sequentially walk 1 to y and 1 to x. It first walks until j is equal to y and then increases i one step. For every step, 'take 4 (drop (4*((j*x-(x-i))-1)) pl))' make sure the right pixel is getting access by dropping every pixel before it and taking 4 elements from the list, consequently a pixel has been stored in the array.

3.4.7 getPixelBox

```
getPixelBox :: PixelPosition -> Dimensions -> Array Dimensions Picture -> Array Dimensions Picture
getPixelBox (i,j) (x,y) pArray | i == 1 && j == 1 = array ((1,1),(2,2)) [((p,q), pArray ! (p,q))           | p <- [1..2], q <- [1..2]]
                               | i == 1 && j == y = array ((1,1),(2,2)) [((p,q), pArray ! (p,y-(q-1)))     | p <- [1..2], q <- [1..2]]
                               | i == x && j == 1 = array ((1,1),(2,2)) [((p,q), pArray ! (x-(p-1),q))     | p <- [1..2], q <- [1..2]]
                               | i == x && j == y = array ((1,1),(2,2)) [((p,q), pArray ! (x-(p-1),y-(q-1))) | p <- [1..2], q <- [1..2]]
                               | i == 1           = array ((1,1),(2,3)) [((p,q), pArray ! (p,j-(2-q)))     | p <- [1..2], q <- [1..3]]
                               | j == 1           = array ((1,1),(3,2)) [((p,q), pArray ! (i-(2-p),q))     | p <- [1..3], q <- [1..2]]
                               | i == x           = array ((1,1),(2,3)) [((p,q), pArray ! (x-(2-p),j-(2-q))) | p <- [1..2], q <- [1..3]]
                               | j == y           = array ((1,1),(3,2)) [((p,q), pArray ! (i-(2-p),y-(2-q))) | p <- [1..3], q <- [1..2]]
                               | otherwise        = array ((1,1),(3,3)) [((p,q), pArray ! (i-(2-p),j-(2-q))) | p <- [1..3], q <- [1..3]]
```

Figure 4: We decided to go with a screenshot of the code here rather as the indentation made the code incredibly difficult to read if put into LaTeX as previously done.

getPixelBox takes three arguments, PixelPosition, Dimensions and an Array, see figure 4. getPixelBox will be called upon by the boxBlur function. This allows the function to find the surrounding pixels of a pixel. If the PixelPosition is one of the corners of the array, a 2x2 array will be made with those pixels. If the PixelPosition is on one off the edges, a 2x3 or 3x2 array will be made, depending

on which corner it is PixelPosition is on. If PixelPosition is not in the corner or on the edge a 3x3 array will be made of the surrounding pixels.

3.4.8 newArray

```
newArray :: Filter -> Dimensions -> Array Dimensions Picture -> Array
    Dimensions Picture
newArray filter (x,y) orgArray = array ((1,1),(x,y)) [((i,j), filter'
    (i,j) (x,y) orgArray) | j <- [1..y], i <- [1..x]]
  where (Just filter') = lookup filter filterList
```

This function's job is to apply a chosen filter to an array and then return a new array after the filter's function has been applied. newArray takes three arguments, Filter, Dimensions and an Array. The function looks up which filter is supposed to be used from filterList. newArray works in the same way as pixelArray with the different that newArray only have to look up the pixel value from the original array, 'orgArray'. The pixel value are then passed as argument to the chosen filter.

3.4.9 arrayToPicture

```
arrayToPicture :: Dimensions -> Array Dimensions Picture -> [Picture]
arrayToPicture (x,y) p = [p ! (i,j) | j <- [1..y], i <- [1..x]]
```

arrayToPicture take dimensions and an array ass arguments. The function builds a list containing Pictures as elements. Because of this loadFileFilter and have to concat the list, so it becomes a Picture type. This take the elements the other way around from the array than how newArrayd does it. It lets i go up to x before it increases j one step. This way the output from arrayToPicture will have the same order as they were read from the image.

3.4.10 saveFile

```
saveFile :: BS.ByteString -> Dimensions -> IO ()
saveFile rgba (x,y) = do
    let bmp = packRGBA32ToBMP x y rgba
    putStrLn "What name do you want for your new image? Don't forget to
        end it with .bmp"
    name <- getLine
    writeBMP name bmp
```

The functions loadFilePixelMover and loadFileFilter both call saveFile after a filter has been applied. saveFile takes the edited image and saves it as a new BMP-file in the same folder as the program. The function starts of by packing the ByteString values into a BMP-file. Secondly it asks the user: "What name do you want for your new picture? Don't forget to end it with .bmp". If the user fails to give the new image a filename that ends with '.bmp' an error appears. Otherwise the image is saved and the program will stop.

3.4.11 flipX

```
flipX :: Dimensions -> Array Dimensions Picture -> Array Dimensions
    Picture
flipX (x,y) orgArray = array ((1,1),(x,y)) [((x-(i-1),j), orgArray !
    (i,j)) | j <- [1..y], i <- [1..x]]
```

flipX takes dimensions and an array as arguments. flipX will flip the 2D array on it's y-axis. In order to change x and y they are renamed to (i,j) where j represents an element from a list 1 to y and i represents an element from a list 1 to x. Every pixels y-value will stay the same but the x-values will be changes with the algorithm x-(i-1). Effectively moving the pixel to a new position the same distance from right edge of the picture as it were from the left edge of the picture. This result the image being flipped for the x values, see figure 5.

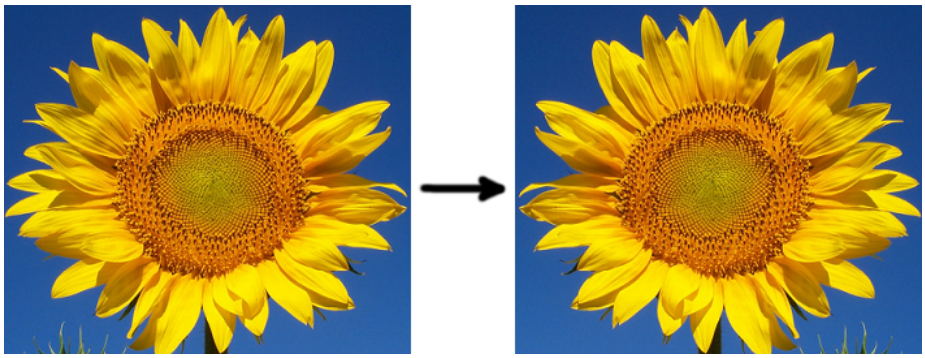

Figure 5: From left to right, original image sunflower.bmp, new image with flip x filter applied.

3.4.12 flipY

```
flipy :: Dimensions -> Array Dimensions Picture -> Array Dimensions
    Picture
flipy (x,y) orgArray = array ((1,1),(x,y)) [((i,y-(j-1)), orgArray !
    (i,j)) | j <- [1..y], i <- [1..x]]
```

flipy takes dimensions and an array as arguments. flipy will flip the 2D array on it's y-axis. In order to change x and y they are renamed to (i,j) where j represents an element from a list 1 to y and i represents an element from a list 1 to x. Every pixels x-value will stay the same but the y-values will be changes with the algorithm y-(j-1). Effectively moving the pixel to a new position the same distance from top of the picture as it were from the bottom of the picture. This result the image being flipped for the y values, see figure 6.

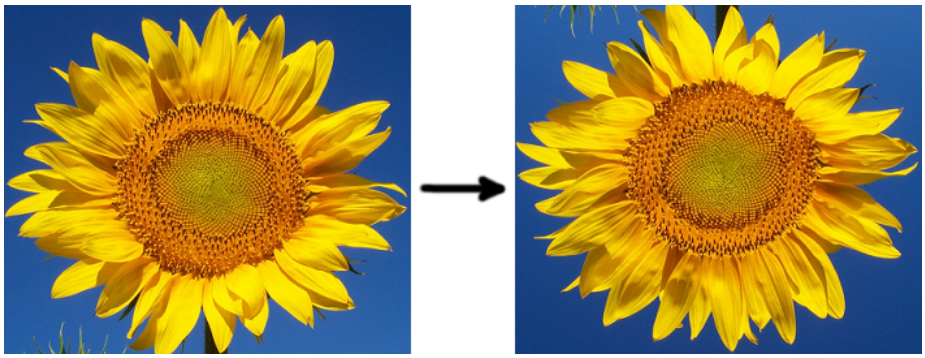

Figure 6: From left to right, original image sunflower.bmp, new image with flip y filter applied.

3.4.13 flipXY

```
flipXY :: Dimensions -> Array Dimensions Picture -> Array Dimensions
    Picture
flipXY (x,y) orgArray = array ((1,1),(x,y)) [((x-(i-1),y-(j-1)),
    orgArray ! (i,j)) | j <- [1..y], i <- [1..x]]
```

flipXY takes dimensions and an array as arguments. flipXY will flip the 2D array on it's x and y-axis. In order to change x and y they are renamed to (i,j) where j represents an element from a list 1 to y and i represents an element from a list 1 to x. The x-values will be changes with the algorithm x-(i-1) and the y-values will be changes with the algorithm y-(j-1). Effectively moving the pixel to a new position the same distance from top of the picture and as it were from the bottom of the picture and from right edge of the picture as it were from the left edge of the picture. This result the image being flipped for the x and y values, see figure 7.

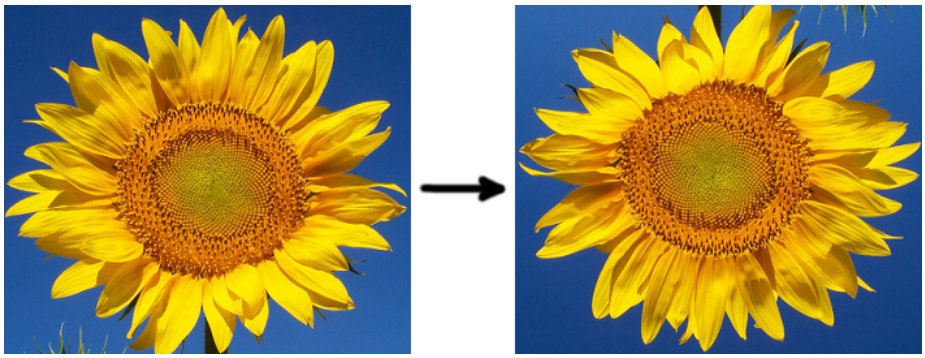

Figure 7: From left to right, original image sunflower.bmp, new image with flip xy filter applied.

3.4.14 rotate90R

```
rotate90R :: Dimensions -> Array Dimensions Picture -> Array Dimensions
    Picture
rotate90R (x,y) orgArray = array ((1,1),(y,x)) [((j,x-(i-1)), orgArray !
    (i,j)) | j <- [1..y], i <- [1..x]]
```

rotate90R takes dimensions and an array as arguments. rotate90R will rotates the image 90 degrees to the right, see figure 8. In order to change x and y they are renamed to (i,j) where j represents an element from a list 1 to y and i represents an element from a list 1 to x. The x and y values in the index also need to be flipped to allow a non square image to be flipped. The x-values will get it position from j and the y-values will be changes with the algorithm x-(i-1).

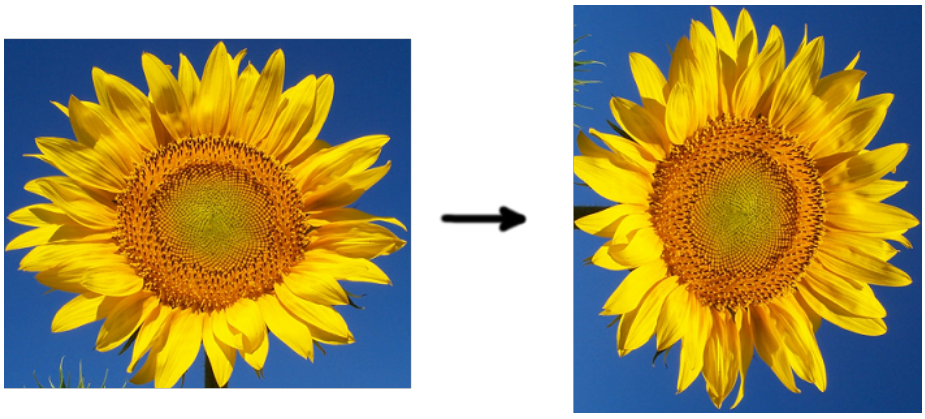

Figure 8: From left to right, original image sunflower.bmp, new image with rotate 90 right filter applied.

3.4.15 rotate90L

```
rotate90L :: Dimensions -> Array Dimensions Picture -> Array Dimensions
    Picture
rotate90L (x,y) orgArray = array ((1,1),(y,x)) [((y-(j-1),i), orgArray !
    (i,j)) | j <- [1..y], i <- [1..x]]
```

rotate90L takes dimensions and an array as arguments. rotate90L will rotates the image 90 degrees to the left, see figure 9. In order to change x and y they are renamed to (i,j) where j represents an element from a list 1 to y and i represents an element from a list 1 to x. The x and y values in the index also need to be flipped to allow a non square image to be flipped. The x-values will get it position from y-(j-1) and the y-values will be changes with the algorithm i.

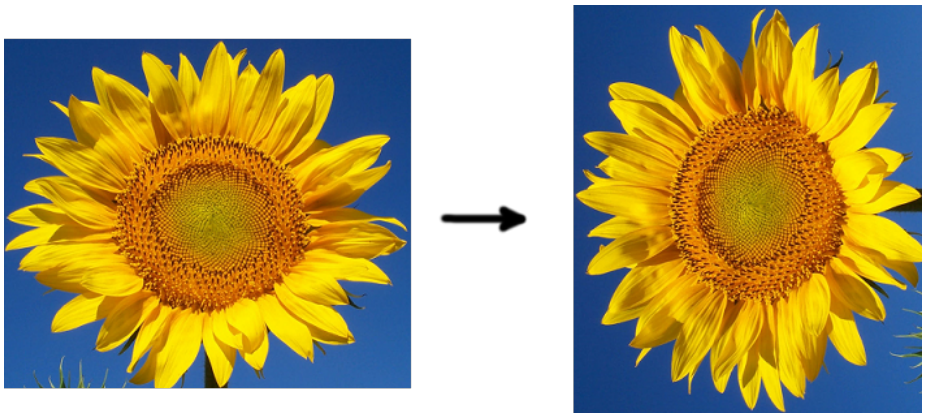

Figure 9: From left to right, original image sunflower.bmp, new image with rotate 90 left filter applied.

3.4.16 rotate180

```
rotate180 :: Dimensions -> Array Dimensions Picture -> Array Dimensions
    Picture
rotate180 d orgArray = rotate90R (swap d) (rotate90R d orgArray)
```

rotate180 takes dimensions and an array as arguments. rotate180 will rotates the image 180 degrees to the right, see figure 10. To achieve this rotate180 calls on rotate90T two times on the array.

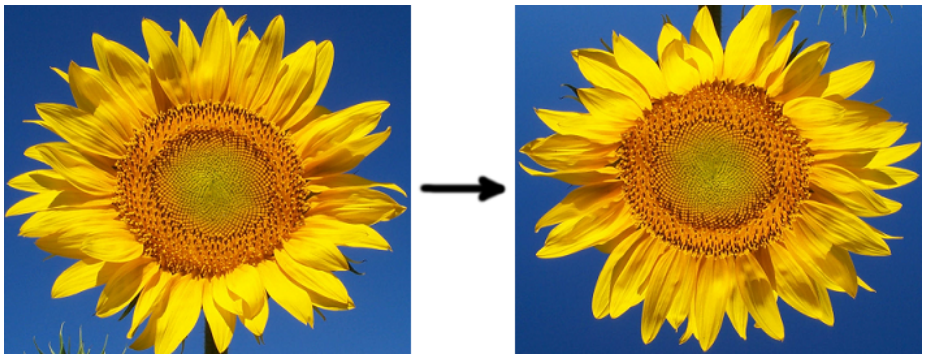

Figure 10: From left to right, original image sunflower.bmp, new image with rotate 180 filter applied.

3.4.17 greyScale

```
greyScale :: PixelPosition -> Dimensions -> Array Dimensions Picture ->
    Pixel
greyScale (i,j) _ orgArray =
  let greyScale' [r, g, b, a] = [avg, avg, avg, a]
       where avg = div r 3 + div g 2 + div b 10
  in  greyScale' (orgArray ! (i,j))
```

The purpose of greyScale is to change the color scheme of an Pixel into grey scale, see figure 11 for the effect applied to a whole image. greyScale takes three arguments, PixelPosition, Dimensions and the array retrieved from the original image. greyScale will be applied to each pixel found within the array separately. The method we chose to go with for creating a grey scale color scheme is known as the luminosity method. The reasoning behind this is that the human eye does not weigh all colors equally, for instance we are more sensitive to green than other colors. Using this method we were therefore able to create a sort of weighted average to account for said perception. The formula is as follows, red is weighted as 0.21, green as 0.72 and blue as 0.07. Unfortunately it is not possible to use fractionals within a ByteString or Word8, we therefore had to compromise by rounding the numbers evenly, however the result was not compromised and the greyScale function appears very true to it's intent.

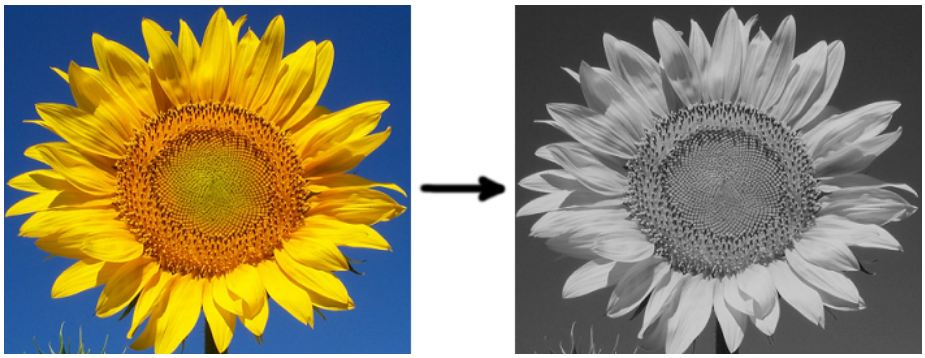

Figure 11: From left to right, original image sunflower.bmp, new image with grey scale filter applied.

3.4.18 brighten

```
brighten :: PixelPosition -> Dimensions -> Array Dimensions Picture ->
    Pixel
brighten (i,j) _ orgArray =
  let brighten' [r, g, b, a] = [newR, newG, newB, a]
            where newR | r < 206 = r+50
                       | otherwise = 255
                  newG | g < 206 = g+50
                       | otherwise = 255
                  newB | b < 206 = b + 50
                       | otherwise = 255
  in brighten' (orgArray ! (i,j))
```

brighten takes PixelPosition, Dimension and an array as arguments. brighten modifies the r, g, b values and thereby change the Pixel color scale to a brighter version, see figure 12 for the effect applied to a whole image. The function starts off with the r value. First it settles if r is lesser than 206, if that is the case the old r value is added with 50. If r was greater then 206 the new r value becomes 255. The same process is being done for the g and b values. The reason brighten checks if r, g or b is greater then 206 is because the final value can not be greater then 255 due to the limitations of ByteString.

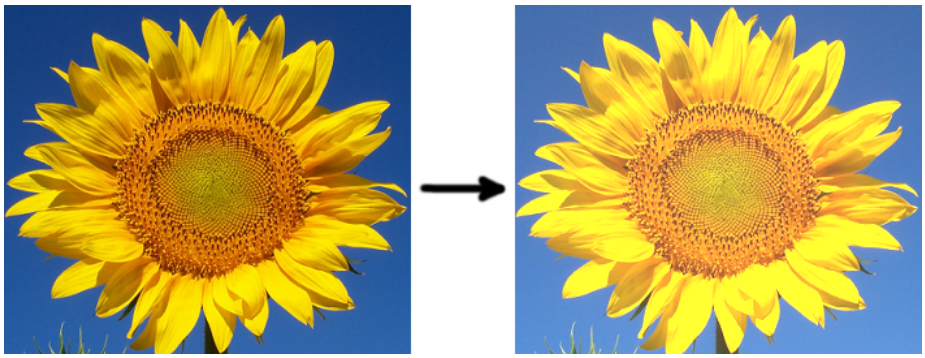

Figure 12: From left to right, original image sunflower.bmp, new image with brighten filter applied.

3.4.19 boxBlur

```
boxBlur :: PixelPosition -> Dimensions -> Array Dimensions Picture ->
    Pixel
boxBlur (i,j) (x,y) pArray = getAverageColorValue $ concat $ elems $
    getPixelBox (i,j) (x,y) pArray
```

boxBlur takes PixelPosition, Dimensions and an array as arguments. boxBlur calls on a helper function called getAverageColorValue and gets a PixelBox from getPixelBox as a agrument. getAverageColorValue returns a Pixel with average colors values from the PixelBox. When applied to all pixels in an Images the results will be a blurred image, see figure 13.

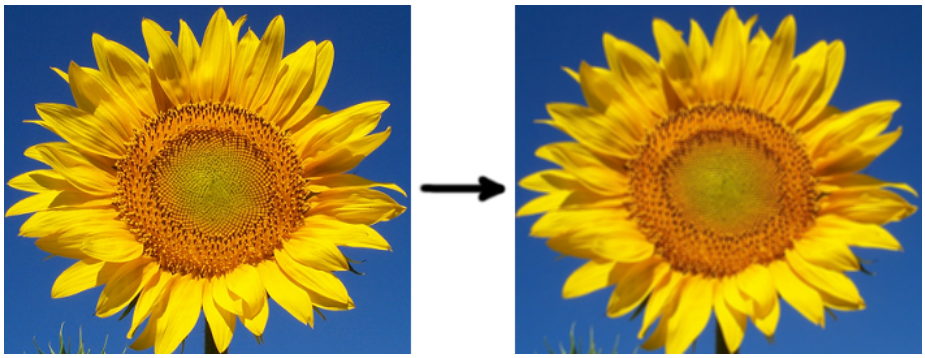

Figure 13: From left to right, original image sunflower.bmp, new image with box blur filter applied.

3.4.20 getAverageColorValue

```
getAverageColorValue :: PixelBox -> Pixel
getAverageColorValue pb =
  let
    getAverageColorValue' [] r' g' b' a' n        = [toEnum (div r' n),
        toEnum (div g' n), toEnum (div b' n), toEnum (div a' n)]
    getAverageColorValue' (r:g:b:a:rest) r' g' b' a' n =
        getAverageColorValue' rest (fromEnum r+r') (fromEnum g+g')
        (fromEnum b+b') (fromEnum a+a') (n+1)
  in
    getAverageColorValue' pb 0 0 0 0 0
```

getAverageColorValue takes PixelBox as an argument. This function gets the average value of the colors based on a list of pixels which it gets from the PixelBox. The function therefore return a Pixel with four values which represent the average r, g, b and a values from the PixelBox. In order to divided the values we had to convert the Word8 type to an Int and then back again after the division.

4 Discussion

4.1 Shortcomings

We initially had bigger plans for the program, we wanted to implement more complex filter. For example, an edge finding filter. But it took longer time to get started with the filters than we had expected. We first had problems getting the data from the image and then we had more problems with organizing the data in an easily accessible manner. Aiming to write more complex filters, we needed a way to access the surrounding pixel of the pixel we were working on. In order to achieve that we wrote the function getPixelBox. With getPixel box we could write the box blur filter, but did not have time to write the edge finding filter.

The program is painfully slow when working with larger images. This is because each pixel first need to be converted to a ByteString, then to a Word8, then the filter gets applied to the pixel. If it is the box blur, up to 8 more pixels needs to get involved to achieve the effect. Then the pixel needs to be converted to a Word8, ByteString and finally saved as an image. We will not get in to the time complexity but imagine an image that is 1000x1000 pixels, that is 1.000.000 pixels that needs to go through each operation. And our program struggles with that. But because our main focus with this project was to deepen our understanding of image processing and how filters work, we did not concern ourselves with the time complicity. But it would be nice if the program could process images a bit faster.

References

[1] *Codec.BMP*. URL: https://hackage.haskell.org/package/bmp-1.2.4.1/docs/Codec-BMP.html.

[2] *Data.Array*. URL: https://hackage.haskell.org/package/array-0.5.2.0/docs/Data-Array.html.