How to change image colours.



A short Explanation on how to change image colours in C#.

Version Control Table

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Name | Details of Changes |
| 09 September 2024 | 0.0.1 | Intelligent Engineer | Original |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Contents

[1. Contents 4](#_Toc177147255)

[2. Introduction 6](#_Toc177147256)

[2.1. Numbering 6](#_Toc177147257)

[2.2. Floating Point Numbers 7](#_Toc177147258)

[2.3. Bits and Bytes 7](#_Toc177147259)

[2.4. Hex Editor 7](#_Toc177147260)

[2.5. Representing Pixel Colours 8](#_Toc177147261)

[3. Creating the test image 9](#_Toc177147262)

[4. Investigating how to change colours 11](#_Toc177147263)

[5. Worked examples of scaling pixels 13](#_Toc177147264)

[5.1. Scaling the first pixel of the test image 13](#_Toc177147265)

[5.1.1. Scaling the Red byte 13](#_Toc177147266)

[5.1.2. Scaling the Green byte 14](#_Toc177147267)

[5.1.3. Scaling the Blue byte 15](#_Toc177147268)

[5.1.4. Scaling the Alpha byte 15](#_Toc177147269)

[5.1.5. Translation (image colour cast) 16](#_Toc177147270)

[5.1.6. Scaled Value for first pixel 16](#_Toc177147271)

[5.1. Using second, third and fourth pixel of test image 17](#_Toc177147272)

[5.2. Using fifth pixel of test image 17](#_Toc177147273)

[5.2.1. Scaling the Red byte 17](#_Toc177147274)

[5.2.1. Scaling the Green byte 17](#_Toc177147275)

[5.2.2. Scaling the Blue byte 18](#_Toc177147276)

[5.2.3. Scaling the Alpha byte 18](#_Toc177147277)

[5.2.1. Translation (image colour cast) 19](#_Toc177147278)

[5.2.1. Scaled Value for fifth pixel 19](#_Toc177147279)

[5.2.1. Check values are correct 19](#_Toc177147280)

[5.3. Alpha Byte 21](#_Toc177147281)

[5.3.1. Alpha Scaling the pixels Alpha component 21](#_Toc177147282)

[5.3.2. Single Colour scaling the Alpha component 22](#_Toc177147283)

[5.3.3. Two Colour scaling the Alpha component 23](#_Toc177147284)

[5.3.4. Full Colour scaling the Alpha component 23](#_Toc177147285)

[5.3.1. Alpha Scaling of a single colour component 24](#_Toc177147286)

[5.3.1. Alpha Scaling of two-colour components 25](#_Toc177147287)

[5.3.1. Alpha Scaling of all colour component 26](#_Toc177147288)

[5.4. Adding Translation (image colour cast) into the mix 27](#_Toc177147289)

[5.4.1. Example 1 27](#_Toc177147290)

[5.4.1. Example 2 27](#_Toc177147291)

[5.4.1. Worked Example 27](#_Toc177147292)

[5.5. Combining Colour scaling, Colour cast and Alpha (opacity) 29](#_Toc177147293)

[5.5.1. Swapping Red and Green 29](#_Toc177147294)

[5.5.1. Swapping Red and Blue 29](#_Toc177147295)

[5.5.1. Swapping Green and Blue 29](#_Toc177147296)

[5.5.2. Sepia 30](#_Toc177147297)

[5.5.3. Polaroid Colour 30](#_Toc177147298)

[5.5.4. Monochrome 30](#_Toc177147299)

[5.5.5. White to Alpha 30](#_Toc177147300)

[6. The Code behind it 31](#_Toc177147301)

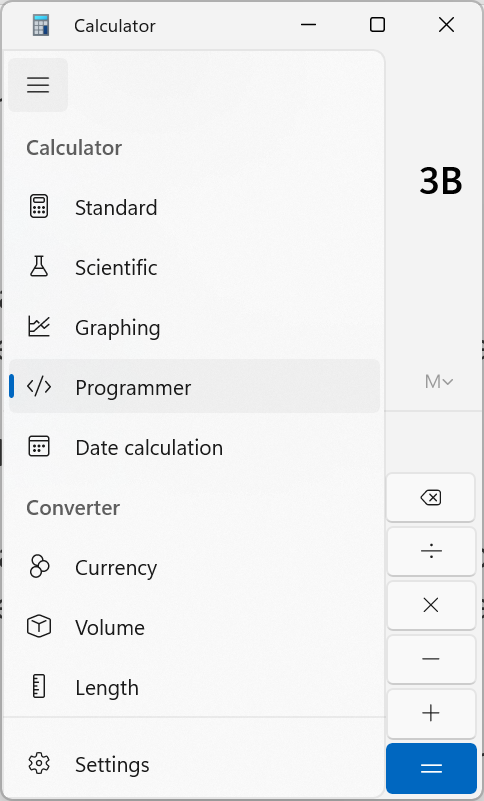
# Introduction

## Numbering

Throughout this explanation we will use three numbering systems. If you are not familiar with these they will be:

* Decimal
  + Which is base 10.
  + Decimal numbers start at 0 through to 9.
  + Decimal numbers are written from right (LSB) to left (MSB) and are derived from the Indian/Arabic numbering systems.
  + At times to identify these we will add the base as a subscript to the value e.g. 2110
* Hexadecimal also just called Hex
  + Which is base 16.
  + Hex numbers start at 0 through to 9 then A through to F.
  + At times to identify these we will add the base as a subscript to the value e.g. 3B16
* Binary
  + Which is base 2.
  + Binary numbers are either 0 or 1.
  + At times to identify these we will add the base as a subscript to the value e.g. 010012

If you are not familiar with these you can convert “Hex to Decimal” and “Decimal to Hex” using the calculator that comes with Windows. Other operating systems will have similar features. To do this change the calculator into Programmer mode, by clicking on “≡” in the top left and selecting “</> Programmer”



On the left around the middle, you will see the word HEX and below it DEC then OCT and BIN. Click on the one you want to work with and enter the number you want to convert. In Hex mode you will have access to the letters A through F. In Binary mode you only have access to 1 or 0. As you enter the number as we have below the value will be updated for the other numbering systems.

|  |  |  |
| --- | --- | --- |
| Hexadecimal | Decimal | Binary |
|  |  |  |

## Floating Point Numbers

A floating-point number is a type of number with a floating decimal point where the decimal point can "float" to any position necessary. Floating-point numbers can be positive or negative numbers with a decimal point.

## Bits and Bytes

In this explanation we will talk about bits and bytes. A byte is made up of 8 bits. A bit is binary (base 2) and can be either 0 or 1. A byte is the hex representation (base 16) of the 8 bits. We saw above that we can use the calculator to translate between the two numbering systems.

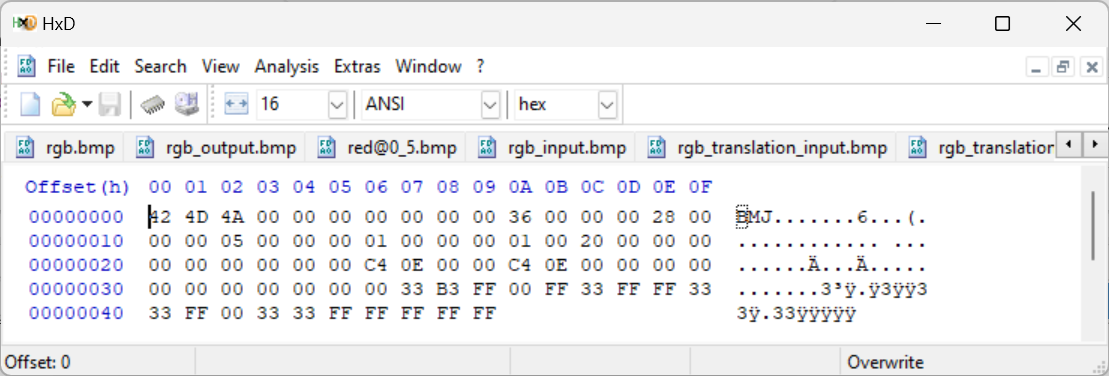
In the above example we see that: 0011110112 = 3B16

A Hex byte can represent 25610 values from 010 to 25510 (Starting with the first item of 010 which is the first item then this making 25510 the 256th item.) This is represented in Hex as 0016 to FF16.

You can deduce that a single byte can never be smaller than 010 nor greater than 25510. Later we will see in the worked examples that we may end up with a value greater than 255 but this is the ceiling and the byte will remain 255. If we end up with less than 0 the byte remains 0.

## Hex Editor

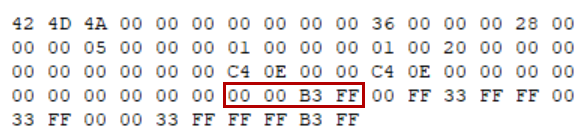
In the explanation we will use a hex editor to look at the bytes inside the saved bitmap image. There are many of these you can use. In this explanation we use HxD which is free and open source and available to download from <https://mh-nexus.de/en/>. Using this app, you can open the image file and look at the bytes as shown below. How to interpret what you see is discussed later in this document.



## Representing Pixel Colours

An image is made up of many pixels. Each pixel has a specific colour. Depending on the type of image you have will dictate how many bytes represent each pixel. In the worked examples we are using 32-bit bitmaps meaning we have 32 bits to represent each pixel’s colour. Each pixel is a solid colour with a Red, Green and Blue component. Each component is 8 bits which is one byte. There is also and Alpha component which determines the opacity of the pixel this is another 8 bits.

With all 4 components (red, green, blue and alpha) 1 byte each we can see that we represent each pixel with 4 bytes. Below we pick out just one pixel from the saved BMP file so you can see the 4 bytes.



The Data can be saved to the file in many ways the app ColourChanger.exe will save them as BGRA.

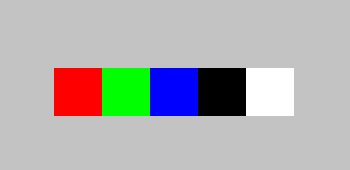
# Creating the test image

There are many explanations of how the ColorMatrix works in C# but none that make sense to me. We will reverse engineer what is happening so we can understand it the way that makes sense.

First, create a small 24bit bitmap in Paint each of the 5 colours is 1x1pixels of solid colour. This is small and even on full magnification difficult to work with.

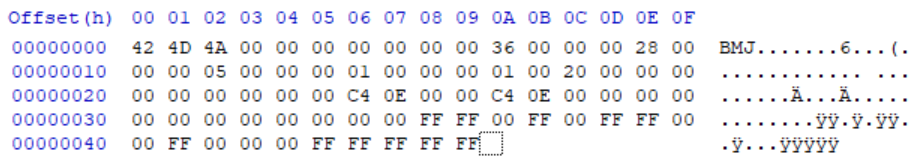
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Alpha | Red | Green | Blue |
| Red | 255 | 255 | 0 | 0 |
| Green | 255 | 0 | 255 | 0 |
| Blue | 255 | 0 | 0 | 255 |
| Black | 255 | 0 | 0 | 0 |
| White | 255 | 255 | 255 | 255 |

This is saved as a 24bit bmp file. This gives us 8 bits for each Red, Green and Blue component but not the Alpha. The image below shows the colours. Ignore the grey border that has been added to make seeing the actual colours easier.



Ideally, we want this as 32bit bmp file and to get that load the saved bmp from paint into the app “ColourChanger.exe” then click “Reset button” and then save the output as “rgb\_input.bmp”. You now have a 32bit bmp file which you can then open as the input file.

Looking at the file in a hex editor we see the following:



Taking the hex bytes which are show in the middle we can then break the file down to see where the actual data is that we are interested in.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Offset in hex** | **Size in bytes** | **Hex string** | **Value** | **Description** |
| **BMP Header** | | | | |
| 00 | 2 | 42 4D | "BM" | ID field (42h, 4Dh) |
| 02 | 4 | 4A 00 00 00 | 74 bytes | Size of the BMP file |
| 06 | 2 | 00 00 | Unused | Application specific |
| 08 | 2 | 00 00 | Unused | Application specific |
| 0A | 4 | 36 00 00 00 | 54 bytes | Offset where the pixel array (bitmap data) can be found |
| **DIB Header** | | | | |
| 0E | 4 | 28 00 00 00 | 40 bytes | Number of bytes in the DIB header (from this point) |
| 12 | 4 | 05 00 00 00 | 5 pixels (left to right order) | Width of the bitmap in pixels |
| 16 | 4 | 01 00 00 00 | 1 pixel (bottom to top order) | Height of the bitmap in pixels |
| 1A | 2 | 01 00 | 1 plane | Number of colour planes being used |
| 1C | 2 | 20 00 | 32 bits | Number of bits per pixel |
| 1E | 4 | 00 00 00 00 | 0 | BI\_BITFIELDS, no pixel array compression used |
| 22 | 4 | 00 00 00 00 | 0 bytes | The image size. This is the size of the raw bitmap data; a dummy 0 can be given for BI\_RGB bitmaps |
| 26 | 4 | C4 0E 00 00 | 3780 pixels/metre horizontal | Horizontal resolution of the image. (pixel per metre, signed integer) |
| 2A | 4 | C4 0E 00 00 | 3780 pixels/metre vertical | Vertical resolution of the image. (pixel per metre, signed integer) |
| 2E | 4 | 00 00 00 00 | 0 colours | Number of colours in the palette |
| 32 | 4 | 00 00 00 00 | 0 important colours | 0 means all colours are important |
| **Start of the Pixel Array (the bitmap Data)** | | | | |
| 36 | 4 | 00 00 FF FF | 0 0 255 255 | Red Pixel (BGRA = 0 0 1 1) |
| 3A | 4 | 00 FF 00 FF | 0 255 0 255 | Green Pixel (BGRA = 0 1 0 1) |
| 3E | 4 | FF 00 00 FF | 255 0 0 255 | Blue Pixel (BGRA = 1 0 0 1) |
| 42 | 4 | 00 00 00 FF | 0 0 0 255 | Black Pixel (BGRA = 0 0 0 1) |
| 46 | 4 | FF FF FF FF | 255 255 255 255 | White Pixel (BGRA = 1 1 1 1) |

The Pixel Array section that we see above is what we are interested in as we try to work out how the ColorMatrix works in changing colours.

# Test Images used in this explanation.

The following is a list of the test images used in this explanation. All images are available in the GitHub repository so you can use the exact images used here to get the same results.

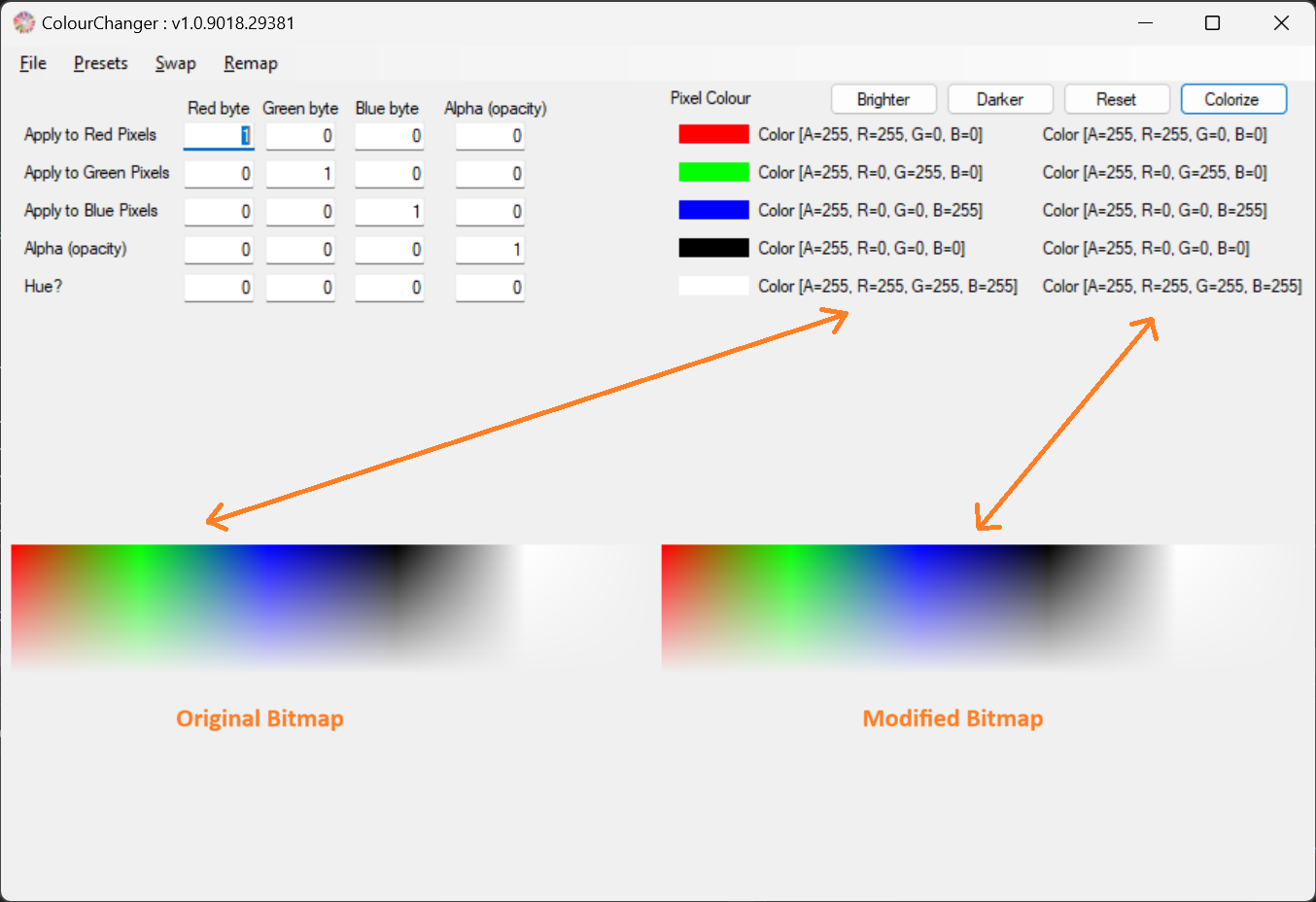
|  |  |  |
| --- | --- | --- |
| **File Name** | **Image** | **Explanation** |
| 100blocks\_input.exe |  | Each colour is exactly 100 pixels high by 100 pixels wide. |
| colourcast.input.exe |  | A small image 568 pixels high by 1221 pixels wide. With the primary colours (Red, Green and Blue) plus areas of no colour (Black) and all colours (white). |
| Rgb\_input.exe |  | Each colour is exactly 1 pixel high by 1 pixel wide. |

# Investigating how to change colours

Use the ColourChanging.exe app but be aware that as we progress through this explanation the interface will change as we become more aware of how the ColorMatrix in C# works.

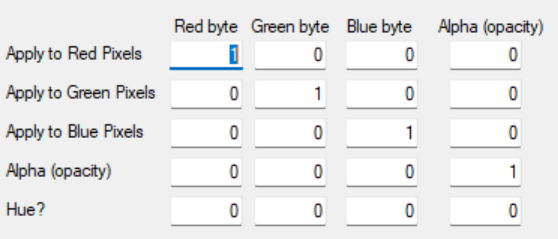
We start the ColourChanging.exe app and load in the “rgb\_input.bmp” we created and saved as explained above. We can see the image of the input and the modified bitmaps. We can also see as shown by the arrows the colour data for each pixel.

Note that the bitmap is small and the app is magnifying the pixels and this is why they appear not to be solid colour. They are solid colour and below in the Pixel Colour column we see the colours of the input pixels.



Now we will start to investigate what happens when we change items in the section shown below. The number that we put in here is a percentage of the original value that will be in the modified bitmap. If we put in 0.2 then that means 20% of original value will be in modified image.

If you put in 1 then that means 100% of the original value will be in the modified image while 0 means 0% of the original value will be in the modified image. In the worked examples we will see how this works with the ARGB values as well as Hue.



Note:

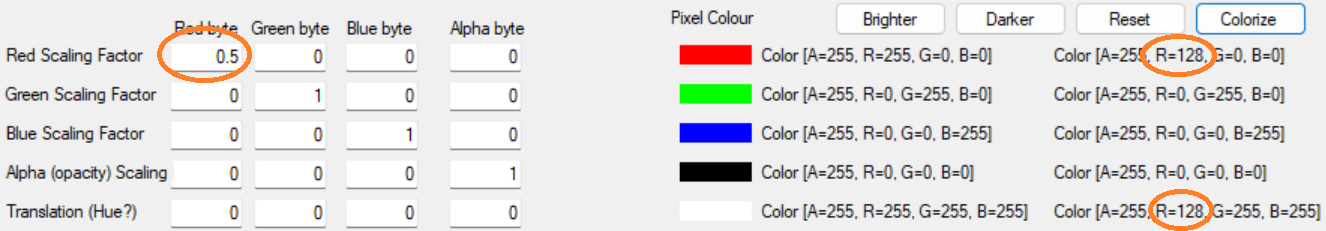
A hex byte has a

* Minimum 8bit value of 000000002 which is 0016 and 010
* Maximum 8bit value of 111111112 which is FF16 and 25510

You can enter any number in the section shown above and it will scale the colour byte to that percentage:

* The meaning of the number is a percentage of the colour byte value:
  + 1 = 100% of the value
  + 0.5 = 50% of the value
  + 0 = 0% of the value
  + 5.5145 = 551.45% of the value
* You can enter values greater than 1 because if the colour byte is 5310 then 100% is still only 5310. In this case we perhaps could have a scaling percentage all the way up to 4.8113% before we reach the ceiling of 25510.
* As soon as the colour byte is equal to 25510 which is FF16 any positive scaling value greater than zero will have no effect on it.
* If you enter values less than 0 the colour byte will be equal to 010 which is 016.
* The numbers you enter here are floating point numbers and can be up to 4 decimal places long e.g. 0.1234.

Using the example below we have R = 255 and we want to scale it 50% of that. Enter 0.5 and you will get the scaled byte scaled to 50% of its original value ∴ 0.5\*255 = 127.5 but this needs to convert to hex so you round up to 128. We can see next that we change the red pixels red component to be 0.5 and this results in the output scaling to 128 as show below:



By scaling the red component of the red pixel, we have also scaled the red component in the white pixel. Save the file and open it in a hex editor and we get the following:

|  |  |
| --- | --- |
| **Image** | **Scaled values** |
|  |  |

The highlighted items are those that tell us the hex values for each pixel and are explained below, be aware that they come out as BGRA rather than ARGB.

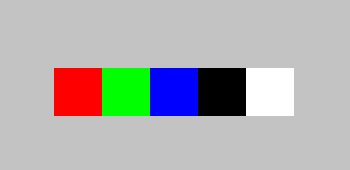
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Blue byte** | **Green byte** | **Red byte** | **Alpha byte** |
| **Red pixel** | 00 | 00 | 80 | FF |
| **Green pixel** | 00 | FF | 00 | FF |
| **Blue pixel** | FF | 00 | 00 | FF |
| **Black pixel** | 00 | 00 | 00 | FF |
| **White pixel** | FF | FF | 80 | FF |

# Worked examples of scaling pixels

We start with the first pixel in the bitmap which happens to be the Red pixel.

* Using this first pixel we have entered that we want to scale its red component by half (0.5).
* We will work through each byte of its RGBA value and working out their new scaled values
* To make it easier to follow we have added coloured borders around certain items which we are going to use.
* For this first pixel we will ignore the translation setting and leave it untouched.
* We will scale each byte of the RGBA in turn starting with R.

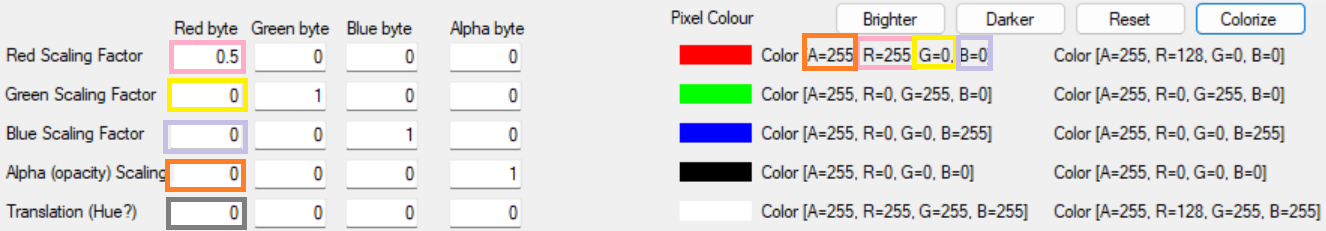
For these worked examples we will be using the test image of 5 pixels as shown below (the grey border has only been added to make it visible in the document it is not in the test image. The test image is 5 pixels wide by 1 pixel high, each colour being 1x1 pixel):



## Scaling the first pixel of the test image

This is pixel(0,0) and in our image this is coloured red with ARGB of Color[A=255, R=255, G=0, B=0]

### Scaling the Red byte



The equation to use is:

(multiply the pink items) add (multiply the yellow items) add (multiply the purple items) add (multiply the orange items) add the grey item which is the colour cast of the image.

The colour cast is always a percentage of full colour, 255, that gets added to ALL pixels in the image to create the image cast. For now, to keep it simple we leave this at 0% and this has no effect on the image. We will explain the concept of the colour cast and how it works it in more depth later on.

Scale red byte value = (255\*0.5) +(0\*0) + (0\*0) + (255\*0) + 0

= 127.5 + 0 + 0 + 0 + 0

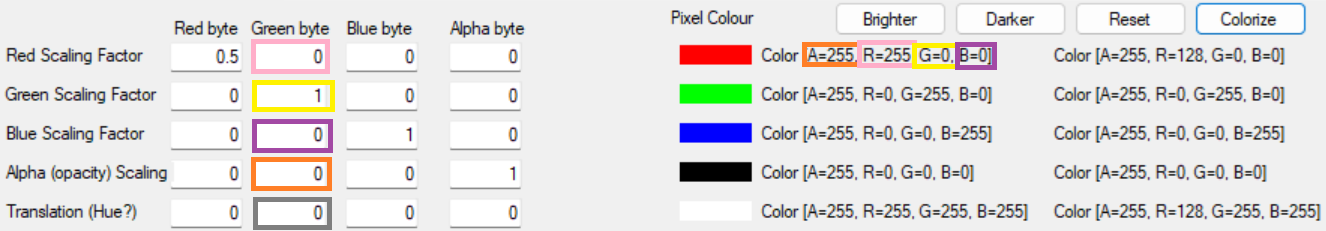
= 127.5

Round up to get hex value ∴ = 12816

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 0 |  |
| **Blue Byte** | 0 |  |
| **Alpha Byte** | 255 |  |
| **Translation** | 0 |  |

### Scaling the Green byte

We now move to the Green byte of the first pixel.



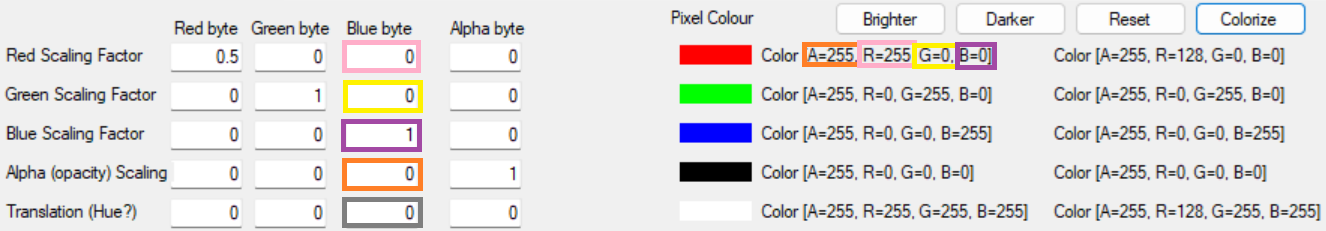
Scaled green byte value = (255\*0) +(0\*1) + (0\*0) + (255\*0) + 0

= 0 + 0 + 0 + 0 + 0

= 016

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 0 | 0 |
| **Blue Byte** | 0 |  |
| **Alpha Byte** | 255 |  |
| **Translation** | 0 |  |

### Scaling the Blue byte



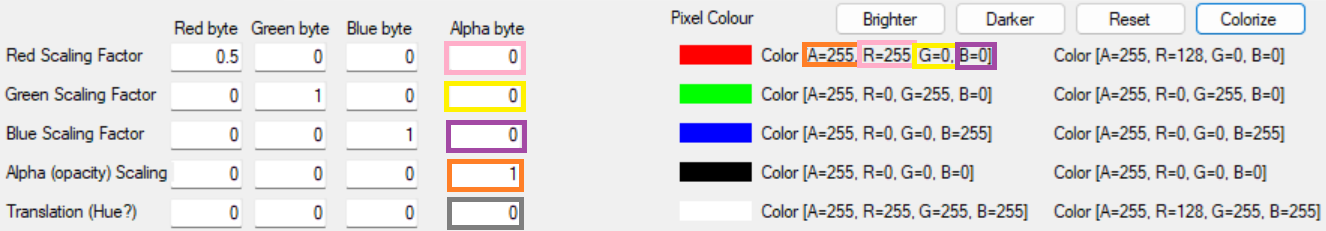
Scaled blue byte value = (255\*0) +(0\*0) + (0\*1) + (255\*0) + 0

= 0 + 0 + 0 + 0 + 0

= 016

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 0 | 0 |
| **Blue Byte** | 0 | 0 |
| **Alpha Byte** | 255 |  |
| **Translation** | 0 |  |

### Scaling the Alpha byte



Scaled Alpha byte value = (255\*0) +(0\*0) + (0\*1) + (255\*1) + 0

= 0 + 0 + 0 + 255 + 0

= 25516

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 0 | 0 |
| **Blue Byte** | 0 | 0 |
| **Alpha Byte** | 255 | 255 |
| **Translation** | 0 |  |

### Translation (image colour cast)

For this part we will ignore this and look at it in the later as changing its value adds a layer of complication which we will deal with once we are happy, we can work out how to scale pixel ARGB values.

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 0 | 0 |
| **Blue Byte** | 0 | 0 |
| **Alpha Byte** | 255 | 255 |
| **Translation** | 0 | 0 |

### Scaled Value for first pixel

As we can see the scaled value of our red pixel in ARGB is now

Color[A=255, R=128, G=0, B=0]

This means the pixel changes it colour to the new scaled value:

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Pixel 1** |  |  |

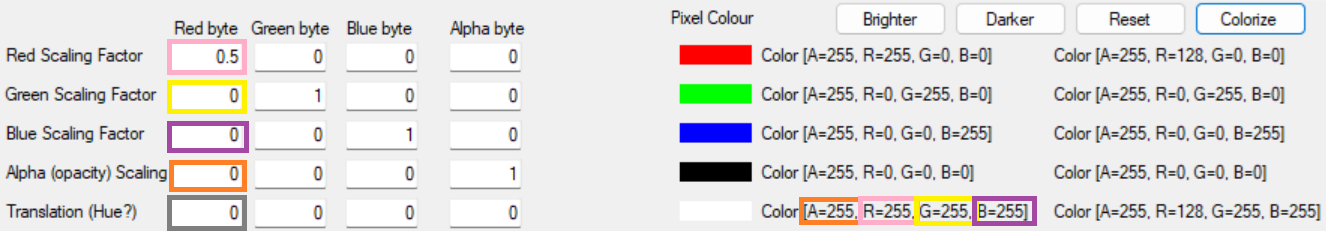
## Using second, third and fourth pixel of test image

As we can see above if the scaling factor is 1 then the byte remains unchanged. If the scaling factor is 0 then the byte will be zero.

Therefore, looking at the 2nd, 3rd and 4th pixel we can see that they will remain unchanged. However, we can see that the 5th pixel will change as we see next.

## Using fifth pixel of test image

### Scaling the Red byte



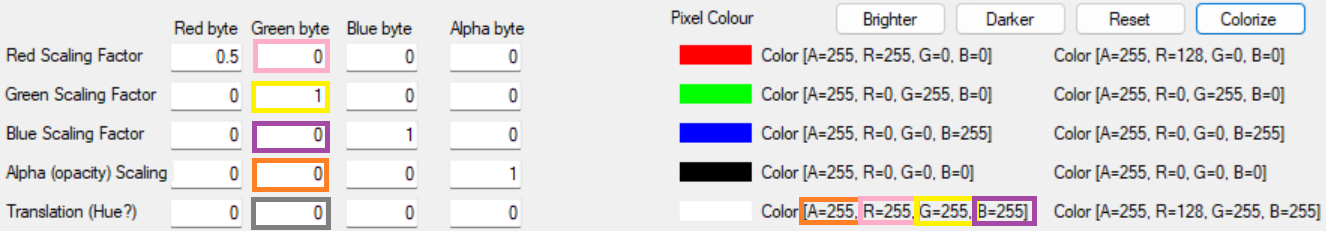
Scaled Red byte value = (255\*0.5) +(255\*0) + (255\*0) + (255\*0) + 0

= 128 + 0 + 0 + 0 + 0

= 12816

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 255 |  |
| **Blue Byte** | 255 |  |
| **Alpha Byte** | 255 |  |
| **Translation** | 0 |  |

### Scaling the Green byte



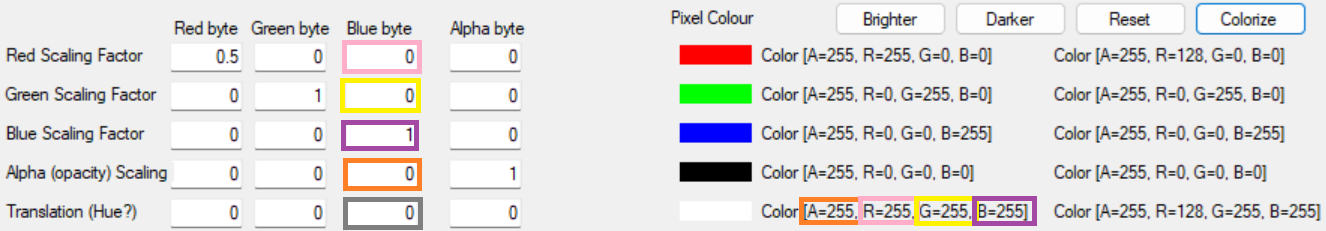
Scaled Green byte value = (255\*0) +(255\*1) + (255\*0) + (255\*0) + 0

= 0 + 255 + 0 + 0 + 0

= 25516

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 255 | 255 |
| **Blue Byte** | 255 |  |
| **Alpha Byte** | 255 |  |
| **Translation** | 0 |  |

### Scaling the Blue byte



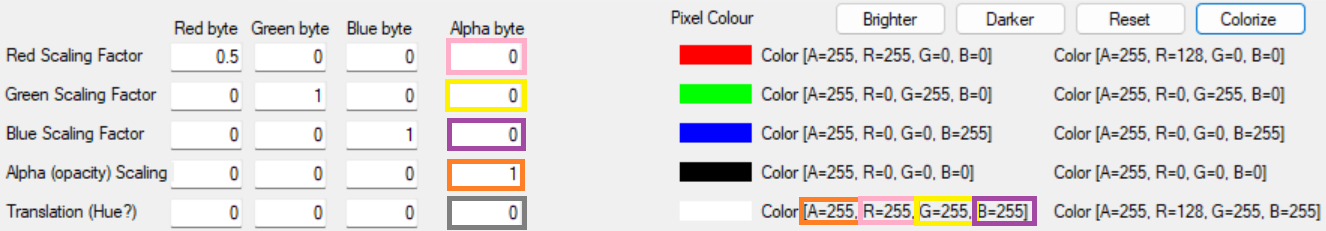
Scaled Blue byte value = (255\*0) +(255\*0) + (255\*1) + (255\*0) + 0

= 0 + 0 + 255 + 0 + 0

= 25516

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 255 | 255 |
| **Blue Byte** | 255 | 255 |
| **Alpha Byte** | 255 |  |
| **Translation** | 0 |  |

### Scaling the Alpha byte



Scaled Alpha byte value = (255\*0) +(255\*0) + (255\*0) + (255\*1) + 0

= 0 + 0 + 0 + 255 + 0

= 25516

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 255 | 255 |
| **Blue Byte** | 255 | 255 |
| **Alpha Byte** | 255 | 255 |
| **Translation** | 0 |  |

### Translation (image colour cast)

For this part we will ignore this and look at it in the later as changing its value adds a layer of complication which we will deal with once we are happy, we can work out how to scale pixel ARGB values.

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Red Byte** | 255 | 128 |
| **Green Byte** | 255 | 255 |
| **Blue Byte** | 255 | 255 |
| **Alpha Byte** | 255 | 255 |
| **Translation** | 0 | 0 |

### Scaled Value for fifth pixel

As we can see the scaled value of our white pixel in ARGB is now

Color[A=255, R=128, G=255, B=255]

This means the pixel changes its colour to the new scaled value (ignore the grey border as it is only added to help you pick out the white pixel on a white background):

|  |  |  |
| --- | --- | --- |
|  | **Original Value** | **Scaled Value** |
| **Pixel 1** |  |  |

### Check values are correct

In the ChangeColour.exe app we create the new image, save it and open it in a hex editor so we can check that the values we have worked out are indeed what appear in the scaled image.

The image is 5 pixels wide by 1 pixel high and the 5 pixels are individually outlined with the colour they are in the image except the 5th one which is the white pixel and here it is outlined in grey to make it obvious. White on white is difficult to see.

|  |  |
| --- | --- |
| **Original** | **Scaled** |
|  |  |

All the above are in Hex bytes (base 16). We did our worked example calculations as decimal (base 10). Here is a small conversion table that will help us understand if we calculated this correctly.

* 0016 = 010
* 8016 = 12810
* FF16 = 25510

As the hex file has the pixel data as BGRA and we calculated as ARGB we need to manipulate our output to be BGRA.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Original** | **Worked example** | **Saved output** |
| Pixel 1 = Red | 00 00 FF FF | 0 0 128 255 | 00 00 80 FF |
| Pixel 2 = Green | 00 FF 00 FF | 0 255 0 255 | 00 FF 00 FF |
| Pixel 3 = Blue | FF 00 00 FF | 255 0 0 255 | FF 00 00 FF |
| Pixel 4 = Black | 00 00 00 FF | 0 0 0 255 | 00 00 00 FF |
| Pixel 5 = White | FF FF FF FF | 255 255 128 255 | FF FF 80 FF |

We can see we calculated it exactly as the ChangeColur.exe app has done. Taking this and drawing it into the bitmap will give the following output.

|  |  |
| --- | --- |
| **Original** | **Scaled** |
|  |  |

## Alpha Byte

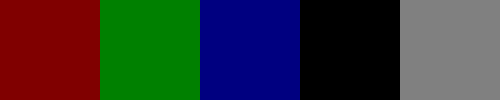
The Alpha byte is responsible for the opacity of the image. This is where it gets a bit tricky to understand the result as you can apply opacity in three ways, in the:

* Yellow row below where you are applying it to the byte of the RGB.
* Blue column you are applying it to the RGB scaling.
* Green box you apply it to the whole image.



If you apply it to the row and the column then you start to get combinations of opacity, lets use some worked examples to help explain this.

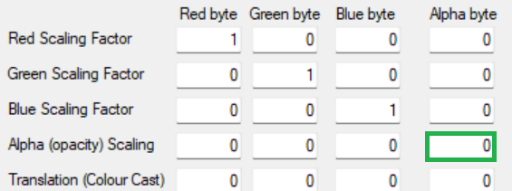
We load in our test image with the 5 pixels that are each a 50% solid colour of Red, Green, Blue, All the colours (White). As No Colour (Black) has no colour it remains as is. We do this so you can see the effect of the opaque feature. E.g. if red was at 100% (25510 or FF16) and we scale it to 110% it will remain at 100% and you would not see what action our changes are having.



Note: 50% of 255 is 127.5 at times you will have seen this is rounded up to 128 and at other times it is rounded down 127. This is done by the “System.Drawing.Imaging.ImageAttributes” and we do not seem to have any control over this. While this may seem to be an issue the human eye will probably not be able to tell the difference between the colour represented by 127 or 128.

### Alpha Scaling the pixels Alpha component

This is by changing the percentage in the text box as outlined in green below

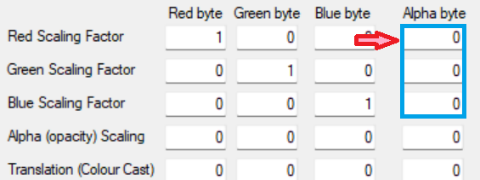


|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| 1 = 100% |  |  |  |
| 0.5 = 50% |  |  |  |
| 0 = 0% |  |  |  |

As we can see the opacity the amount of light we see from the image will change but the value of the actual RGB will not change.

### Single Colour scaling the Alpha component

We do this by changing the values in the blue column below in the red scaling row pointed to by the arrow. It does not matter which scaling colour we choose as we choose the result will follow for any.



|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| Red  Alpha = 2 = 200% |  |  |  |
| Red  Alpha = 1.5 = 150% |  |  |  |
| Red  Alpha = 1 = 100% |  |  |  |
| Red  Alpha = 0.5 = 50% |  |  |  |
| Red  Alpha = 0 = 0% |  |  |  |

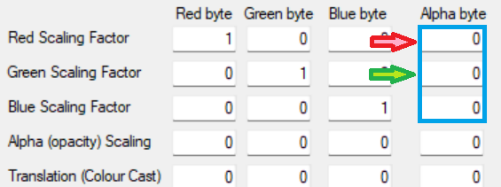
We can see that this scaling has an effect on all pixels that have a red component greater than 0 in them.

The item to note here is that the colours are set at 50% which is 127 or 128 depending on whether it is rounded up or down. This is with 100% opacity. This means that we can increase opacity by 200% before the Alpha byte tops out at 255 again it could be 254 or 255 depending on rounding. We can deduce from this that the Alpha scaling is connected to the actual value of the original colour byte. If the colour byte is at 50% then even if we set Alpha for the scaling to 100% it will be 100% of 50% and will be 127. In the above example anything above 200% has topped out a at 255 and gets ignored.

All individual colours will react the same to what we see here so next, we will investigate scaling Alpha on more than one colour.

### Two Colour scaling the Alpha component

We do this by changing the values in the blue column below in the red and green scaling row as pointed to by the arrows. It does not matter which scaling colour combination we choose the result will follow for any.

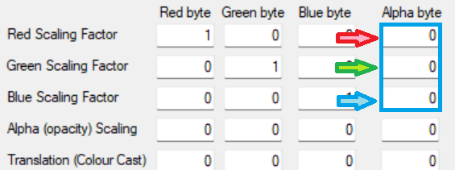


|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| Red and Green Alpha = 2 = 200% |  |  |  |
| Red and Green Alpha = 1.5 = 150% |  |  |  |
| Red and Green Alpha = 1 = 100% |  |  |  |
| Red and Green Alpha = 0.5 = 50% |  |  |  |
| Red and Green Alpha = 0 = 0% |  |  |  |

We can see that this scaling has an effect on all pixels that have a red and/or green component greater than 0 in them e.g. the white pixel. The Blue and Black pixel do not have a green or red component so they remain invisible.

### Full Colour scaling the Alpha component

We do this by changing the values in the blue column below in the red, green and blue scaling row. The results we get may not be what we are expecting from the work already done.

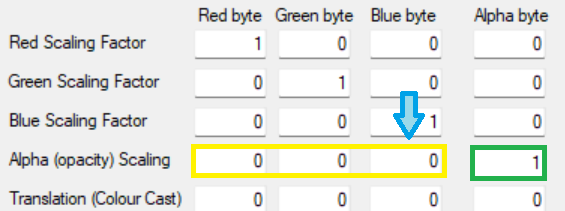


|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| Red, Green, Blue  Alpha = 2 = 200% |  |  |  |
| Red, Green, Blue  Alpha = 1.5 = 150% |  |  |  |
| Red, Green, Blue  Alpha = 1 = 100% |  |  |  |
| Red, Green, Blue  Alpha = 0.5 = 50% |  |  |  |
| Red, Green, Blue  Alpha = 0 = 0% |  |  |  |

As black has not Red, Green or Blue component it will always be invisible.

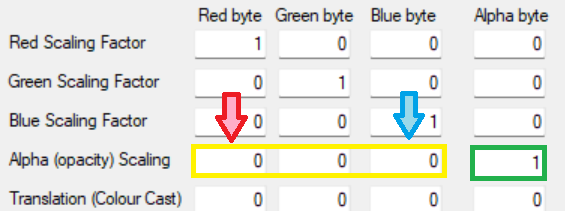
### Alpha Scaling of a single colour component

To allow us now to see any changes when we adjust the scaling percentages in the yellow row, we will need to set the value in the Green box greater than 0. For this part of the explanation, we will choose 100% and enter 1 for this.



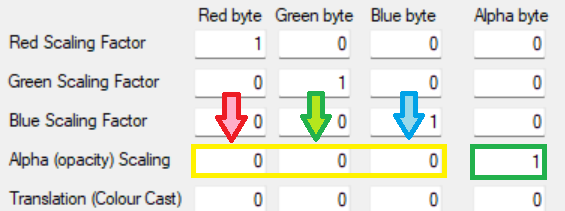
|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| Red, Green, Blue  Alpha = 2 = 200% |  |  |  |
| Red, Green, Blue  Alpha = 1.5 = 150% |  |  |  |
| Red, Green, Blue  Alpha = 1 = 100% |  |  |  |
| Red, Green, Blue  Alpha = 0.5 = 50% |  |  |  |
| Red, Green, Blue  Alpha = 0 = 0% |  |  |  |

### Alpha Scaling of two-colour components



|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| Red, Green, Blue  Alpha = 2 = 200% |  |  |  |
| Red, Green, Blue  Alpha = 1.5 = 150% |  |  |  |
| Red, Green, Blue  Alpha = 1 = 100% |  |  |  |
| Red, Green, Blue  Alpha = 0.5 = 50% |  |  |  |
| Red, Green, Blue  Alpha = 0 = 0% |  |  |  |

### Alpha Scaling of all colour component



|  |  |  |  |
| --- | --- | --- | --- |
| **Setting** | **Original** |  | **Modified** |
| Red, Green, Blue  Alpha = 2 = 200% |  |  |  |
| Red, Green, Blue  Alpha = 1.5 = 150% |  |  |  |
| Red, Green, Blue  Alpha = 1 = 100% |  |  |  |
| Red, Green, Blue  Alpha = 0.5 = 50% |  |  |  |
| Red, Green, Blue  Alpha = 0 = 0% |  |  |  |

## Adding Translation (image colour cast) into the mix

This is simply the colour cast that we will ADD to ever pixel over the whole image even if the colour is not present in the pixel we are working with.

This is a percentage of the full value of the colour (25510 or FF16) and NOT the value of that colour byte of the pixel we are working with.

### Example 1

We are working with a pixel that has:

Color[A=255, R=128, G=0, B=0]

Now we want to ADD a Green cast to pixel of 20%. This is 20% of full green which has a byte value of 25510 or FF16. The modified Pixel will now be:

Color[A=255, R=128, G=51, B=0]

### Example 2

We are working with a pixel that has:

Color[A=255, R=128, G=0, B=0]

We want to ADD a Red cast to the image of 20%. This is 20% of full Red byte value of 25510 or FF16. 20% of full value is5110 or 3316.

Note: This 20% was not of the existing red value of 12810 but will be added to that value (128+51 = 179)

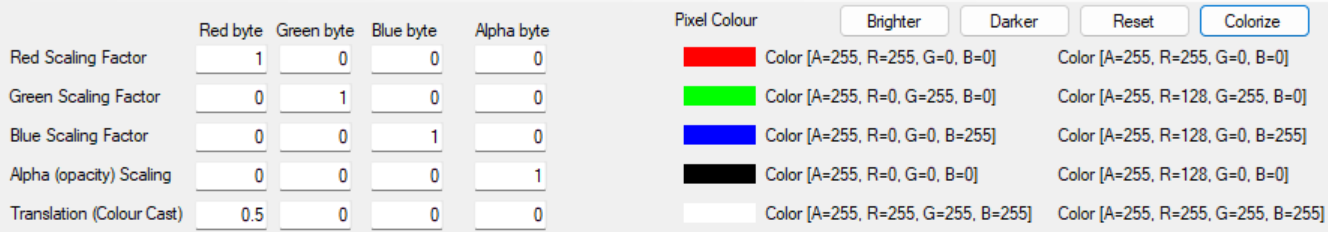
The modified Pixel will now be:

Color[A=255, R=179, G=51, B=0]

### Worked Example

We take or test image and we set the Image to have a red cast of 50%. This will add 128 to every pixels red component. If the red components value goes above 255 it is capped at 255.

Under Red byte translation enter 0.5 for 50%. This will be a value of 128 (50% of 255) and this will be added to the existing red component. Each byte will have a ceiling of 255 and any amounts over that get ignored.



We can now see the effect that this cast will have on the image. (ignore the grey boarder as this is just added to allow you to pick out the image that is 1 pixel high by 5 pixels wide)

|  |  |
| --- | --- |
| **Original** | **Modified with 50% Red cast** |
|  |  |

We can apply this to a simple image and see the different effects:

|  |  |  |
| --- | --- | --- |
| **Cast Value** | **Original** | **Modified** |
| 50% Red |  |  |
| 20% Green |  |  |
| 60% Blue |  |  |
| 30% Red  75% Green |  |  |
| 30% Red  25% Green  60% Blue |  |  |
| 90% Red  90% Green  90% Blue |  |  |

## Combining Colour scaling, Colour cast and Alpha (opacity)

Now we combine everything and we can see how we can change the colours of an image. There are almost infinite ways of doing this and here we see just a few examples. You can experiment to get the look you want from your image.

### Swapping Red and Green

We will swap just the full Red with the full Green. You see in the modified image and the Pixel data that the red and green have swapped all else remains the same.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

### Swapping Red and Blue

We will swap just the full Red with the full Blue. You see in the modified image and the Pixel data that the red and blue have swapped all else remains the same.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

### Swapping Green and Blue

We will swap just the full Green with the full Blue. You see in the modified image and the Pixel data that the green and blue have swapped all else remains the same.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

### Sepia

Sepia is a reddish-brown colour (RGB = 112,66,20).



It is named after the rich brown pigment derived from the ink sac of the common cuttlefish Sepia. In an image this is combined with the colour pixels to give a brownish tinge to the image often used to make the image look old.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

### Polaroid Colour

Polaroid photos often had a warm palette with colours that can shift towards pink in highlights and green in shadows.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

### Monochrome

A monochrome photo is an image which has had all colour removed, can also be called Grey Scale. In photography these photos are called black and white photos. Black and White images are a different type of image as we will see later.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

### White to Alpha

In this we change the White colour to be transparent and the only pixel we will see is the one with no colour.

|  |  |  |  |
| --- | --- | --- | --- |
| **Settings** | **Original** | **Pixel Data: Original - Modified** | **Modified** |
|  |  |  |  |

# The Code behind it