Section 2 – Framework

Contents

[2.1.1 Project Decomposition 4](#_Toc17218116)

[2.1.1.1 Decomposition diagram 4](#_Toc17218117)

[2.1.1.2 Diagram Justification 5](#_Toc17218118)

[2.1.2 Usability 5](#_Toc17218119)

[2.1.2.1 UI Design 5](#_Toc17218120)

[2.1.2.2 UI Design Features 6](#_Toc17218121)

[Canvas 6](#_Toc17218122)

[Scroll Bars 6](#_Toc17218123)

[Zoom Bar 6](#_Toc17218124)

[2.1.3 Inputs, Processing, Outputs and Storage 6](#_Toc17218125)

[2.1.4 Class Design 7](#_Toc17218126)

[2.1.4.1 Image 7](#_Toc17218127)

[Properties 7](#_Toc17218128)

[Methods 7](#_Toc17218129)

[Constructor 8](#_Toc17218130)

[2.1.4.2 ZoomSettings 9](#_Toc17218131)

[Properties 9](#_Toc17218132)

[Constructor 9](#_Toc17218133)

[2.1.4.3 WorkSpace 10](#_Toc17218134)

[Properties 10](#_Toc17218135)

[Methods 10](#_Toc17218136)

[Constructor 11](#_Toc17218137)

[2.1.4.2 Axis 12](#_Toc17218138)

[2.1.5 Algorithm Design 13](#_Toc17218139)

[Algorithm 2.1 Creating Image Class 13](#_Toc17218140)

[Pseudocode 13](#_Toc17218141)

[Algorithm 2.2 Populating Image Class 14](#_Toc17218142)

[Pseudocode 14](#_Toc17218143)

[Algorithm 2.3 Resizing Picture Box 14](#_Toc17218144)

[Pseudocode 14](#_Toc17218145)

[Algorithm 2.4 Displaying Pixels 15](#_Toc17218146)

[Flowchart 15](#_Toc17218147)

[Pseudocode 16](#_Toc17218148)

[Algorithm 2.5 Setting Pixels at Runtime 18](#_Toc17218149)

[Pseudocode 18](#_Toc17218150)

[Algorithm 2.6 Determining PictureBox Location 19](#_Toc17218151)

[Diagram 19](#_Toc17218152)

[Pseudocode 19](#_Toc17218153)

[Algorithm 2.7 Stopping window becoming too small 20](#_Toc17218154)

[Pseudocode 20](#_Toc17218155)

[Algorithm 2.8 Deciding which pixels to draw 20](#_Toc17218156)

[Why differentiate between pixel types? 23](#_Toc17218157)

[**Decision:** Which algorithm should be used for determining status of pixels? 23](#_Toc17218158)

[Algorithm 2.8A Determining borders of pixels 24](#_Toc17218159)

[Algorithm 2.8B Finding Green pixels 27](#_Toc17218160)

[Algorithm 2.8C Finding Yellow Pixels 28](#_Toc17218161)

[Algorithm 2.9 Draw Zoomed Part 32](#_Toc17218162)

[Algorithm 2.9A Converting between File Pixels & Display Pixels 32](#_Toc17218163)

[Algorithm 2.9B Drawing Green Pixels 39](#_Toc17218164)

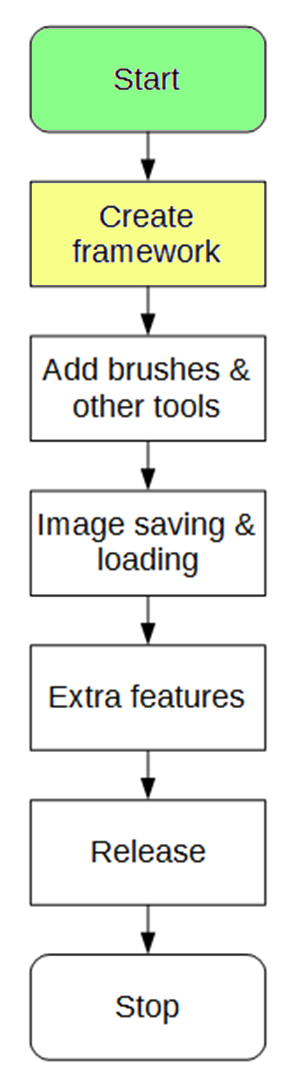
[Algorithm 2.9C Drawing Yellow Pixels 40](#_Toc17218165)

[Algorithm 2.10 Determining Bar Size 42](#_Toc17218166)

[Algorithm 2.10A Determining whether bars are visible or not. 43](#_Toc17218167)

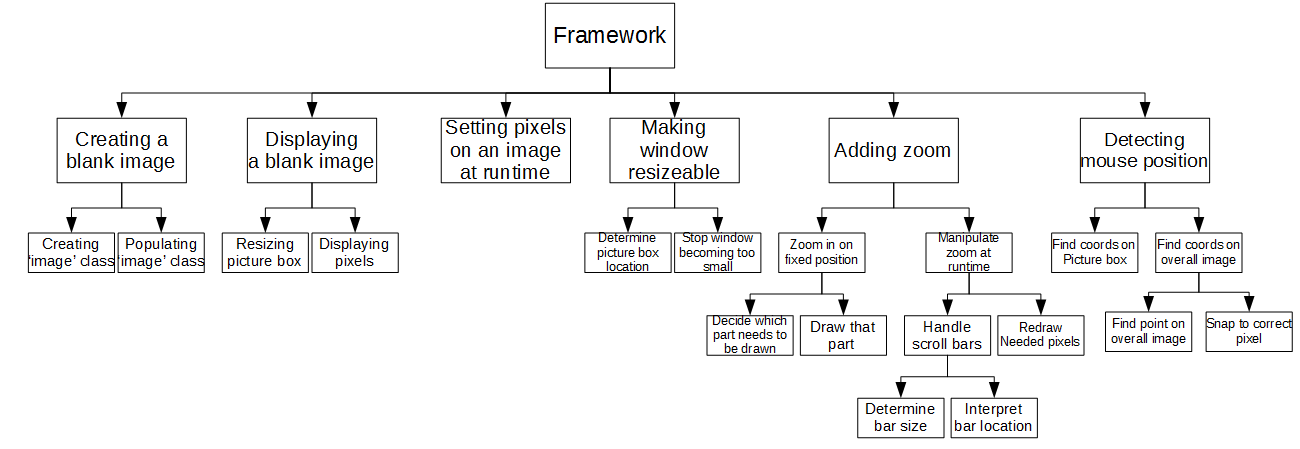
2.1 Design

The framework includes a lot of the internal design for the program. Classes, basic GUI and important functions will be designed here. After the framework is complete, the other functions should be able to be easily added on top.



# 2.1.1 Project Decomposition

## 2.1.1.1 Decomposition diagram



5

Each lowest level block has a label, which will be referred to later when each algorithm is designed.

For example algorithm 2.1 is algorithm 1 in phase 2 of the project.

15

14

13

12

11

10

9

8

7

6

4

3

2

1

## 2.1.1.2 Diagram Justification

This will detail why each part of the top level of the diagram is needed for this stage in the development

|  |  |  |
| --- | --- | --- |
| **Title** | **Justification** | **Fulfils** |
| Creating a blank image | The very first thing that will need to be coded is defining an empty image, which will involve creating a base ‘Image’ class, and populating the image class with default values, with a flexible constructor. | C1 |
| Displaying a blank image | After the image has been defined, it will need to be displayed in a window. Only a white box will be displayed at this point, and it will not need to move yet. | B1 |
| Setting pixels on an image at runtime | After the image is successfully displayed, with the use of a temporary button, pixels of differing colours will need to set at runtime, showing that the image can change. | A2 |
| Making window resizable | Currently all tests are being performed on a static window of fixed size, now the program will need to appropriately place the picture box in the window. The size will not be changed at this point (as zoom is not added yet) but the picture box should be appropriately placed. The window should also be prevented from becoming smaller than the image. | B1 |
| Adding zoom | Finally in this phase, zooming of the image will need to be added. As this is a very important part of the editor, it *must* be added at this early stage, as it would be difficult to add later. Zooming contains the decision of which pixels to draw, and at what size. | B9 |
| Detecting mouse position | Almost all brushes and features added later will in some way involve the mouse being clicked on the image. This makes it imperative that detecting mouse clicks is added at this early stage, no brushes can be added until this is complete. | A2 |

# 2.1.2 Usability

At the end of this design phase, the UI of the program will be very basic, as none of the tools will be added. The stakeholder input is not needed, as this will not reflect the final UI of the project.

Canvas

## 2.1.2.1 UI Design

Scroll bars

Zoom bar

## 2.1.2.2 UI Design Features

### Canvas

The main canvas will need to be visible to the user. The canvas will display the image in its current state, depending on the zoom and positioning of the image at that time. This will take up the main proportion of the window, as it is what the user will interact with most.

### Scroll Bars

The scroll bars have been added to allow the user to move around through their image, and there will be two for X and Y movement. The size of the bars will depend on the window size, and current zoom.

Scroll bars are suitable here due to allowing movement through a fixed range

### Zoom Bar

The zoom bar will allow the user to change his current zoom of the image, and will start at 100% (as Criteria A1 dictates that zooming out is not needed) and will go to a maximum value.

# 2.1.3 Inputs, Processing, Outputs and Storage

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Processing** | **Output** | **Storage** |
| Mouse click at position on canvas | Detect where the mouse was clicked on the canvas | A pixel of colour is set at that position on the canvas | Image data stored in RAM, the current zoom and viewing location |
| Change of the value of the zoom bar | Calculate new pixel size, new image position and redraw the desired pixels | The image at a different level of zoom | Image data stored in RAM, the previous zoom and viewing location |
| Change of the value of the scroll bars | Calculate which portion of the image must now be displayed | A different portion of the image | Image data stored in RAM, the current zoom and previous viewing location |

The table is very small at this point in time due to the limited functionality of the program. At later points in development there will be more potential inputs.

# 2.1.4 Class Design

## 2.1.4.1 Image

<class>

Image

colours[,]

width

height

zoomSetting

GetPixel()

SetPixel()

GetDisplayImage()

The image class will store the main properties of the image.

### Properties

|  |  |  |
| --- | --- | --- |
| **Property** | **Datatype** | **Justification** |
| colours[,] | 2D array of type ‘Color’ | An array of colours is what will be needed to store the property of each pixel on the grid. The array will make use of the existing ‘Color’ class in C#, *so that I do not reinvent the wheel.* |
| width | Integer | Stores the width of the current image. |
| height | Integer | Stores the height of the current image. |
| zoomSetting | [ZoomSettings](#_2.1.4.2_ZoomSettings) | Stores the current settings describing the zoom of the image. |

### Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Params** | **Return type** | **Justification** |
| GetPixel() | X, int, X coord of pixel to get  Y, int, Y coord of pixel to get | Colour | Will get an existing colour from a specific point in the image, regardless of its size. |
| SetPixel() | X, int, X coord of pixel to get  Y, int, Y coord of pixel to get  Colour, Color, colour of pixel to get | None | Will set a new colour onto the grid, similar to GetPixel. |
| GetDisplay-Image() | None | Image (C#) | Will return the C# base ‘Image’ class, for use in the displaying picture box. |

### Constructor

class image {

Color[,] pixels

integer width

integer height

ZoomSetting zoomSettings

constructor(\_width, \_height) {

width = \_width

height = \_height

pixels = new array of Color(width,height)

// *gives every pixel a default white colour*

foreach Color in Pixels {

Color = White

}

zoomSettings = new ZoomSetting()

}

}

## 2.1.4.2 ZoomSettings

<class>

ZoomSettings

centreLocation

zoomAmount

ZoomSettings contains all the properties about the current zoom on the image. This is separate from the image class to enforce proper **encapsulation**, as the level of zoom is not directly tied to the image.

### Properties

|  |  |  |
| --- | --- | --- |
| **Property** | **Datatype** | **Justification** |
| centre-Location | Point | Stores the location of the pixel in the middle of the zoom. The middle location is stored to make the zoom feel more natural when zooming in and out (the middle pixel remains the same) |
| zoom-Amount | Integer | Stores the amount that the image is currently zoomed in by. 1 = 1x = 100% zoom, 2 = 200% zoom etc. |

### Constructor

class ZoomSettings {

Point centreLocation

Integer zoomAmount

constructor(\_centreLocation) {

centreLocation = \_centreLocation

The default zoom will be none - 1x.

zoomAmount = 1

}

}

## 2.1.4.3 WorkSpace

WorkSpace is a class that encapsulates everything about the current working area. It stores the width and height of the window it needs to display to, holds an image to display, and calculates the display properties for the scroll bar, which it will send to the [ZoomSettings](#_2.1.4.2_ZoomSettings) class

<class>

Workspace

image

width

height

form

displayBox

IsScrollBarVisible()

GetScrollBarMax()

GetScrollBarValue()

SetScrollBarValue()

### Properties

|  |  |  |
| --- | --- | --- |
| **Property** | **Datatype** | **Justification** |
| image | [Image](#_2.1.4.1_Image) | Stores the image that is currently being edited. |
| width | Integer | Stores the width of the current working area (not the width of the image). |
| height | Integer | Stores the height of the current working area. |
| form | Form | Links the workspace to a Windows Form, to allow for easy interraction |
| displayBox | PictureBox | Links the workspace to the Picture Box that it will use for displaying |

### Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Params** | **Return type** | **Justification** |
| IsScrollBar-Visible | axis, [Axis](#_2.1.4.2_Scrollbar), the X or Y bar. | Boolean | This determines whether the selected scroll bar needs to be visible, if the workspace is larger than the image then no scroll bars are needed. |
| GetScroll-BarMax() | axis, [Axis](#_2.1.4.2_Scrollbar), the X or Y bar. | Integer | Calculates what the maximum for the selected scroll bar needs to be set to. |
| GetScroll-BarValue() | axis, [Axis](#_2.1.4.2_Scrollbar), the X or Y bar. | Integer | Calculates what the current value for the selected scroll bar needs to be set to. |
| SetScroll-BarValue() | axis, [Axis](#_2.1.4.2_Scrollbar), the X or Y bar.  value, Integer, the value to set the scroll bar to. | None | Sets the value of the scroll on the selected X or Y axis. This employs **abstraction** as the lower classes have hidden the specifics of zooming, and all that is needed to change the zoom location is progress through a bar. |

### Constructor

class WorkSpace {

Image image

Integer width

Integer height

Form form

PictureBox displayBox

constructor(\_image, \_form) {

image = \_image

The ‘Workspace’ object contains the associated PictureBox, which it will add to the form.

This means that the base form inputted will need to be blank

form = \_form

width = width of form

height = height of form

displayBox = new PictureBox

add displayBox to form

}

}

## 2.1.4.2 Axis

The scrollbar enum is a small set of constants for deciding whether to manipulate the X axis or the Y axis. This enum has no purpose by itself but makes the code more **readable** and **maintainable** by making it clear which axis is being manipulated.

<enum>

Axis

X

Y

|  |  |
| --- | --- |
| **Member** | **Justification** |
| X | Denotes that the X axis has been selected. |
| Y | Denotes that the Y axis has been selected. |

# 2.1.5 Algorithm Design

## Algorithm 2.1 Creating Image Class

As there is no program ‘flow’ in defining a class, there is no flowchart

### Pseudocode

class image {

Color[,] pixels

integer width

integer height

ZoomSetting zoomSettings

}

## Algorithm 2.2 Populating Image Class

To now populate the image class, a constructor has been added to fill in some values.

### Pseudocode

class image {

Color[,] pixels

integer width

integer height

ZoomSetting zoomSettings

constructor(\_width, \_height) {

width = \_width

height = \_height

pixels = new array of Color(width,height)

// *gives every pixel a default white colour*

foreach Color in Pixels {

Color = White

}

zoomSettings = new ZoomSetting(middle of image)

}

}

## Algorithm 2.3 Resizing Picture Box

This ellipsis denotes previously designed code, showing that this code is part of the existing ‘Image’ class

As this algorithm is very linear, no flowchart diagram is needed

### Pseudocode

class Image {

...

ResizePictureBox(box) {

*//* *uses the width and height properties to set properties of the //* *picture box*

box.width = width

box.height = height

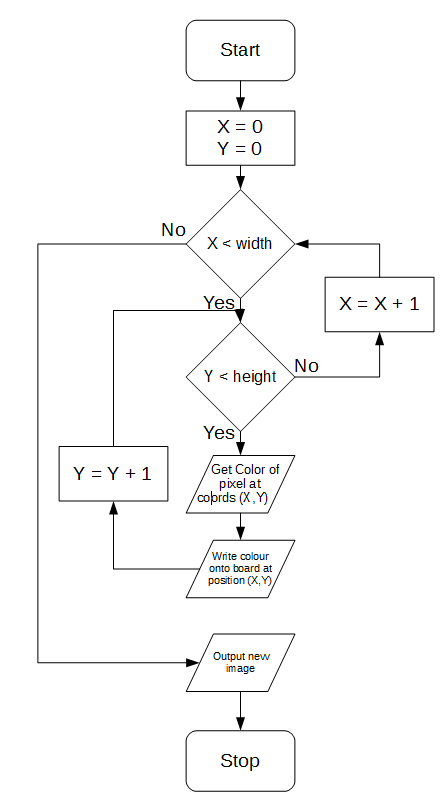
*// this ensures the picture box can hold the pixels*

}

}

## Algorithm 2.4 Displaying Pixels

### Flowchart



### Pseudocode

For the pseudocode of this algorithm, I have two options for outputting

DisplayToPictureBox(box) {

*// creates a ‘Graphics’ object for drawing directly onto the picture box*

Graphics GFX = box.CreateGrapics()

for x = 1 to width {

for y = 1 to height {

colour = pixels[x,y]

GFX.SetPixel(x,y,colour)

}

}

}

GetDisplayImage() {

*// creates a C# ‘image’ object to store the output image*

Drawing.Image image = new Drawing.Image(width,height)

for x = 1 to width {

for y = 1 to height {

colour = pixels[x,y]

Image.SetPixel(x,y,colour)

}

}

}

I have decided to use algorithm B for my program, as using an algorithm that outputs a standard object that contains all the necessary info, and is compatible with most workspace objects, is much more desirable than writing directly onto the picture box.

This also means the image data is easier to store and transmit, as it will be outputted fully contained in an image object.

Algorithms 2.3 and 2.4 will be combined into a single combined function, which encompasses the entirety of displaying the image:

class Workspace {

Image displaying will be part of the ‘Workspace’ class, as it encapsulates both generating the image and placing it in the correct box

...

public Display {

ResizePictureBox(displayBox)

displayBox.DisplayImage = GetDisplayImage()

}

The only public facing function is ‘Display’. The end user will not need to see any other of the functions.

private [ResizePictureBox](#_Algorithm_2.3_Resizing)(box) {

*//* *uses the width and height properties to set properties of the //* *picture box*

box.width = width

box.height = height

*// this ensures the picture box can hold the pixels*

}

private [GetDisplayImage](#_Algorithm_2.4_Displaying)() {

// creates a C# ‘image’ object to store the output image

Drawing.Image image = new Drawing.Image(width,height)

for x = 1 to width {

for y = 1 to height {

colour = pixels[x,y]

Image.SetPixel(x,y,colour)

}

}

}

## Algorithm 2.5 Setting Pixels at Runtime

### Pseudocode

class Image {

...

SetPixel(x,y,colour) {

pixels[x,y] = colour

[Display()](#Display)

}

}

All that is needed for this function is to call the existing Display() function, which will handle all of the image displaying.

## Algorithm 2.6 Determining PictureBox Location

Form

### Diagram

imageHeight

formWidth

formHeight

imageWidth

Canvas

The purple ❌ denotes what the location of the picture box must be. Looking at the diagram, it becomes clear that the X position of the ❌ must be:

½formWidth  
 ½imageWidth

So this to find the X, the formula is ½formWidth - ½imageWidth or ½(formWidth – imageWidth)

### Pseudocode

class WorkSpace {

...

RelocatePictureBox {

Integer X = (width – image.width) / 2

Integer Y = (height – image.height) / 2

displayBox.Location = new Location(X,Y)

}

}

## Algorithm 2.7 Stopping window becoming too small

### Pseudocode

form.minimumWidth = image.width

form.minimumHeight = image.height

This prevents the form from (at this point) ever being smaller than the image, as zooming is not yet implemented.

## Algorithm 2.8 Deciding which pixels to draw

Assuming there is an image of 20px by 20px:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

If the screen has a size of 10x10px, centred on the pixel at 5,5, then the zoomed area will be:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| formHeight |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | formWidth |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn

If the centre location is moved to 10,5, the zoomed area will now be:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | formHeight  formWidth |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn

However, if the zoom is doubled, then the resulting window will be:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  | formWidth | formHeight |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn, Partially Drawn

The user will, in this example, see this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

From these illustrations, two potential types of pixel emerge:

* Those that need to be drawn in full (Green)
* Those that need to only partially be drawn (Yellow)
* Those that do not need to be drawn (White)

Note that the bracketed terms will be use to refer to a type of pixel (For example a Green pixel is a pixel that needs to be drawn in full)

### Why differentiate between pixel types?

White pixels do not need to be drawn, so it is useful to be able to ignore them

Green pixels are much easier to draw than Yellow pixels (as they are all square), so drawing a Green pixel requires a less complicated algorithm. Using a less complicated algorithm on the majority of pixels should save time, and make the image render faster.

### **Decision:** Which algorithm should be used for determining status of pixels?

In this case, there are two potential sorts of algorithm to use when telling the renderer which pixels are Yellow and which are Green

#### Potential Algorithm A: Checking whether a given pixel is Green or Yellow

Would be given the location of a pixel and the current zoom settings, and would return whether that pixel is Green, Yellow or White. The renderer would iterate over every pixel, making a decision based on the return of this function on each pixel.

If White 🡪 Do nothing  
If Green 🡪 Draw a square  
If Yellow 🡪 Draw a rectangle

Advantages:

* Good for checking one individual pixel

Disadvantages

* In bulk, may perform the same calculation many times, not very efficient

#### Potential Algorithm B: Returning a list of all Yellow pixels, and a list of all Green pixels

Would, from the current zoom settings, return a list of every Yellow pixel, and a list of all Green pixels. A list of white pixels is not necessary (nothing needs to be done with white pixels, they are not visible)

Advantages:

* Means the calculations to determine where yellow pixels are should only be done **once**, significantly reducing the amount of calculations
* It is easy to iterate over each set of pixels with a different algorithm once they have been separated.

Disadvantages

* Takes a long time to calculate whether one individual pixel is yellow or green

#### Decision

In this case, I have opted for **algorithm B**, as the image may be very large, the less complex scalability of algorithm B will suit the program much more. It doesn’t matter that it is not easy to check an individual pixel as when rendering it is more important to draw as many pixels as possible.

### Algorithm 2.8A Determining borders of pixels

From our illustrations before, it becomes clear two types of border can be drawn:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn, Partially Drawn

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

The Red border denotes where the drawing must stop, the yellow border denotes where partially drawn pixels must stop.

By abstracting the rectangles into two points, this diagram can be drawn:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn, Partially Drawn

#### Red Crosses

As the window is 10px by 10px, and the image (at 2x zoom) is now 40px by 20px, with a centre at (what is now) 20,10, it becomes clear (by eye) that the top-left red cross must be at 15px, 5px, and the bottom-left cross at 25px,15px.

Thus the calculations become:

* The X is centreloc.X ± ½form.width.
  + The ± depends on whether we are looking for the left or right cross
  + In the above example this would be 20-½(10) = 20 – 5 = 15

* The Y is centreloc.Y ± ½form.height.
  + The ± depends on whether we are looking for the top or bottom cross
  + In the above example this would be 10-½(10) = 10 – 5 = 5

#### Orange Crosses

The orange crosses need to take into account the zoom.

By eye, the top-left red cross must be at 16px, 6px, and the bottom-left cross at 24px, 14px.

The calculations will need to require some sort of rounding in order to make sure they are aligned onto a pixel, and the top-left Orange cross is always further across than the Red cross, however the bottom-right Orange cross is always less far across than the bottom-right Red cross.

In order to do this, the coord can be divided by the zoom, rounded, and then multiplied back again.

* Top-Left cross
  + X is (topLeftOrangeArrow.X / zoom), rounded up, multiplied by zoom
  + X = (topLeftOrangeArrow.X / zoom).RoundUp \* zoom
  + Y is (topLeftOrangeArrow.Y / zoom), rounded up, multiplied by zoom
  + Y = (topLeftOrangeArrow.Y / zoom).RoundUp \* zoom
  + It is rounded up to ensure the point is always higher
* Bottom-Right cross
  + X is (bottomRightOrangeArrow.X / zoom), rounded down, multiplied by zoom
  + X = (bottomRightOrangeArrow.X / zoom).RoundDown \* zoom
  + Y is (bottomRightOrangeArrow.Y / zoom), rounded down, multiplied by zoom
  + Y = (bottomRightOrangeArrow.Y / zoom).RoundDown \* zoom
  + It is rounded down to ensure the point is always lower

### Algorithm 2.8B Finding Green pixels

**Note**

At this point in time, there are two sorts of ‘pixel’. There is a pixel on the image file, and a pixel on the users screen. This becomes a problem when zooming is applied, as 1 file pixel will translate to 4 displayed pixels (at 2x zoom).

To clear up confusion, there will be two terms used to refer to pixels:

* A ‘file pixel’ (fpx) refers to a pixel in the image file, stored in the original array of colours
* A ‘display pixel’ (dpx) refers to a pixel being displayed on the user’s screen. All pixels up to this point have been display pixels.

To switch between file pixels and display pixels, divide or multiply by zoom.

To find the location of the green pixels, all that is needed is to find the pixels captured in the orange rectangle.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

#### Pseudocode

class Workspace {

...

GetGreenPixels {

fileTopLeftPoint = orangeRectangle.topLeftCorner

fileBottomRightPoint = orangeRectangle.bottomRightCorner

for x = fileTopLeftPoint.X to fileBottomRightPoint.X

for y = fileTopLeftPoint.Y to fileBottomRightPoint.Y

[DrawGreenPixel(x,y)](#_Pseudocode)

next y

next x

### Algorithm 2.8C Finding Yellow Pixels

To find the location of the green pixels, it is more difficult, as their shape is non-rectangular. It seems that four loops will be needed (to iterate along each side).

To help do this, the yellow segment will be split into 4 smaller segments (around each side)

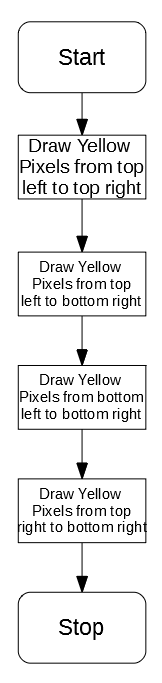
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Note that (while this is not possible to show in the diagram), each pixel in the corner will be drawn twice, as the four groups overlap in the corner, meaning there will be 4 extra calls of the DrawYellowPixel() function. However this is manageable as there will be significantly more green and yellow file pixels drawn.

For comparison, whilst drawing green file pixels has a **big O notation** of around , drawing these pixels has **time complexity** of , significantly shorter.

#### Program Flow

The order for drawing pixels is described in this flowchart:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Where the arrows denote the order in which yellow boxes are determined.

#### Pseudocode

GetYellowPixels {

fileTopLeftPoint = redRectangle.topLeftPoint

fileBottomRightPoint = redRectangle.bottomRightPoint

for X = fileTopLeftPoint.X TO fileBottomRightPoint.X – 1\*

DrawYellowPixel(X,fileTopLeftPoint.Y)

next X

for Y = fileTopLeftPoint.Y TO fileBottomRightPoint.Y – 1

DrawYellowPixel(fileTopLeftPoint.X,Y)

next X

for X = fileTopLeftPoint.X TO fileBottomRightPoint.X – 1

DrawYellowPixel(X,fileBottomRightPoint.Y - 1)

next X

for Y = fileTopLeftPoint.Y TO fileBottomRightPoint.Y – 1

DrawYellowPixel(fileBottomRightPoint.X - 1,Y)

next X

}

\* The reasoning for subtracting 1 from all the bottom points on the rectangle is that it is necessary to the be able to find the coords of its adjacent yellow display pixel.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

However, it is possible to **improve** the prior algorithm, by having the program draw both sets of parallel yellow pixels in one go, rather than looping twice, resulting in:

GetYellowPixels {

fileTopLeftPoint = redRectangle.topLeftPoint

fileBottomRightPoint = redRectangle.bottomRightPoint

for X = fileTopLeftPoint.X TO fileBottomRightPoint.X – 1\*

DrawYellowPixel(X,fileTopLeftPoint.Y)

DrawYellowPixel(X,fileBottomRightPoint.Y - 1)

next X

for Y = fileTopLeftPoint.Y TO fileBottomRightPoint.Y – 1

DrawYellowPixel(fileTopLeftPoint.X,Y)

DrawYellowPixel(fileBottomRightPoint.X - 1,Y)

next X

}

Which is significantly smaller.

## Algorithm 2.9 Draw Zoomed Part

To do this, two sorts of pixels will need to be drawn, Green and Yellow. Because of this, this algorithm will be split into two parts, drawing Green and Yellow.

### Algorithm 2.9A Converting between File Pixels & Display Pixels

While making these algorithms, two important functions will need to be designed. An algorithm to convert from a File Points (a location in the file), to a display pixel (a pixel on the screen).

To design these algorithms, the outputs of a form in several positions will be considered.

In this case, the zoom centre is at (10fpx, 5fpx), at 2x zoom.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Form |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Thus, the form’s view will be:

(Each square in this diagram is 2dpx by 2dpx)

(12fpx, 7fpx)

(9dpx, 9dpx)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | (8fpx, 3fpx)  (1dpx, 1dpx) |  |  |  |  |
|  |  |  |  |  |  |
|  |  | (10fpx, 5fpx)  (5dpx, 5dpx) |  |  |  |
|  |  |  |  |  |  |

The expected display locations (in dpx) have been labelled, as well as their file positions (in fpx).

In another example, the centre location is moved to file location (11fpx, 6fpx)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Each square in this diagram is 2dpx by 2dpx

(13fpx, 8fpx)

(9dpx, 9dpx)

(9fpx, 4fpx)

(1dpx, 1dpx)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | (11fpx, 6fpx)  (5dpx, 5dpx) |  |  |  |
|  |  |  |  |  |  |

The expected display locations (in dpx) have been labelled, as well as their file positions (in fpx).

From these diagrams some conclusions can be drawn:

* The centre location X must always stay in the middle of the form.
* Only points close to the centre location are drawn on the diagram
* If the zoom is increased, fewer points are drawn.

Considering this, algorithms can start to be planned:

#### File Pixels to Display Pixels

##### Flowchart

**Note, I can’t put a flowchart here right now, replace the bullet with some flow later**

* Calculate the file displacement from the centre (in terms of X and Y) by subtracting that point’s file coords from the coords of the centre location.
* Multiply the displacements by the current zoom
* Add these displacements to the display location of the centre coord.

For an example, considering the first diagram again:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Thus, the form’s view will be:

(Each square in this diagram is 2dpx by 2dpx)

(12fpx, 7fpx)

(9dpx, 9dpx)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | (8fpx, 3fpx)  (1dpx, 1dpx) |  |  |  |  |
|  |  |  |  |  |  |
|  |  | (10fpx, 5fpx)  (5dpx, 5dpx) |  |  |  |
|  |  |  |  |  |  |

Considering the top-left X.

File Displacement X = 8 – 10 = -2  
File Displacement Y = 3 – 5 = -2

Multiplied by Zoom Displacement X = 2 \* -2 = -4  
Multiplied by Zoom Displacement Y = 2 \* -2 = -4

Added to centre location X = 5 + -4 = 1  
Added to centre location Y = 5 + -4 = 1 = Display Location of (1dpx, 1dpx)

These results are consistent with the predicted display location.

##### Pseudocode

From this algorithm, the follow pseudocode can be produced:

FilePointToDisplayPoint(filePoint) {

displacementX = filePoint.X – centreFilePoint.X

displacementY = filePoint.Y – centreFilePoint.Y

displacementX = displacementX \* zoom

displacementY = displacementY \* zoom

newX = centreDisplayPoint.X + displacementX

newY = centreDisplayPoint.Y + displacementY

return new Point(newX, newY)

}

#### Display Pixels to File Pixels

In order to construct this algorithm, it becomes clear that the inverse of the [File Pixels to Display Pixels](#_File_Pixels_to_1) algorithm should be employed.

##### Flowchart

This is the same steps as the File Pixels to Display Pixels [flowchart](#_Flowchart), but in reverse order.

* Find the displacement between the display location and the centre coord (in terms of X and Y) by subtracting that point’s file coords from the coords of the centre location.
* Divide the displacements by the current zoom
* Add the displacements to the file location of the centre coord.

##### Pseudocode

The pseudocode follow a similar format as the previous pseudocode:

FilePointToDisplayPoint(displayPoint) {

displacementX = displayPoint.X - centreDisplayPoint.X

displacementY = displayPoint.Y - centreDisplayPoint.Y

This will use an integer division, so only whole numbers (rounded down) will be returned from this.

displacementX = displacementX / zoom

displacementY = displacementY / zoom

newX = displacementX + centreFilePoint.X

newY = displacementY + centreFilePoint.Y

return new Point(newX, newY)

}

#### Extra Note

These two functions now make a complete loop allowing conversions to and from file pixels and display pixels.

However there will be some information loss, as the [Display Pixel to File Pixel conversion code](#_Pseudocode_1) will round display pixels to the nearest pixel, this is because of this snippet of the pseudocode:

displacementX = displacementX / zoom

displacementY = displacementY / zoom

This occurs as **integer division** is used instead of floating point division. The decision to use integer division was made so that File Pixels followed the **simple rule** that File Pixels must always be whole numbers (as well as Display Pixels). This is done to:

* **Reduce confusion.** The concept of ‘halfway through’ a File Pixel isn’t natural. File Pixels are usually understood to be the smallest indivisible component of the image, so there cannot be half a pixel.
* **Reduce code error.** Keeping File Pixels as integers means that all references to the same file pixel are equivalent. This avoids the potential error of checks failing since 10.0 != 10.5.
* **Remove unnecessary information.** The knowledge that the user has clicked halfway across a File Pixel isn’t useful, as no edits can be made to half pixels, so can be safely ignored.

### Algorithm 2.9B Drawing Green Pixels

To draw Green Pixels, the operation can be done very quickly using the in-built Draw Rectangle tool in C#, and only squares will need to be drawn.

The zoom is used twice here as this is the width and height of the rectangle.

#### Pseudocode

DrawGreenPixel(x,y) {

displayPoint = [FilePointToDisplayPoint(x,y)](#_File_Pixels_to)

DrawRectangle(displayPoint.X,displayPoint.Y,zoom,zoom,colours[x,y])

}

### Algorithm 2.9C Drawing Yellow Pixels

To draw Yellow Pixels, the process can be achieved by using a ClampNumber function. ClampNumber will receive an input number, a minimum and a maximum. If the input number is inbetween the minimum and maximum, it will return the input. If the input is less than the minimum or greater than the maximum, it will return the minimum or maximum respectively.

In other words, the pseudocode for ClampNumber is:

ClampNumber(input, minimum, maximum) {

IF input < minimum THEN

RETURN minimum

ELSE IF input > maximum THEN

RETURN maximum

ELSE

RETURN input

END IF

}

In this case, each corner of the pixel to be drawn will be ‘clamped’ inside of the red rectangle (inside of the display form). This means no pixels will be drawn outside of the screen.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

#### Pseudocode

DrawYellowPixel(x,y) {

baseDisplayPoint = [FilePointToDisplayPoint(x,y)](#_File_Pixels_to)

Zoom is added here to find the location of the bottom of the display pixel.

displayPointTopRightX = ClampNumber(baseDisplayPoint.X, redRectangleTopRightX, redRectangleBottomLeftX)

displayPointTopRightY = ClampNumber(baseDisplayPoint.Y, redRectangleTopRightY, redRectangleBottomLeftY)

displayPointBottomLeftX = ClampNumber(baseDisplayPoint.X + zoom, redRectangleTopRightX, redRectangleBottomLeftX)

displayPointBottomLeftY = ClampNumber(baseDisplayPoint.Y + zoom, redRectangleTopRightY, redRectangleBottomLeftY)

pixelWidth = displayPointBottomLeftX – displayPointTopRightX;

pixelHeight = displayPointBottomLeftY – displayPointTopRightY;

DrawRectangle(displayPointTopRightX, displayPointTopRightY, pixelWidth, pixelHeight, colours[x,y])

}

## Algorithm 2.10 Determining Bar Size

Now that the zooming has been implemented, there now needs to be a way to manipulate the zoom at runtime. As described on the original GUI design, the zoom will be manipulated using two bars at the edges of the screen:

Scroll bars

### Algorithm 2.10A Determining whether bars are visible or not.

The first consideration must be made to determine whether the scroll bars need to be shown. In examples where the image (with zoom applied) is smaller than the viewing screen, the scroll bars are not needed as the entire image is viewable at once. This means it must be possible to check whether the scroll bars need to be displayed, before attempting to display them.

For example, in this scenario, where the green represents the viewable image, there is obviously no need to put scroll bars in; there is nothing to scroll through.

Conversely, if the image is much more zoomed in, then the scroll bars become needed to help select which part of the image to look at:

The total size of the image can be calculated by multiplying the file size of the image in that axis (X or Y) by the zoom. If this is larger than the display form, the corresponding bar needs to be displayed.

Thus this makes the pseudocode:

IsBarVisible([Axis](#_2.1.4.2_Axis) axis) {

imageSizeInAxis = image.getFileSizeInAxis(axis) \* zoom

IF displayForm.getDisplaySizeInAxis(axis) > imageSizeInAxis {

RETURN true

ELSE

RETURN false

END IF

}

This code made reference to the function *getFileSizeInAxis(axis)*. This is a function that would return the size of the file in that axis (the axis being X or Y).

### Algorithm 2.10B Determining Bar Size

The size of the bar is determined by its defined width, and its maximum (and minimum) value. The minimum value will always be 1 (the top of the image).

In this GUI example, if the window was only able to display half of the image, the bars should look like:

Or, in specific numbers, if the image’s size was 20dpx by 10dpx, but the display form was only 10dpx by 5dpx (half on each axis), this is what the result from the bars should be.

This means the bar should take up ½ of its available space. As 10dpx / 20dpx = ½

However, an **easier implementation** for this sort of bar would be to (in the X axis) set the Maximum of the bar to the Display Size of the image, and set the Bar’s width to the Display Size of the form. This means that the bar’s display code will automatically resize the bar to its appropriate width, making the code much simpler.

Thus the pseudocode is:

SetupBarSize(progressBar, Axis) {  
 progressBar.Maximum = image.getDisplaySizeInAxis(Axis)  
 progressBar.BarSize = displayForm.getDisplaySizeInAxis(Axis)  
}

It’s important here to set the maximum before the bar’s size, or else the bar might be given a large size than the maximum – causing an error

## Algorithm 2.11 Interpreting Bar Location

This algorithm consists of two parts. Since the portion of the image that the user sees is entirely defined by the [centre location](#centreLocation), there will need to be a way to determine the bar location from the current centre location.

Coming back to the prior example of the window:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn, Partially Drawn

In this example, there are many potential places where the centre could be placed, illustrated here:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Key: Fully Drawn, Partially Drawn

It is needed to be able to have the green rectangle place-able at any point in the image. This ensures that any pixel can be seen **in full** (hence why viewing with the yellow rectangle is not enough).

An algorithm for determining the dimensions of the green rectangle [have already been designed](#_Algorithm_2.8B_Finding), and so the

