Section 2 – Framework

General Contents

- [Decomposition Diagram](#)
- [Usability Diagram](#)
- [IPO Diagrams](#)
- [Class Design #1](#)
- [Algorithm Design](#)
  - Contains smaller Unit Tests
- [Class Design #2](#)
  - This supersedes the previous Class Design
- [Class Relation Diagrams](#)
- [Other key data structures](#)
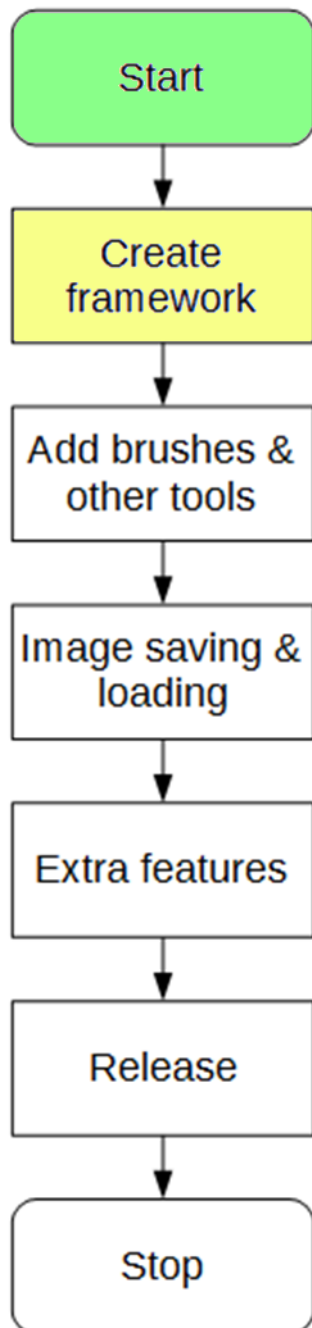- [Testing Table](#)
- [Testing Plan](#)

Contents

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.

```
┌─────────────────┐
│      Start      │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│     Create      │
│    framework    │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│  Add brushes &  │
│   other tools   │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│ Image saving &  │
│     loading     │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│  Extra features │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│     Release     │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│      Stop       │
└─────────────────┘
```

2.1.1 Success Criteria Fulfilment Plan

In this section, the following success criteria are planned to be completed:

Not completed
To be done this section
Completed

Feature	Proof	Code
Section A - Brushes		
Variable brush width	Screenshot of strokes of the same brush showing different widths	A1
Hard brushes	Screenshot showing the hard edge of the brush (colour to no colour)	A2
Shape creation tools	Screenshot showing the shape toolbar and a small selection of drawn shapes	A3
Fill (bucket) tool	Screenshot showing a before and after of filling a large area	A4
Single pixel pencil	Screenshot showing a stroke of the single pixel brush	A5
Rubber	Screenshot showing a densely packed picture being rubbed out	A6
Section B – Other editing tools		
Image viewer	Screenshot of a currently being viewed image	B1
Bitmap image editor	Screenshot of a zoom in on the image showing the pixels	B2
RGB colour picker	Screenshot showing a system for entering an RGB colour	B3
RGB direct input	Screenshot showing the user entering "FF0000" (or equivalent) and the programming outputting red	B4
Layer system	Screenshot of layer navigator	B5
Rectangle selection tool	Screenshot showing a rectangle selection on the image	B6
Magic selection tool	Screenshot showing a complex selection around non-linear shape	B7
Transparent pixels	Screenshot showing a layer with blank pixels (one layer on top of another). Partial transparency is not required	B8
Zoom in (no zoom out)	Screenshot of an image at smallest zoom, followed by a screenshot at max zoom showing a portion of an image much smaller	B9
Text	Screenshot of the text "Hello World" on the image	B10
Eyedropper tool	Screenshot of an imported image, with the colour stroke of a colour taken from that image beneath it	B11
Image effects	*Screenshot of an image before and after an effect is applied*	B12
Rotating Images	*Screenshot of an image in 4 different rotations, normal, 90°, 180° and 270°*	B13
Clipping masks	*Screenshot of an image being clipped onto a complex selection*	B14
Section C – File System		
Creating a new image	Screenshot of a blank 300x300 square image	C1
Importing images	Screenshot of the file browser showing an image preview, and screenshot showing the image in the program	C2

Exporting images	Screenshot showing a custom image in the program, followed by an image showing the file browser showing the image in a folder	C3
Supporting PNG and JPEG	Screenshot showing the file browser which accepts both PNG and JPEG images	C4
Saving and loading from a proprietary format	Screenshot showing the user saving an image, screenshot of the image in the file browser, and the program after the image is loaded	C5
Section D – Usability		
Program should be stable and not crash.	A complete testing table, showing no failed tests, followed 75% yes response to asking stakeholders "Did you encounter any errors while using the program?"	D1
Program should be easy to use	75% yes response to asking stakeholders "Did you find the program easy to use?"	D2
Features should be easily accessible	From the default state of the program, any feature will need to be activated by no less than 4 clicks	D3

2.1.2 Section Decomposition

2.1.2.1 Decomposition diagram

```
                                    ┌───────────┐
                                    │ Framework │
                                    └───────────┘
```

Framework

- **Creating a blank image**
  - Creating 'image' class — 1
  - Populating 'image' class — 2
- **Displaying a blank image**
  - Resizing picture box — 3
  - Displaying pixels — 4
- **Setting pixels on an image at runtime** — 5
- **Making window resizeable**
  - Determine picture box location — 6
  - Stop window becoming too small — 7
- **Adding zoom**
  - Zoom in on fixed position
    - Decide which part needs to be drawn — 8
    - Draw that part — 9
  - Manipulate zoom at runtime
    - Handle scroll bars
      - Determine bar size — 10
      - Interpret bar location — 11
    - Redraw Needed pixels — 12
- **Detecting mouse position**
  - Find coords on Picture box — 13
  - Find coords on overall image
    - Find point on overall image — 14
    - Snap to correct pixel — 15

> Each lowest level block has a label, which will become an algorithm designed in this document.
>
> These algorithms will then be combined together.

2.1.2.2 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

2.1.3 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.

2.1.3.1 UI Design

Canvas

Scroll bars

Zoom bar
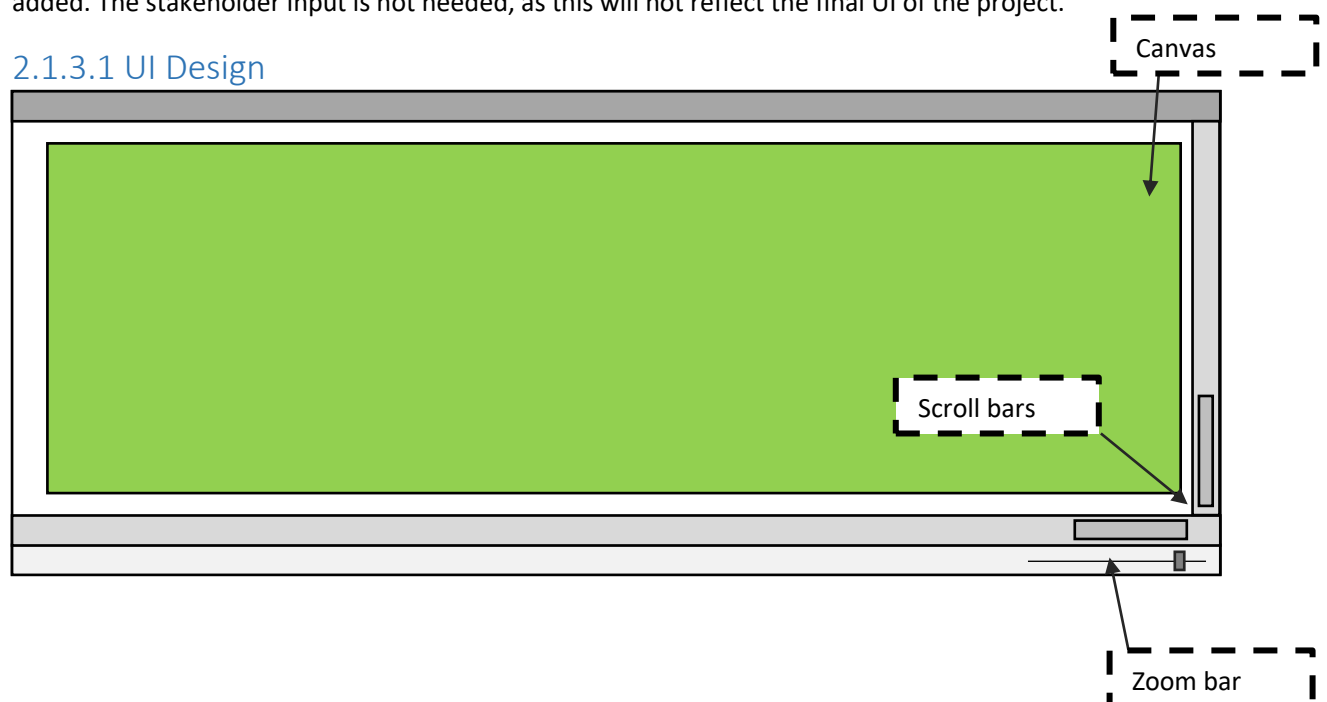
2.1.3.2 UI Design Features

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.

2.1.4 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.

2.1.5 Class Design

*Note. This class design has been redeveloped. The updated design can be found **here***

2.1.5.1 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	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()

      }

}
```

2.1.5.2 ZoomSettings
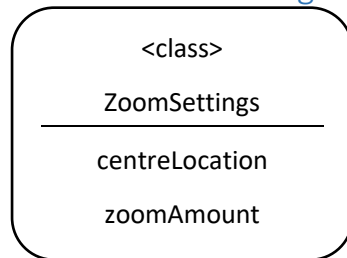
```
           <class>

         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

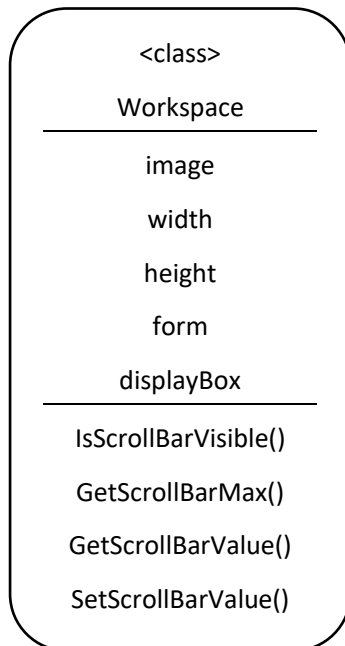| Property	Datatype	Justification
centre-Location	Point	Stores the location of the pixel in the middle of the zoom. The middle location is stored to make the zoom feel more natural when zooming in and out (the middle pixel remains the same)
zoom-Amount	Integer	Stores the amount that the image is currently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

Constructor

```
class ZoomSettings {

      Point centreLocation

      Integer zoomAmount

      constructor(_centreLocation) {

            centreLocation = _centreLocation
            zoomAmount = 1

      }

}
```

The default zoom will be none - 1x.

2.1.5.3 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 class

```
            <class>
           Workspace
         ─────────────
             image
             width
             height
             form
           displayBox
         ─────────────
       IsScrollBarVisible()
       GetScrollBarMax()
       GetScrollBarValue()
       SetScrollBarValue()
```

Properties

Property	Datatype	Justification
image	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, 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, the X or Y bar.	Integer	Calculates what the maximum for the selected scroll bar needs to be set to.
GetScroll-BarValue()	axis, Axis, 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, 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

        form = _form

        width = width of form

        height = height of form

        displayBox = new PictureBox

        add displayBox to form

    }

}
```

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

2.1.5.2 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
```
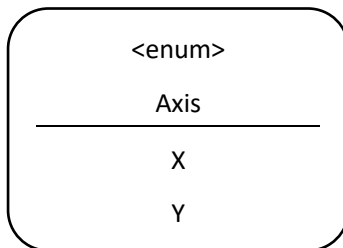
Member	Justification
X	Denotes that the X axis has been selected.
Y	Denotes that the Y axis has been selected.

2.1.6 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)

     }

}
```

Unit Testing

Test Data	Test Type	Expected Output	Description
new Image(10,10)	Valid	An image of size 10fpx, 10fpx	This tests if a normal image can be created
new Image(10,6)	Valid	A 10fpx, 5fpx image	This tests if a non-rectangular image can be created
new Image(10,5)	Valid	A 10fpx, 2fpx image	This tests when an odd dimension is entered
new Image(10,1)	Extreme Valid	A 10fpx, 1fpx image	This tests if a very short image can be created
new Image(1,10)	Extreme Valid	A 1fpx, 10fpx image	This tests if a very thin image can be created
new Image(1,1)	Extreme Valid	A 1fpx, 1fpx image	This tests when a very small image is created
new Image(10,0)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no height is rejected
new Image(0,10)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no width is rejected
new Image(0,0)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no width or height is rejected
new Image(10000,10)	Extreme Invalid	The parameters are rejected and no image is created	This tests if a width which is much too large is rejected

new Image(10,10000)	Extreme Invalid	The parameters are rejected and no image is created	This tests if a height which is much too large is rejected
new Image(-1,10)	Invalid	The parameters are rejected and no image is created	This tests if a negative width is rejected.
new Image(10,-1)	Invalid	The parameters are rejected and no image is created	This tests if a negative height is rejected.
new Image(10)	Erraneous	The parameters are rejected and no image is created	This tests if an image missing a height is rejected.
new Image("10","10")	Erraneous	The parameters are rejected and no image is created	This tests if an image with invalid numbers is rejected

Algorithm 2.3 Resizing Picture Box

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

    }

}
```
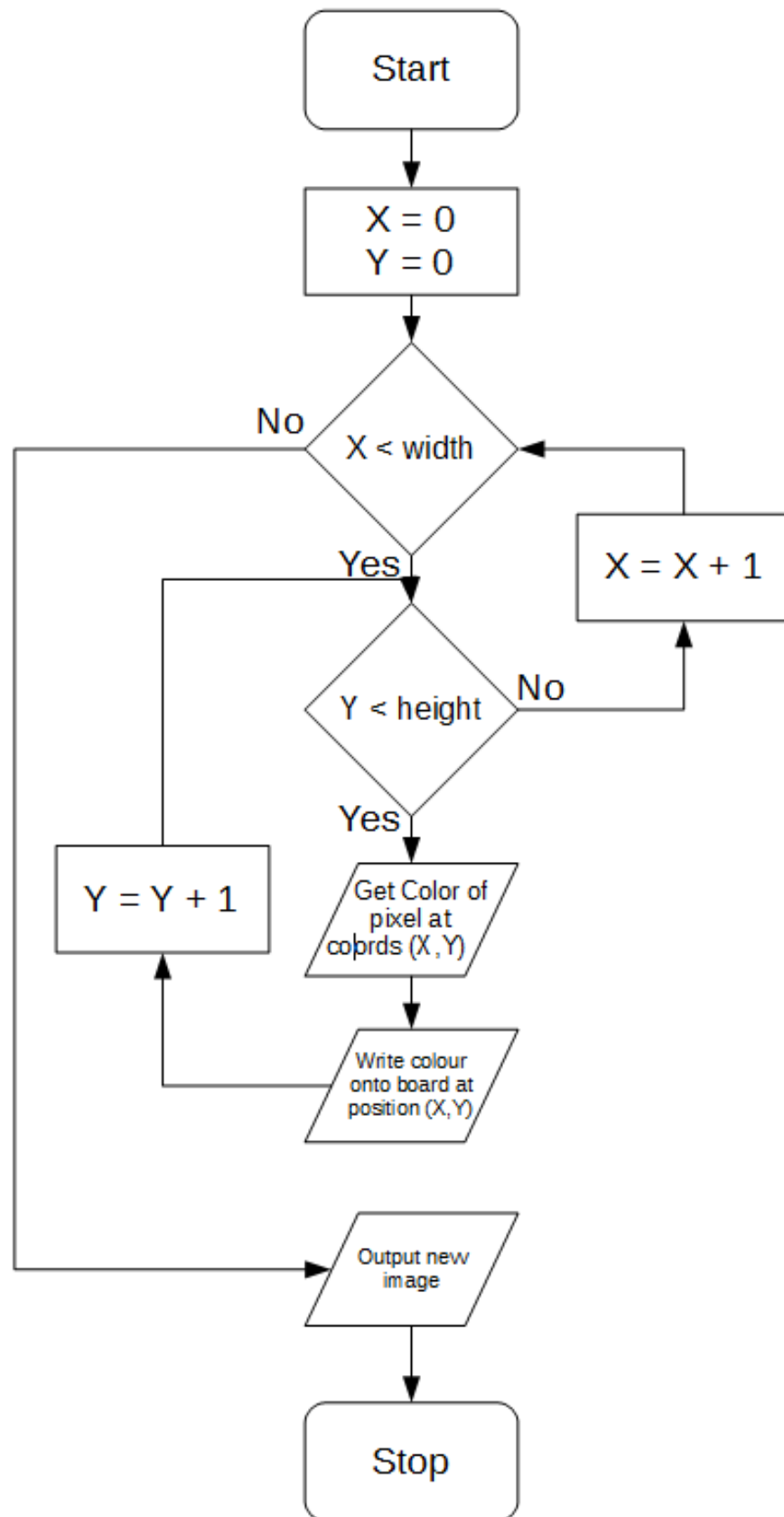
Unit Testing

Test Data	Test Type	Expected Output	Description
Input Box(10,10)	Valid	An image of size 10fpx, 10fpx	This tests if a normal image can be created
Input Box(10,6)	Valid	A 10fpx, 5fpx image	This tests if a non-rectangular image can be created
Input Box(10,5)	Valid	A 10fpx, 2fpx image	This tests when an odd dimension is entered
Input Box(10,1)	Extreme Valid	A 10fpx, 1fpx image	This tests if a very short image can be created
Input Box(1,10)	Extreme Valid	A 1fpx, 10fpx image	This tests if a very thin image can be created
Input Box(1,1)	Extreme Valid	A 1fpx, 1fpx image	This tests when a very small image is created
Input Box(10,0)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no height is rejected
Input Box(0,10)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no width is rejected
Input Box(0,0)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no width or height is rejected
Input Box(10000,10)	Extreme Invalid	The parameters are rejected and no image is created	This tests if a width which is much too large is rejected
Input Box(10,10000)	Extreme Invalid	The parameters are rejected and no image is created	This tests if a height which is much too large is rejected
Input Box(-1,10)	Invalid	The parameters are rejected and no image is created	This tests if a negative width is rejected.
Input Box(10,-1)	Invalid	The parameters are rejected and no image is created	This tests if a negative height is rejected.
Input Box(10)	Erraneous	The parameters are rejected and no image is created	This tests if an image missing a height is rejected.
Input Box("10","10")	Erraneous	The parameters are rejected and no image is created	This tests if an image with invalid numbers is rejected

Algorithm 2.4 Displaying Pixels

Flowchart

```
                    Start
                      |
                      v
                  X = 0
                  Y = 0
                      |
                      v
      No        < X < width >  <----  X = X + 1
      |               |                    ^
      |              Yes                    |
      |               |                     |
      |               v                     |
      |         < Y < height >  --No-------->
      |               |
      |              Yes
      |               |
      |               v
  Y = Y + 1      Get Color of
      ^           pixel at
      |          coords (X, Y)
      |               |
      |               v
      |          Write colour
      |<---------onto board at
                 position (X, Y)

                 Output new
   ------------>   image
                      |
                      v
                    Stop
```

Pseudocode

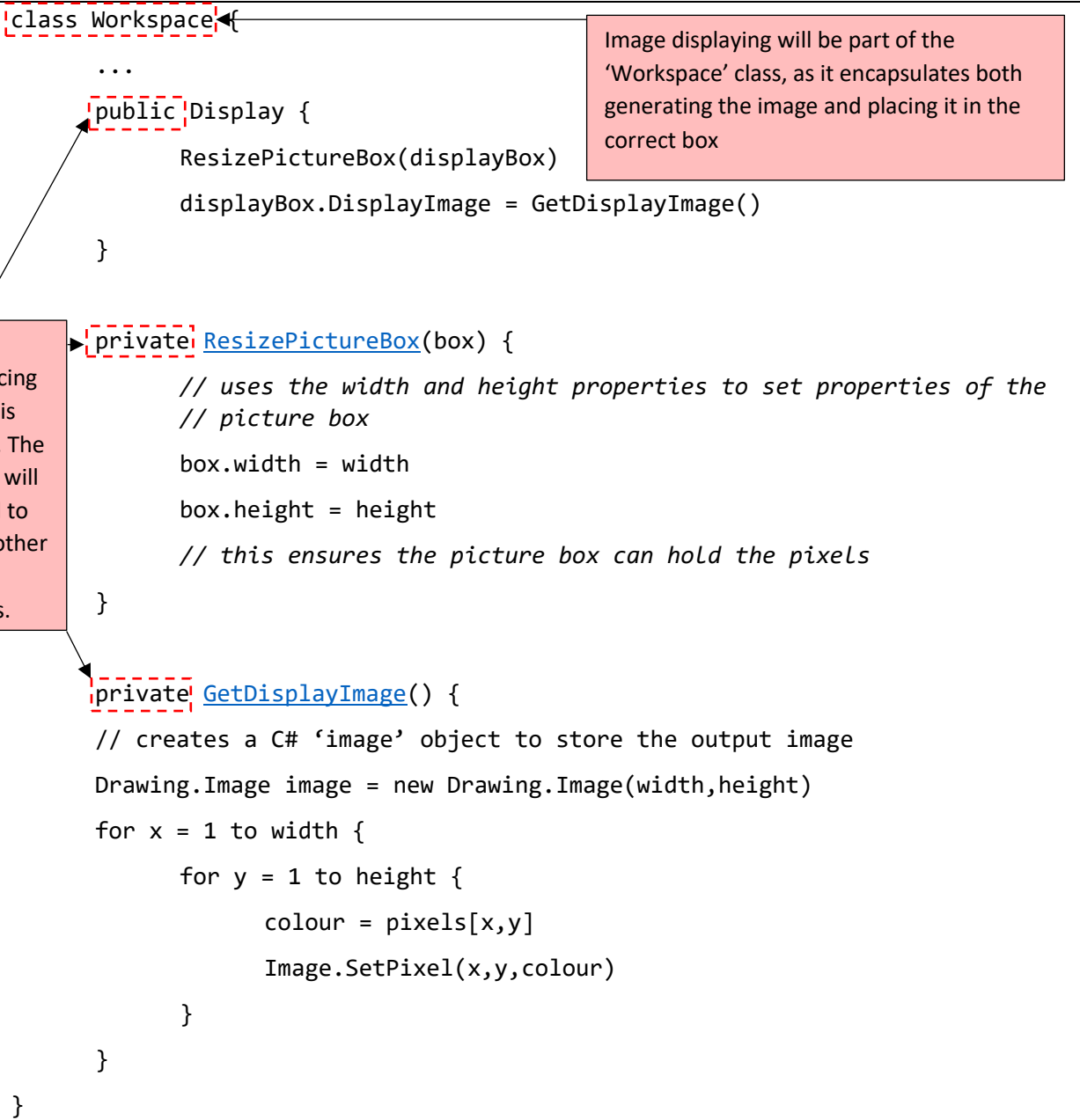For the pseudocode of this algorithm, I have two options for outputting

```
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)

        }

    }

    RETURN image

}
```

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 {
    ...
    public Display {
        ResizePictureBox(displayBox)
        displayBox.DisplayImage = GetDisplayImage()
    }


    private 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
    }


    private 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)
            }
        }
    }
}
```

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

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

Algorithm 2.5 Setting Pixels at Runtime

Pseudocode

```
class Image {

    ...

    SetPixel(x,y,colour) {

        pixels[x,y] = colour

        Display()

    }

}
```

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

Unit Testing

This assumes the image has a size of 10px by 10px

Test Data	Test Type	Expected Output	Description
SetPixel(5,5,Black)	Valid	A pixel near the middle of the image is set to black	This tests if a normal pixel can be set
SetPixel(5,5,Green)	Valid	A pixel near the middle of the image is set to yellow	This tests if a pixel can be set to other colours than black
SetPixel(0,0,Black)	Extreme Valid	A pixel in the top-left corner is set to black	This tests if the top-left corner can be set
SetPixel(9,9,Black)	Extreme Valid	A pixel in the bottom-right corner is set to black	This tests if the bottom-right corner can be set
SetPixel(-1,0,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far left is rejected
SetPixel(0,-1,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far up is rejected
SetPixel(10,0,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far right is rejected
SetPixel(0,10,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far down is rejected

Algorithm 2.6 Determining PictureBox Location

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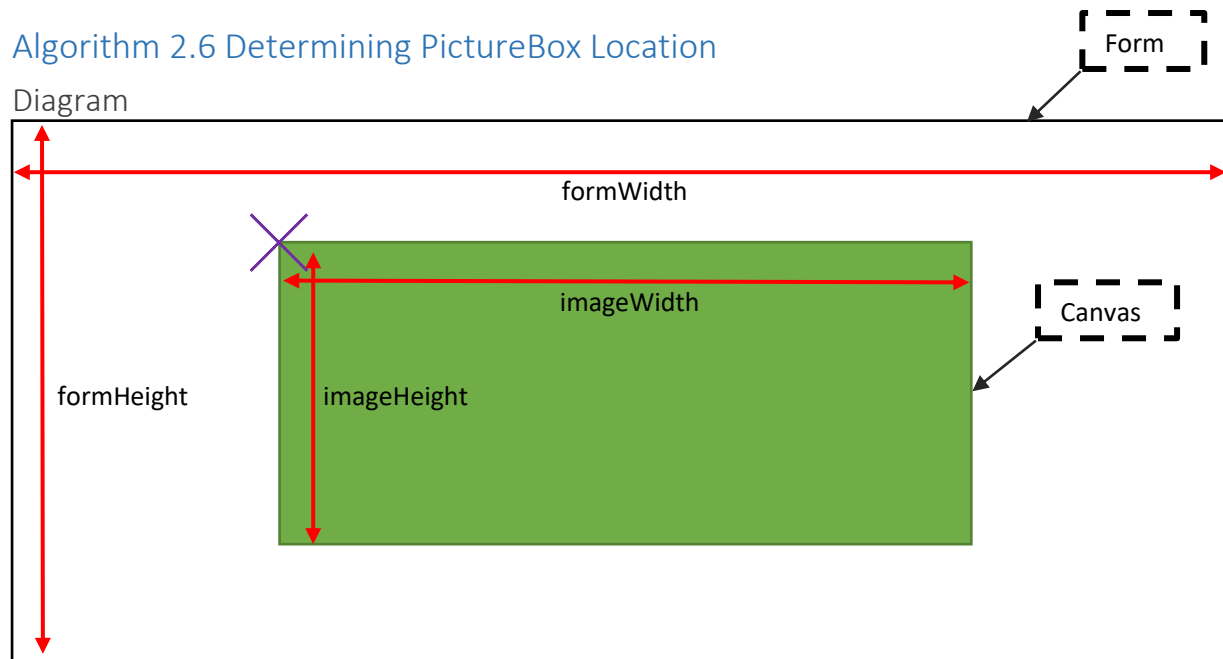
The purple ✖ denotes what the location of the picture box must be. Looking at the diagram, it becomes clear that the X position of the ✖ must be:

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.

Unit Testing

Test Data	Test Type	Expected Output	Description
Form is started at normal size	Valid	Image Displays in the middle of the form	This tests that the program can start and display the image
Form is maximized	Extreme Valid	Image displays in the middle of the large form	This tests if the image is displayed when the form is very large
Form is minimized	Extreme Valid	No image is displayed (as form is currently invisible)	This tests that the program can be minimized safely
Form is resized to smallest possible	Extreme Valid	The form cannot be made smaller than the image	This tests that the form can never be made smaller than the image
Form is resized to minimum width maximum height	Extreme Valid	A very thin form displays the image	This tests if a form with an extreme height but minimum width is accepted
Form is resized to minimum height maximum width	Extreme Valid	A very short form displays the image	This tests if a form with an extreme width but minimum height is accepted
The form's size is rapidly changed.	Extreme Valid	The image is very quickly moved around but remains centred	This tests that under a stress test the image is displayed correctly

Algorithm 2.8 Deciding which pixels to draw

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. Note that display pixels are bound to the screen, so a display pixel of 0,0 could be a pixel anywhere on the image

Algorithms to switch between these two sorts of pixel are here.

From this point on, algorithms involving zoom and zoom centring will be implemented.

Algorithm 2.8A 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).

Assuming there is an image of 20fpx by 10fpx:
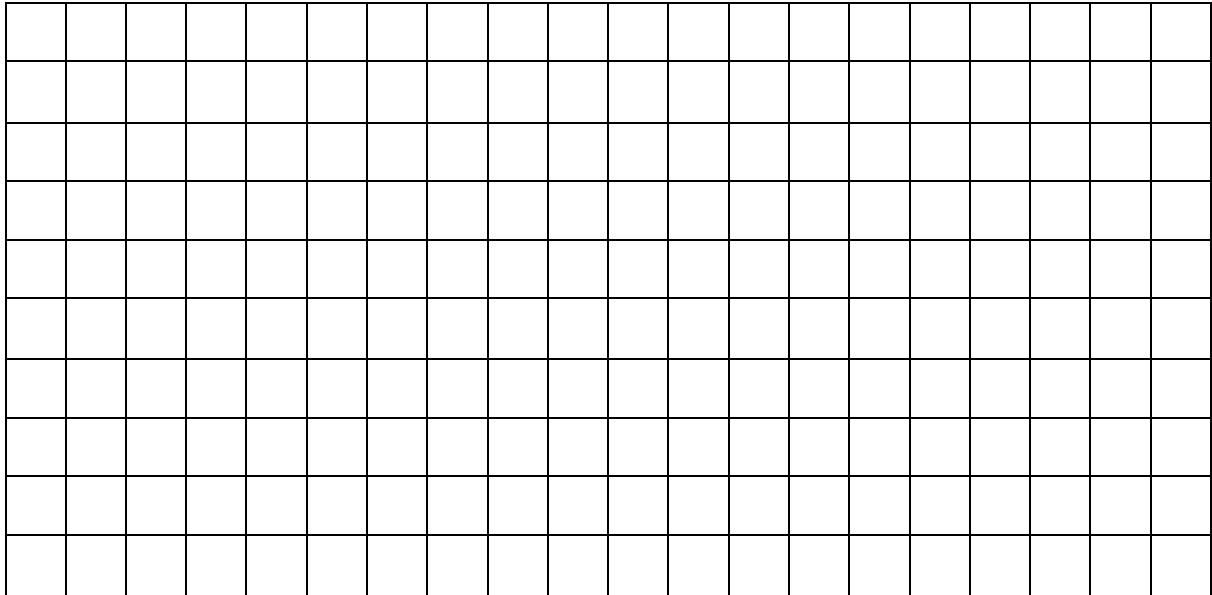
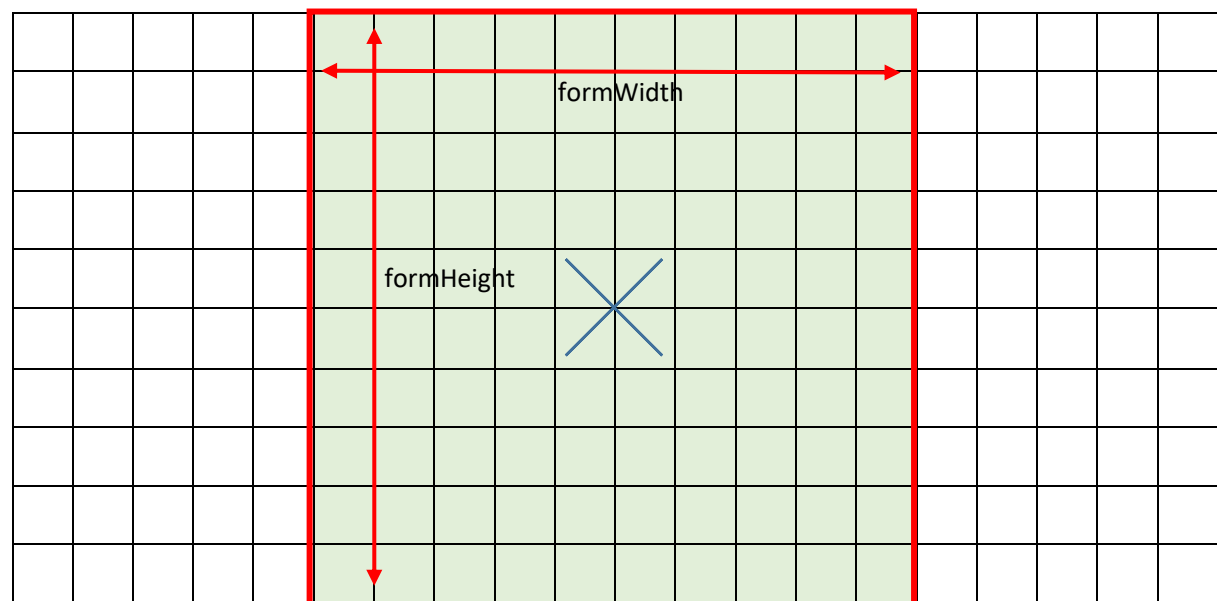If the screen has a size of 10fpx10fpx, centred on the pixel at 5fpx,5fpx, then the zoomed area will be:
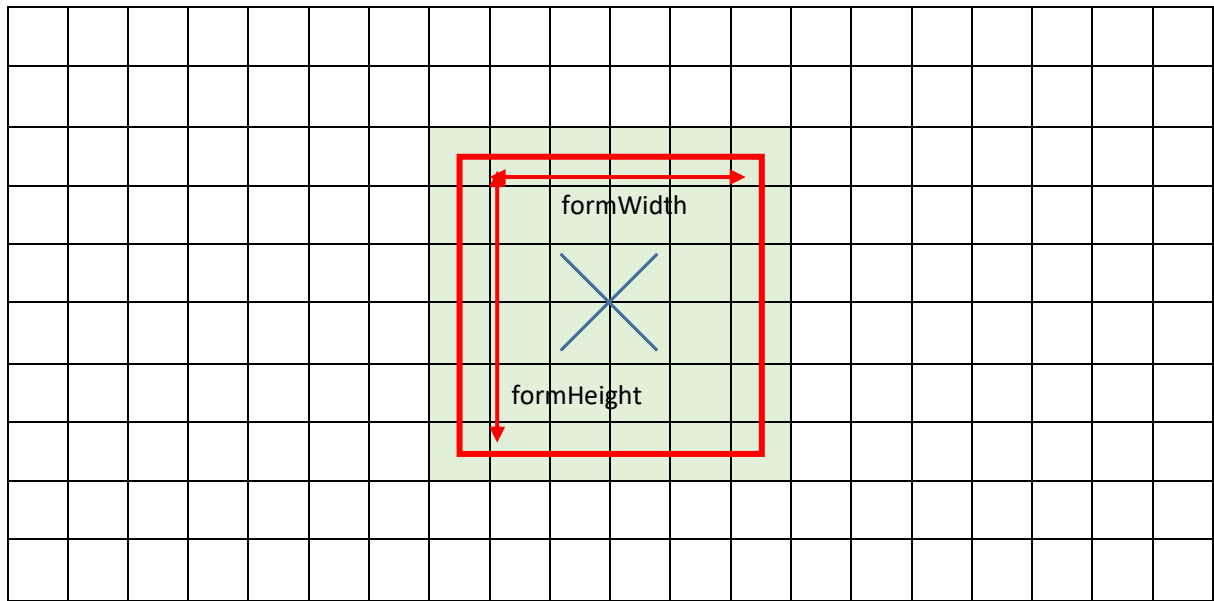
formWidth

formHeight
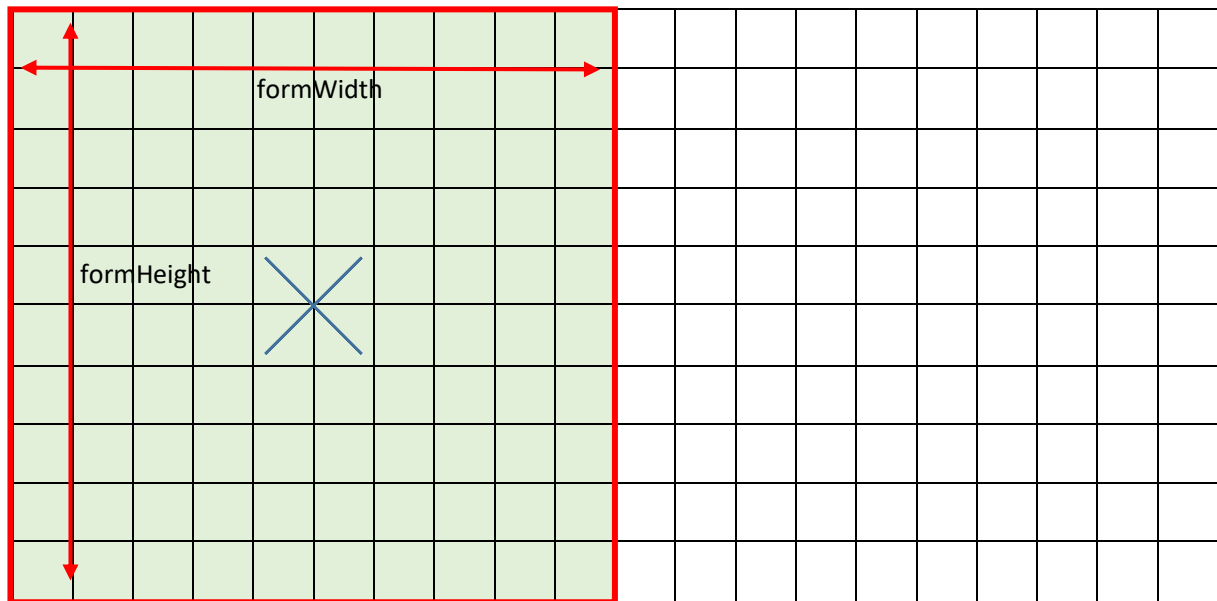
Key: Fully Drawn

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

formWidth

formHeight

Key: Fully Drawn

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

The user will, in this example, see this:
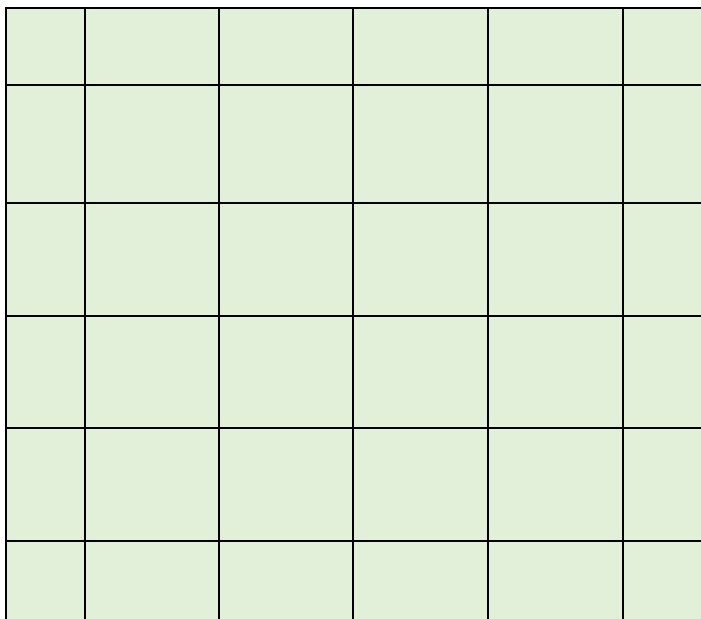
A green pixel represents a pixel that must be drawn to the screen

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.

Thus, the form's view will be:
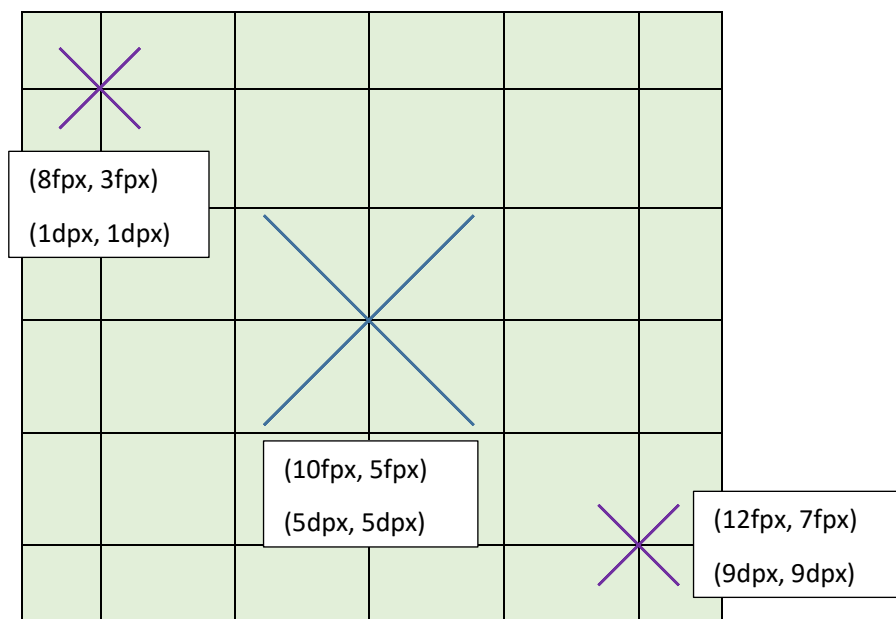
(Each square in this diagram is 2dpx by 2dpx)

(8fpx, 3fpx)
(1dpx, 1dpx)

(10fpx, 5fpx)
(5dpx, 5dpx)

(12fpx, 7fpx)
(9dpx, 9dpx)

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)
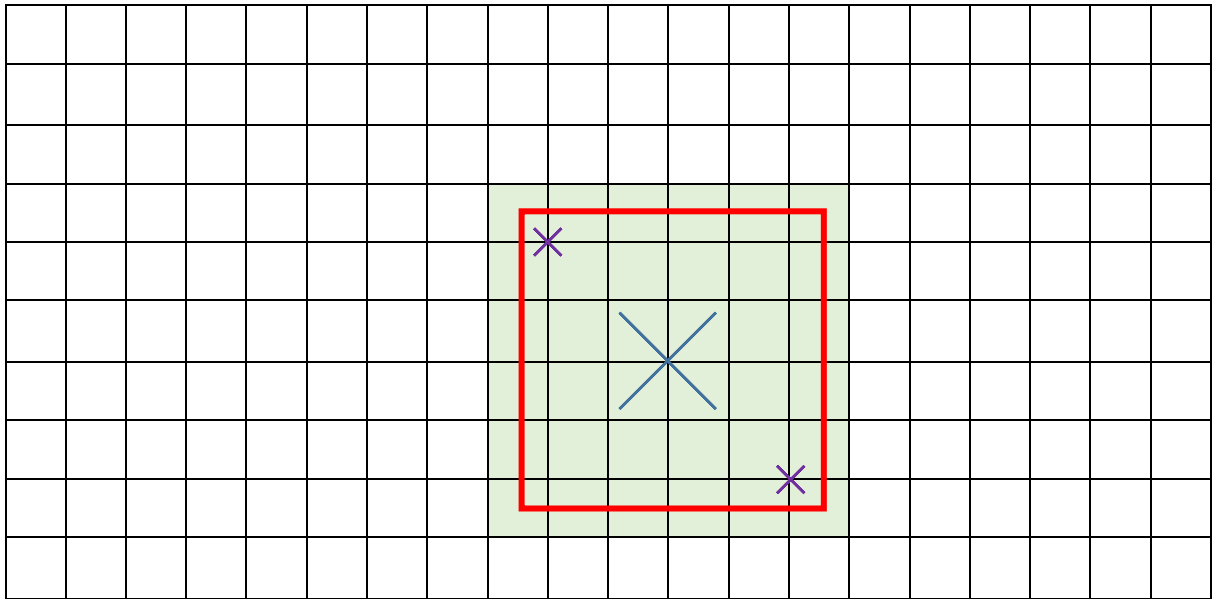
Each square in this diagram is 2dpx by 2dpx
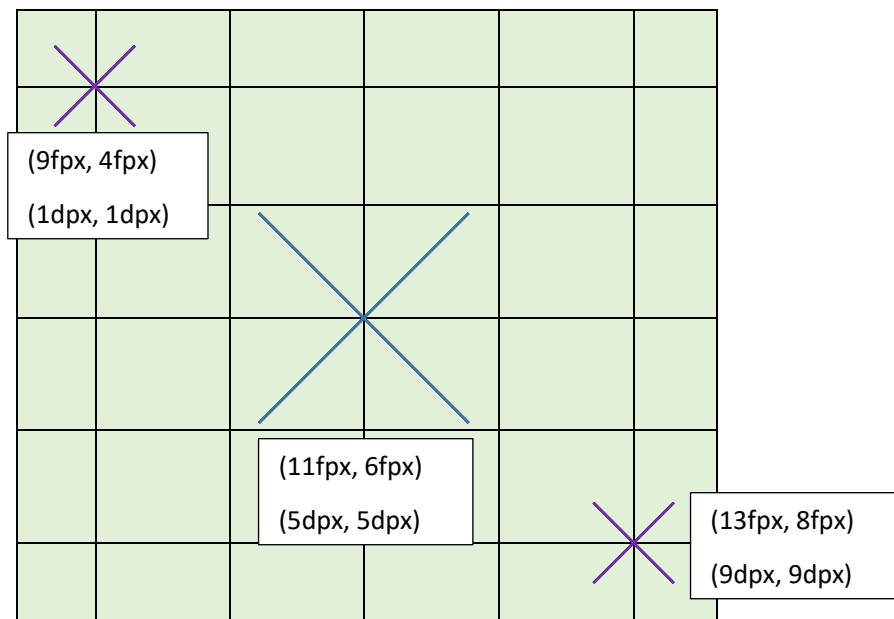
The expected display locations (in dpx) have been labelled, as well as their file positions (in fpx).

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

Start

Calculate the File
Displacement

Multiply Displacement
by Zoom

Add Displacement to
Display Centre
Location

Stop
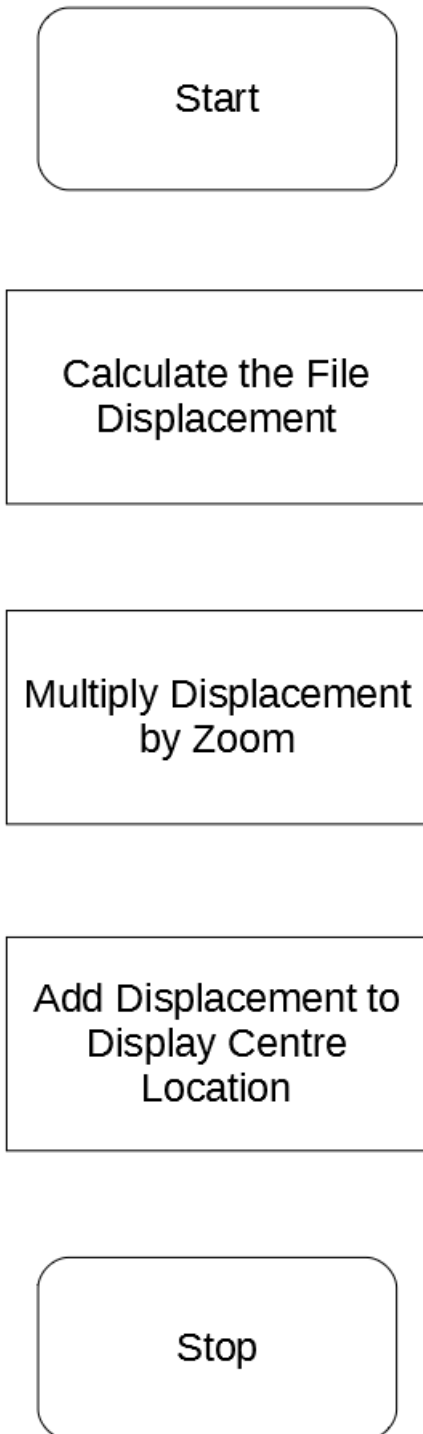
For an example, considering the first diagram again:

Thus, the form's view will be:

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

(8fpx, 3fpx)

(1dpx, 1dpx)

(10fpx, 5fpx)

(5dpx, 5dpx)

(12fpx, 7fpx)
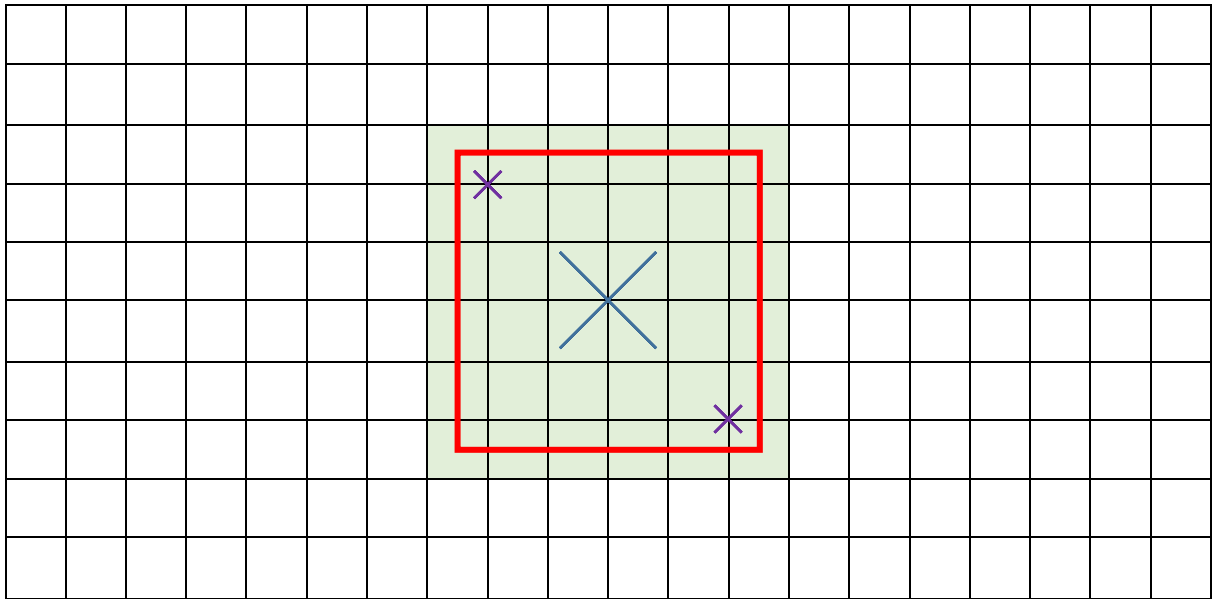
(9dpx, 9dpx)
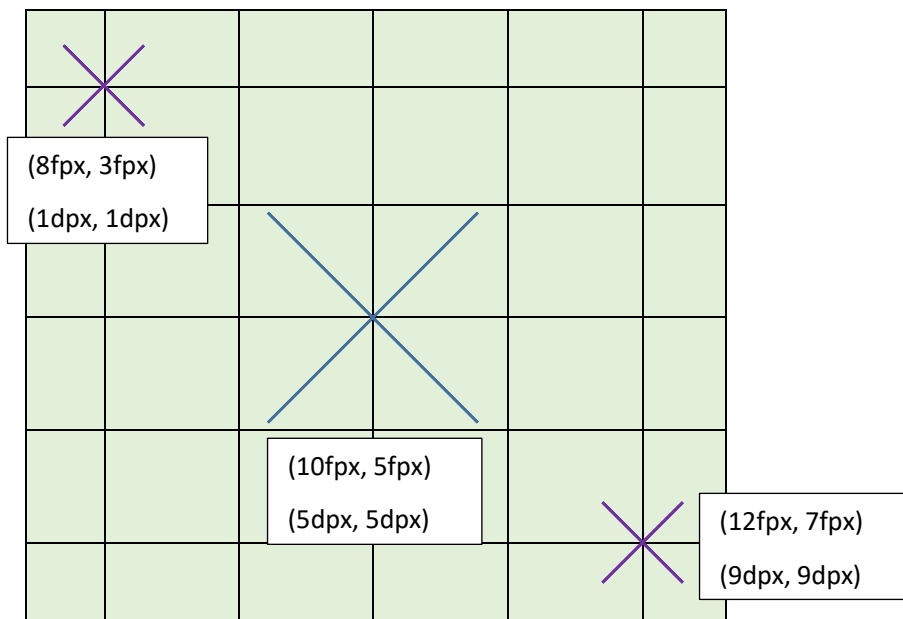
Considering the top-left X.

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)
}
```

Unit Testing

This assumes the image has a size of 20fpx by 10fpx and the zoom centre is at (10fpx, 5fpx) and zoom is at 2x

Test Data	Test Type	Expected Output	Description
DisplayPoint(10,5)	Valid	(5,5)	This tests if a point at the same location as the zoom centre is returned as the centre location of the zoom centre
DisplayPoint(8,3)	Valid	(1,1)	This tests if a point to the top left of the centre location is positioned appropriately in the image
DisplayPoint(12,7)	Valid	(9,9)	This tests if a point to the bottom right of the centre location is positioned appropriately in the image
DisplayPoint(7,2)	Extreme Valid	(-1,-1)	This tests if negative coords will be outputted. Whilst this may seem like a glitch this is necessary for some of the displaying code to display half size pixels.
DisplayPoint(13,8)	Extreme Valid	(11,11)	This tests if coords larger than display form will be outputted. This is also necessary for some of the displaying code to display half size pixels.
DisplayPoint(0,0)	Extreme Valid	(-15,-5)	This tests if a point in the top right corner has a relative display point calculated
DisplayPoint(19,9)	Extreme Valid	(23,13)	This tests if a point in the bottom left corner has a relative display point calculated
DisplayPoint(-1,-1)	Extreme Invalid	Throws out of bounds error	This tests if a point that is out of the bounds of the grid is not accepted
DisplayPoint(20,10)	Extreme Invalid	Throws out of bounds error	This tests if a point that is out of the bounds of the grid is not accepted

Display Pixels to File Pixels

In order to construct this algorithm, it becomes clear that the inverse of the File Pixels to Display Pixels algorithm should be employed.

Flowchart

This is the same steps as the File Pixels to Display Pixels 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


    displacementX = displacementX / zoom

    displacementY = displacementY / zoom


    newX = displacementX + centreFilePoint.X

    newY = displacementY + centreFilePoint.Y


    return new Point(newX, newY)

}
```

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

Unit Testing

Test Data	Test Type	Expected Output	Description
FilePoint(5,5)	Valid	(10,5)	This tests if a point at the same location as the zoom centre is returned as the centre location of the zoom centre
FilePoint(1,1)	Valid	(8,3)	This tests if a location near the top of the display box is returned as above the centre location
FilePoint(12,7)	Valid	(9,9)	This tests if a location near the bottom of the display box is returned as below the centre location
FilePoint(-1,-1)	Extreme Valid	(7,2)	This tests if a file point away from the top left of the image is returned as its appropriate file point
FilePoint(11,11)	Extreme Valid	(13,8)	This tests if a file point away from the bottom right of the image is returned as its appropriate file point
FilePoint(-15,-5)	Extreme Valid	(0,0)	This tests if a point at the top left of the file will be returned as such
FilePoint(23,13)	Extreme Valid	(19,9)	This tests if a point at the bottom right of the file will be returned as such
FilePoint(6,6)	Valid	(10,5)	This tests if a point that is halfway across a pixel is rounded down to its nearest file location
FilePoint(4,4)	Valid	(9,4)	This tests the same as above
FilePoint(-17,-7)	Extreme Invalid	This point is rejected as it would output (-1,-1)	This tests if a Display Point that would output an invalid file point is rejected
FilePoint(25,15)	Extreme Invalid	This point is rejected as it would output (10,5)	This tests if a Display Point that would output an invalid file point is rejected

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 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.

Why differentiate between pixel types?

White pixels do not need to be drawn, so it is useful to be able to ignore them. When the image is zoomed in, many pixels do not need to be drawn at all. Ignoring them will significantly increase performance, especially with larger images at high zoom.

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 Visible and which are not

Potential Algorithm A: Checking whether a given pixel is Visible

Would be given the location of a pixel and the current zoom settings, and would return whether that pixel is Visible or not. 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 Visible → Draw that pixel

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 Visible Pixels

Would, from the current zoom settings, return a list of all Visible 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 which pixels are visible is only done **once**. This significantly reduces the overheads that may be caused by doing the same sums repeatedly.
- 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 Visible or not, as you'd had to check the entire list to see whether it belongs there.

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 2.8B Determining borders of pixels

From our illustrations before, two borders can be drawn. A red border to denote the edge of what the user can see, and an orange border to denote the range of pixels that must be drawn.
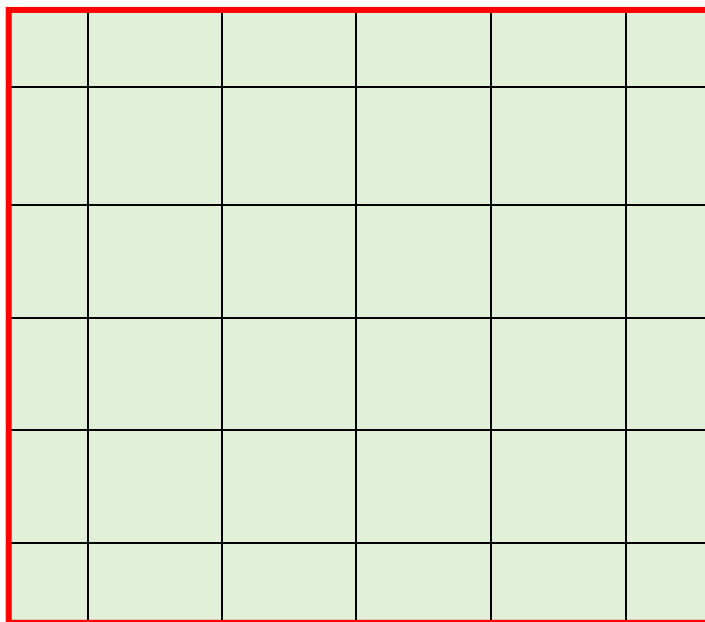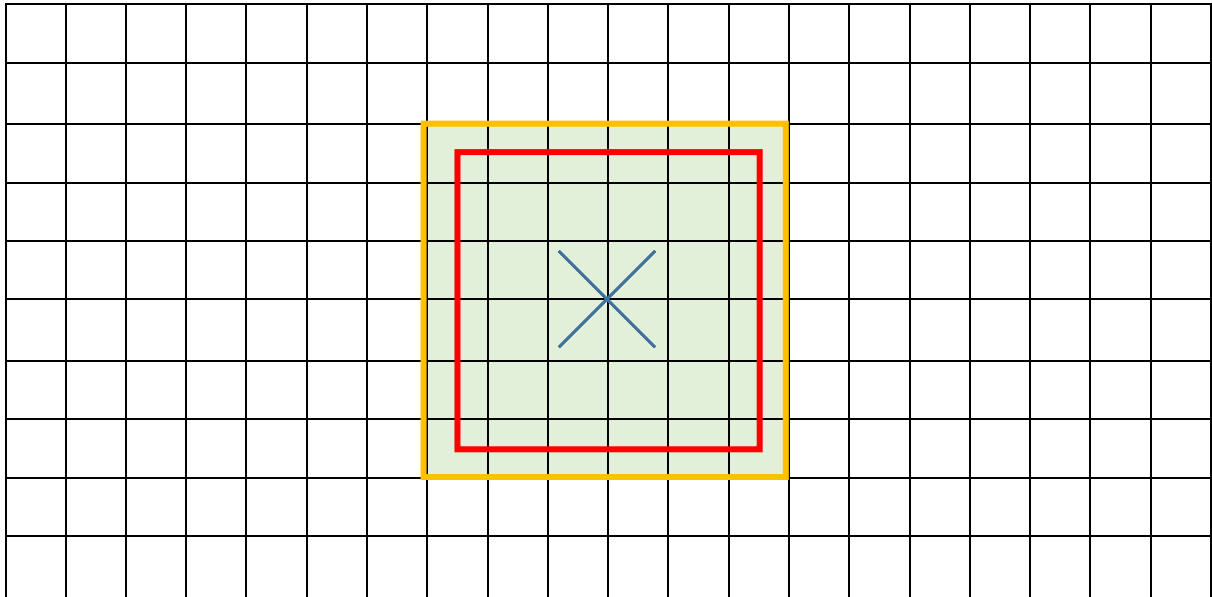
The Red border denotes where the drawing must stop.

By abstracting the rectangles into two points, this diagram can be drawn:
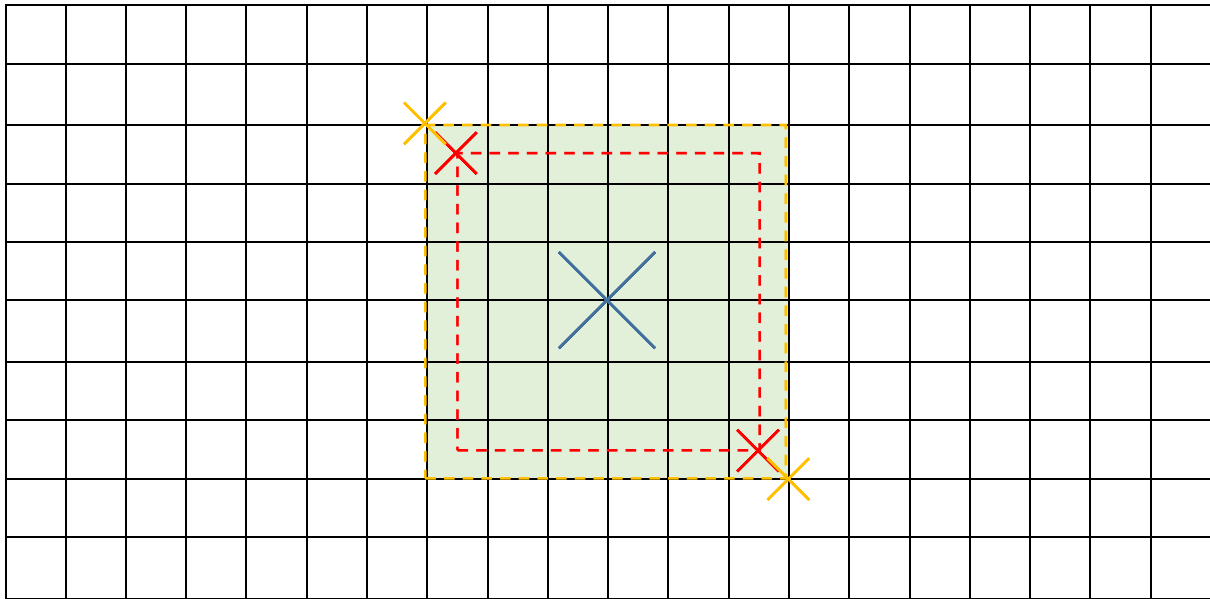
The advantage of these crosses is because it means that if we want to check whether a pixel needs to be drawn or not, all we need to do is check whether it is inbetween the two orange crosses.

Determining Location of Orange Crosses

The centre of this rectangle must be the centre location of the zoom. This implies that calculations must be done relative to the centre location.

In the X Axis firstly, to figure out how many pixels the current form could display in that axis, the program can divide the form size by the display size of one pixel (which is the same as the zoom), and then rounded down. This gives us the amount of file pixels the form can display.

In this example the process would be 10 / 2 (display form is 10dpx across and the current zoom is 2x).

However, if this number is a whole number (for example 5), this can pose a problem if we try to put 5 pixels on each side of the centre location, as 5 cannot be split evenly into two. To find the amount of pixels to display on each side of the centre location, the amount of file pixels must be divided by 2 again, and then rounded **up**. (This is to make sure we draw as many as is needed). In this case the output would be 3, which is the desired result for the example above. (There are 3 visible pixels on each side of the cross).

Thus to find the file coords of Orange Crosses (depending on whether you + or -)

- The X is centreloc.X ± Ceiling(Floor(form.width/zoom) / 2)
    - The ± depends on whether we are looking for the left or right cross
    - In the above example this would be 10 + Ceiling(Floor(10/2)/2) = 10 + Ceiling(Floor(5)/2) = 10 + Ceiling(2.5) = 10 + 3 = 13.

- The X is centreloc.Y ± Ceiling(Floor(form.height/zoom) / 2)
    - The ± depends on whether we are looking for the left or right cross
    - In the above example this would be 5 + Ceiling(Floor(10/2)/2) = 5 + Ceiling(Floor(5)/2) = 5 + Ceiling(2.5) = 5 + 3 = 8.

Algorithm 2.8C Finding Green pixels

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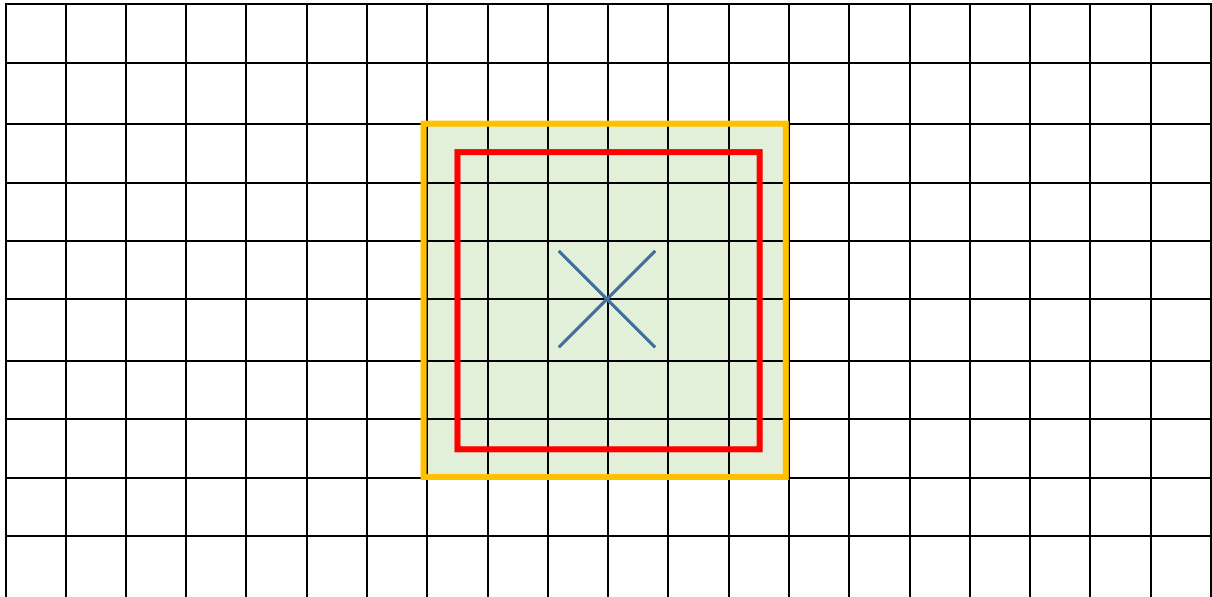
Pseudocode

```
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)

            next y

        next x

    }

}
```

This algorithm was designed above.

Algorithm 2.9 Draw Zoomed Part

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

Pseudocode

```
DrawGreenPixel(x,y) {
        displayPoint = FilePointToDisplayPoint(x,y)
        DrawRectangle(displayPoint.X,displayPoint.Y,zoom,zoom,colours[x,y])
}
```

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

Another advantage of the in-built Draw Rectangle tool is that it is resistant to potentially bad inputs. As some Visible pixels location may start past the boundary of the display window, this could result in a **negative** Display Pixel location being inputted, or a Display Pixel location that is larger than the dimensions of the form. The Draw Rectangle tool handles these and will not draw pixels that it cannot display.

Unit Testing

Test Data	Test Type	Expected Output	Description
Form is started at normal size	Valid	Image Displays in the middle of the form	This tests that the program can start and display the image
Form is maximized	Extreme Valid	Image displays in the middle of the large form	This tests if the image is displayed when the form is very large
Form is minimized	Extreme Valid	No image is displayed (as form is currently invisible)	This tests that the program can be minimized safely
Form is resized to smallest possible	Extreme Valid	The form cannot be made smaller than the image	This tests that the form can never be made smaller than the image
Form is resized to minimum width maximum height	Extreme Valid	A very thin form displays the image	This tests if a form with an extreme height but minimum width is accepted
Form is resized to minimum height maximum width	Extreme Valid	A very short form displays the image	This tests if a form with an extreme width but minimum height is accepted
The form's size is rapidly changed.	Extreme Valid	The image is very quickly moved around but remains centred	This tests that under a stress test the image is displayed correctly

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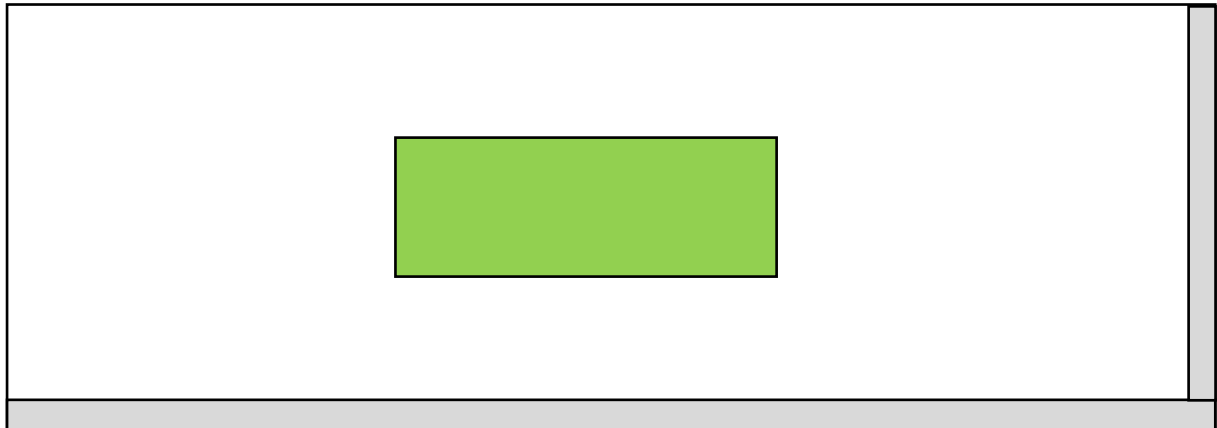
Conversely, if the image is much more zoomed in, then the scroll bars become needed to help select which part of the image to look at:

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.

Pseudocode

```
IsBarVisible(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).

Unit Testing

Test Data	Test Type	Expected Output	Description
Form is started	Valid	The form is started as the same size as the image and no bars are visible	This makes sure that form starts in an unzoomed way
Form width is reduced to less than image	Extreme Valid	The horizontal scroll bar becomes visible	This makes sure that the scroll bar becomes visible
Form width is then increased	Extreme Valid	The horizontal scroll bar disappears	This makes sure that the scroll bar then becomes invisible
Form height is reduced to less than image	Extreme Valid	The vertical scroll bar becomes visible	This makes sure that the scroll bar becomes visible
Form height is then increased	Extreme Valid	The vertical scroll bar disappears	This makes sure that the scroll bar then becomes invisible
Form height and width is both reduced	Extreme Valid	Both scroll bars becomes visible	This makes sure both bars can be visible at the same time
Form height is then increased	Extreme Valid	The vertical scroll bar becomes invisible but horizontal scroll bar remains	This tests than one scroll bar can disappear but the other remains
Form width is then increased	Extreme Valid	The horizontal scroll bar then becomes invisible	This tests than one scroll bar can disappear but the other remains

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, there will need to be a way to determine the bar location from the current centre location. First, the total number of possible locations will need to be determined.

Algorithm 2.11A Determining possible centre locations

Coming back to the prior example of the window:
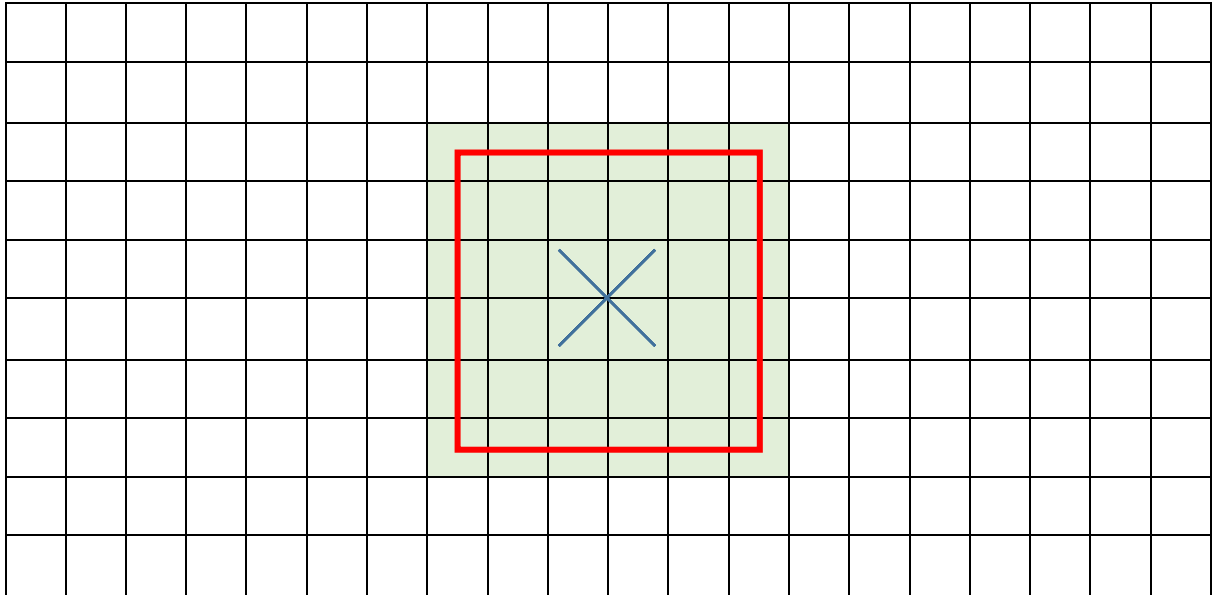
To recap, the image size is 20dpx by 10dpx, and the display window has dimensions of 10dpx by 5dpx.

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

Key: Fully Drawn, Partially Drawn
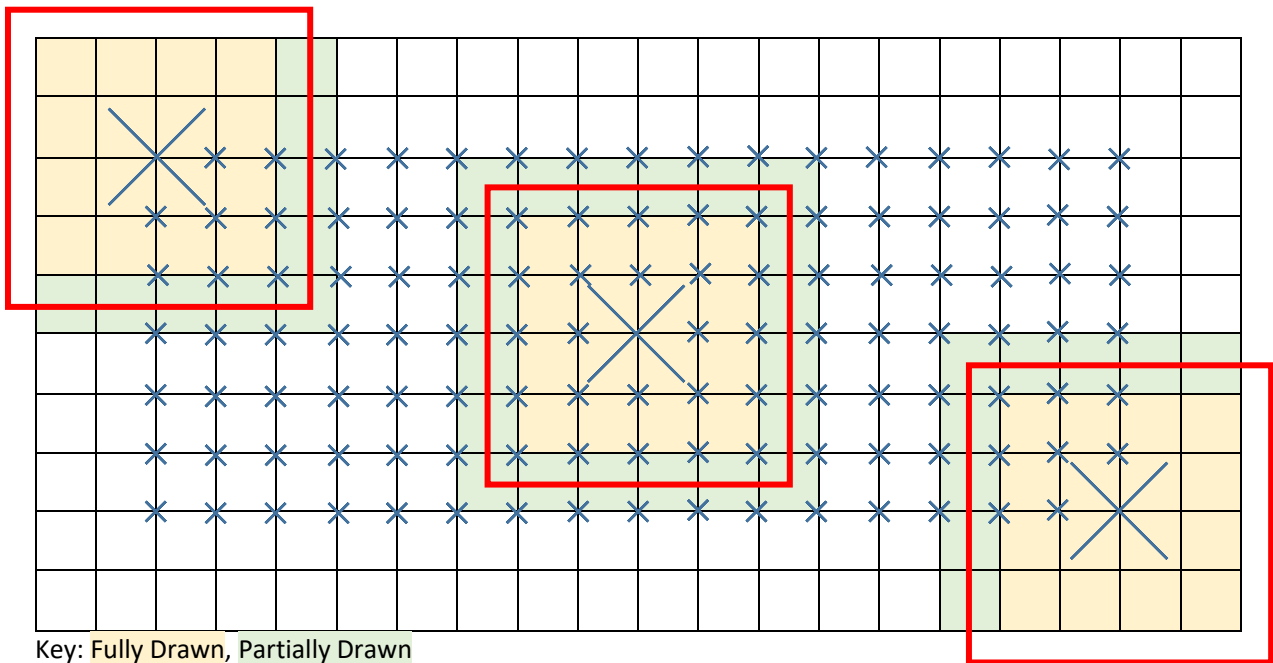
In this diagram a second shading has been introduced, yellow. This denotes fully drawn pixels. It is necessary that every pixel is fully viewable, so this yellow rectangle must be able to touch all pixels.

However, it is not necessary or useful to place the centre location any closer to the edge of the image, as it does show the user any more information. Or, to sum up:

Scrolling in a direction should only be possible if scrolling in that direction reveals pixels that were not visible before.

Thus, the crosses in the diagram denote all the possible valid locations that the centre locations can be placed.

The amount of crosses in each X axis is 17, whilst the theoretical maximum number of crosses is 21 (one on each vertical line). This means we need to calculate that 4 crosses need to be taken away. It is not necessary to know from where the crosses are taken, as they will always be taken equally from each side (in this example two crosses are taken away from each side). This means that the output of our sum must **always be a multiple of 2**.

The width of the Yellow Rectangle above will always be equal to the amount of crosses taken away, so it then becomes necessary to find the dimensions of the yellow rectangle. The algorithm for this is nearly identical to the algorithm for finding the green rectangle, except it is rounded down at the end rather than rounded up.

Pseudocode

```
DetermineCrossesInAxis(Axis axis) {

    MissingCrossesInAxis = Floor(Floor(form.getSizeInAxis(axis)/zoom)/2)*2

    MaxCrossesInAxis = image.getSizeInAxis(axis)

    return MaxCrossesInAxis – MissingCrossesInAxis

}
```

Algorithm 2.11B Upgrading scroll bar

Now that the potential amount of centre locations has been determined, there is an issue with the scroll bar's size code, as the calculations to determine its size become more complex.

Consider this small 3fpx by 2fpx image, at 1x zoom and a 2dpx by 2dpx view window with a centre location of 1fpx, 1fpx:
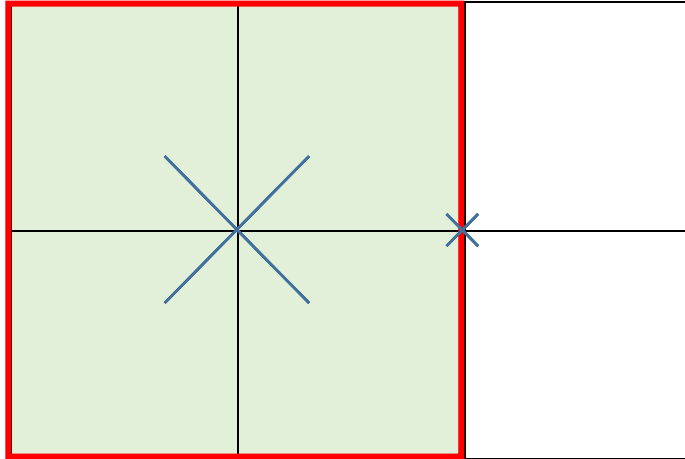
There is one other potential centre location (at 2fpx, 1fpx), and so the horizontal scroll bar should have two potential settings. However, there are multiple potential scroll bars that have two potential settings:

- &
- &
- &

Which one should be selected?

In this case, since the viewer is display $\frac{2}{3}$ of the image, the bar should take up two thirds of its scroll bar.

To calculate the width, since the scroll bar's size is designed to represent the size of the display form, they can be aligned like so:
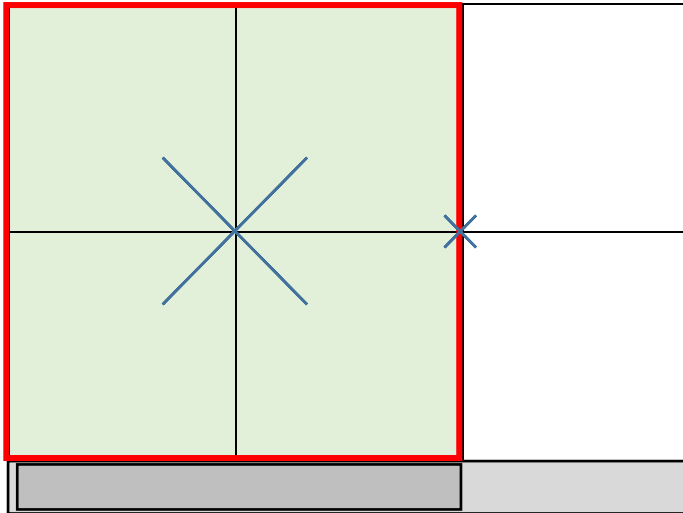
From this it becomes clear that the width of the bar must be equal to the width of the display form (which is equal to the amount of missing crosses), and have 3 potential values (where 1 is taken up by the width of the scroll bar).

Pseudocode

```
UpgradedSetupBarSize(progressBar, Axis) {

    MissingCrossesInAxis = Floor(Floor(form.getSizeInAxis(axis)/zoom)/2)*2

    progressBar.barSize = MissingCrossesInAxis

    progressBar.max = image.getFileWidth(Axis)

}
```

Algorithm 2.11C Determining Centre Location from Bar Value

The bar is now limited for how many locations it can be present in. In the above example there are only two potential positions:
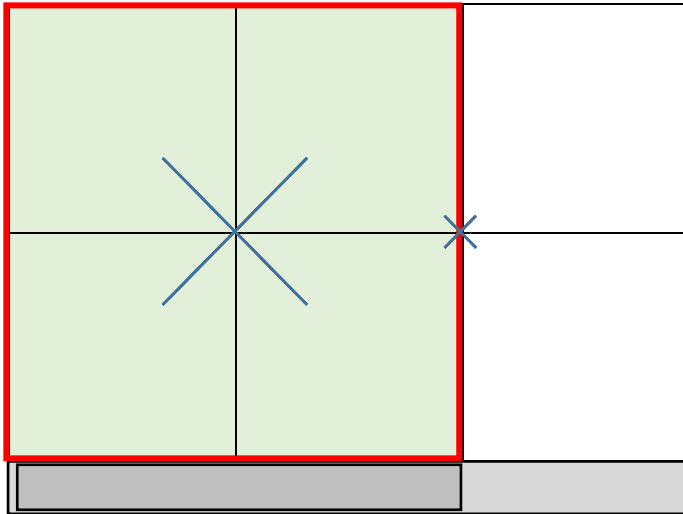
Internally the bar has a value of 0 (shown) and 1 (when moved to the right).

So to find the needed centre location in that axis, all that is needed is to add the bar's value to the location of the first cross.

Pseudocode

```
GetCentreLocationFromBar(progressBar bar, Axis axis) {

    firstCrossPosition = Floor(Floor(form.getSizeInAxis(axis)/zoom)/2)

    RETURN bar.value + firstCrossPosition
}
```

Algorithm 2.11D Determining Bar Value from Centre Location

In some scenarios (such as when originally creating or resizing the display window) it may become necessary to be able to determine what the value of the scroll bar should be from the centre location (and other settings).

To do this, we can use the inverse of the previous algorithm, where after determining the position of the first cross, we subtract it from the current centre location to find out where the bar should be.

Pseudocode

```
SetBarFromCentreLocation(progressBar bar, Axis axis) {

    firstCrossPosition = Floor(Floor(form.getSizeInAxis(axis)/zoom)/2)

    bar.value = centreLocation.getSizeInAxis(axis) - firstCrossPosition

}
```

Unit Testing

In this case, image size is 20fpx by 10fpx and zoom centre is at 10fpx, 5fpx

Test Data	Test Type	Expected Output	Description
Form is resized to half the size of image	Valid	Bar is half size of bounds and in its centre position	This tests if the bar can display normally
Form is resized to its smallest position	Extreme Valid	Bar is small but still central.	This tests if the centre location remains in its position when the window size is reduced
Form is resized back to half size of image	Valid	Bar is half size of bounds and in its centre position	This tests if the centre location remains in its position when the window size is increased
Horizontal Scroll bar is moved to far left	Extreme Valid	The far left of the image is displayed, but no more	This tests that the far left of the image can be displayed and also that it stops there
Horizontal Scroll bar is moved to far right	Extreme Valid	The far right of the image is displayed, but no more	This tests that the far right of the image can be displayed and also that it stops there
Vertical Scroll bar is moved to highest	Extreme Valid	The highest of the image is displayed, but no more	This tests that the highest of the image can be displayed and also that it stops there
Vertical Scroll bar is moved to lowest	Extreme Valid	The lowest of the image is displayed, but no more	This tests that the lowest of the image can be displayed and also that it stops there
Horizontal scroll bar is moved far right, then form size increased	Extreme Valid	Centre locations is moved to the left when needed	This tests that the centre location is moved when it no longer becomes valid due to form size increasing
Form is resized to smallest, then maximized	Extreme Valid	The bars disappear and the view is normal	This tests that the program will cope if form size is rapidly increased

Algorithm 2.12 Redrawing needed pixels

For the moment, all pixels will be redrawn. If this becomes particularly cumbersome (during development) an algorithm may be designed to determine which pixels are new. However as this algorithm may be difficult to design, and computationally expensive to execute (may end up being slower than brute-force), this will only be investigated if necessary.

Algorithm 2.13 Determining mouse position on picture box

The last task that needs to be implemented now is handling interaction with the mouse. This will be integral for any user interaction with the program.

The first task is to determine where the mouse location occurs relative to the picture box, this is because the only information that Windows provides is the mouse's location on the entire screen. It must be necessary to find the location of the display form on the screen.

This can be solved by using C#.NET's in-built Click event. This returns the location of the mouse relative to the picture box.

Demonstration Code

```csharp
void PictureBox1Click(object sender, EventArgs e)
{
    MouseEventArgs me = (MouseEventArgs)e;

    MessageBox.Show(String.Format("X: {0} Y: {1}", me.X,me.Y));
}
```

When clicking at the top left corner of the picture box results in:
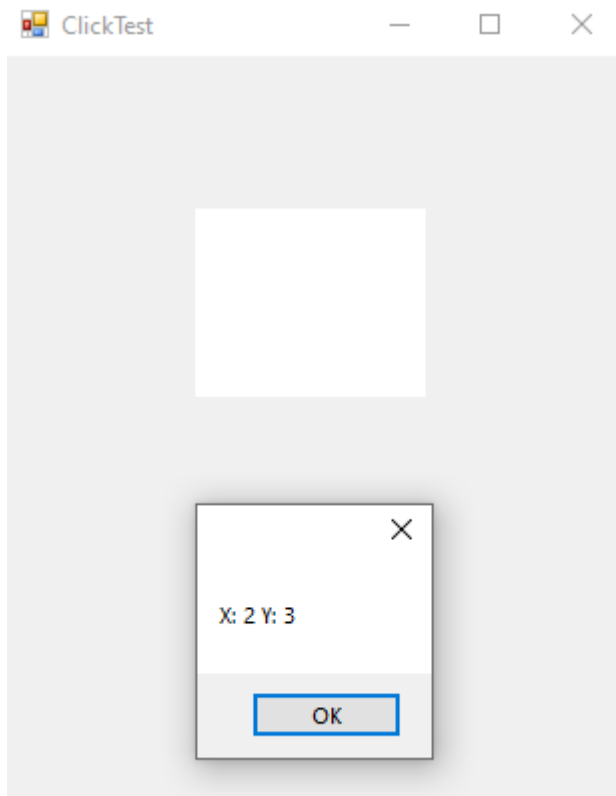
Showing that this click event will give the location of the mouse on the picture box, irrespective of where it is on the form.

This happens to also be the display pixel that the mouse resides in on the form, so there is no necessary conversion there.

Algorithm 2.14 + 2.15 Finding Location on overall image

As the location of the mouse pointer correlates to a display pixel, then finding the corresponding file pixel is easy, as an algorithm to convert from display pixels to file pixels has already been designed, so no extra design is needed here.

2.1.7 Class Design Revisited

As the algorithms have all been designed, they will need to be allocated to their respective class. Some new classes will also need to be designed as their needs have increased.

Class relation diagrams can be found [here](#).

However, for the underlying framework, two new interfaces will be introduced:

2.1.7.1 IPicturePoint

This will be the superclass of all classes that refer to a location on an image. This ensures that any point referred to in code will be able to transfer between file and display points.
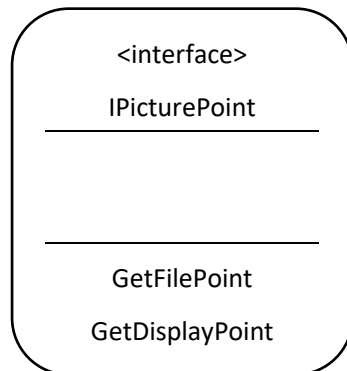
```
          <interface>

          IPicturePoint
     _____


     _____

          GetFilePoint

         GetDisplayPoint
```

Methods

Method	Params	Return type	Justification
GetFilePoint	None	FilePoint	Will return the point represented as a File Point
GetDisplay-Point	None	DisplayPoint	Will return the point represented as a Display Point

2.1.7.2 FilePoint

```
        <class>
        FilePoint
─────────────────────────
         fileX
         fileY
─────────────────────────
      *GetFilePoint
     *GetDisplayPoint
       GetFileValue
```
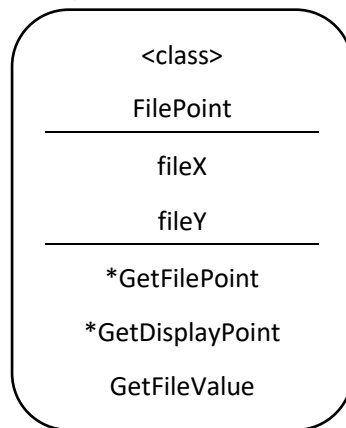
This class represents the **entity** of a file point existing on the image. It inherits from IPicturePoint so it must define its own functions for converting itself to a display point, when needed.

Properties

Property	Datatype	Justification
fileX	(private) Integer	The X position of this point in the file It is private to enforce getting the coords through GetAxisValue
fileY	(private) Integer	The Y position of this point in the file It is private to enforce getting the coords through GetAxisValue

Methods

Method	Params	Return type	Justification
GetFilePoint	None	FilePoint	Will return itself (as this is a file point)
GetDisplay-Point	None	DisplayPoint	Will convert itself into a display point
GetFile-Value	axis, Axis, the axis to get the coord in	Integer	Will return the value of the coord in the X or Y Axis

Constructor

```
Constructor (int newFileX, int newFileY) {

      fileX = newFileX

      fileY = newFileY

}
```

2.1.7.3 DisplayPoint

```
         <class>

       DisplayPoint
     _____
        displayX

        displayY
     _____
      *GetFilePoint

     *GetDisplayPoint

     GetDisplayValue
```
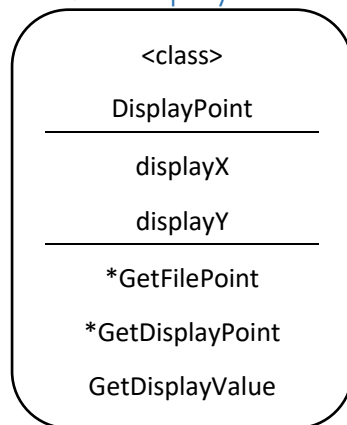
This class represents the **entity** of a display point existing on the image. It inherits from IPicturePoint so it must define its own functions for converting itself to a file point, when needed.

Properties

Property	Datatype	Justification
displayX	(private) Integer	The X position of this point on the display form. It is private to enforce getting the coords through GetDisplayValue
displayY	(private) Integer	The Y position of this point on the display form. It is private to enforce getting the coords through GetDisplayValue

Methods

Method	Params	Return type	Justification
GetFilePoint	None	FilePoint	Will convert itself into a file point
GetDisplay-Point	None	DisplayPoint	Will return itself (as this is a display point)
GetDisplay-Value	axis, Axis, the axis to get the coord in	Integer	Will return the value of the coord in the X or Y Axis

Constructor

```
Constructor (int newFileX, int newFileY) {

      fileX = newFileX

      fileY = newFileY

}
```

Why have two classes for file and display points?

As a file point and a display point are separate in this design, it makes sense to also separate them in the class design. This increases readability as a function can require or return a file point, and it makes conversions between file and display points much more explicit.

2.1.7.4 IPictureRectangle
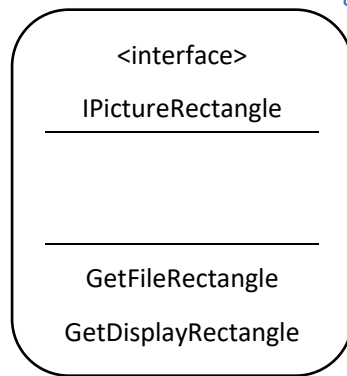
```
╭─────────────────────────────╮
│        <interface>          │
│     IPictureRectangle       │
│  ─────────────────────────  │
│                             │
│                             │
│  ─────────────────────────  │
│      GetFileRectangle       │
│    GetDisplayRectangle      │
╰─────────────────────────────╯
```

Methods

Method	Params	Return type	Justification
GetFile-Rectangle	None	FileRectangle	Will return the rectangle represented as a File Rectangle
GetDisplay-Rectangle	None	DisplayRectangle	Will return the rectangle represented as a Display Rectangle

2.1.7.5 FileRectangle

```
<class>

FileRectangle
─────────────
topLeftCorner

bottomRightCorner
─────────────
*GetFileRectangle

*GetDisplayRectangle

GetTopLeftCorner

GetBottomRightCorner

GetFileSizeInAxis
```
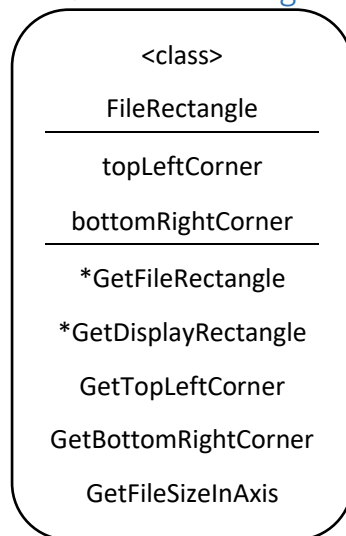
Properties

Property	Datatype	Justification
topLeft-Corner	(private) FilePoint	The location of the top left corner in the file. It is private to enforce getting the location through GetTopLeftCorner
bottomRight-Corner	(private) FilePoint	The Y position of this point in the file. It is private to enforce getting the location through GetBottomRightCorner

### Methods

Method	Params	Return type	Justification
*GetFile-Rectangle	None	FileRectangle	Will return itself as this is already a file rectangle
*GetDisplay-Rectangle	None	DisplayRectangle	Will convert itself into a display rectangle
GetTop-LeftCorner	None	FilePoint	Will return the location in the file of the top left corner
GetBottom-RightCorner	None	FilePoint	Will return the location in the file of the bottom right corner
GetFile-SizeInAxis	axis, Axis, the axis to get the coord in	Integer	Will return the size (width or height) depending on the inputted axis

Constructor

```
Constructor(fileTopLeftX, fileTopLeftY, fileBottomRightX, fileBottomRightY) {

    topLeftCorner = new FilePoint(fileTopLeftX, fileTopLeftY)

    bottomRightCorner = new FilePoint(fileBottomRightX, fileBottomRightY)

}
```

2.1.7.5 DisplayRectangle

```
        <class>

     DisplayRectangle

      topLeftCorner

    bottomRightCorner

    *GetFileRectangle

    *GetDisplayRectangle

     GetTopLeftCorner

    GetBottomRightCorner

    GetDisplaySizeInAxis
```
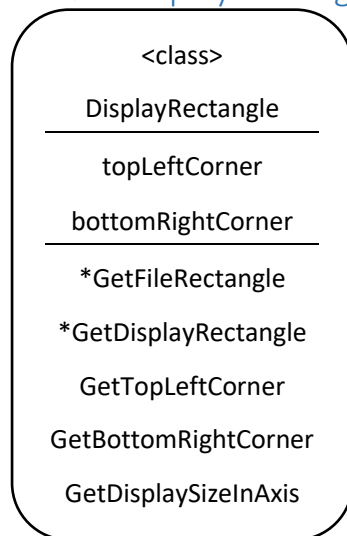
Properties

Property	Datatype	Justification
topLeft-Corner	(private) DisplayPoint	The location of the top left corner in the Display It is private to enforce getting the location through GetTopLeftCorner
bottomRight-Corner	(private) DisplayPoint	The Y position of this point in the Display It is private to enforce getting the location through GetBottomRightCorner

Methods

Method	Params	Return type	Justification
*GetFile-Rectangle	None	FileRectangle	Will convert itself into a File Rectangle
*GetDisplay-Rectangle	None	DisplayRectangle	Will return itself as this is already a Display Rectangle
GetTop-LeftCorner	None	DisplayPoint	Will return the location in the Display of the top left corner
GetBottom-RightCorner	None	DisplayPoint	Will return the location in the Display of the bottom right corner
GetDisplay-SizeInAxis	axis, Axis, the axis to get the coord in	Integer	Will return the size (width or height) depending on the inputted axis

Constructor

```
Constructor(DisplayTopLeftX, DisplayTopLeftY, DisplayBottomRightX,
DisplayBottomRightY) {

     topLeftCorner = new DisplayPoint(DisplayTopLeftX, DisplayTopLeftY)

     bottomRightCorner = new DisplayPoint(DisplayBottomRightX,
DisplayBottomRightY)

}
```

2.1.7.6 Image

```
┌─────────────────────────────┐
│          <class>            │
│                             │
│           Image             │
│  ─────────────────────────  │
│         colours[,]          │
│                             │
│           width             │
│                             │
│           height            │
│                             │
│        zoomSetting          │
│                             │
│      attachedWorkspace      │
│  ─────────────────────────  │
│         GetPixel()          │
│                             │
│         SetPixel()          │
│                             │
│     ToDisplayRectangle()*   │
│                             │
│      ToFileRectangle()*     │
└─────────────────────────────┘
```

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	Stores the current settings describing the zoom of the image.
attached-Workspace	Workspace	The workspace in which this image exists.

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 set	None	Will set a new colour onto the grid, similar to GetPixel.
ToDisplay-Rectangle()	None	DisplayRectangle	Will return the Display size of the image
ToFile-Rectangle()	None	FileRectangle	Will return the File size of the image

2.1.7.7 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	FilePoint	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 {

      FilePoint centreLocation

      Integer zoomAmount

      constructor(_centreLocation) {

            centreLocation = _centreLocation
            zoomAmount = 1

      }

}
```

The default zoom will be none - 1x.

2.1.7.8 Workspace

```
        <class>

        Workspace
    ─────────────────
      displayPictureBox

       attachedForm

        displayWidth

        displayHeight

        LEFT_PADDING

       RIGHT_PADDING

        TOP_PADDING

      BOTTOM_PADDING
    ─────────────────
         GetPixel

         SetPixel

     GetDisplaySizeInAxis

     GetDisplayRectangle

         Display

        HandleResize
```
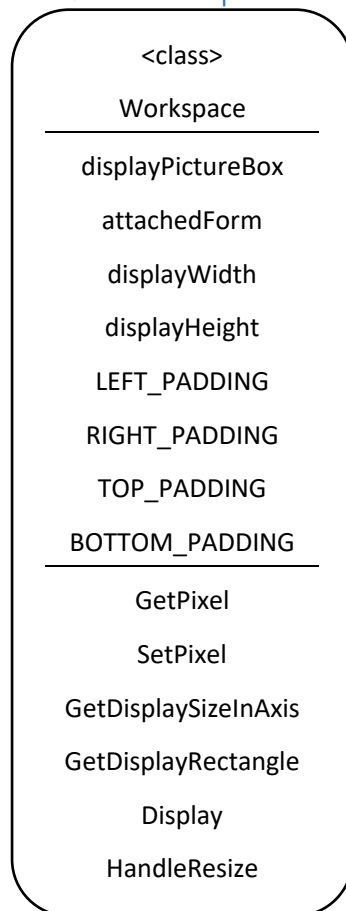
Properties

Property	Datatype	Justification
Display-PictureBox	PictureBox	The .NET Picture Box that this workspace will use to display itself into.
attachedForm	Form	The form that this workspace is attached to. This will also provide access to the form controls.
displayWidth	Integer	The current width of pixels that can be used for displaying in. Will be updated each time the form changes size
displayHeight	Integer	The current height of pixels that can be used for displaying in. Will be updated each time the form changes size
LEFT_PADDING	(constant) Integer	The amount of padding on the left hand side. Used for calculations involving where to place the picture box
RIGHT_PADDING	(constant) Integer	The amount of padding on the right hand side
TOP_PADDING	(constant) Integer	The amount of padding on the top side
BOTTOM_PADDING	(constant) Integer	The amount of padding on the bottom side

Methods

Method	Params	Return type	Justification
GetPixel()	X, int, X coord of pixel to get Y, int, Y coord of pixel to get	Colour	A mirror of Image's GetPixel, also included here for convenience
SetPixel()	X, int, X coord of pixel to get Y, int, Y coord of pixel to get Colour, Color, colour of pixel to set	None	A mirror of Image's SetPixel, also included here for convenience
GetDisplay-SizeInAxis	None	Integer	Returns the amount of display pixels in a certain axis
GetDisplay-Rectangle	None	DisplayRectangle	Returns a Display Rectangle that represents the entire display area.
Display	None	None	Calls the code to display the image
Handle-Resize	None	None	Called when the form resizes, move all of the display components into the correct positions.

2.1.7.9 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
```
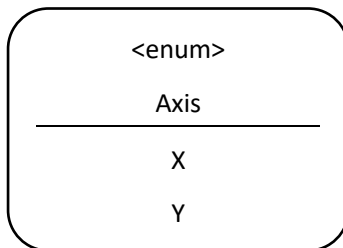
Members

Member	Justification
X	Denotes that the X axis has been selected.
Y	Denotes that the Y axis has been selected.
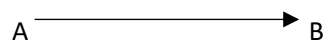
2.1.8 Class Relation Diagram

This demonstrates how the classes described in 2.1.7 will interact with each other to provide a full **abstraction stack** to make image editing easy later on.

2.1.8.1 Point abstraction stack

```
              +-----------------------+
              |     <interface>       |
              |                       |
              |     IPicturePoint     |
              |  _____  |
              |                       |
              |  _____  |
              |                       |
              |     GetFilePoint      |
              |    GetDisplayPoint    |
              +-----------------------+
                    ^          ^
                   /            \
                  /              \
  +-----------------------+   +-----------------------+
  |       <class>         |   |       <class>         |
  |                       |   |                       |
  |       FilePoint       |   |     DisplayPoint      |
  |  _____  |   |  _____  |
  |        fileX          |   |       displayX        |
  |        fileY          |   |       displayY        |
  |  _____  |   |  _____  |
  |     GetFilePoint      |   |     GetFilePoint      |
  |    GetDisplayPoint    |   |    GetDisplayPoint    |
  |     GetFileValue      |   |    GetDisplayValue    |
  +-----------------------+   +-----------------------+
```

Key:
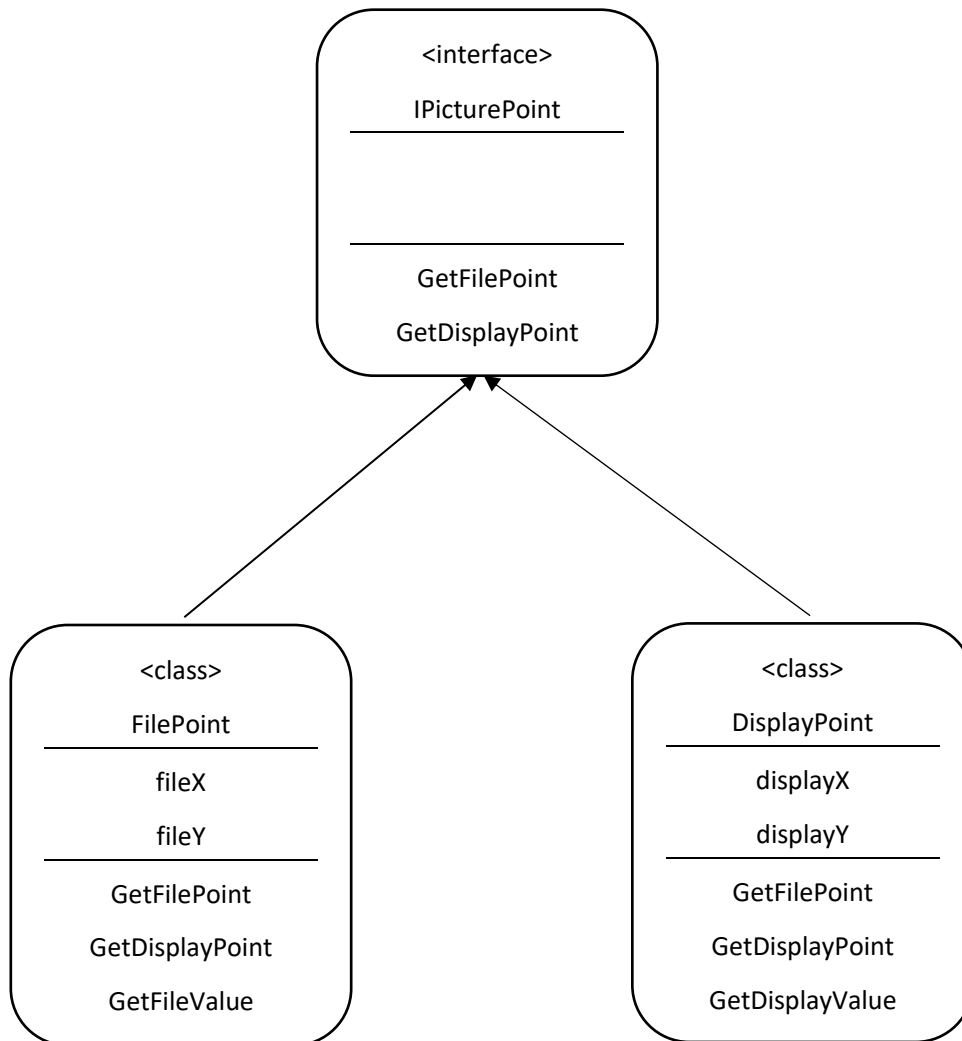
A ————————————▶ B
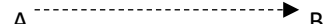
A inherits from B

2.1.8.2 Rectangle abstraction stack

```
            ┌──────────────────────────┐
            │       <interface>        │
            │    IPictureRectangle     │
            │ ──────────────────────── │
            │                          │
            │ ──────────────────────── │
            │    GetFileRectangle      │
            │   GetDisplayRectangle    │
            └──────────────────────────┘
```

```
┌──────────────────────┐                    ┌──────────────────────┐
│       <class>        │                    │       <class>        │
│      FilePoint       │                    │     DisplayPoint     │
│ ──────────────────── │                    │ ──────────────────── │
│        fileX         │                    │       displayX       │
│        fileY         │                    │       displayY       │
│ ──────────────────── │                    │ ──────────────────── │
│     GetFilePoint     │                    │     GetFilePoint     │
│   GetDisplayPoint    │                    │   GetDisplayPoint    │
│     GetFileValue     │                    │   GetDisplayValue    │
└──────────────────────┘                    └──────────────────────┘
```

```
┌──────────────────────┐                    ┌──────────────────────┐
│       <class>        │                    │       <class>        │
│    FileRectangle     │                    │   DisplayRectangle   │
│ ──────────────────── │                    │ ──────────────────── │
│     topLeftCorner    │                    │     topLeftCorner    │
│   bottomRightCorner  │                    │   bottomRightCorner  │
│ ──────────────────── │                    │ ──────────────────── │
│   *GetFileRectangle  │                    │   *GetFileRectangle  │
│ *GetDisplayRectangle │                    │ *GetDisplayRectangle │
│   GetTopLeftCorner   │                    │   GetTopLeftCorner   │
│ GetBottomRightCorner │                    │ GetBottomRightCorner │
│   GetFileSizeInAxis  │                    │ GetDisplaySizeInAxis │
└──────────────────────┘                    └──────────────────────┘
```

Key:

A ───────────────────► B

A inherits from B

A ─ ─ ─ ─ ─ ─ ─ ─ ─ ─► B

A contains references to B

2.1.8.3 Overall stack

<class>

Image

---

colours[,]

width

height

zoomSetting

attachedWorkspace

GetPixel()

SetPixel()

<class>

ZoomSettings

---

centreLocation

zoomAmount

<interface>

IPictureRectangle

GetFileRectangle

GetDisplayRectangle

<class>

Workspace

---

displayPictureBox

attachedForm

displayWidth

displayHeight

LEFT_PADDING

RIGHT_PADDING

TOP_PADDING

BOTTOM_PADDING

GetPixel

SetPixel

GetDisplaySizeInAxis

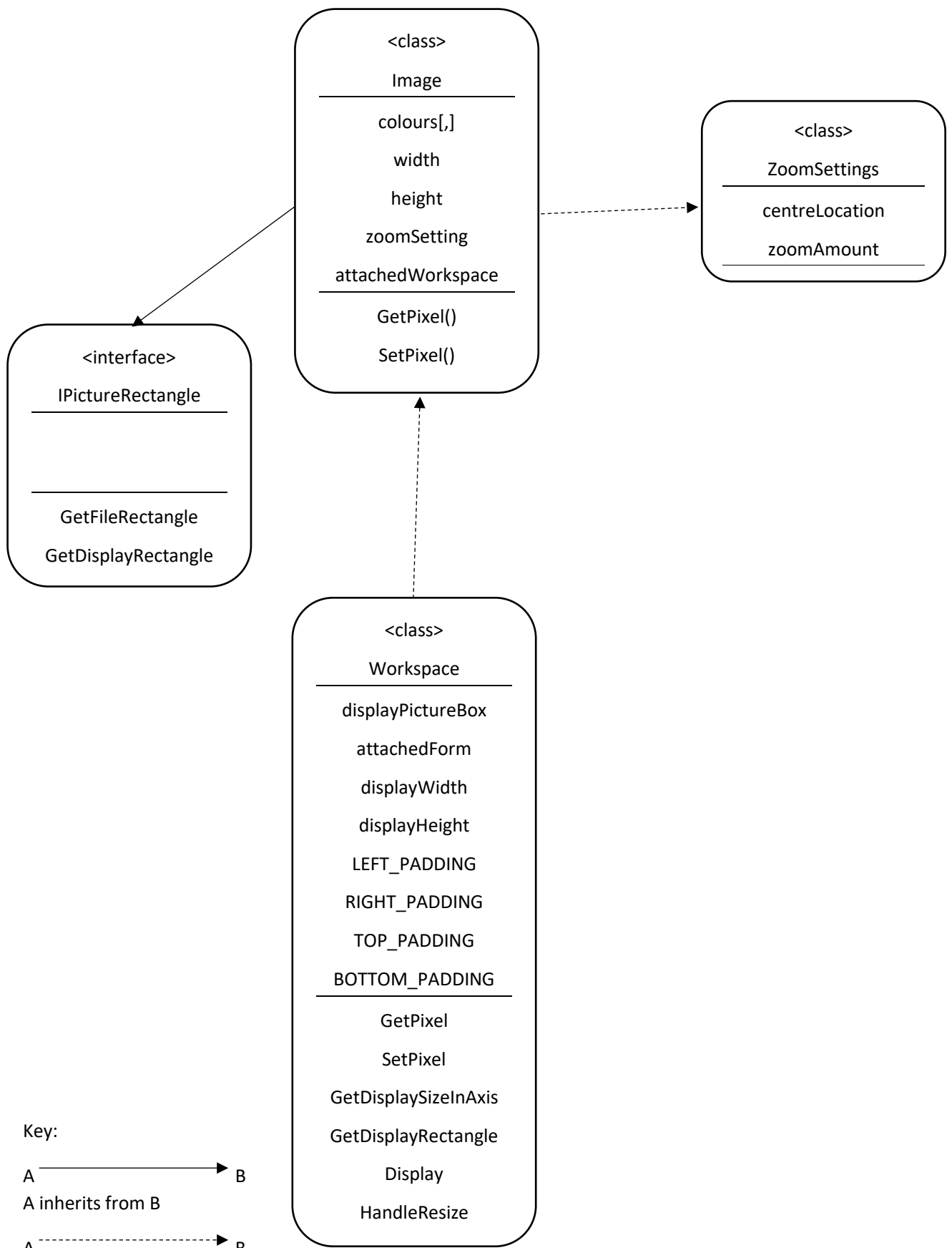GetDisplayRectangle

Display

HandleResize

Key:

A ⎯⎯⎯⎯⎯⎯→ B

A inherits from B

A ----------→ B

A contains references to B

2.1.9 Key Data Structures

The MainForm will contain one data structure. A list of workspaces.

Property	Datatype	Justification
open-Workspaces	List of Workspaces	This stores any current workspaces the program has open.

This means that each instance of the WorkSpace class will **encapsulate** an entire window. This means that there can be multiple distinct instances of a program open at once.

2.1.20 Full Section Testing

This is a repeat of the unit tests from before, but tested again to ensure cooperation between the modules

Test Data	Test Type	Expected Output	Description	ID
new Image(10,10)	Valid	An image of size 10fpx, 10fpx	This tests if a normal image can be created	1
new Image(10,6)	Valid	A 10fpx, 5fpx image	This tests if a non-rectangular image can be created	2
new Image(10,5)	Valid	A 10fpx, 2fpx image	This tests when an odd dimension is entered	3
new Image(10,1)	Extreme Valid	A 10fpx, 1fpx image	This tests if a very short image can be created	4
new Image(1,10)	Extreme Valid	A 1fpx, 10fpx image	This tests if a very thin image can be created	5
new Image(1,1)	Extreme Valid	A 1fpx, 1fpx image	This tests when a very small image is created	6
new Image(10,0)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no height is rejected	7
new Image(0,10)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no width is rejected	8
new Image(0,0)	Extreme Invalid	The parameters are rejected and no image is created	This tests if an image with no width or height is rejected	9
new Image(10000,10)	Extreme Invalid	The parameters are rejected and no image is created	This tests if a width which is much too large is rejected	10
new Image(10,10000)	Extreme Invalid	The parameters are rejected and no image is created	This tests if a height which is much too large is rejected	11
new Image(-1,10)	Invalid	The parameters are rejected and no image is created	This tests if a negative width is rejected.	12
new Image(10,-1)	Invalid	The parameters are rejected and no image is created	This tests if a negative height is rejected.	13
new Image(10)	Erraneous	The parameters are rejected and no image is created	This tests if an image missing a height is rejected.	14
new Image("10","10")	Erraneous	The parameters are rejected and no image is created	This tests if an image with invalid numbers is rejected	15
SetPixel(5,5,Black)	Valid	A pixel near the middle of the image is set to black	This tests if a normal pixel can be set	16
SetPixel(5,5,Green)	Valid	A pixel near the middle of the image is set to yellow	This tests if a pixel can be set to other colours than black	17
SetPixel(0,0,Black)	Extreme Valid	A pixel in the top-left corner is set to black	This tests if the top-left corner can be set	18

SetPixel(9,9,Black)	Extreme Valid	A pixel in the bottom-right corner is set to black	This tests if the bottom-right corner can be set	19
SetPixel(-1,0,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far left is rejected	20
SetPixel(0,-1,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far up is rejected	21
SetPixel(10,0,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far right is rejected	22
SetPixel(0,10,Black)	Extreme Invalid	No pixel is set as it is out of bounds	This tests if a pixel too far down is rejected	23
Form is started at normal size	Valid	Image Displays in the middle of the form	This tests that the program can start and display the image	24
Form is maximized	Extreme Valid	Image displays in the middle of the large form	This tests if the image is displayed when the form is very large	25
Form is minimized	Extreme Valid	No image is displayed (as form is currently invisible)	This tests that the program can be minimized safely	26
Form is resized to smallest possible	Extreme Valid	The form cannot be made smaller than the image	This tests that the form can never be made smaller than the image	27
Form is resized to minimum width maximum height	Extreme Valid	A very thin form displays the image	This tests if a form with an extreme height but minimum width is accepted	28
Form is resized to minimum height maximum width	Extreme Valid	A very short form displays the image	This tests if a form with an extreme width but minimum height is accepted	29
The form's size is rapidly changed.	Extreme Valid	The image is very quickly moved around but remains centred	This tests that under a stress test the image is displayed correctly	30
Form is resized to half the size of image	Valid	Bar is half size of bounds and in its centre position	This tests if the bar can display normally	31
Form is resized to its smallest position	Extreme Valid	Bar is small but still central.	This tests if the centre location remains in its position when the window size is reduced	32
Form is resized back to half size of image	Valid	Bar is half size of bounds and in its centre position	This tests if the centre location remains in its position when the window size is increased	33
Horizontal Scroll bar is moved to far left	Extreme Valid	The far left of the image is displayed, but no more	This tests that the far left of the image can be displayed and also that it stops there	34
Horizontal Scroll bar is moved to far right	Extreme Valid	The far right of the image is displayed, but no more	This tests that the far right of the image can be displayed and also that it stops there	35

Vertical Scroll bar is moved to highest	Extreme Valid	The highest of the image is displayed, but no more	This tests that the highest of the image can be displayed and also that it stops there	36
Vertical Scroll bar is moved to lowest	Extreme Valid	The lowest of the image is displayed, but no more	This tests that the lowest of the image can be displayed and also that it stops there	37
Horizontal scroll bar is moved far right, then form size increased	Extreme Valid	Centre locations is moved to the left when needed	This tests that the centre location is moved when it no longer becomes valid due to form size increasing	38
Form is resized to smallest, then maximized	Extreme Valid	The bars disappear and the view is normal	This tests that the program will cope if form size is rapidly increased	39

2.1.21 Testing Plan

The unit tests will be completed as described by the documentation in the order that they are described. If any test in the unit is failed, then the error(s) will be rectified and then the entire unit will be tested again. This is to ensure no chance of a 'domino effect' of errors, where fixing one may trigger another. This approach of unit testing will be **iteratively applied** until the entire unit is passed, at which point design will start on the next unit.

During development, there will also be the practice of self-testing and diagnosing. I will note changes to code (designed here or not) and the reason for their changing in a **development diary**.

The testing for the program will be recorded in the following table

Test	ID	Expected Result	Actual Result	Comment
The data inputted into the program	0	What the program should output if it is working correctly	What the program outputs, which may differ from Expected	A comment on the performance of the test, ad why it might have failed