Section 3 – Brushes & Tools

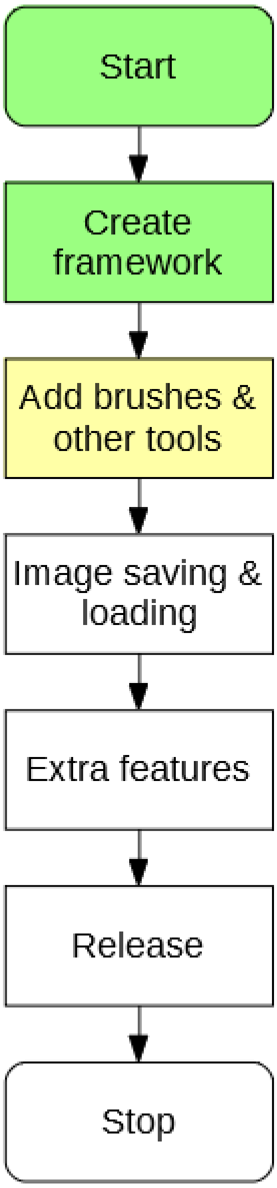
General Contents

They will go here

Contents

3.1 Design

The brushes and tools section will contain many of the basic image editing tools, based upon the framework made in the previous section.



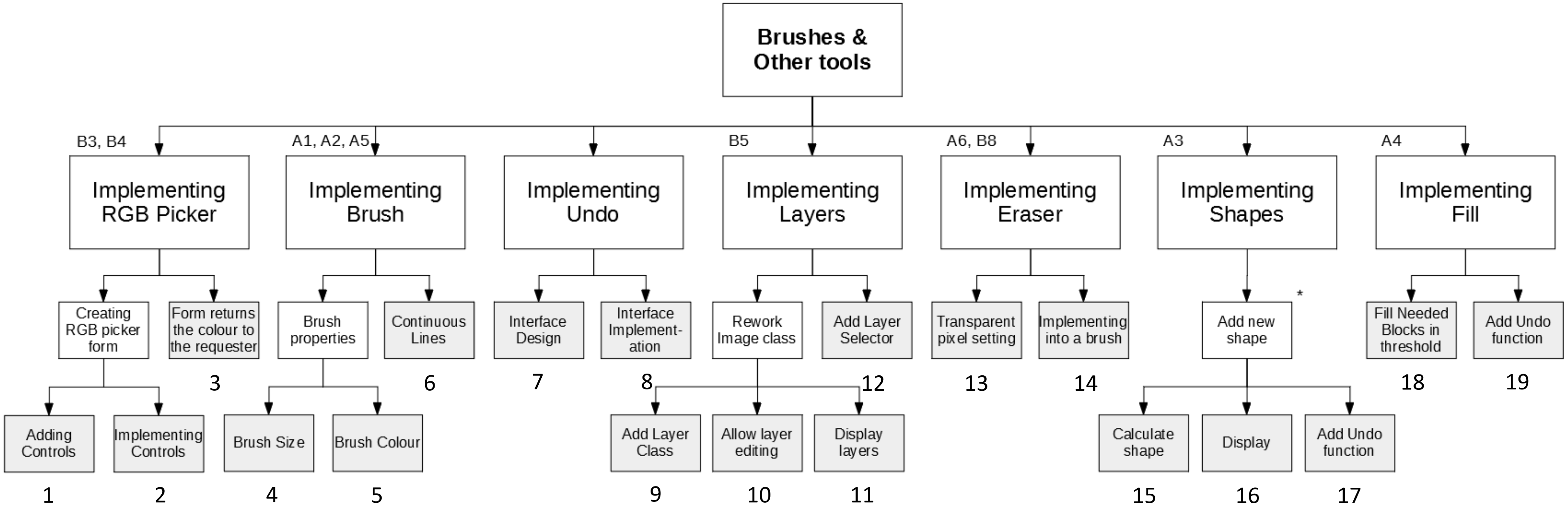
3.1.1 Success Criteria Fulfilment Plan

In this section the following success criteria are planned to be 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		
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
Transparent pixels	Screenshot showing a layer with blank pixels (one layer on top of another). Partial transparency is not required	B8

This should leave a reasonably functional (though not complete) editing program by the end of the section.

3.1.2 Section Decomposition



Each lowest level block has a label, which will become an algorithm designed in this document.

These algorithms will then be combined together.

3.1.3 Class Design

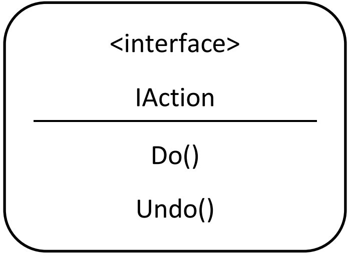
The necessary classes for this section are:

- IAction (for undo)
- PixelAction
- ITool
- IToolProperty
- NumericalProperty
- ColourProperty
- LineTool
- Layer

3.1.3.1 IAction

The IAction interface will encapsulate a generic action in the program. This enforces that every action in the program must have the possibility to be both done and undone.

Class Diagram



Methods

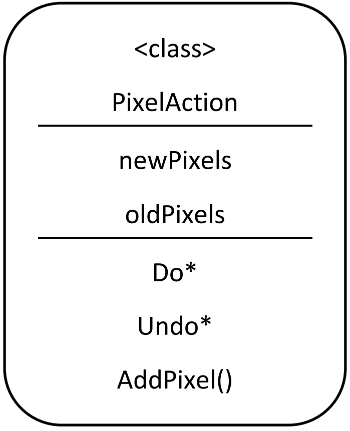
Method	Params	Return type	Justification
Do()	Workspace workspace	None	Will do the action implemented by the action on the workspace
Undo()	Workspace workspace	None	Will undo the action implemented by the action on the workspace

3.1.3.2 PixelAction

Implements IAction

Will encapsulate the concept of an action applied to the pixels of the image.

Class Diagram



<class>

PixelAction

newPixels

oldPixels

Do*

Undo*

AddPixel()

Properties

Property	Datatype	Justification
newPixels	Dictionary of FilePoint to Pixels	Stores a record of all the changes made to pixels
oldPixels	Dictionary of FilePoint to Pixels	Stores a record of what the pixels used to look like

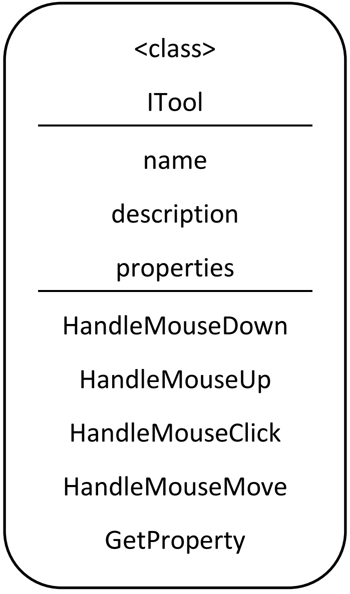
Methods

Method	Params	Return type	Justification
Do()	Workspace workspace	None	Inherited from IAction
Undo()	Workspace workspace	None	Inherited from IAction
AddPixel	FilePoint pixelLocation, Colour oldColour, Colour newColour	None	Adds a change of pixel to the record

3.1.3.3 ITool

The ITool interface will implement the generic idea of a tool. This will be used by the tool checking code to decide what tool is active.

Class Diagram



<class>

ITool

name

description

properties

HandleMouseDown

HandleMouseUp

HandleMouseClick

HandleMouseMove

GetProperty

Properties

Property	Datatype	Justification
name	String	Stores the name of the tool to be displayed
description	String	Stores the description of the tool to be displayed
properties	List of ToolProperties	Stores the properties of that tool

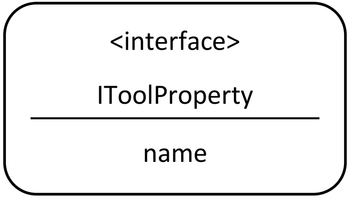
Methods

Method	Params	Return type	Justification
HandleMouseDown	FilePoint clickLocation, MouseButton button	None	Will run code when the mouse is pressed down
HandleMouseUp	FilePoint clickLocation, MouseButton button	None	Will run code when the mouse is released
HandleMouseClick	FilePoint clickLocation, MouseButton button	None	Will run code when the mouse is clicked
HandleMouseMove	FilePoint oldLocation, FilePoint newLocation	None	Will run code when the mouse is moved from one position to another
GetProperty	String propertyName	ToolProperty	Will return the specific property when asked for its name

3.1.3.4 IToolProperty

The IToolProperty encapsulates a generic property that a tool may have, and can be further defined by specific implementations of the class. This means that there will be a common format for all tools.

Class Diagram



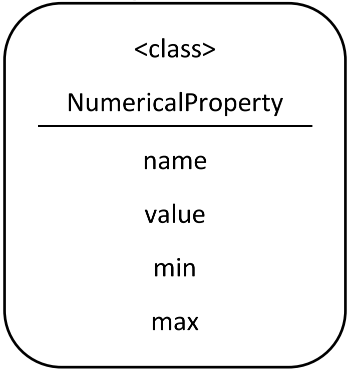
Properties

Property	Datatype	Justification
name	String	Stores the name of the property of the tool (e.g. size)

3.1.3.5 NumericalProperty

The numerical property encapsulates a property that is represented by a number (e.g. thickness).

Class Diagram



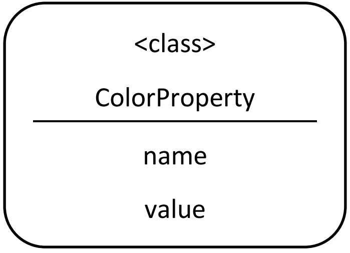
Properties

Property	Datatype	Justification
name	String	Stores the name of the property of the tool (e.g. size)
value	Integer	Stores the value of the property
min	Integer	Stores the minimum value of the property
max	Integer	Stores the maximum value of the property

3.1.3.6 ColorProperty

The numerical property encapsulates a property that is represented by a color (e.g. brush colour).

Class Diagram

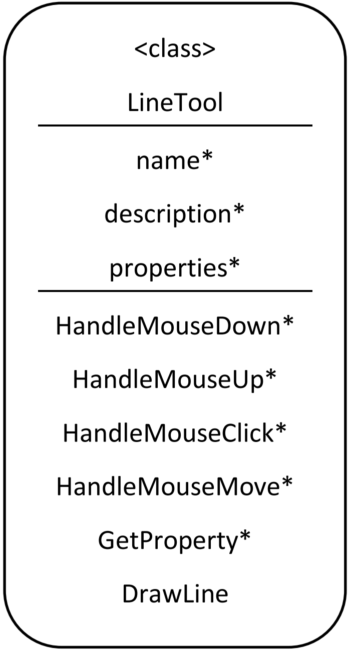


Properties

Property	**Datatype**	**Justification**
name	String	Stores the name of the property of the tool (e.g. size)
value	Color	Stores the value of the property

3.1.3.7 LineTool

Class Diagram



Properties

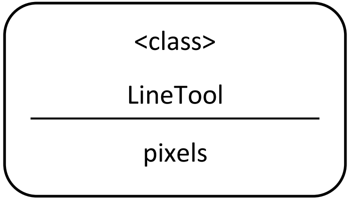
Property	**Datatype**	**Justification**
name	String	Inherited from ITool
description	String	Inherited from ITool
properties	List of ToolProperties	Inherited from ITool

Methods

Method	**Params**	**Return type**	**Justification**
HandleMouseDown	FilePoint clickLocation, MouseButton button	None	Inherited from ITool
HandleMouseUp	FilePoint clickLocation, MouseButton button	None	Inherited from ITool
HandleMouseClick	FilePoint clickLocation, MouseButton button	None	Inherited from ITool
HandleMouseMove	FilePoint oldLocation, FilePoint newLocation	None	Inherited from ITool
GetProperty	String propertyName	ToolProperty	Inherited from ITool
DrawLine	None	None	Draws the line when required

3.1.3.8 Layer

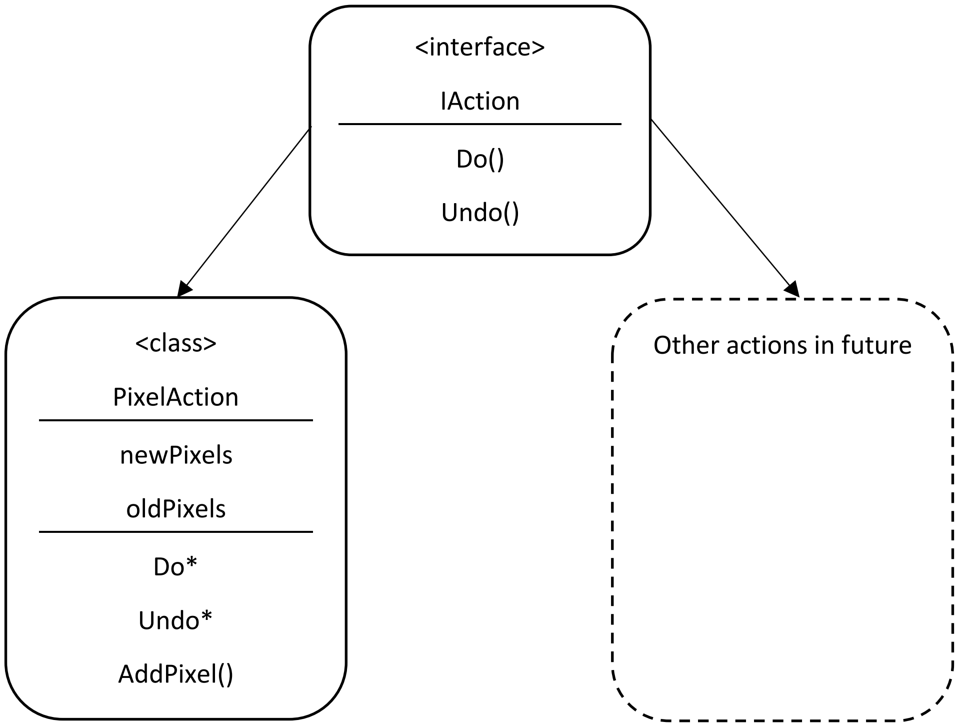
Class Diagram

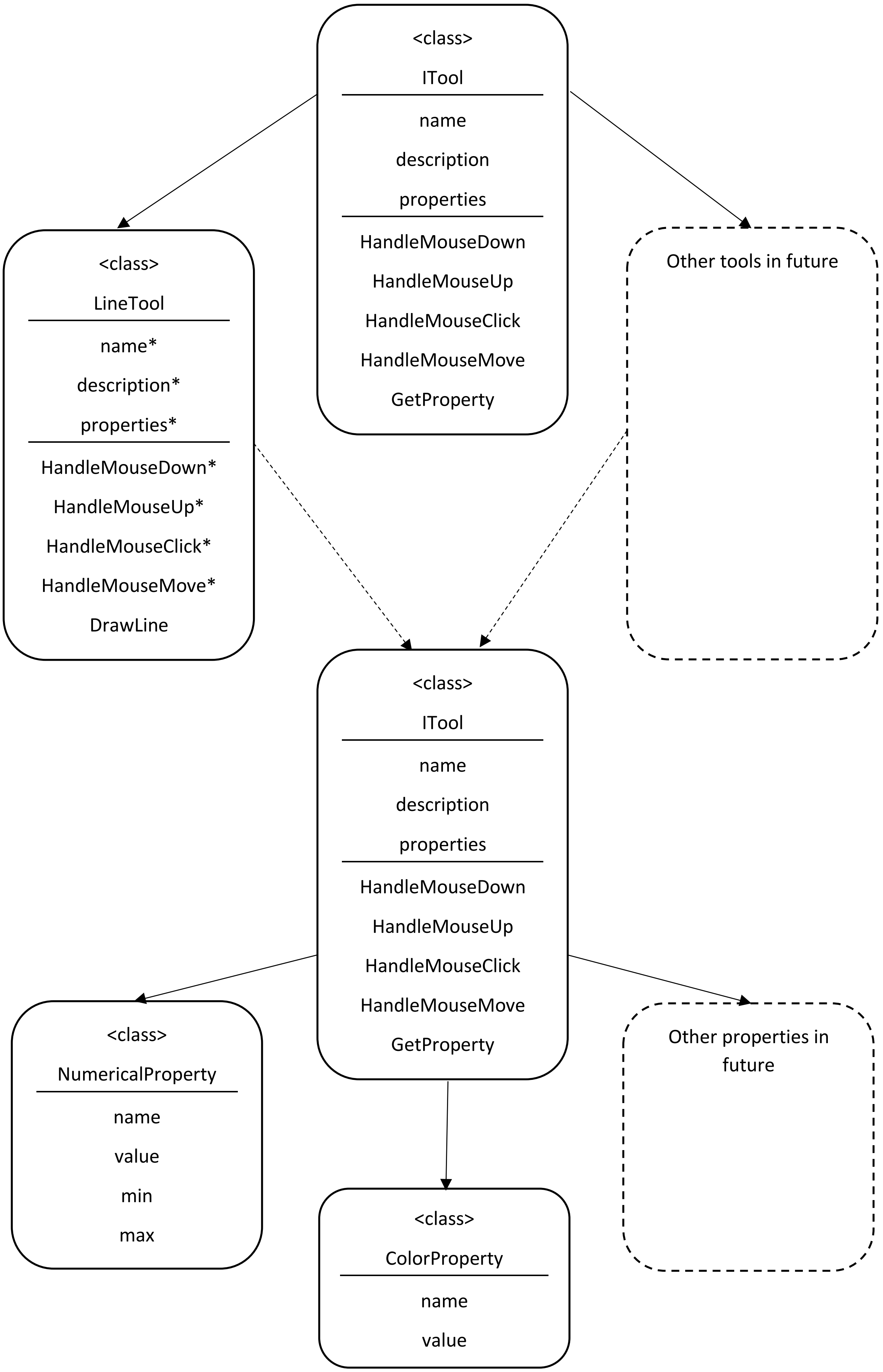


Properties

Property	Datatype	Justification
pixels	Array of Colours	Stores the colours needed in this class

3.1.4 Class Relation Diagram

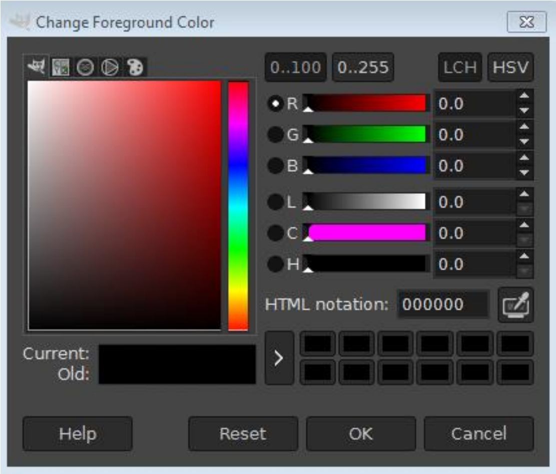



<class>
ITool
name
description
properties
HandleMouseDown
HandleMouseUp
HandleMouseClick
HandleMouseMove
GetProperty
<class>
LineTool
name*
description*
properties*
HandleMouseDown*
HandleMouseUp*
HandleMouseClick*
HandleMouseMove*
DrawLine
Other tools in future
<class>
ITool
name
description
properties
HandleMouseDown
HandleMouseUp
HandleMouseClick
HandleMouseMove
GetProperty
<class>
NumericalProperty
name
value
min
max
<class>
ColorProperty
name
value
Other properties in future

3.1.5 Algorithm Design

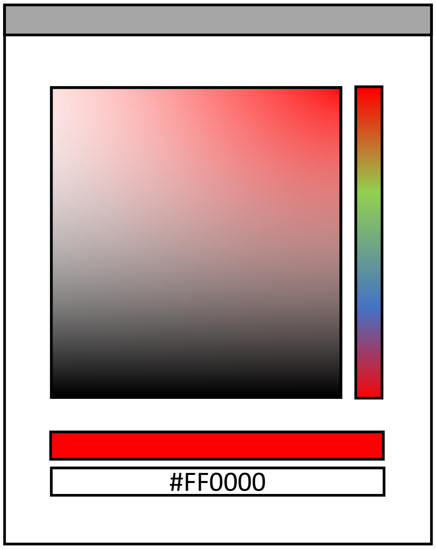
Algorithm 3.1 Form Controls

The algorithm form controls will be, from my analysis, inspired by the GIMP colour picker:



R 0.0
G 0.0
B 0.0
L 0.0
C 0.0
H 0.0
HTML notation: 000000

Initial Design

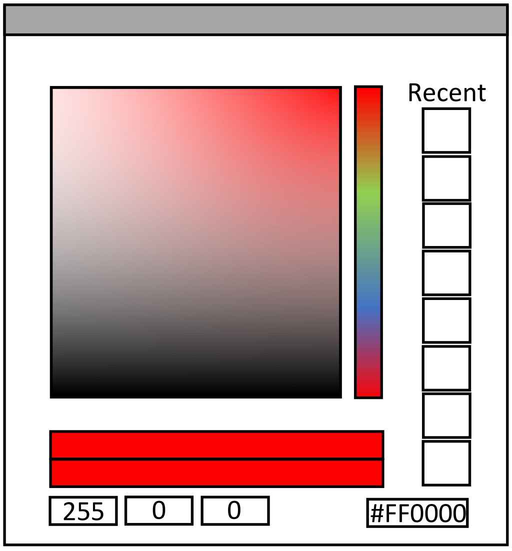


Stakeholder feedback

I showed the initial design to my stakeholders, and they had the following changes to be made:

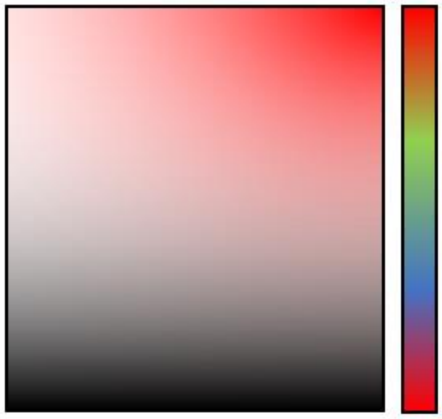
- “An option to see your old colour would be good”
- “If there were separate editable boxes for the R, G and B values that would be nice”
- “Is the box for the code editable? It should be”
- “There should be a place that stores your recently used colours”

Updated Design

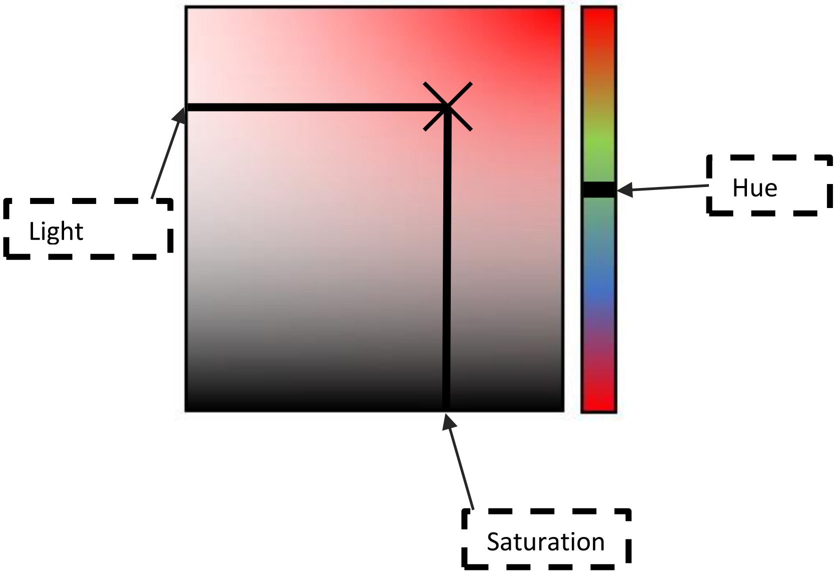


Algorithm 3.2 Implementing Controls

3.2A Displaying Current Colour

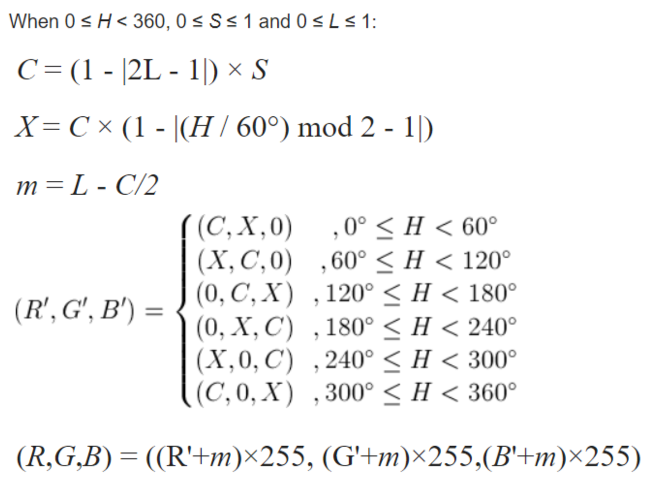


The colour space can be designed using the HSL colour format. This is where colour is split into Hue (H), Saturation (S) and Light (L). On the diagram these 3 colours can be represented like so:



3.2B Converting HSL to RGB

The algorithm for converting between HSL and RGB is



When $0 \le H < 360$, $0 \le S \le 1$ and $0 \le L \le 1$:

$$C = (1 - |2L - 1|) \times S$$

$$X = C \times (1 - |(H / 60°) \bmod 2 - 1|)$$

$$m = L - C/2$$

$$(R', G', B') = \begin{cases} (C, X, 0) & , 0° \leq H < 60° \\ (X, C, 0) & , 60° \leq H < 120° \\ (0, C, X) & , 120° \leq H < 180° \\ (0, X, C) & , 180° \leq H < 240° \\ (X, 0, C) & , 240° \leq H < 300° \\ (C, 0, X) & , 300° \leq H < 360° \end{cases}$$

$$(R,G,B) = ((R'+m)\times 255, (G'+m)\times 255,(B'+m)\times 255)$$

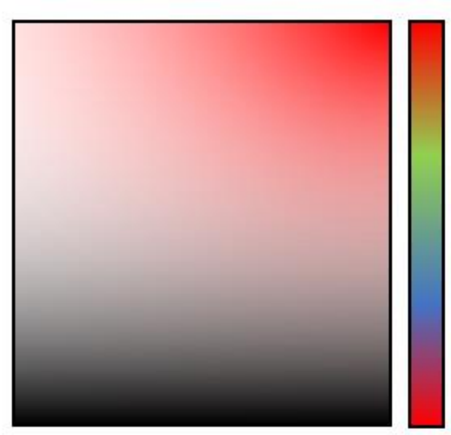
Pseudocode

Implementing the above diagram as pseudocode gives:

```
HSLtoRGB(H, S, L) {
        C = (1 – Abs((2 * L) – 1)) * S
        X = C * (1 – Abs(((H / 60) % 2) -1))
        M = L – C/2
        IF H < 60 THEN
                r = C, g = X, b = 0
        ELSE IF H >= 60 && H < 120 THEN
                r = X, g = C, b = 0
        ELSE IF H >= 120 && H < 180 THEN
                r = 0, g = C, b = X
        ELSE IF H >= 180 && H < 240 THEN
                r = 0, g = X, b = C
        ELSE IF H >= 240 && H < 300 THEN
                r = X, g = 0, b = C
        ELSE IF H >= 300 && H < 360 THEN
                r = C, g = 0, b = X
        END IF
        R = (r + m) * 255
        G = (g + m) * 255
        B = (b + m) * 255
        RETURN new Colour(R,G,B)
}
```

3.2C Generating the colour square

Now that HSL can be converted to RGB, the colour square can be generated reasonably simply:



Pseudocode

```
FOR x = 0 to 100
      FOR y = 0 to 100
            currentColour = HSLtoRGB(hue, x/100, y/100)
            setPixel(x,y,currentColour)
      NEXT
NEXT
```

The other controls are much more straightforward in design, so there is no merit in designing them here.

Unit Test

Test	*ID*	*Expected Result*	*Comment*
Set Hue to Red, click top-left corner	1	White	This tests that the top left corner is functional
Click top-right corner	2	Red	This tests that the top left corner is functional
Click bottom-left corner	3	Black	This tests that the bottom left corner is functional
Click bottom-right corner	4	Black	This tests that the bottom right corner is functional
Click three random positions on the colour square	5	Correct corresponding colour outputs	This tests that other areas on the colour square are valid
Click three random positions on the hue slider	6	Correct corresponding colour square, and output also changes	This tests that the hue slider can be changed and the rest of the form updates
Set RGB values to 255,0,0	7	Red, hue adjusts appropirately	This tests that a standard colour can be implemented
Set RGB values to 255,255,255	8	White	This tests that a maximum brightness colour can be implemented

Set RGB values to 0,0,0	9	Black	This tests that the darkest colour can be entered
Set RGB values to -1,0,0	10	Not accepted, round -1 to 0	This tests that an incorrect value is adjusted up appropriately
Set RGB values to 256,255,255	11	Not accepted, round 256 to 255	This tests that an incorrect value is adjusted down appropriately
Set RGB values to 255,255	12	Not accepted, missing value substituted for 0	This tests that if a value is left out, it will be subsituted

Algorithm 3.3 Returning Colour

This will be implemented using the knowledge that in C# all classes (including colour) are passed by reference.

Pseudocode

So in the workspace form, the colour designer can be called by:

```
RGBPicker newPicker = new RBGPicker(colortochange)
```

RGBPicker's constructor will look like so:

```
RGBPicker(color) {
      storedColour = color
      show form
}
```

Then RGBPicker will return the colour by setting storedColour to the result.

Algorithm 3.4 Brush Size

3.4A Storing Current Brush

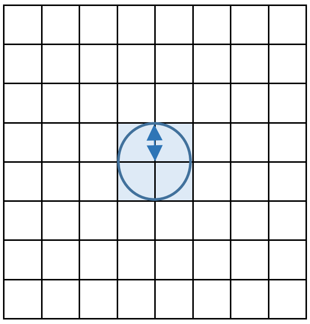
In order to determine the size of the brush, there must be a way to store what the brush looks like. In order to do this, there must be a store for the current brush.

Pseudocode

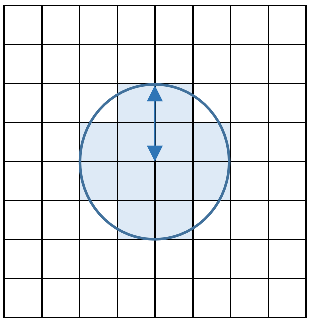
```
List currentBrush = new List of Points
currentBrush.Add(0,0) // brush has a set point at the origin
currentBrush.Add(-2,-2) // brush has a point 2 above and 2 to left of origin
currentBrush.Add(2,2) // brush has a point 2 below and 2 to right of origin
```

3.4B Constructing current brush

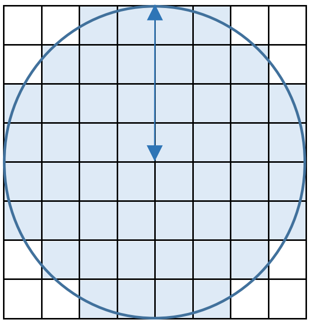
When generating brushes from certain sizes, it can be primarily planned by laying expectations:



Brush size 1



Brush size 2

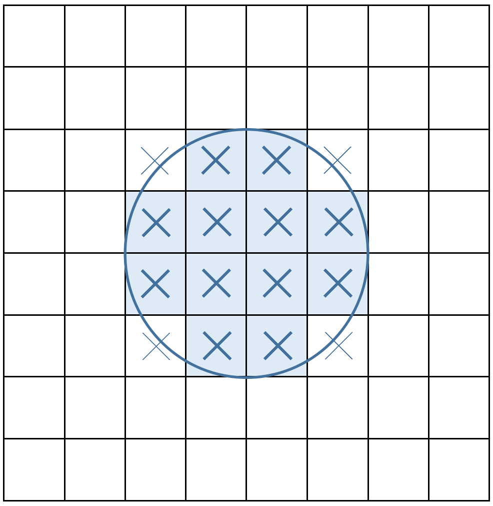


Brush size 4

In this case, it is clear that the size of the brush is determined by its radius.

In order to start generating the brushes, an **exhaustion algorithm** can be used. In this case, every possible pixel that can be a part of the brush is tried, and if it is inside the circle, it is added to the dimensions of the brush.

In the case of the Brush Size 2, every point marked with an x is checked.

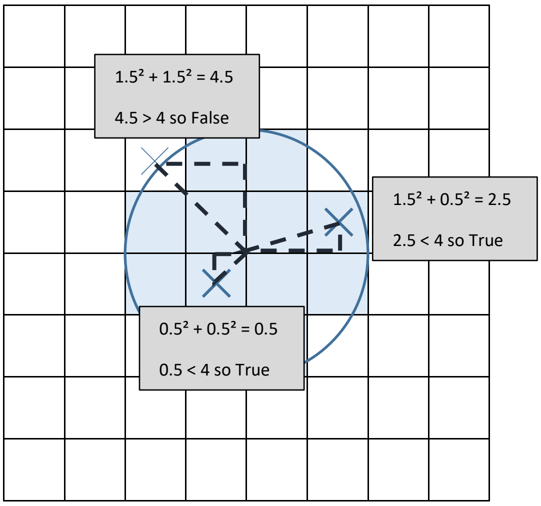


Generating the locations of these crosses is simple, the coordinated of each pixel is generated, and then 0.5 is added to centre it inside the pixel. This makes the pseudocode:

```
FOR x = radius*-1 TO brushSize
      FOR y = radius*-1 TO brushSize
            IF IsPointInCircle(x+0.5, y+0.5) THEN
                  currentBrush.Add(x,y)
            END IF
      NEXT
NEXT
```

To check whether a point is inside a circle, simple Pythagoras can be used. The points distance to the original can be calculated ($x^2 + y^2$), and if it is less than the brushSize^2, then the point is inside the circle.

So again coming back to the size 2 brush:



In algorithmic form:

```
IsPointInCircle(x,y) {
      IF ((x*x)+(y*y) <= (brushSize*brushSize)) THEN
            RETURN true
      ELSE
            RETURN false
      END IF
}
```

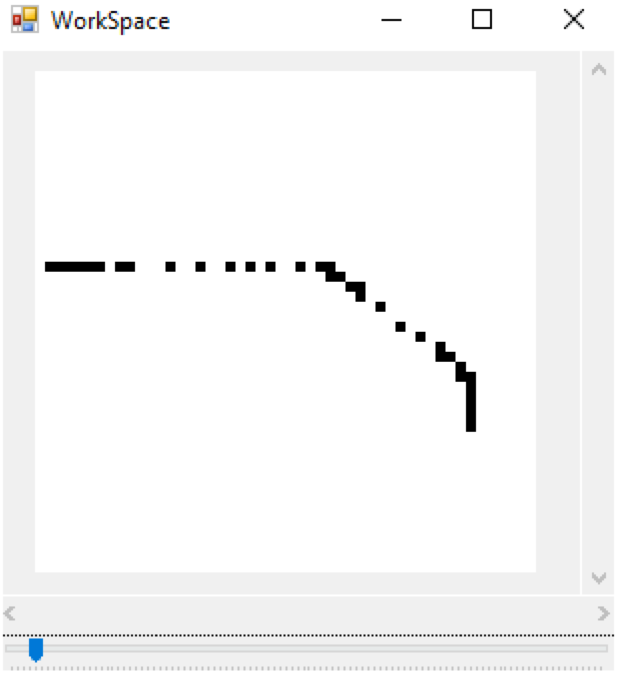
Algorithm 3.5 Brush Colour

When needing to change the brush colour, the RGBPicker form can be created, and edited from there.

Algorithm 3.6 Continuous Lines

This code is to fix a problem that was noted during the first session of stakeholder feedback.

> "The current brush is a very simple affair, however it means that when drawing larger lines there is the potential for gaps. This will be resolved later when the brush is fully implemented"



3.6A Bresenham's Line Algorithm

In order to implement this, straight lines must be drawn between any gaps in the pixels. In order to draw straight lines in this case, an algorithm known as **Bresenham's Line Algorithm** can be used. It is a well-established and efficient way of drawing a line between two points.

Psuedocode

```
DrawLine(x1,y1,x2,y2) {
    IF x1 > x2 THEN
        tempX = x1
        tempY = y1
        x1 = x2
        y1 = y2
        x2 = tempX
        y2 = tempY
    END IF

    IF (x2 – x1) > Abs((y2 – y1)) // If change X is bigger than change Y
        A = 2 x (y2 – y1)
        B = A – (2 x (x2 – x1))
        P = A – (x2 – x1)
        currentY = y1

        FOR currentX = x1+1 TO x2
            IF P < 0 THEN
                P += A
            ELSE
                IF (y2 > y1) THEN
                    currentY++
                ELSE
                    currentY--
                END IF
                SetPixel(currentX,currentY)
                P += B
            END IF
        NEXT
    ELSE
        A = 2 x (x2 – x1)
        B = A – (2 x (y2 – y1))
        P = A – (y2 – y1)
        currentX = x1
```

If the second pixel is further left than the first, they are swapped. This decreases the amount of potential point configurations

```
        FOR currentY = y1+1 TO y2
            IF P < 0 THEN
                P += A
            ELSE
                IF (x2 > x1) THEN
                    currentX++
                ELSE
                    currentX--
                END IF
                SetPixel(currentX,currentY)
                P += B
            END IF
        NEXT
    END IF
}
```

3.6B Setting the pixels

When the brush is moved, the above algorithm can be used to set pixels between the points

Pseudocode

```
HandleMouseMove(oldPoint, newPoint) {
    SetPixel(oldPoint.x, oldPoint.y)
    DrawLine(oldPoint.x, oldPoint.y, newPoint.x, newPoint.y)
    SetPixel(newPoint.x, newPoint.y)
}
```

Unit Test

Test	*ID*	*Expected Result*	*Comment*
Draw a 1 width line across middle of image	1	A line appears and is displayed on the image	This tests that a normal line can be drawn on the image
Draw a 1 width line at the edge of the image	2	A line is drawn at the edge	This tests that a normal line can be drawn at the edge of the image
Draw a 4 width line across middle of image	3	A thicker line is displayed on the image	This tests that a thick line can be drawn on the image
Draw a 4 width line at the edge of the image	4	A thicker line is displayed at the edge	This tests that a thick line can be drawn at the edge of the image
Move the mouse rapidly drawing a 1 width line	5	The drawn line has no visible breaks	This tests whether the filled line algorithm works correctly

Move the mouse rapidly drawing a 4 width line	6	The drawn line has no visible breaks	This tests whether the filled line algorithm works correctly with thicker brushes

Algorithm 3.7 Designing Undo Interface

The interfaces should be designed and implemented in accordance to class design 3.1.3.1 & 3.1.3.2. The reason for the implementation this way is to have a common method of performing (and undoing) any action in the program.

Algorithm 3.8 Implementing Undo Interface

3.8A New methods

The workspace should then include three new public methods:

```
PerformAction(action) {
        action.Do(this)
        RecordAction(action)
}
PerformActionSilent(action) {
        Action.Do(this)
}
RecordAction(action) {
        pastActions.add(action)
        futureActions.flush()
}
```

The purpose of these two data structures will be explained in 3.8B

The three methods are **justified** as:

- PerformAction is the general use action that will be performed and then recorded (in case it needs to be undone)
- PerformSilentAction is used in the case that an action is done to the image that should *not* be recorded, should be used very sparsely and in conjunction with RecordAction
- RecordAction is used after a silent action is performed, to make sure the action is recorded. These two alternatives are used when an action consists of many smaller actions, but only the larger change should be recorded (e.g. a brush draw may consist of many individual smaller line actions, but only the stroke as a whole should be recorded).

3.8B Dealing with previous actions

In order to implent the undoing and redoing that is needed by the program, two data structures are needed, being the pastActions and futureActions

Property	**Datatype**	**Justification**
pastActions	Stack of Actions	Stores the actions previously performed by the program. Only the last action needs to be accessed so a stack is needed
futureActions	Stack of Actions	Stores the actions that will be performed in the future, necessary for redoing (undo moves into the 'past', redo moves back from the past to its 'future', which is the present)

In diagrammatic form, if the user draws three lines then the contents of the data structures should be:

pastActions	LineAction 1	LineAction 2	LineAction 3
futureActions			

Then if the user undoes two of those lines then two actions should be moved into the future:

pastActions	LineAction 1	
futureActions	LineAction 3	LineAction 2

Then if the user chooses to redo a line the first action from the future is popped

pastActions	LineAction 1	LineAction 2
futureActions	LineAction 3	

Finally, if the user makes a new line the futureActions stack is cleared, it's no longer relevant.

pastActions	LineAction 1	LineAction 2	LineAction 4
futureActions			

Thus, the pseudocode for adding actions, undoing and redoing becomes:

```
RecordAction(action) {
    pastActions.Push(action)
    futureActions.Flush()
}
Undo() {
    lastAction = pastActions.Pop()
    lastAction.Undo()
    futureActions.Push(lastAction)
}
Redo() {
    nextAction = futureActions.Pop()
    nextAction.Do()
    pastActions.Push(nextAction)
}
```

Algorithm 3.8C Implementing PixelAction

The PixelAction interface is designed to encapsulate any action that changes the pixels of the image, so implementing its Do() and Undo() methods is comparatively easy.

Pseudocode

```
Do() {
    FOR EACH pixel IN newPixels
        SetPixel(pixel.location,pixel.colour)
    NEXT
}

Undo() {
    FOR EACH pixel IN oldPixels
        SetPixel(pixel.location,pixel.colour)
    NEXT
}
```

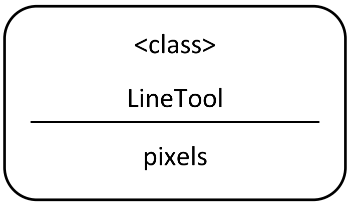
Unit Test

Test	*ID*	*Expected Result*	*Comment*
Undo an action (on default image)	1	Cannot be done	Tests that an action cannot be undone if no action has been perfoemd
Redo an action (on default image)	2	Cannot be done	Tests that an action cannot be redone if there is no future
Perform an action	3	Action is performed onto image	Tests that actions can still be performed
Undo the action	4	Action is undone	Tests that an action can be undone
Redo the action	5	Action is redone	Tests that an action can be redone
Undo the action	6	Action is undone	Tests that a previously redone action can be undone
Redo the action	7	Action is redone	Tests that a previously undone action can be redone
Perform another action	8	Action is done	Tests that actions can still be done from here
Undo both actions	9	Both actions redone	Tests that multiple actions can be redone
Undo an action	10	Cannot be done	Tests that the program stops undoing when there are no previous actions
Redo both actions	11	Both actions redone	Tests that both actions can be redone
Redo an action	12	Cannot be done	Tests that the program stops redoing when there is no future

Undo an action, and preform a new action	13	New action is performed on top of old action	Tests that actions can be performed on a previously undone action
Redo an action	14	Cannot be done	Tests that the future is cleared when a new action is performed

Algorithm 3.9 Implementing Layer Class

The structure of the layer class can be implemented in accordance to 3.1.3.5



Then the Image class should be updated to contain a list of layers, as well as a reference to its currentLayer

Property	Datatype	Justification
layers	List of Layers	Stores all of the image's current layers
currentLayer	Layer	Contains a reference to the layer that is currently being edited

Algorithm 3.10 Allow layer editing

In order to achieve this, there must firstly be a way to change which layer is selected. This can be done using the following code:

```
ChangeLayer(newLayerID) {
        currentLayer = layers[newLayerID]
}
```

Then to begin editing the layer, the SetPixel and GetPixel functions must be edited

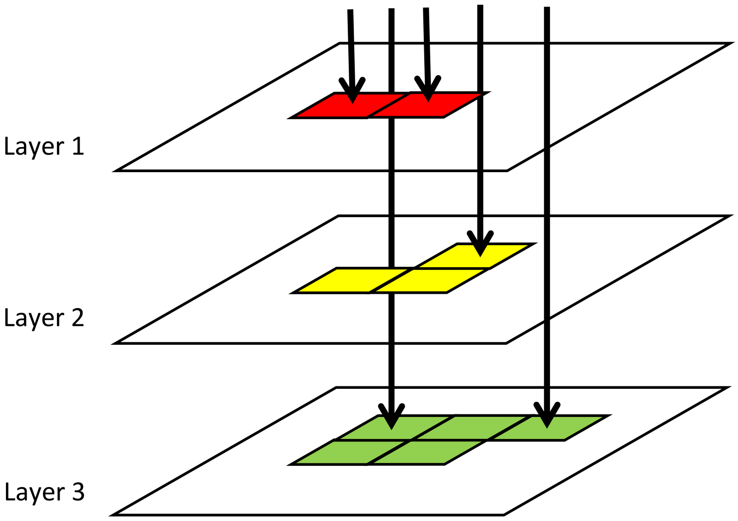
```
SetPixel(location, colour) {
        currentLayer.pixels[location] = colour
}

GetPixel(location) {
        RETURN currentLayer.pixels[location] = colour
}
```

This means that **no functionality is lost** and despite internal changes, externally nothing has changed, meaning that the code is more compatible

Algorithm 3.11 Display layer

In order to create the image now, when displaying a pixel each layer must be iterated through to find the first non-blank pixel, or in other words the process must look like:



Where the layer blocks the view to the layer below it, when a non-transparent pixel is seen.

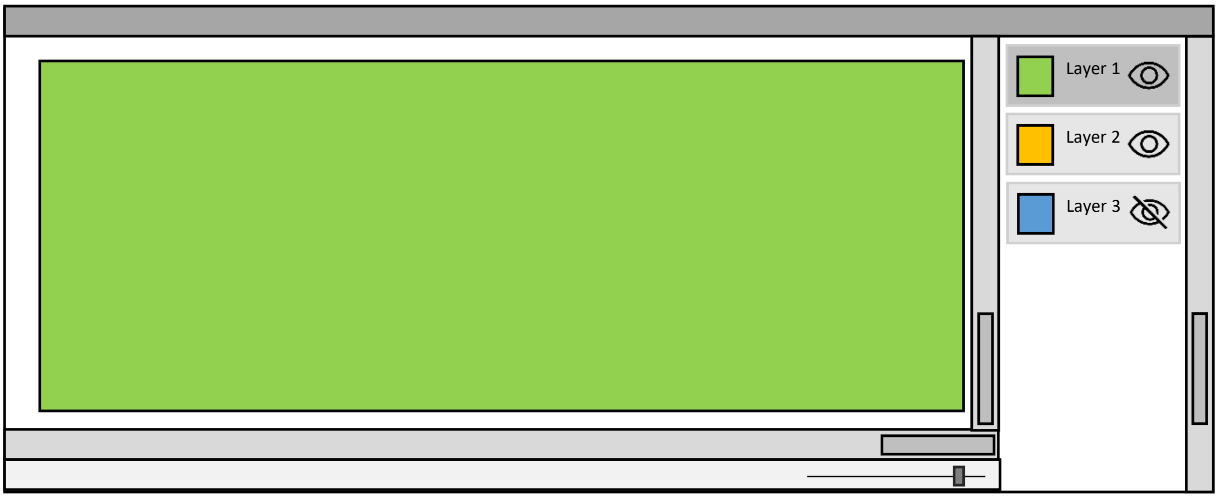
This makes the pseudocode:

```
DisplayPixel(x,y) {
      FOR EACH layer IN layers
            IF layer.pixels[x,y] is transparent
                  STOP REPEAT
            ELSE
                  //Draw pixel on image (as before)
            END IF
      NEXT
}
```

Algorithm 3.12 Add layer selector

In order to allow the user to manipulate the layer, a layer tool must be added that allows the user to select which layer is active, deselect the layer, and show/hide a layer. A potential design for this could look like:

Initial Design



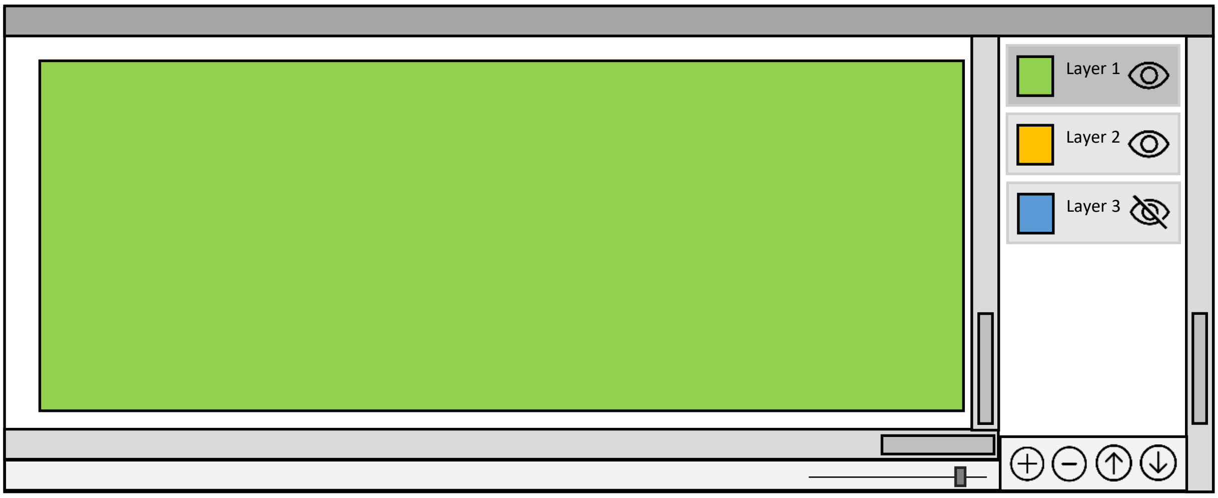
Stakeholder feedback

After showing the stakeholders the design, I got the following feedback:

- “Looks solid, but will you be able to delete a layer?”
- “How can I deselect a layer?”
- “What about the rearranging tools, deleting and adding new layers?”

Addressing this feedback leads to:

Updated Design



3.11A Drawing icon for layer

Each layer must contain a small icon to let the user know roughly what the layer looks like, drawing this image is much simpler as the entire image will always be displayed, the only difficulty will be downscaling the image to its smaller size.

A simple way to resolve this is to iterate through each pixel and find out which file pixel to draw from it, making the pseudocode:

```
DrawIcon() {
      FOR x = 1 TO iconHeight
            FOR y = 1 TO iconHeight
                  imageX = (x * imageWidth) / iconWidth
                  imageY = (y * imageHeight) / iconHeight
                  DrawRectangle(x,y,1,1,colours[imageX,imageY])
            NEXT
      NEXT
}
```

Unit Test

Test	*ID*	*Expected Result*	*Comment*
Select the top layer and draw onto it	1	There is drawing on layer	This tests whether a layer can be drawn onto
Attempt to delete the layer	2	The layer cannot be deleted as it is the only one	This tests whether the program is prevented from having 0 layers
Select lower layer and draw onto it at same location as top layer	3	There is drawing on lower layer, but it is obviously below	This tests whether the user can tell what layer they are drawing onto
Move lower layer up	4	The lower layers moves on top of previous top layer	This tests whether the drawing adjusts when the layers are moved
Move the highest layer up	5	Should be impossible as it cannot go up	This tests whether the highest layer cannot be moved too high
Move the lowest layer down	6	Should be impossible as it cannot go down	This tests whether the lowest layer cannot be moved too low
Add a new layer	7	A new (empty) layer is created	This tests whether a new layer can be made
Add 10 new layers	8	Many new layers are added	This tests whether the system can handle many layers
Layers are labelled then randomly rearranged	10	The layers are moved	This tests whether the program can handle moving many layers
The 11 layers are deleted	11	The layers disappear in the order in which they are added in the list	This tests that the order is maintained when removing many rearranged elements from the list
The final layer is deleted	12	Prevented as there is only one layer	This tests whether the removal prevention code still works

Algorithm 3.13 & 3.14 Implementing eraser

This was initially thought to be a more complex class, however the eraser can simply be implemented as a special LineTool, where its colour is set to Color.Transparent, which has an alpha (opaqueness) value of 0.

Algorithm 3.15 Calculating Shapes

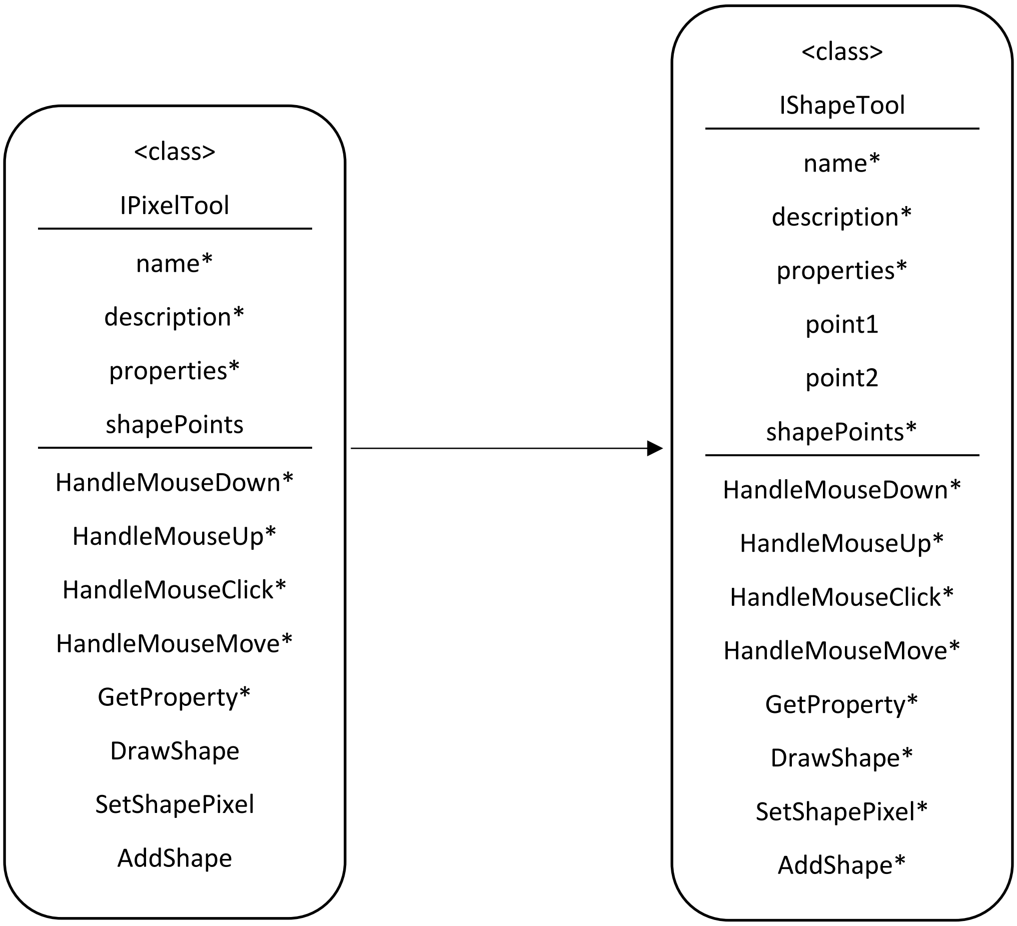
In order for the shape tool to work, there must be some simple shape calculation tools. Drawing from one of the image editing tools outlined in the analysis, **Paint**, there will be four implemented shapes:

- Line
- Rectangle
- Circle

3.15A Generic Shape Calculations

Common Class

In order for shapes to be implemented easily, there will be a base class that they inherit from IShapeTool, which inherits from IPixelTool, which inherits from ITool. This abstract class will encapsulate the generic process of adding a shape; click once, click somewhere else, shape is made. It will leave the calculations to make the shape up to its child classes. This means that the class will be implemented by:



The new properties are:

Property	**Datatype**	**Justification**
point1	FilePoint	The location of the top-left corner of the shape
point2	FilePoint	The location of the bottom-right corner of the shape
shapePoints	List of FilePoints	The points of the pixels that the shape will set

Method	**Params**	**Return type**	**Justification**
DrawShape	None (uses local Variables)	None	Draws the shape
AddShapePoint	Integer x, Integer y	None	Adds a point to be set to the shape. Will be outputted when AddShape is called
AddShape	None	None	Outputs the completed shape to the image

Handling Clicks and resolving points

Handling the Click events is reasonably simple, just requiring a check for whether the first point has been set or not

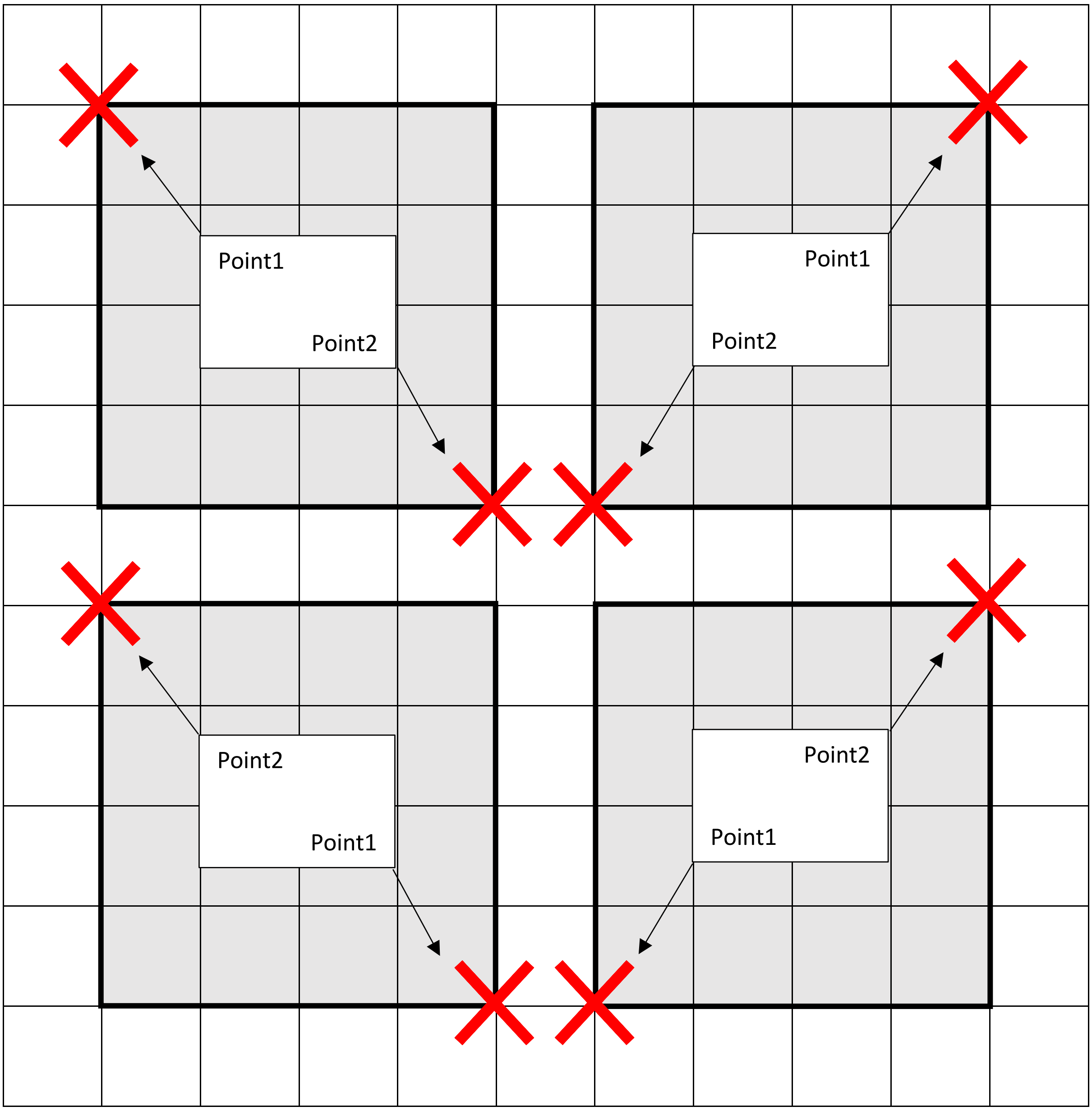
```
HandleMouseClick(clickLocation, button) {
      IF point1 is null THEN
            point1 = clickLocation
      ELSE
            point2 = clickLocation
            ResolveLocations()
            DrawShape()
            AddShape()
      END IF
}
```

After the second point has been set, the shape is created. This requires two steps:

- Resolving the two points to make them uniform
- Drawing it

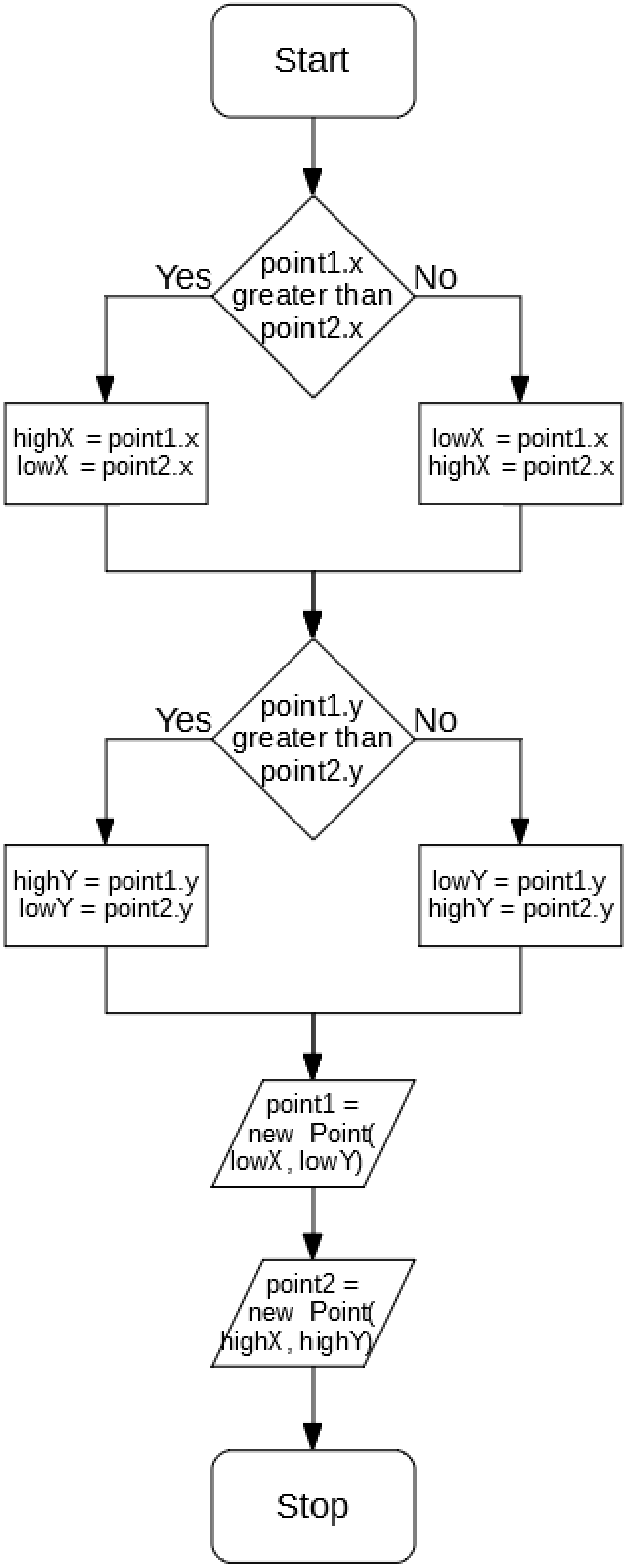
Resolving points

There is a need to resolve points, because it is unknown the order in which the user will create the two points, and there are four potential outcomes:



Having four potential situations is not desirable, as it means programming for four cases. To resolve this, the points can be modified so that, no matter what, they look like the top-left situation.

In order to do that, the general algorithm must involve finding the highest and lowest out of each category, or in a flow chart:



This now means that point1 will always have lower coords than point2

3.15B Line Drawing

LineDrawing can be accomplished by using Bresenham's Line Algorithm code implemented previously in Algorithm 3.6, requiring no additional pseudocode

3.15C Rectangle Drawing

Drawing a rectangle between two points can be accomplished using iterations:

```
DrawShape() {
    FOR x = point1.x TO point2.x
        FOR y = point1.y TO point2.y
            AddShapePoint(x,y)
        NEXT y
    NEXT x
}
```

3.15D Circle Drawing

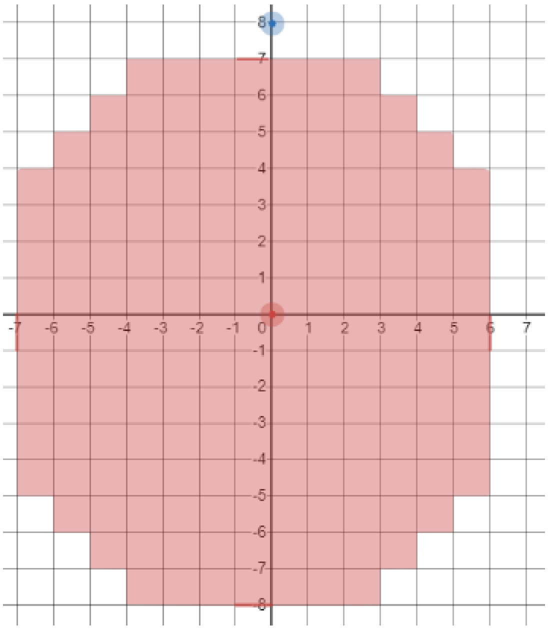
Circle drawing is a more in-depth process, but can be implemented reasonably simply using a similar process to Algorithm 3.4. All potential candidate points are checked as to whether they appear in the circle (or ellipse) generated, if so, draw it.

This means a condition for an ellipse will need to be defined. In order to do this a standard representation can be used:

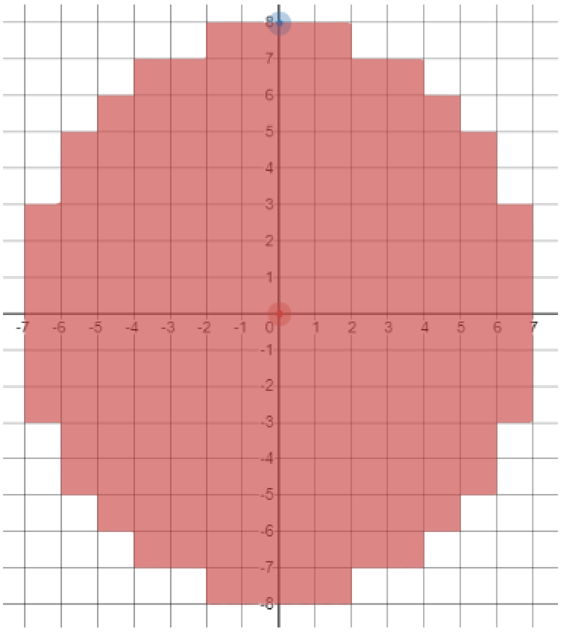
$$\frac{(x-(x'+0.5))^2}{w^2}+\frac{(y-(y'+0.5))^2}{h^2}\leq 1$$

Representation of a sphere of width w and height h with a centre of (x',y')

0.5 is subtracted from each point in order to make sure the *centre* of each pixel is checked, as this most accurately represents where the pixel should go, and means there is no bias between top left pixels and bottom right pixels.



Without correction



With correction

So every potential x and y can be checked to see if it satisfies that condition, if so, draw the circle

```
DrawShape() {
      centreLocX = (point1.X + point2.X)/2 + 0.5
      centreLocY = (point1.Y + point2.Y)/2 + 0.5
      widthSquared = (point2.X – point1.X) ^ 2
      heightSquared = (point2.Y – point1.Y) ^ 2



      FOR x = point1.X TO point2.X
            FOR y = point1.Y TO point2.Y
                  IF ((x – centreLocX) ^ 2) / widthSquared +
                     ((y – centreLocY) ^ 2) / heightSquared <= 1 THEN
                        AddShapePoint(x,y)
                  END IF
            NEXT y
      NEXT x
}
```

Unit Test

Test	*ID*	*Expected Result*	*Comment*
Create line with points (5,5) & (10,10)	1	A line is drawn from (5,5) to (10,10)	This tests that a line can be created using an above left first point
Create line with points (5,10) & (10,5)	2	A line is drawn from (5,10) to (10,5)	This tests that a line can be created using an below left first point
Create line with points (10,5) & (5,10)	3	A line is drawn from (10,5) to (5,10)	This tests that a line can be created using an below right first point
Create line with points (10,10) & (5,5)	4	A line is drawn from (10,10) to (5,5)	This tests that a line can be created using an above right first point
Create line with points (5,5) & (10,8)	5	A shorter line is drawn	This tests that a shorter, non-square line can be drawn
Create line with points (5,5) & (8,10)	6	A thinner line is drawn	This tests that a thinner, non-square line can be drawn
Create line with points (5,5) & (10,5)	7	A single row is drawn	This tests that a shape can be drawn that has same Y
Create line with points (5,5) & (5,10)	8	A single column is drawn	This tests that a shape can be drawn that has same X
Create rectangle with points (5,5) & (10,10)	9	A rectangle is drawn from (5,5) to (10,10)	This tests that a rectangle can be created using an above left first point
Create rectangle with points (5,10) & (10,5)	10	A rectangle is drawn from (5,10) to (10,5)	This tests that a rectangle can be created using an below left first point
Create rectangle with points (10,5) & (5,10)	11	A rectangle is drawn from (10,5) to (5,10)	This tests that a rectangle can be created using an below right first point
Create rectangle with points (10,10) & (5,5)	12	A rectangle is drawn from (10,10) to (5,5)	This tests that a rectangle can be created using an above right first point
Create rectangle with points (5,5) & (10,8)	13	A shorter rectangle is drawn	This tests that a shorter, non-square rectangle can be drawn
Create rectangle with points (5,5) & (8,10)	14	A thinner rectangle is drawn	This tests that a thinner, non-square rectangle can be drawn
Create rectangle with points (5,5) & (10,5)	15	A single row is drawn	This tests that a shape can be drawn that has same Y
Create rectangle with points (5,5) & (5,10)	16	A single column is drawn	This tests that a shape can be drawn that has same X
Create circle with points (5,5) & (10,10)	17	A circle is drawn from (5,5) to (10,10)	This tests that a circle can be created using an above left first point
Create circle with points (5,10) & (10,5)	18	A circle is drawn from (5,10) to (10,5)	This tests that a circle can be created using an below left first point
Create circle with points (10,5) & (5,10)	19	A circle is drawn from (10,5) to (5,10)	This tests that a circle can be created using an below right first point

Create circle with points (10,10) & (5,5)	20	A circle is drawn from (10,10) to (5,5)	This tests that a circle can be created using an above right first point
Create circle with points (5,5) & (10,8)	21	A shorter circle is drawn	This tests that a shorter, non-square circle can be drawn
Create circle with points (5,5) & (8,10)	22	A thinner circle is drawn	This tests that a thinner, non-square circle can be drawn
Create circle with points (5,5) & (10,5)	23	A single row is drawn	This tests that a shape can be drawn that has same Y
Create circle with points (5,5) & (5,10)	24	A single column is drawn	This tests that a shape can be drawn that has same X

Algorithm 3.16 Display & Algorithm 3.17 Log Undo

This can be accomplished by implementing the AddShapePoint function, as it needs to log any set pixels into a list

```
AddShapePoint(x,y) {
      shapePoints.Add(new FilePoint(x,y))
}
```

This means that any points are logged and then eventually outputted. To output them, this list can be iterated through to create a PixelAction object which will be executed by the image.

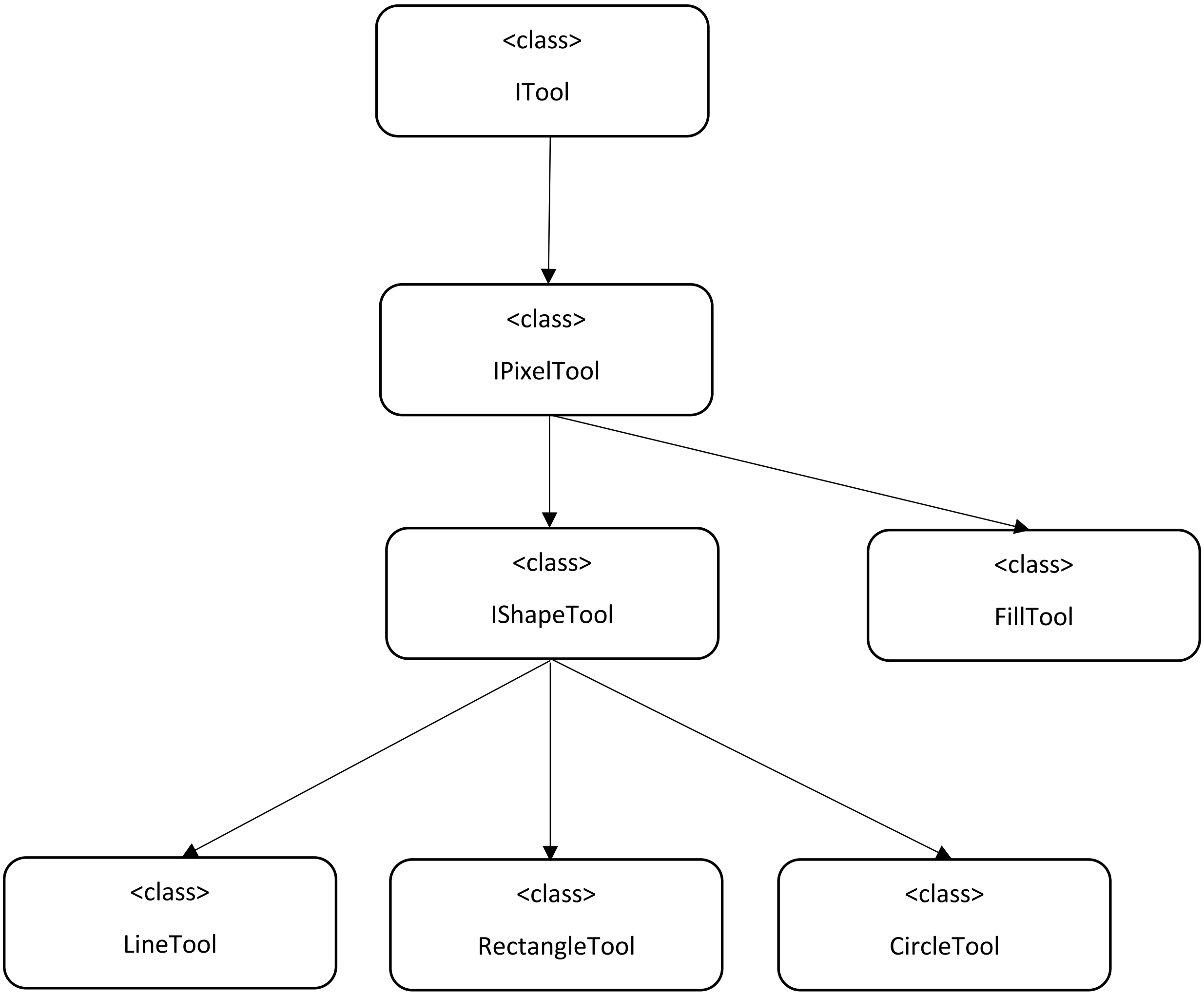
```
AddShape() {
      newAction = new PixelAction()
      newColour = GetProperty(“colour”)
      FOR EACH shapePoint IN shapePoints
            oldColour = image.GetPixel(x,y)
            newAction.AddPixel(x,y,oldColour,newColour)
      NEXT shapePoint
      image.PerformAction(newAction)
}
```

This demonstrates how the existing action classes can be used to safely perform an edit to the image.

Algorithm 3.17 & Algorithm 3.18 Fill Tool

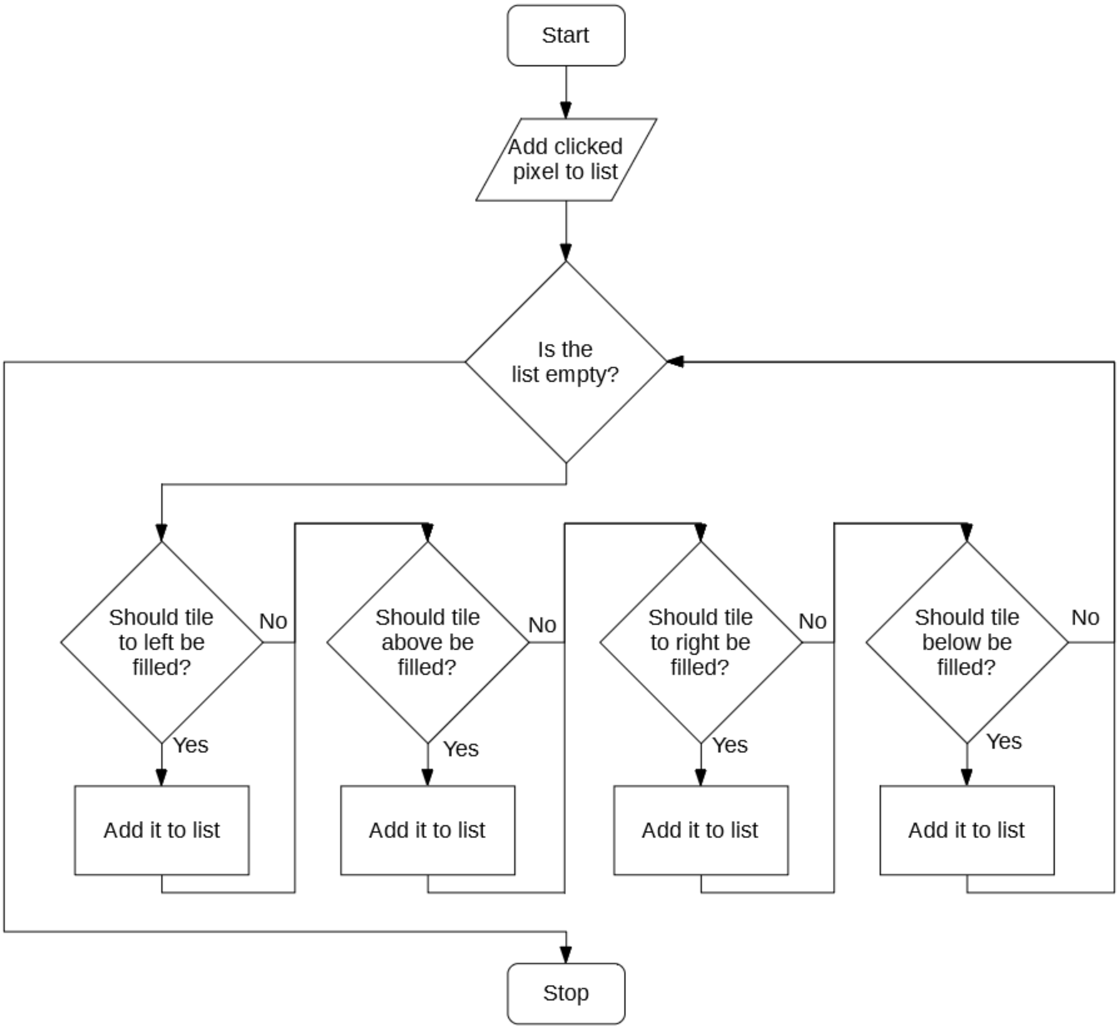
In order to implement the fill tool, a simple algorithm will be used. This is to save on programming time and a faster algorithm may not make much difference.

The FillTool will inherit from the existing IPixelTool. This makes the inheritance tree for the current tools:

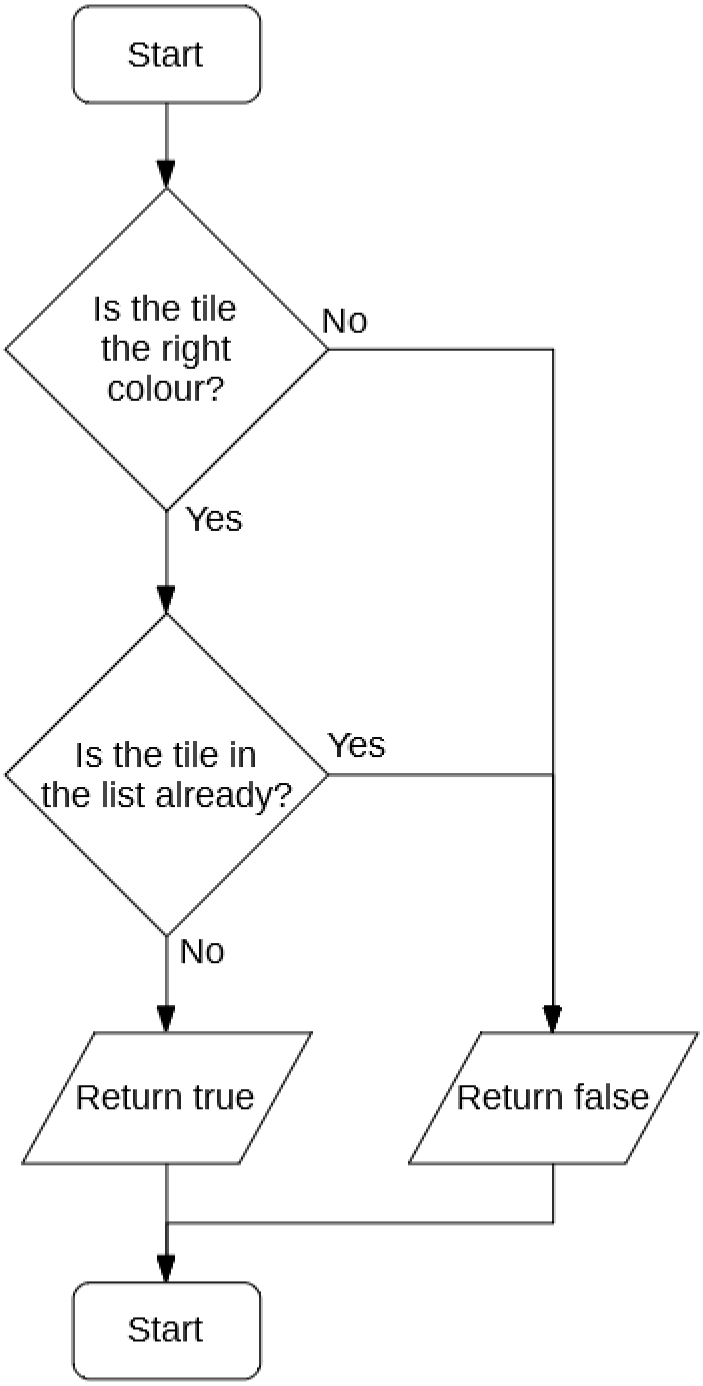


In order to create the fill algorithm, a simple structure will be used. When clicking on a starting point, that point will be parsed, and then the four adjacent points will be parsed, and so on, until the algorithm is complete.

In flowchart form, the algorithm for this becomes:


Start
Add clicked pixel to list
Is the list empty?
Should tile to left be filled?
No
Yes
Add it to list
Should tile above be filled?
No
Yes
Add it to list
Should tile to right be filled?
No
Yes
Add it to list
Should tile below be filled?
No
Yes
Add it to list
Stop

In order to determine whether a tile should be added to the list, the following algorithm can be used:


Start
Is the tile the right colour?
No
Yes
Is the tile in the list already?
Yes
No
Return true
Return false
Start