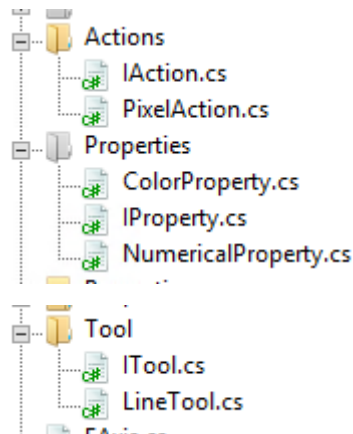


3.2 Development

05/11/2019 Class Design

Today I implemented the necessary classes for the next phase of SIMP. They have been organised into folders. They will be implemented later:

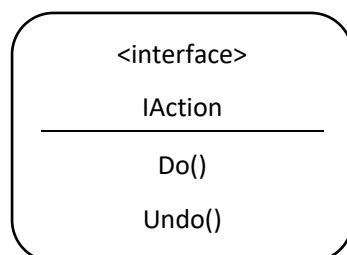


06/11/2019 Class Implementation

The following classes have been implemented:

IAction

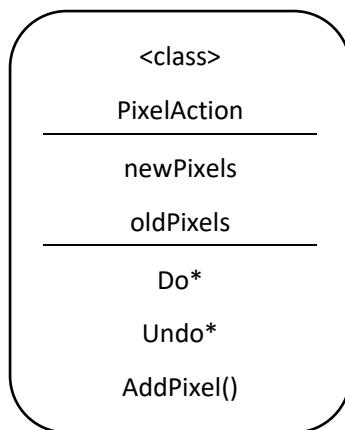
IAction has been implemented with its two functions in accordance to 3.1.3.1:



```
public interface IAction
{
    void Do();
    void Undo();
}
```

PixelAction

PixelAction has been partly implemented, its Do() and Undo() will be implemented later in development, in accordance to 3.1.3.2:



```
public class PixelAction : IAction
{
    private Dictionary<FilePoint,Color> oldPixels;
    private Dictionary<FilePoint,Color> newPixels;

    public PixelAction()
    {
        oldPixels = new Dictionary<FilePoint, Color>();
        newPixels = new Dictionary<FilePoint, Color>();
    }

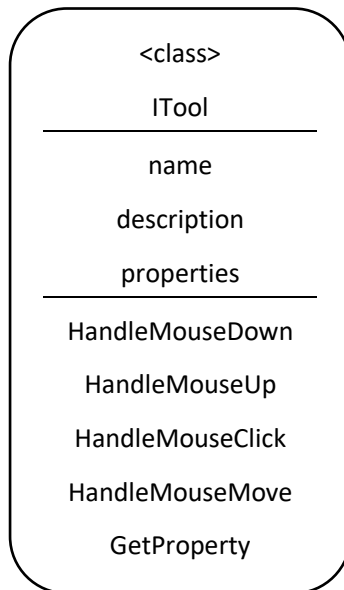
    public void AddPixel(FilePoint pixelLocation, Color oldColour, Color newColour) {
        // saves the old and new colours in the dictionary
        oldPixels[pixelLocation] = oldColour;
        newPixels[pixelLocation] = newColour;
    }

    public void Do(Workspace workspace) {
        //TODO: PixelAction Do()
        throw new NotImplementedException();
    }

    public void Undo(Workspace workspace) {
        //TODO: PixelAction Undo()
        throw new NotImplementedException();
    }
}
```

ITool

ITool has received a major change, it has been changed from an Interface to an Abstract Class. This is because the GetProperty method is common code for each class, there should be no need for each class to implement its own GetProperty method. Thus an Abstract Class is a better fit:



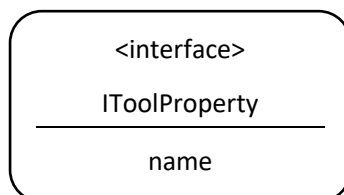
```
public abstract class ITool
{
    public string name;
    public string description;
    public List<SIMP.Properties.IProperty> properties

    public abstract void HandleMouseDown(FilePoint clickLocation, MouseButtons button);
    public abstract void HandleMouseUp(FilePoint clickLocation, MouseButtons button);
    public abstract void HandleMouseClicked(FilePoint clickLocation, MouseButtons button);
    public abstract void HandleMouseMove(FilePoint oldLocation, FilePoint newLocation);

    public SIMP.Properties.IProperty GetProperty(string propertyName) {
        //TODO: ITool GetProperty()
        throw new NotImplementedException();
    }
}
```

IProperty (IToolProperty)

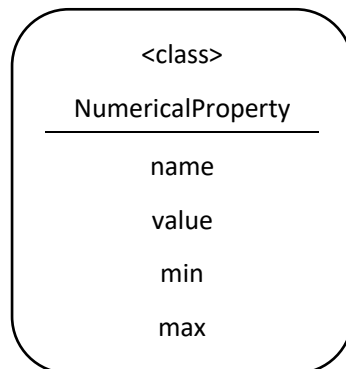
The simple IToolProperty interface has been implemented, though has been renamed to IProperty (as there is no obligation for it to be attached to a tool), and has also been changed to a static class, in accordance to 3.1.3.4:



```
public abstract class IProperty
{
    public string name;
}
```

NumericalProperty

NumericalProperty has been implemented in accordance to 3.1.3.5:



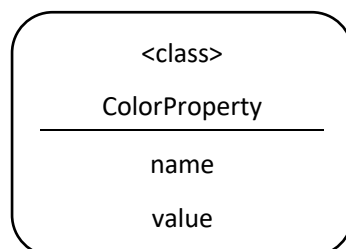
```
public class NumericalProperty : IProperty
{
    public int value;
    public int min;
    public int max;

    public NumericalProperty(string name, string value, string min, string max)
    {
        this.name = name;
        this.value = value;
        this.min = min;
        this.max = max;
    }
}
```

(Name is inherited from IProperty)

ColorProperty

NumericalProperty has been implemented in accordance to 3.1.3.6:



```
public class ColorProperty : IProperty
{
    public Color value;

    public ColorProperty(string name, Color value)
    {
        this.name = name;
        this.value = value;
    }
}
```

LineTool

The skeleton of LineTool has been implemented in accordance to 3.1.3.7:

```
public class LineTool : ITool
{
    public LineTool(string name, string description, Color color)
    {
        this.name = name;
        this.description = description;

        this.properties.Add(new ColorProperty("Color",color));
        this.properties.Add(new NumericalProperty("Brush Size",1,1,100));
    }

    public override void HandleMouseDown(FilePoint clickLocation, MouseButtons button) {
        //TODO: LineTool HandleMouseDown()
        throw new NotImplementedException();
    }

    public override void HandleMouseUp(FilePoint clickLocation, MouseButtons button){
        //TODO: LineTool HandleMouseUp()
        throw new NotImplementedException();
    }

    public override void HandleMouseClicked(FilePoint clickLocation, MouseButtons button) {
        //TODO: LineTool HandleMouseClicked()
        throw new NotImplementedException();
    }

    public override void HandleMouseMove(FilePoint oldLocation, FilePoint newLocation) {
        //TODO: LineTool HandleMouseMove()
        throw new NotImplementedException();
    }

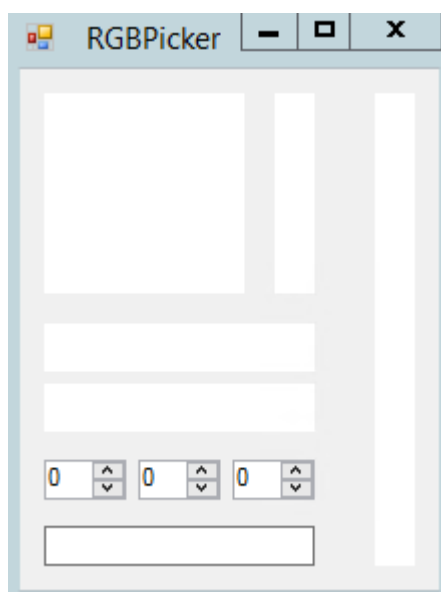
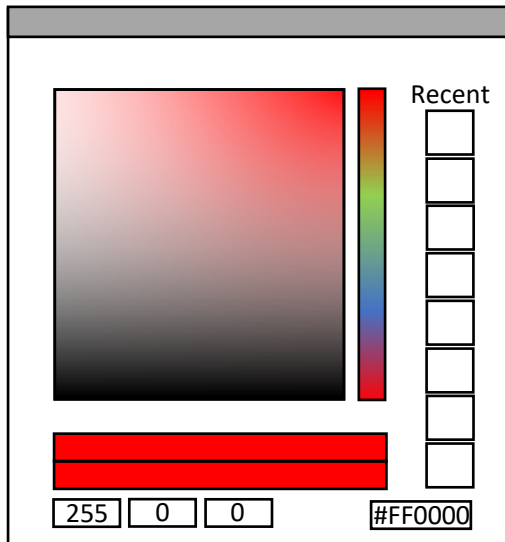
    public void DrawLine() {
        //TODO: LineTool DrawLine()
        throw new NotImplementedException();
    }
}
```

Layer

09/11/2019 RGB Picker design

Form Design

Today the basic outline of the RGB picker has been implemented, in accordance to the updated design in Algorithm 3.1:



There has been a single change made, the hex code input box has been moved to below the color input boxes, as it more closely part of that section. However none of the code for the form has been completed.

HSL to RGB

The algorithm for converting from HSL notation to RGB notation has been implemented, in accordance to Algorithm 3.2B:

```
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)  
}
```

```
public static Color HSLtoRGB(double h, double s, double l) {  
    double c = (1 - Math.Abs((2*l)-1)) * s;  
    double x = c * (1-Math.Abs(((h/60)%2)-1));  
    double m = 1-(c/2);  
    double r,g,b;  
    if (h < 60) {  
        r = c; g = x; b = 0;  
    }  
    else if (h >= 60 && h < 120) {  
        r = x; g = c; b = 0;  
    }  
    else if (h >= 120 && h < 180) {  
        r = 0; g = c; b = x;  
    }  
    else if (h >= 180 && h < 240) {  
        r = 0; g = x; b = c;  
    }  
    else if (h >= 240 && h < 300) {  
        r = x; g = 0; b = c;  
    }  
    else {  
        r = c; g = 0; b = x;  
    }  
    int R = (int)((r+m) * 255);  
    int G = (int)((g+m) * 255);  
    int B = (int)((b+m) * 255);  
    return Color.FromArgb(R,G,B);  
}
```

Generating Colour Square

The code to generate the Colour Square has been implemented, in accordance to Algorithm 3.2C:

```
private void UpdateColourSquare() {  
    float hue = _startColor.GetHue();  
    Bitmap newImage = new Bitmap(100,100);  
  
    // generate each pixel  
    for (float x = 0; x < 100; x++) {  
        for (float y = 0; y < 100; y++) {  
            // uses 1-() in order to flip in y axis  
            Color currentColour = SIMPRGB.HSLtoRGB(hue,x/100,1-(y/100));  
            newImage.SetPixel((int)x,(int)y,currentColour);  
        }  
    }  
  
    picColourSquare.Image = newImage;  
}
```

However the output square is different to that expected in the design:



This is due to the fact that the analysed program, GIMP, uses HSV colour notation rather than HSL. The calculation for HSV differ slightly, resulting in a different square. HSV cannot be used in SIMP without extra work due to the fact that the internal C# libraries for Color use HSL notation, as the Hue of a colour can be gotten with a pre-written function. This means the program will run faster.

Other Graphics

Code has been written for the other graphics objects, making the GUI currently look like so:



10/11/2019 RGB Picker functionality

Numerical Inputs

Some simple code for updating the current colour when a numeric input has been implemented. Each one checks the Boolean updating before proceeding, to make sure that the boxes aren't being updated by code:

```
void NumRValueChanged(object sender, EventArgs e)
{
    if (_updating) {
        return;
    }

    _currentColor = Color.FromArgb((byte)numR.Value, _currentColor.G, _currentColor.B);

    Update();
}

void NumGValueChanged(object sender, EventArgs e)
{
    if (_updating) {
        return;
    }

    _currentColor = Color.FromArgb(_currentColor.R, (byte)numG.Value, _currentColor.B);

    Update();
}

void NumBValueChanged(object sender, EventArgs e)
{
    if (_updating) {
        return;
    }

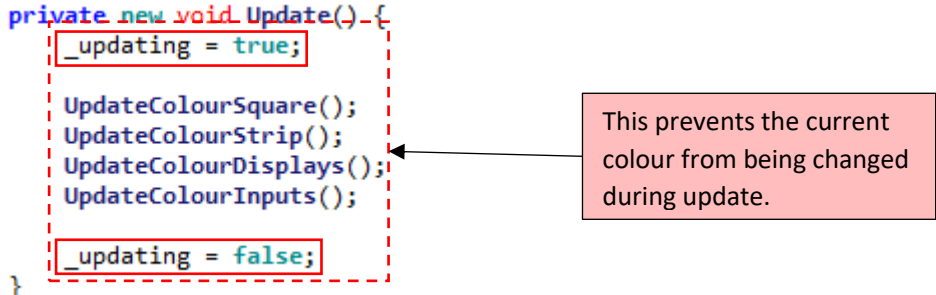
    _currentColor = Color.FromArgb(_currentColor.R, _currentColor.G, (byte)numB.Value);

    Update();
}

private new void Update()-{
    _updating = true;

    UpdateColourSquare();
    UpdateColourStrip();
    UpdateColourDisplays();
    UpdateColourInputs();

    _updating = false;
}
```



Colour Strip Input

Gathering input from the Colour Strip is also reasonably simple. As the strip is 100px tall simply multiplying the Y by 100 gets the wanted hue:

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

    float hue = ((float)me.Location.Y)*3.6f;
    _currentColor = _currentColor.SetHue(hue);

    Update();
}
```

Colour Square Input

The input from the Color Square can also be easily extracted, with some arithmetic on the X and Y of the clicklocation:

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

    float hue = _currentColor.GetHue();
    float saturation = ((float)me.Location.X)/100;
    float lightness = 1-((float)me.Location.Y)/100;

    _currentColor = SIMPRGB.HSLtoRGB(hue,saturation,lightness);

    Update();
}
```

RGB Display

When the first code for the RGB display was implemented, a problem arose. This was because when the code converted a number to hexadecimal, it did not pad the number, meaning that a hex code could be generated that was less than 6 in width:

```
private void UpdateColourInputs() {
    numR.Value = _currentColor.R;
    numG.Value = _currentColor.G;
    numB.Value = _currentColor.B;

    txtRGB.Text = string.Format("#{0}{1}{2}", _currentColor.R.ToString("X"), _currentColor.G.ToString("X"), _currentColor.B.ToString("X"));
}
```



This can be fixed by adding some extra padding to each part of the hex code:

```
private void UpdateColourInputs() {
    numR.Value = _currentColor.R;
    numG.Value = _currentColor.G;
    numB.Value = _currentColor.B;

    string R = _currentColor.R.ToString("X").PadLeft(2, '0');
    string G = _currentColor.G.ToString("X").PadLeft(2, '0');
    string B = _currentColor.B.ToString("X").PadLeft(2, '0');

    txtRGB.Text = string.Format("#{0}{1}{2}", R, G, B);
}
```

The parts are now padded

Problem with returning

When originally designing the RGBPicker feature, it was assumed that since a colour was an object, it would be passed by reference. Thus any changes to the colour would reflect in its original function. However this is not true for C#'s colour class, which is passed by value. This means that changing the reference won't work.

Solution - Delegates

Instead, a **delegate** can be used, which defined by the caller and will handle changing the colour when necessary:

```
public delegate void ColorCallback(Color newColour);
```

Thus, when a new Save button is clicked, this delegate is called, sending the new colour back to the caller and allowing it to do whatever is needed with the new colour:

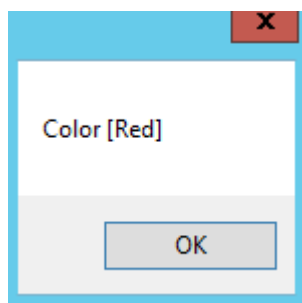
```
void BtnSaveClick(object sender, EventArgs e)
{
    // calls the callback to be handled by caller
    _updateCallback(_currentColor);

    Close();
}
```

Meaning a call can be made like this:

```
RGBPicker newPicker = new RGBPicker(Color.Red, delegate(Color newColour) {
    MessageBox.Show(newColour.ToString());
});
```

The colour is indeed correctly returned after pressing 'Save':



13/11/2019 Defining Tools

Storing Tools

The simple brush has been added to the workspace, though only in basic class form. This was done by defining a list, and adding the relevant info for the linetool:

```
// Defines tools
tools.Add(new LineTool("Brush", "Simple Brush", Color.Black));
```

Displaying Tools

However, it then becomes necessary to display the possible tools on the side of the program. Some simple code can be written to accomplish this:

```
// Adds buttons
int buttonY = 0;
foreach (ITool tool in tools) {
    Button newButton = new Button();
    newButton.Size = new Size(24, 24);
    newButton.Location = new Point(0, buttonY);
    newButton.Name = tool.name;
    newButton.Click += new EventHandler(ToolButtonClick);
    buttonY += 24;

    panTools.Controls.Add(newButton);
}
```

Then when the button is pressed:

```
void ToolButtonClick(object sender, EventArgs e) {
    Button buttonSender = (Button)sender;

    // disables every button except for pressed one
    foreach (Button button in panTools.Controls) {
        button.Enabled = true;
    }
    buttonSender.Enabled = false;

    // selects the tool
    foreach (ITool tool in tools) {
        if (tool.name.Equals(buttonSender.Name)) {
            currentTool = tool;
        }
    }
}
```

This means that currentTool will store the current tool being used.

Hooking Tools

This means that code can be taken out of the current click events, and moved to be encapsulated by the tool:

Before:

```
void DisplayBoxMouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left) {
        DisplayPoint clickLocation = new DisplayPoint(e.Location.X,e.Location.Y);
        image.SetPixel(clickLocation,Color.Black);
        UpdateDisplayBox(true);
    } else if (e.Button == MouseButtons.Right) {
        DisplayPoint clickLocation = new DisplayPoint(e.Location.X,e.Location.Y);
        image.SetPixel(clickLocation,Color.White);
        UpdateDisplayBox(true);
    }
}

void DisplayBoxMouseMove(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left) {
        DisplayPoint clickLocation = new DisplayPoint(e.Location.X,e.Location.Y);
        image.SetPixel(clickLocation,Color.Black);
        UpdateDisplayBox(true);
    } else if (e.Button == MouseButtons.Right) {
        DisplayPoint clickLocation = new DisplayPoint(e.Location.X,e.Location.Y);
        image.SetPixel(clickLocation,Color.White);
        UpdateDisplayBox(true);
    }
}
```

After:

```
void DisplayBoxMouseDown(object sender, MouseEventArgs e)
{
    DisplayPoint clickLocation = new DisplayPoint(e.Location.X,e.Location.Y);
    currentTool.HandleMouseDown(image.DisplayPointToFilePoint(clickLocation),e.Button);
}

DisplayPoint oldLocation = new DisplayPoint(0,0);
void DisplayBoxMouseMove(object sender, MouseEventArgs e)
{
    //TODO: Finish the clicking stuff
    DisplayPoint newLocation = new DisplayPoint(e.Location.X,e.Location.Y);
    currentTool.HandleMouseMove(image.DisplayPointToFilePoint(oldLocation),image.DisplayPointToFilePoint(newLocation));

    oldLocation = newLocation;
}
```

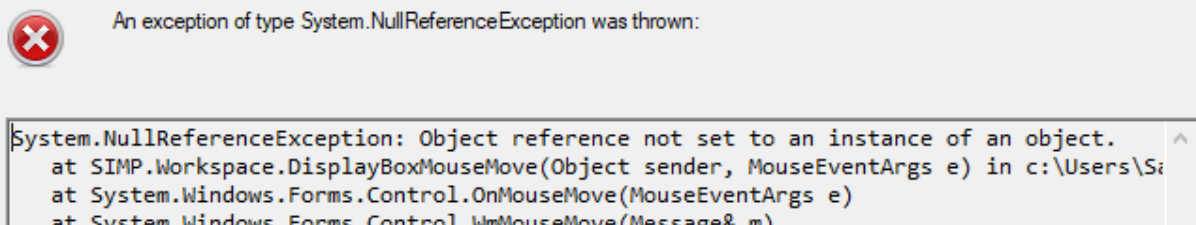
Which uses a new constructor for FilePoint

```
public FilePoint(Point p) : this(p.X,p.Y) {
    // no extra implementation
}
```

Fixing a NullReferenceError

However, a problem emerges with this implementation. At the start of the program `currentTool` is unset and thus null, so when any reference to this before a tool is selected causes a `NullReferenceError` to occur.

Unhandled exception



However, to make it so that there is a clean way of showing that there is a blank tool, a `BlankTool` class can be created and statically stored in the encompassing `ITool` class:

```
internal class BlankTool : ITool
{
    internal BlankTool()
    {
        //nothing
    }

    // no implementation
    public override void HandleMouseDown(FilePoint clickLocation, MouseButton button){}
    public override void HandleMouseUp(FilePoint clickLocation, MouseButton button){}
    public override void HandleMouseClick(FilePoint clickLocation, MouseButton button){}
    public override void HandleMouseMove(FilePoint oldLocation, FilePoint newLocation){}
}
```

In `ITool`:

```
//blanktool
public static ITool BlankTool;

static ITool() {
    BlankTool = new BlankTool();
}
```

And so the Blank Tool can be set like so:

```
currentTool = ITool.BlankTool;
```

Meaning that the crash is avoided.

16/11/2019 Implementing LineTool

A simple implementation for the tool can be coded, using a Boolean for whether the brush is down or not:

```
public override void HandleMouseDown(FilePoint clickLocation, MouseButton button) {
    //TODO: LineTool HandleMouseDown()
    _mouseDown = true;
}

public override void HandleMouseUp(FilePoint clickLocation, MouseButton button){
    //TODO: LineTool HandleMouseUp()
    _mouseDown = false;
}

public override void HandleMouseMove(FilePoint oldLocation, FilePoint newLocation) {
    //TODO: LineTool HandleMouseMove()
    Color myColor = Color.Black;
    if (_mouseDown) {
        myWorkspace.image.SetPixel(newLocation, myColor);
        myWorkspace.UpdateDisplayBox(true);
    }
}
```

Which is able to draw. The tool will be iteratively improved on until the tool is complete.

Implementing Colour Parameter

The first change that can be made is implementing the Colour property into the tool, which uses the GetProperty() method:

```
public override void HandleMouseMove(FilePoint oldLocation, FilePoint newLocation) {
    //TODO: LineTool HandleMouseMove()
    if (_mouseDown) {
        Color myColor = ((ColorProperty)GetProperty("Color")).value;
        myWorkspace.image.SetPixel(newLocation, myColor);
        myWorkspace.UpdateDisplayBox(true);
    }
}
```

Casts an IProperty to a ColorProperty to get its value

However GetProperty is not implemented, so must be implemented like so:

```
public SIMP.Properties.IProperty GetProperty(string propertyName) {
    //TODO: ITool GetProperty()
    foreach (IProperty property in properties) {
        if (property.name.Equals(propertyName)) {
            return property;
        }
    }
    throw new KeyNotFoundException("Couldn't find property " + propertyName);
}
```

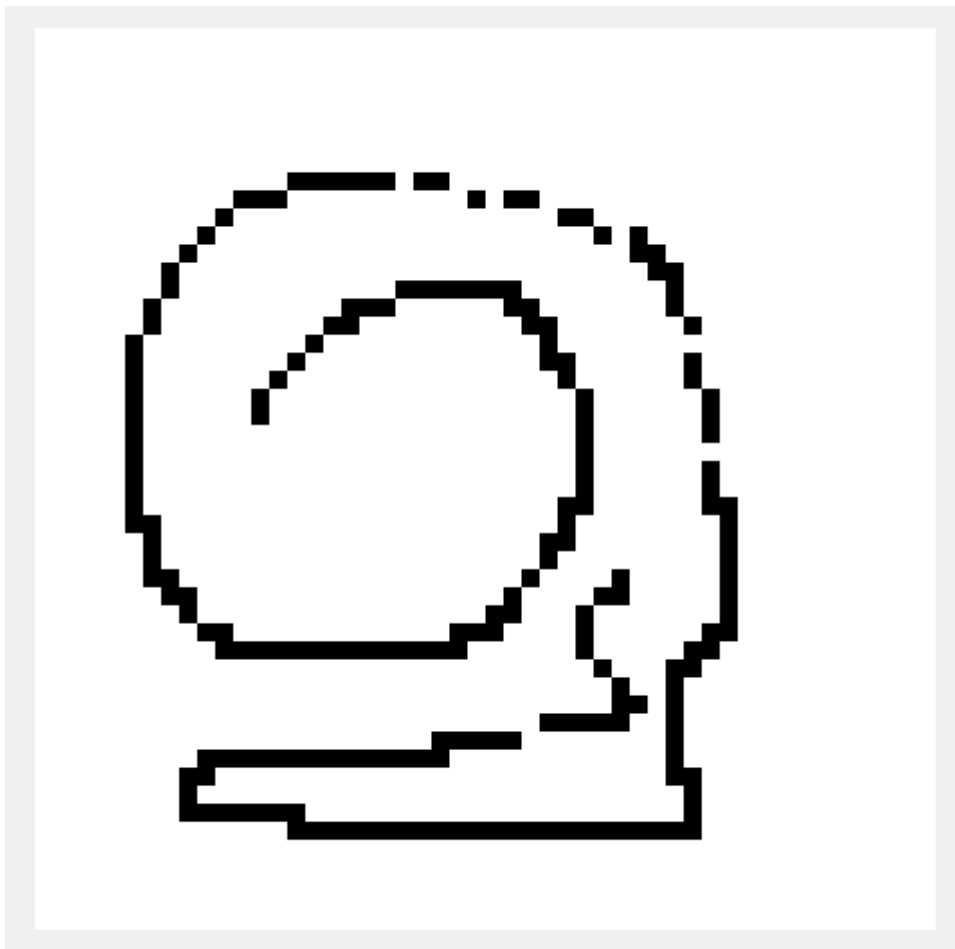

Connecting Pixels – Bresenham's

First, a check for whether two pixels is adjacent is implemented. This is because two adjacent pixels don't require Bresenham's Algorithm, it can just be simply drawn:

```
private bool IsAdjacent(FilePoint point1, FilePoint point2) {  
    if (Math.Abs(point1.fileX - point2.fileX) > 1) {  
        return false;  
    }  
    if (Math.Abs(point1.fileY - point2.fileY) > 1) {  
        return false;  
    }  
    return true;  
}
```

Then, Bresenham's algorithm can be implemented in accordance to Algorithm 3.6A

However, it was discovered that there was a problem with the implemented algorithm, it sometimes left a gap:



The problem turned out to be that the last pixel is not set by Bresenham's Algorithm, so it needs to be set manually, with an extra line of code:

```
myWorkspace.image.SetPixel(newLocation,myColor);  
DrawLine(oldLocation,newLocation);
```

This makes sure all lines are now connected:

