Introduction

For this assignment, I had the idea of implementing a 2D Turing Machine where the tape alphabet consisted of colour data, and the input tapes consisted of pixels. In this manner, the input of the machine could be a bitmap image, and the output could be images which have been modified by the machine.

Aim

The aim of the experiment is to see what type of visual effects can be made to an image with the most simple of processing: a Turing Machine. This aim is very broadly defined in order to allow experimentation and to see where the results can be taken. Rather than doing detailed planning on how the machine will be implemented, the implementation was done in an iterative manner, attempting to improve on the behaviour of the machine with each iteration.

Design

- The machine is written in C#.NET. This language and platform was chosen because .NET provides a simple but mature image manipulation library and very mature and simple GUI creation.
- Input to the machine can be an image of any of the following types: BMP, GIF, JPG, PNG and TIFF. After being opened, these files will be converted to Bitmaps in memory, then saved as the original file type once processing is complete.
- The Machine will begin at pixel position x=0, y=0, which in a Bitmap image corresponds to the top left hand pixel.
- The machine will run a fixed number of iterations then halt. How this relates to the behaviour of a Turing Machine is described below.

Turing Machine Equivalence

Tape Alphabet

The tape alphabet consists of 24 bits. Logically inside the transition function (more below), these 24 bits are treated as 3 bytes. Each byte will correspond to the RGB (Red, Green, Blue) components of that tape position (pixel) – Note that internally, .NET actually stores colour as a 32-bit ARGB value. The Alpha byte is ignored by the program.

Tape Dimensions

Due to the nature of the type of machine implemented, the dimensions of the tape can be of any size, and corresponds to the pixel dimensions of the input image.

Machine States
The machine state was developed in an iterative manner, but in the final versions of the machine, consists of:

- State information used to update pixel colour information (in later versions, 3 bytes corresponding to RGB values)
- State information storing the direction in which the machine is moving through the 2D tape (moving left or right and up or down)
- State information storing how many iterations the machine has made since it was begun, used to determine when to halt

**Transition Function Equivalence**

To be truly equivalent to a Turing machine, the transition function takes the machine state and the current tape position data as input, and can

- Halt (ie. The function is undefined for those inputs)
- Update the current tape position
- Update the machine state
- Move left or right

To make the implementation of this function simpler, it was split into four separate parts, which as a whole, can still be treated as a single transition function, making it Turing Machine equivalent. These parts are:

- Halting check. This part of the transition function checks if the number of iterations (stored as part of the state), is greater than the desired number, and halts if it is. This is logically equivalent of the Transition Function being undefined (ie causing a halt).
- Update tape data. This changes the data at the current tape position
- Update state. This changes the machine state
- Update tape position. This will move the tape left, right, up or down (up and down are equivalent of moving the second tape left or right)
Method

Mark One

The first simple and naive implementation of the machine behaved as follows:

- The machine state was stored as a single byte
- The R, G and B pixels of the current element were updated by XORing each of them with the state.
- The state was updated by XORing the previous state with the R, G and B values of the current element
- The next x and y tape coordinates were selected randomly:
  - The x coordinate was the R byte multiplied by the state byte, modulus the image width
  - The x coordinate was the G byte multiplied by the state byte, modulus the image height

As can be imagined, the result of this original version of the machine was to create an image that in no way resembled anything, it was simply noise, with bits of the original image barely discernable in the background. Before and after pictures are displayed below.

Illustration 2: Original Image

Illustration 1: After 10,000 iterations of Mk.I

Illustration 3: After 100,000 iterations of Mk.I

Illustration 4: After 1,000,000 iterations of Mk.I

While a proof of concept that the machine could update the pixels of an image, slowly
degrading an image until it is white noise is not very interesting.

**Mark Two**

The next model of the machine took its inspiration from the Turing Trains described in the assignment spec. Instead of the x and y coordinates of the next tape element to be examined being selected randomly, the direction to move (increment x or y) was chosen by XORing the R, G and B bytes of the state together, then moving in the x axis if the result was less than 128, or on the y axis otherwise.

This raised the issue of what to happen when an x or y coordinate ran off the end of the image. To keep things simple, at this stage I chose to simply wrap around to the start of the image in this case.

In the images generated, the paths of the Turing Trains through the image could be clearly seen moving diagonally downward from left to right. The images were still being slowly degraded to white noise. In illustration 5, vertical lines can be seen, suggesting that the tape movement was not truly random, an issue that was to prevail for a number of versions of the machine.

**Mark Three**

In order to update the pixels to a more meaningful value, I realised that the state would need to store just a single byte. The next version of the program contained a 3 byte state, each byte corresponding to the 3 RGB bytes of the input alphabet. Each element processed had its R G and B values replaces with those of the state (ie. The values of the previously examined element)
The results are certainly more interesting than the previous designs, in that they don’t tend towards noisy chaos.

**Mark Four**

Rather than have each element simply take on the value of the previous, this version of the machine used a more complicated rule for updating each element:

- If the Red value of the state was more intense than the Red value of the current element, it made the current elements Red more intense (ie increase the value of the byte representing Red.
- If the Red value of the state was less intense than the Red value of the current element, it made the current elements Red less intense.
- The above two steps were also followed for both the Green and Blue bytes.

The results of this step were somewhat surprising:

Obviously, not all parts of the image were being touched by the Turing Train. Close observation of Illustration 10 shows that the trains spent long runs travelling either vertically or
horizontally, the same issue that first manifested itself in Mark II. Clearly, there was a problem with the amount of randomness in the way whether to move x or y was chosen.

**Mark Five**

To improve the randomness of the direction choosing, the algorithm was improved from:

```java
byte seed = (byte)(stateR ^ stateG ^ stateB);
if (seed < 128) {move in x axis}
else {move in y axis}
```

to

```java
byte seed = (byte)((stateR ^ stateG ^ stateB)* i);
if (seed < 128) {move in x axis}
else {move in y axis}
```

where i is the counter of the number of iterations that the machine has made. This increase in pseudorandomness drastically changed the output of the program. The results were quite appealing – they resemble watercolour versions of the original image.

It was at this stage that I also began to play with the amount that each byte was raised or lowered. I call this amount the "byte offset".

All the results are quite aesthetically pleasing, resembling (with a bit of imagination) watercolour versions of the original picture.
It was at this point that I tried the machine on some different input files. An example is below:

The effect on this image is not quite as aesthetically pleasing. In addition, close examination of the bottom part of the image shows a trend in the grain in diagonal lines from top left to bottom right.

**Mark Six**

To minimise this diagonal grain, I decided to rethink the decision to have the trains wrap around the image. It might be better to have them “bounce” off the edges of the image. In order to implement this, I added two flags to the state to represent the trains movements left to right and top to bottom, which could be flipped when they hit an edge.
The black parts of the image are still grainy, but there is no longer any obvious diagonal lines

Mark Seven

Finally, I came up with the idea of instead of modifying an element’s Red byte based on the intensity of the state’s Red byte, modifying it based on one of the other two state bytes. I called this “rotating” the colours, and (arbitrarily) defined the following as “clockwise” rotating:

- The Red byte of an image has its intensity changed based on the Green of the state
- The Green byte of an image has its intensity changed based on the Blue of the state
- The Blue byte of an image has its intensity changed based on the Red of the state

When the colours interact in the opposite order to the above, I defined that as “counter clockwise”.

Following are some images generated by this final version of the machine:

Illustration 17: Byte Offset 16, Bouncing Edges
It is obvious comparing Illustrations 20 and 21 that bouncing trains seem to result in less streaking across the image than wrapping trains.
It is interesting to compare Illustrations 23 and 25. They both exhibit similar colour intensities. Perhaps due to the fact that it took twice as many steps, but only adjusting half as much colour intensity with each step, the colours of 25 are smoother than those of 23, and don't exhibit as much diagonal criss crosses from the travel of the trains.
The only part of the transition function I hadn't played with was the state updating algorithm. The original design had the state only storing the colour values of the previous pixel, like so (the update of the state was the last thing carried out before the tape movement):

```cpp
state.R = currPix.R;
state.G = currPix.G;
state.B = currPix.B;
```

I modified this to retain some information from the previous state, like so:

```cpp
state.R = (currPix.R + state.R) / 2;
state.G = (currPix.G + state.G) / 2;
state.B = (currPix.B + state.B) / 2;
```

I called the previous method of pixel updating "Previous Pixel", the new version "Pixel Feedback". This had the interesting effect of slightly increasing the softening watercolour effect, but with a smoother, more uniform colour distribution across areas of the image which were of similar tones. This is illustrated below:

Illustration 27: Byte Offset=8, 2 million iterations, Bouncing Trains, Previous Pixel
Illustration 26: Byte Offset=8, 2 million iterations, Bouncing Trains, Pixel Feedback
As an aside, the most painstaking part of designing the algorithm for the transition function was getting enough randomness into the tape-moving. Some examples of early works in progress are below:

Illustration 28: Poor randomness of tape movement - the train follows similar paths each time

Illustration 29: An attempt to fix the tape movement algorithm only makes it worse - nearly all moves are on the x axis

Illustration 30: Even without train bouncing, a more random tape movement algorithm is clearly superior

Illustration 31: Implementing train bouncing does not make up for the poor tape movement
GUI

While developing the program, I added a GUI in order to make it easier to experiment with various values and settings without recompiling the program each time. In addition, at the suggestion of James Harland, I added some code that could dump frames from the bitmap in memory as it was changed, which allows videos to be created using VirtualDub.

All settings should be obvious based on the definitions given previously in the document, with the exception of “Iterations per pixel”. This is a guide which shows the number of iterations divided by the total number of pixels in the image, which gives an idea of how much of a change each pixel is likely to experience.
Appendix A – Included Files

- **g2DTuring.zip** This ZIP file contains all the source files required to compile and run the GUI. This is a Visual Studio 2008 solution.
- **pine1.avi** A video of an image file being changed by the turing trains, with colours rotating.
- **bb0.avi** A video of an image file being changed by the turing trains.

**NOTE:** To play these videos, you may need an updated codec pack or a new version of Windows Media Player.

- **g2DTuring.exe** This is the compiled application. It requires the .NET 2.0 framework or later to run. This is a “demonstration” quality application, and as a result may contain bugs or issues – Caveat Emptor.

Appendix B – Source Code

This section contains source code used to run the program. For the benefit of the marker, I have removed all parts of the code relating to the GUI, multithreading, IO and movie making, only leaving the code related to the actual Turing Machine. The complete source code can be found in the zip file containing the Visual Studio Solution.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Threading;

namespace g2DTuring
{
    public partial class Form1 : Form
    {
        private static string fileName;
        private static Bitmap displayImage;

        const byte A = 0; // the Alpha value of a Color, unused in this program but required to construct a new Color

        public void turingTrain(BackgroundWorker worker)
        {
            // load the "tape" into program memory
            Bitmap tape = new Bitmap(fileName);

            // The coordinates of the first blanks on the tapes
            Point posnOfTapeBlanks = new Point(tape.Width, tape.Height);

            // The x and y coordinates of the two tapes, beginning at 0,0
            Point tapePosn = new Point(0, 0);

            // The state of the turing machine
            MachineState state = new MachineState();

            while (true) // the main loop of the turing machine
            {
                // read the tape
                Color currentTapeElement = tape.GetPixel(tapePosn.X, tapePosn.Y);

                // A special case of the transition function to force the machine to halt
                if (state.Iterations >= numLoops)
                {
                    // Code...
                }
            }
        }
    }
}
```
break; //halt the machine
}

//modify the current element on the tape
tape.SetPixel(tapePosn.X, tapePosn.Y, tapeElementUpdateFunction(currentTapeElement,
state));

//update the machine state
state.updateState(currentTapeElement, StateChangeAlgPrevPixel);

//move the tape position
tapePosn = updateTapePos(state, tapePosn, posnOfTapeBlanks);
}
saveOutput(tape, fileName);

private Point updateTapePos(MachineState state, Point tapePosn, Point tapeBlankPositions)
{
    byte seed = state.getSeed();
    if (Bouncing)
        #region bouncing off walls train
        {
            if (seed < 128)
            {
                if (state.LTR)
                {
                    if (tapePosn.X == tapeBlankPositions.X - 1)
                        state.swapLTR();
                    else
                        tapePosn.X++;
                }
                else
                {
                    if (tapePosn.X == 0)
                        state.swapLTR();
                    else
                        tapePosn.X--;
                }
            }
            else
            {
                if (state.TTB)
                {
                    if (tapePosn.Y == tapeBlankPositions.Y - 1)
                        state.swapTTB();
                    else
                        tapePosn.Y++;
                }
                else
                {
                    if (tapePosn.Y == 0)
                        state.swapTTB();
                    else
                        tapePosn.Y--;
                }
            }
        }
        #endregion
    else
        #region wrapping around walls train
        {
            if (seed < 128)
            {
                if (tapePosn.X == tapeBlankPositions.X - 1)
                    tapePosn.X = 0;
                else
                    tapePosn.X++;
            }
            else
            {
                if (tapePosn.Y == tapeBlankPositions.Y - 1)
                    tapePosn.Y = 0;
                else
                    tapePosn.Y++;
            }
        }
        #endregion
    return tapePosn;
}
private Color tapeElementUpdateFunction(Color c, MachineState state)
{
    byte R, G, B; //temporary variables to allow easier implementation of the transition
    function
    if (!RotateCols)
    {
        #region alter the component colour values based on the same colour in the state
        if (state.R < c.R)
            R = lowerByte(c.R);
        else
            R = raiseByte(c.R);
        if (state.G < c.G)
            G = lowerByte(c.G);
        else
            G = raiseByte(c.G);
        if (state.B < c.B)
            B = lowerByte(c.B);
        else
            B = raiseByte(c.B);
        #endregion
    }
    else
    {
    #region alter the component colour values based on adjacent colours in the state
        if (RotateCW)
        {
            if (state.R < c.G)
                R = lowerByte(c.R);
            else
                R = raiseByte(c.R);
            if (state.G < c.B)
                G = lowerByte(c.G);
            else
                G = raiseByte(c.G);
            if (state.B < c.R)
                B = lowerByte(c.B);
            else
                B = raiseByte(c.B);
        }
        else
        {
            if (state.R < c.B)
                R = lowerByte(c.R);
            else
                R = raiseByte(c.R);
            if (state.G < c.R)
                G = lowerByte(c.G);
            else
                G = raiseByte(c.G);
            if (state.B < c.G)
                B = lowerByte(c.B);
            else
                B = raiseByte(c.B);
        }
        #endregion
    }
    return Color.FromArgb(A, R, G, B);
}

private byte raiseByte(byte old)
{
    if (old + byteOffset > 255)
        return (byte)255;
    else
        return (byte)(old + byteOffset);
}

private byte lowerByte(byte old)
{
    if (old < byteOffset)
        return (byte)0;
    else
        return (byte)(old - byteOffset);
}

class MachineState
{
    private bool ltr, ttb; //directions of movement: left to right, top to bottom
private byte stateR;
private byte stateG;
private byte stateB;
private int iterations;

public int Iterations
{
    get { return iterations; } }

public byte R
{
    get { return stateR; }
}

public byte G
{
    get { return stateG; }
}

public byte B
{
    get { return stateB; }
}

public bool LTR
{
    get { return ltr; }
}

public bool TTB
{
    get { return ttb; }
}

public MachineState()
{
    ltr = true;
    ttb = true;
    stateR = 0;
    stateG = 0;
    stateB = 0;
    iterations = 0;
}

internal void updateState(Color c, bool prevPixel)
{
    if (prevPixel)
    {
        stateR = c.R;
        stateG = c.G;
        stateB = c.B;
    }
    else
    {
        stateR = (byte)((stateR + c.R) / 2);
        stateG = (byte)((stateG + c.G) / 2);
        stateB = (byte)((stateB + c.B) / 2);
    }
    iterations++;
}

internal byte getSeed()
{
    return (byte)((stateR ^ stateG ^ stateB) * iterations);
}

internal void swapLTR()
{
    ltr = !ltr;
}

internal void swapTTB()
{
    ttb = !ttb;
}