Introduction

A further application of Alan Turing’s concept, the two-dimensional Turing machine further exploits the infinite stretch of the conceptual tape. By introducing the machine to the screen visually using squares of colour to simulate data written to the tape, it can be shown how different processes can interact, help and hinder one another to complete a task. The concept of determinism is also explored, whereby multiple trains behave the same until manipulated by each other.

Implementation

Implemented using C++ and OpenGL, the machine’s two-dimensional tape (represented by a matrix of squares) is drawn onto the screen. These squares have two states, either active or inactive, represented internally by a number. When a square is inactive, it is not drawn and its position on the tape appears black. When set to active, a coloured square is displayed.

As it appears in the window, the tape is 128 by 128 squares in size by default, although this can be adjusted. The size of each ‘square’ inside the tape is dependent on the size of the window. Enlarging the window increases the size of each square so the tape fills the entirety of the space available. This may also be modified.

This implementation includes the capability of having multiple heads, or ‘trains’, running at once. Each of these heads may read from and write to the data on the tape in each transition. A head may be in one of four states at any given time, which allows it to be ‘facing’ a particular direction on the tape. The states, read-write operations and transitions for this machine can be summarised in the diagram below.

In its initial state, \( q_0 \), the head is facing north. Reading a symbolic ‘off’ (represented by -1) instructs the head to write a symbolic ‘on’ (represented by a unique number assigned to each head, shown as ‘N’ in the diagram), moves west and changes its state to \( q_3 \). Otherwise, reading a symbolic ‘on’ from the tape instructs the head to write the ‘off’ symbol, move east and transition to state \( q_1 \). We can see from the
diagram that \( q_0 \) represents up, or north; \( q_1 \) represents right, or east; \( q_2 \) represents down, or south; and \( q_3 \) represents left, or west. Therefore we may generalise these rules to say that whenever a head encounters a square in the active state, it changes it to its inactive state and rotates 90° clockwise. Similarly, finding a square in its inactive state, the head changes it to be active and rotates itself 90° counter-clockwise. Then, the head is moved forward one square on the tape in the direction it is facing and the cycle starts over.

If a head moves off the edge of the screen, it is wrapped around to appear on the opposite side. This simulates the tape as being infinitely expansive in both dimensions.

The code is written so that the heads take turns in reading and writing from the tape each cycle. This avoids the problem of having two heads occupying the same position on the tape and reading/writing at the same time, which was an issue with the first version of the application. In the latest version, each head reads, writes and transitions state before the next is allowed to have access to the tape.

**Discussion**

The heads start out by making seemingly random movements over the tape and creating an intricate pattern which is bunched into a tight area surrounding the starting position (figure 1). It becomes apparent with more than one head (or train) operating at once that the machine is deterministic (which is also indicated by the diagram in the above section). The heads all make the same movements, provided they do not interfere with one another.

![Random clumps](image)

**Figure 1: Random clumps.**

After a number of transitions, each head replicates a pattern which spawns out and away from the main cluster (figure 2).
Figure 2: Replicating patterns.

The pattern continues (wrapping around if it reaches the edge of the window) until the heads start reading what other heads of written. The pattern again becomes chaotic (figure 3).

Figure 3: Trains interfere with each other.

Occasionally, a head will follow along the outside of a pattern it or another head has made, or even erase it from the tape, while continuing to make its own diverging patterns until the entire tape is filled with random, but deterministic, data (figure 4).
This type of machine is known as *Langton’s Ant* (Pegg, 2002). The repeating structures that branch out from the main cluster are called “highways”.

**Conclusion**

By experimenting with a visual representation of Alan Turing’s computational machines, it can be seen that a set of simple rules called ‘transitions’ enables different sets of processes complete tasks may aid each other. Multiple trains of logic can be used in a computational system to speed up processing time. Furthermore, it has been shown that the set of logical transitions within the machine is fully deterministic since each head replicates the same pattern until interfered with. The patterns created become more chaotic and less predictable when the amount of heads is increased and the start positions are randomised.

**References**


**Appendix**

**C++ Code**

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>

#define TILE_SIZE 0.0078125
#define GRID_SIZE 128
#define HEADS 2
```
#define TRANSITION_TIME 0

int grid[GRID_SIZE][GRID_SIZE];

int headX[HEADS];
int headY[HEADS];

float headRed[HEADS];
float headGreen[HEADS];
float headBlue[HEADS];

enum direction
{
    NORTH,
    EAST,
    SOUTH,
    WEST
};
direction headDirection[HEADS];

void initialiseGrid();
void drawTile(float, float, float);
void toggleTile(int, int);
void transition();

<main.cpp>

// Computing Theory Assignment 2
// Ryan Tessier - 3164095
#include "main.h"

void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    int x, y;

    for (y = 0; y < GRID_SIZE; y++)

        for (x = 0; x < GRID_SIZE; x++)

            {g1PushMatrix();
             glTranslatef(-1.0 + TILE_SIZE + (x * TILE_SIZE * 2), 1.0 - TILE_SIZE - (y * TILE_SIZE * 2), -1.0);

                if (grid[x][y] != -1)

                    { drawTile(headRed[grid[x][y]], headGreen[grid[x][y]], headBlue[grid[x][y]]);
                    }

             glPopMatrix();
          }

    glutSwapBuffers();
}

void initialiseGrid()
{
```c
int i, j;
for (i = 0; i < GRID_SIZE; i++)
{
    for (j = 0; j < GRID_SIZE; j++)
    {
        grid[j][i] = -1;
    }
}

void drawTile(float red, float green, float blue)
{
    glColor3f(red, green, blue);
    glBegin(GL_QUADS);
    glVertex3f(-TILE_SIZE, TILE_SIZE, TILE_SIZE);
    glVertex3f(-TILE_SIZE, -TILE_SIZE, TILE_SIZE);
    glVertex3f(TILE_SIZE, -TILE_SIZE, TILE_SIZE);
    glVertex3f(TILE_SIZE, TILE_SIZE, TILE_SIZE);
    glEnd();
}

void toggleTile(int head, int x, int y)
{
    if (x >= 0 && x < GRID_SIZE && y >= 0 && y < GRID_SIZE)
    {
        if (grid[x][y] != -1)
        {
            grid[x][y] = -1;
        }
        else
        {
            grid[x][y] = head;
        }
    }
}

void moveHead(int head)
{
    // Move the head depending on its state
    if (headDirection[head] == EAST)
    {
        if (headX[head] < GRID_SIZE - 1)
        {
            headX[head]++;
        }
        else
        {
            headX[head] = 0;
        }
    }
    else if (headDirection[head] == SOUTH)
    {
        if (headY[head] < GRID_SIZE - 1)
        {
            headY[head]++;
        }
        else
        {
            headY[head] = 0;
        }
    }
    else if (headDirection[head] == WEST)
    {
```
```c
{  if (headX[head] > 0)  
    headX[head]--;  
  else  
    headX[head] = GRID_SIZE - 1;  
}  
else  
{  if (headY[head] > 0)  
    headY[head]--;  
  else  
    headY[head] = GRID_SIZE - 1;  
}

void transition()  
{  int i;  
  for (i = 0; i < HEADS; i++)  
  {  if (grid[headX[i]][headY[i]] != -1)  
      {  if (headDirection[i] == EAST)  
          headDirection[i] = SOUTH;  
        else if (headDirection[i] == SOUTH)  
          headDirection[i] = WEST;  
        else if (headDirection[i] == WEST)  
          headDirection[i] = NORTH;  
        else  
          headDirection[i] = EAST;  
      }  
    else  
    {  if (headDirection[i] == EAST)  
        headDirection[i] = NORTH;  
      else if (headDirection[i] == NORTH)  
        headDirection[i] = WEST;  
      else if (headDirection[i] == WEST)  
        headDirection[i] = SOUTH;  
      else  
          
```
{  
    headDirection[i] = EAST;
}

moveHead(i);
toggleTile(i, headX[i], headY[i]);

glutPostRedisplay();
}

void idle()  
{  
    transition();
}

int main(int argc, char **argv)  
{  
    initialiseGrid();

    int i;
    for (i = 0; i < HEADS; i++)  
    {
        headX[i] = rand() % GRID_SIZE;
        headY[i] = rand() % GRID_SIZE;
        headDirection[i] = NORTH;
        headRed[i] = (rand() % 11) / 10.0;
        headGreen[i] = (rand() % 11) / 10.0;
        headBlue[i] = (rand() % 11) / 10.0;
    }

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
    glutCreateWindow("Turing Machine");
    glutIdleFunc(idle);
    glutDisplayFunc(display);
    glutMainLoop();

    return EXIT_SUCCESS;
}