One vs Two Dimensional tape

Input and output are necessary features of all practical computer programs, and each one is useless without the other. For example, what good would a calculator program be if it correctly computed the user's equations, but did not output the result back to the user? In turn, what good would a highly efficient video conversion program be if a user could not input their own files into it? For the concept of Turing machines, input and output is related to the one-dimensional tape within the machine. This restriction to one-dimensional tape is only a physical limitation, as using offset values would allow a Turing machine to imitate having a two-dimensional tape. Using a two-dimensional tape offers advantages for many algorithms, especially making it easier to relate certain data to other data. Two-dimensional tapes can also be used for visual representation, outputting data in a grid format. This report explores the use of two-dimensional tapes for just that purpose, using a grid of images as the input and output of the machine and the Turing principle of checking one input at a time and possibly making changes to the state and also possibly outputting new data. This is all examined by creating a Turing Trains program that emulates a 'train' on it's path.

Grid and image sizes

The grid of images used in the program (the 'tape' of the machine) is of size 10 x 15, thus there are 150 panels for the machine to traverse. The screen dimensions of the entire panel is set to 800 x 600 initially, though this can be altered in run-time. This size is large enough to be easily viewed, but not too large to prevent viewing on most systems. Image size is 40 x 40, allowing an easily visible border to be drawn around each of the images to highlight the train and it's carriages. There are 6 different images, as well as a blank image.

Preliminary Experimentation

The first attempt at producing a two-dimensional tape Turing machine involved using a 2 x 3 grid, with only one square highlighted every step of the program. This program used a two-dimensional array of boolean values to represent which square the machine was currently analyzing. Another two-dimensional array (of the same size) contained the data (an integer value) stored at each square. These integers were used to determine what image was displayed at each square in the grid. With every move, the value within the current location was checked, and based on that value as well as the current state (an integer of value 1 or 2), the machine would alter the data in the current position, then move in one of the four possible directions. This program initially exhibited a problem with predictability; the machine would only pass through 5 of the 6 squares, and would do so in a fairly predictable manner. This was due to the fact that the current state altered between 1 and 2 every step, thus producing a pattern in the path taken. This was fixed by setting the state change to occur based on the input; on certain inputs the state would change but on others it would not. Each step of the Turing machine is invoked by clicking the mouse button anywhere on the panel.
Creating the Train Program

The second attempt at exploring two-dimensional tape Turing machines involved creating a Turing Trains program, that would not only highlight the currently observed square, but also leave the previous 10 observed squares highlighted as the 'carriages' of the train. This program also uses two-dimensional arrays to represent the location of the current square and the value stored within. There is also another array used, one that stores the locations of the carriages of the train. This array is added to with the current location just before the machine transitions to the next square, and once the array is full, the first one to be added is removed. This process makes the carriages 'follow' the train along it's path. A new variable, an integer representing the direction the train is travelling, is used to ensure that the train can only travel forward, left or right from it's current position and direction. Two other features implemented into the program were a 'halt' condition, and initial input options. When the program is run, the user is prompted to enter a value from 0 to 6, to initialise all the images to one of the images 1 through 6, or the blank image 0. The user is also given the option to enter -1, in which case the images are randomized. The 'halt' condition of the machine is based on the engine of the train colliding with one of it's carriages. In the event that this occurs, a message is displayed, and the program quits.

There were a few problems associated with the second attempt, namely finding an appropriate method to remember the location of the carriages after the machine has moved on to a new square. The first possibility was a stack, but unlike the first in-last out concept of a stack, managing the storage of the locations of the carriages required a first in-first out implementation. Thus an algorithm to shift the carriage array values backward in the array, to eliminate the first element in the array and then add the new one to the last index, was written and implemented.

Solving the 'Wrap Around' issue

For Turing machines, as with all current computing systems, available memory is an issue, especially with larger programs. Turing machines use the theory of an 'infinite tape' to prevent the limitation, but this is obviously not applicable in practical applications. Another solution is to use a tape that connects to itself in a continuous feed. This method is not practical when all the data that has been output must also be read, but for a case such as the Turing Trains example, data can be overwritten without worry and thus a continuous tape is very useful. For this two-dimensional scenario, there is also the issue of what to do when the machine tries to move off the screen (the end of the tape). This program solves the problem by moving the machine to the opposite side of the grid, making it appear as if the machine were wrapping around the 'track'. This is implemented within the program by setting the current location to either the beginning or the end of the location array (on either the current row or column), based on the direction that the train is currently moving.

Tested input

The first example tested on the program was with the entire grid set to blank (image 0). The beginning of the tape is set to the top-left corner of the grid. This Turing machine's inner-definitions state that when the current state is 1 (as it is set to initially), and the input is 0, the train drives forward. The initial direction of the train is set to 1, which means up, so the train moves up one square. Because the machine is already at the top of the
screen, this movement takes the machine to the very bottom of the first column. The Turing machine changes the image at the top-left from blank to image 4 (Gimli). From here the machine continues the same pattern of traveling up and altering the image to Gimli, as the state is not changed when currentState == 1 and input == 0. The carriages are left on screen, highlighting the route the train has taken. The train continues until it reaches the top left again, where it collides with the carriage still in the top left of the grid, and the collision message is displayed. The program exits.

The second example tested uses a grid set to the image 1 (Frodo). The machine code determines that upon input 1 and state 1, turn left, so the machine moves left, which takes it off screen and as such it wraps around to the very right-top corner. The current state is changed to 2. On input 1 and state 2, the machine turns right, which takes the machine off the top of the screen this time. This results in the machine being at the bottom-right of the screen. The machine continues in a zigzag pattern (left then up) through the grid, leaving a trail of some new images, and some images remaining the same. The pattern continues until the machine hits one of the new images, and proceeds to move in predetermined directions based on the machine until it collides with one of it's carriages.

The third example uses a grid completely set to image 6 (the ring). The machine remains in state 1 for the duration of execution, as the input of 6 with the current state as 1 does not result in a state change. The machine therefore follows the instructions to turn left, and repeats this until it collides with itself. Once again, the machine wraps around the grid when it moves off screen.

The final example is the most interesting of them all. This example sets each image in the input grid to a random input between 0 and 6. Due to the randomization, this example is difficult to re-create, but in the same manner, the results are much more interesting. For the tested example, the train snaked around the bottom left-corner of the grid, before driving forward to the very right side, then traveling down and wrapping around to the top. The train then zigzagged it's way towards the centre of the grid before colliding with itself.

**What has this shown?**

One of the most striking results from this experiment was the predictability of hard-coded Turing machine logic. In most cases this is not only a benefit, it's a necessity, to ensure that the same process (e.g. calculation, string conversion, etc...) can be performed time and time again with the same results. However, in situations such as this Turing Trains example, it is far more interesting to the user to have the machine act differently each time. This way the output is not as predictable. To achieve this, random variables, or possibly random state changes and random output, must be utilised. This results in a more appropriate output for this program, where the user can at least seem to receive a unique output each time they run the program. This experiment also highlighted the usefulness of multiple variables in the creation of programs. Turing machines are limited by making decisions based on the current state and the input at the current location. While Turing machines can still perform all the calculations and operations that current programs use, it requires a lot more code and round-about methods to achieve. As is evident in such a program as this Turing Trains example, where all sorts of
information needs to be considered and altered, having multiple variables to represent relevant information makes everything a lot simpler, and presents functionality that would be very difficult to convert back down to Turing machine logic. Features such as these multiple variables, as well as the use of two-dimensional arrays themselves, make coding a lot simpler for programmers. Despite Turing machines being able to compute the same algorithms as in current programs, they require much more effort, and even emulating two-dimensional arrays is far more difficult than it is with even an older language such as C. Thus, it is obvious how far programming has come, and how much easier we have things now with the ability to use all sorts of variables, and let the language itself manage a lot of the work. Turing machines may be a theoretical notion from the past, but they are still relevant today in revealing how different programming used to be, and how many current systems would be almost impossible to manage without the luxury of modern programming languages.