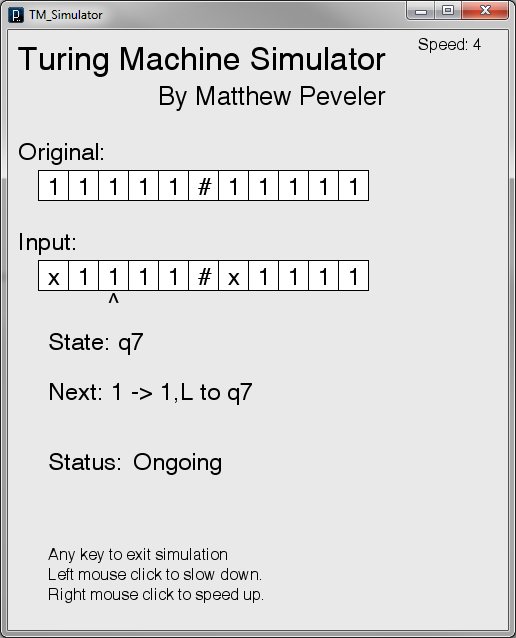
Matthew Peveler

CSC460: Theory of Computation

3/29/2012

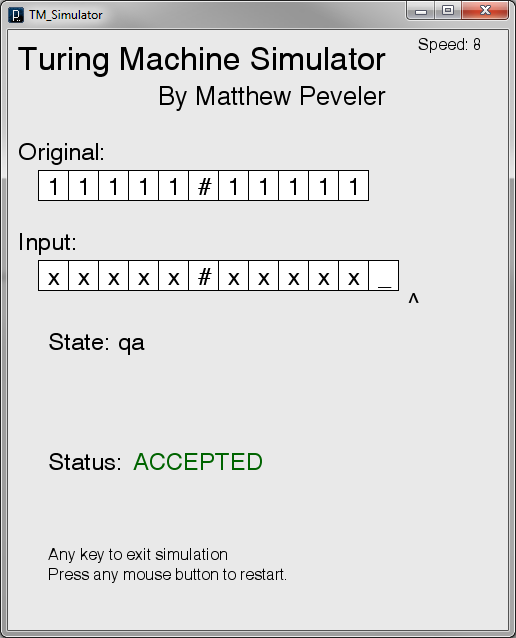
**Technical Report: Turing Machine Simulator**

The Turing Machine Simulator was written using the Processing platform, as it is Java-based, but with a graphical focus. This allowed me to completely focus on the logic of the program, and use very easy system calls to create the graphics and text on the screen as needed, as opposed to attempting to get graphics out of Java proper or C++. The downside of course was that the code has to be kept as small as possible on each draw function, so as to not bog down the system, as Processing does feature some more overhead on the system, when compared to Java. The small nature of the program though ended up making this not really an issue. Talking about the TM, it is a single tape TM, the very basic type of TM that could be used, and the first one introduced in the textbook.

 When executing the Simulator, it does some set-up before drawing. During the set-up phase of the execution, it reads in both the desired input, from “tm\_input.txt”, and the list of the state transitions for the TM, from “tm\_states.txt”. For the states, it reads in a line at a time, each line corresponding to a new transition state. There’s currently a HashMap object at the same time that’s ensuring that if I read in a value I already have, I then get the tmState object out of the HashMap, add the line to the state, and then continue on. If it isn’t a duplicate, then I create a new tmState that’s constructed with the line data. In both cases, the tmState is then put into the HashMap. The HashMap is mapped such that the key is equal to the state names, and the value is the tmState objects. tmState is the main class of the program, as it contains all the data for each state, as well as having functions that allow me to get the next state, alter the input, and get the new direction, and all other necessary things from that I read in from the tm\_state.txt file. Due to difficulties I encountered later on with the HashMap function, the program demonstrates some minor inefficiency as I then have an array initialized to the number of non-duplicate states. I then iterate through the HashMap, taking out each tmState object, and putting it into an empty index of this new array. I then re-set up HashMap such that the key is a string that corresponds to the state name, and then the value is an integer that corresponds to the index where the state is in that array. After that, we set the desired size of the sketch (minimal size is 500 width, anything above that is calculated based off length of input string), and the set-up period is over.

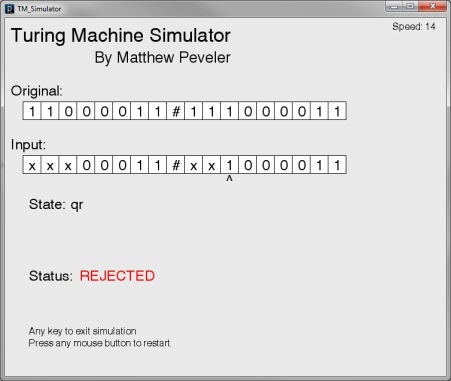
During the actual execution of the program, it displays the name of the program, my name, as well as directions on using it (such as right mouse button speeds up execution, left slows it down, both changing the frame rate by 2), and then the original input string, and what is currently on the tape (input). It then loads the needed transition into tmState based off where the iterator head is pointing. After that, it gets details about what we’re doing (direction we’re heading, whether or not the next state is a reject or accept state, etc. After that, we display a bit more text (such as our current state, and the transition function we’re following next), and then loop through the draw function again. If we hit a accept state, we set a “done” variable to true, with if he hit a “reject” state (or there’s nowhere to go from the current state) we set a “failed” variable to true. If either of these are equal to true, then we end the loop after the next draw, displaying a status of “ACCEPTED” or “REJECTED” as appropriate. Once draw has stopped looping, we wait for either the use to hit a mouse button (to restart the animation) or a key on the keyboard to exit.

1. Machine in mid-simulation

 During input, the TM operates on the input variable, which is of the String type. It passes the input over to the current state that the TM is in, to operate on the string, replacing the character the iterator is pointing to with whatever the transition function specifies, returning the modified string to the where the function was called. During execution, if the iterator ever exceeds the input string (goes to the right of it, and the next state reads in a space, then a space character (“\_”) get appended to the end of input, and the screen is resized if needed.

If the simulation is restarted, then the input is set back to the original input, and everything re-initialized to zero, and the FPS put back down to 4, and then letting the program run again from the beginning Then the program begins drawing again, starting at the beginning. At any time, if the keyboard is hit, the program exits execution.

2. TM accepts string, shows how input can add \_ to input

 Running the program is as easy as either using the “TM\_Simulator.exe” application file for Windows or the “TM\_Simulator” file for Linux (though the Linux one is unconfirmed, with the Windows 100% working). The README contained within the program contains how the tm\_states.txt file should be laid out so that all the desired states are read in perfectly. Examples of two TMs are listed in the “Test States.txt” file, with working inputs, as well as what the states should be for the tm\_states.txt file. For any other questions that you may have, they should be answered by the accompanying README.txt file found within the folder. Any further questions should be posted on the Github repo.

3. TM fails on string, so rejects