Blockmotion - Project Report

Rhys Evans (rhe24@aber.ac.uk)

9th May 2017

Contents

1 Introduction

In this document I will discuss the process of designing, implementing and testing a program called 'Blockmotion'. Blockmotion is a very minimal animation suite wherein footage is loaded from a plain text document and is displayed in a JavaFx window. The user can then manipulate the footage with various transformations such as 180°rotation and directional movement.

In my project I have successfully implemented all the requirements specified in the brief with relative ease. Some implemented features had to be revisited before submission for optimisation and debugging, this was the most challenging and time consuming aspect. In addition to the basic requirements I have also implemented several extra features for flair, these include:

- Support for the colours Green and Red
- Option to set the slide number for all directions
- Ability to add frame to footage from menu
- Ability to delete frame from footage
- File name appears as title of JavaFx window

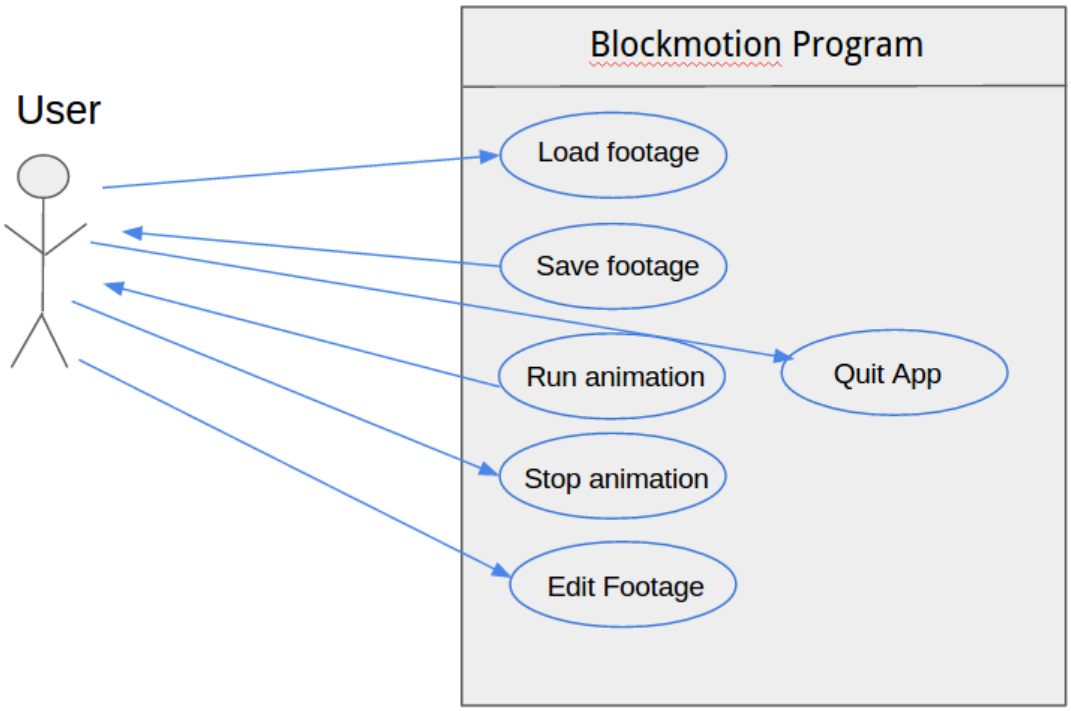

Figure 1: UML Use-Case Diagram

2 Design

2.1 Overview

The program is split into 3 packages: 'Gui', 'Model' and 'Transformation', each package and their classes are described in figure 2 below. A more detailed description of each class and it's relationships is given in the subsections to follow, along with some explanations of particular design choices.

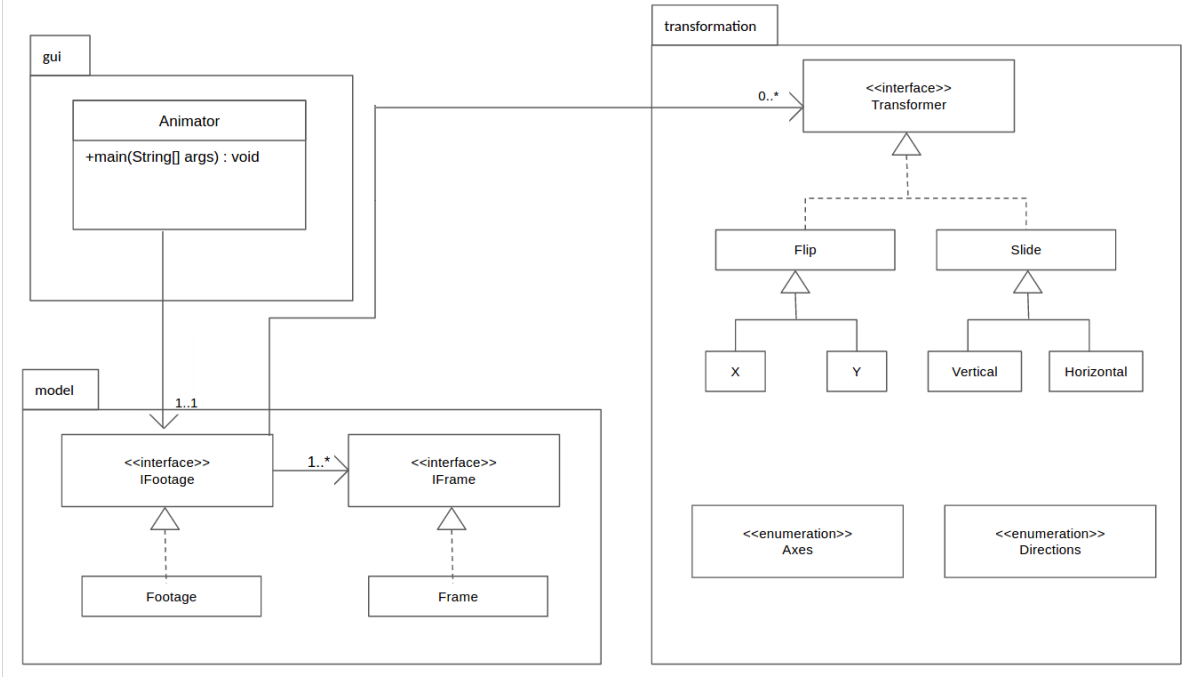

Figure 2: Blockmotion Class Diagram

2.2 Package - Gui

2.2.1 Animator

The 'Animator' class contains the main method and all of the code required to run the footage animation, menus and JavaFx window. This class contains a mixture of methods that were provided in the brief and methods that I implemented to run the menus. I included several helper methods in order to shorten the switch statements within the menu methods, for example I included load and save methods that catch IO Exceptions and deal with bad input. The Animator class has an association with the Footage class in order to load, save and edit a given footage.

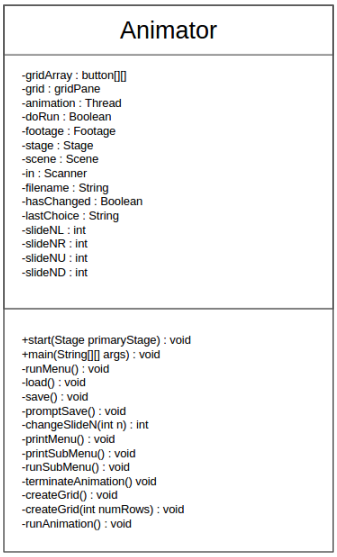

Figure 3: Gui Package Class Diagram

2.3 Package - Model

2.3.1 Frame

The 'Frame' class implements the 'IFrame' interface given in the template code. This class represents a single frame in a footage, therefore each footage class requires at least one frame class. Each frame object has a 2D array of chars that is used to store each 'pixel' of the frame, this allows the frame to be represented as a matrix and manipulated as such during transformation.

2.3.2 Footage

The 'Footage' class implements the 'IFootage' interface. This class is associated to the frame class due to a given footage consisting of one to many frames, this relationship is represented by an array list of frames within the footage class. Using this array list individual frames can be easily indexed and edited, thus simplifying the load, save and transform operations. The footage class also has an association with the transformer classes, this is represented by a transform method. This method takes a transformer class as a parameter and applies that transformation to each frame individually using a foreach loop.

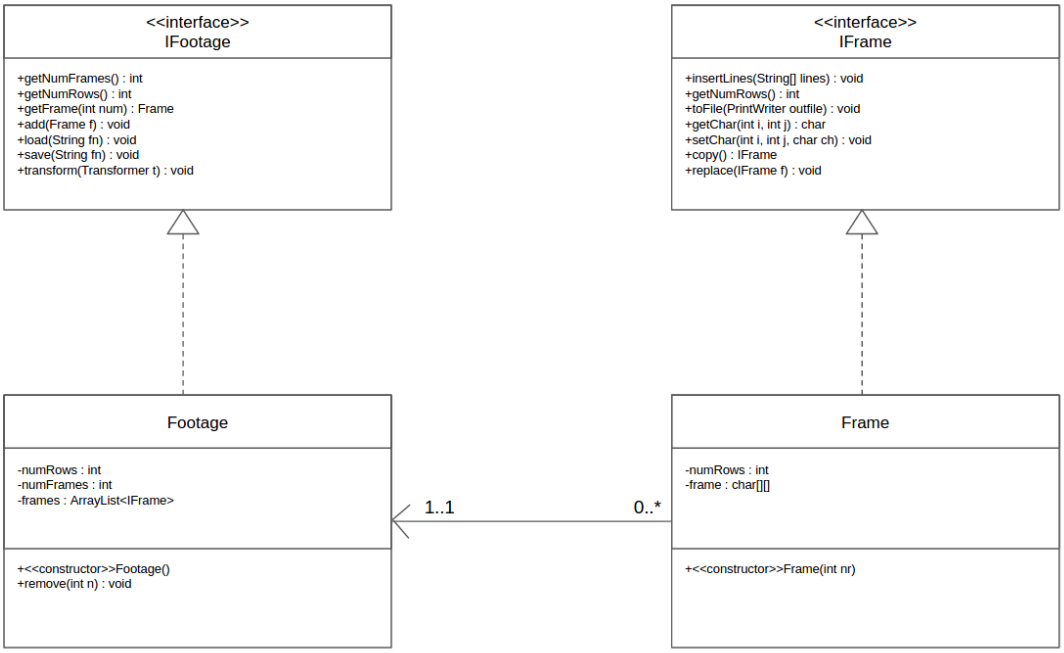

Figure 4: Model Package Class Diagram

2.4 Package - Transformation

2.4.1 Slide

The 'Slide' class implements the 'Transformer' interface and is a base class to the directional classes. 'Slide' contains a 'transform' method which is inherited by both subclassess, the direction of movement is declared in the constructor of the subclasses. For each sublcass there can be two directions, for example horizontal movement can either be left or right, this is represented by either a -1 or 1. Below is the pseudo-code for the sliding algorithm.

Figure 5: Pseudo-Code for sliding algorithm

```
1
2                   copy frame
3                   for all positions in the frame do
4                   new horizontal position = old horizontal position + (number
                        to slide by * direction of horizontal movement)
5                   new vertical position = old vertical position + (number to
                       slide by * direction of vertical movement)
6
7                   if new horizontal position >= width of screen then
8                   new horizontal position %= width of screen
9                   endif
10
11                  if new horizontal position < 0 then
12                  new horizontal position += width of screen
13                  endif
14
15                  if new vertical position >= height of screen then
16                  new vertical position %= height of screen
17                  endif
18
19                  if new vertical position < 0
20                  new vertical position += height of screen
21                  endif
22
23                  assign new horizontal and vertical positions to the copy of
                        the frame
24
25                  end loop
26
27                  replace original frame with the copied frame
```

2.4.2 Vertical

The 'Vertical' class is a subclass of 'Slide', it is responsible for sliding the footage vertically. It has a constructor that specifies the direction of movement (Up or Down) then assigns the relevant value to the 'transform' method in the 'Slide' class It uses an enum type to specify the direction.

2.4.3 Horizontal

The 'Horizontal' class is a subclass of 'Slide' and is responsible for sliding the footage horizontally.

2.4.4 Flip

The 'Flip' class implements the 'Transformer' interface, it deals with all rotational transformations. Similarly to the 'Slide' class it has two subclasses, one for each axis of rotation.

2.4.5 X

The 'X' class is a subclass of 'Flip', this class is responsible for flipping the footage vertically.

2.4.6 Y

The 'Y' class is a subclass of 'Flip', it is responsible for flipping the footage horizontally. Below is a sequence diagram that represents the 'flip horizontal' transformation.

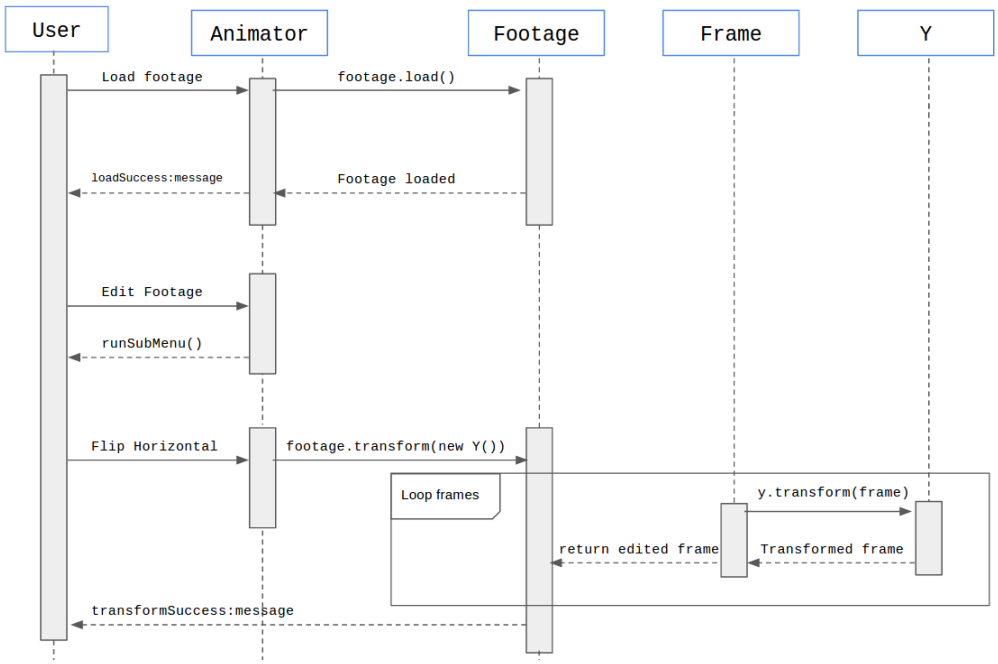

Figure 6: Sequence Diagram for Horizontal Flip

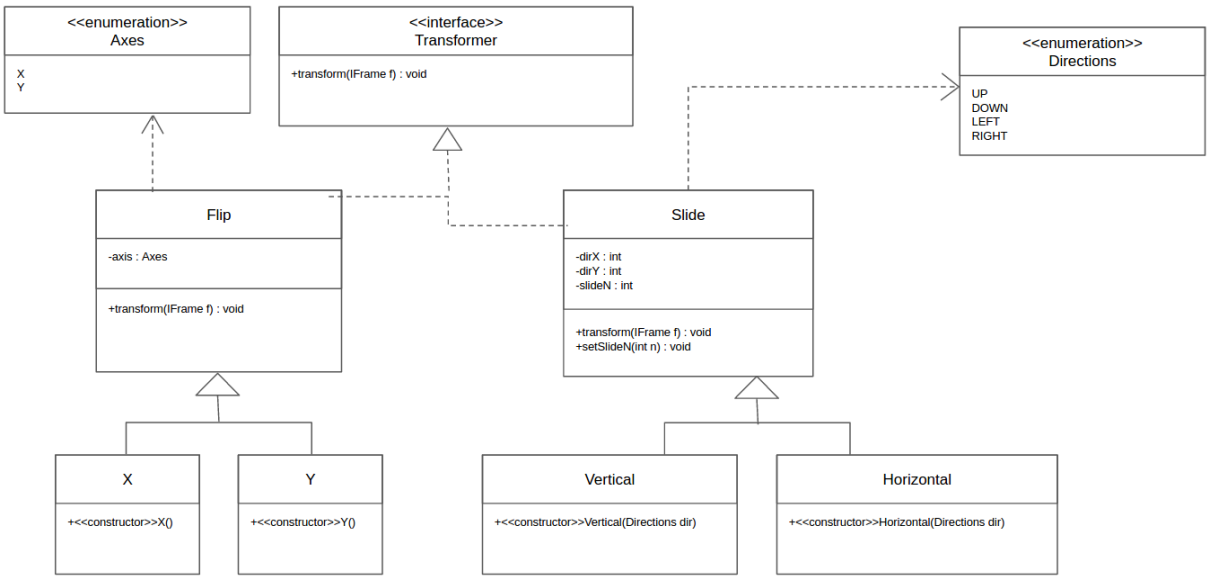

Figure 7: Transformation Package Class Diagram

3 Testing

Throughout the development of the program I made a conscious effort to test each feature thoroughly immediately after implementing it. In order to test the program's functionality I used both the test data provided in the template and created two files of my own. Creating the extra test data was necessary to test the two new colours I added support for. I also drew up a test table of all the program's functional requirements, this allowed me to keep track of what was functional and what was yet to be finished. Below I will discuss each feature that was implemented and provide screenshots of the program running, this includes extra features that i implemented for flair.

3.1 Main Menu

The menu was the first feature I implemented, it functions as expected and can be exited using the 'Q' option.

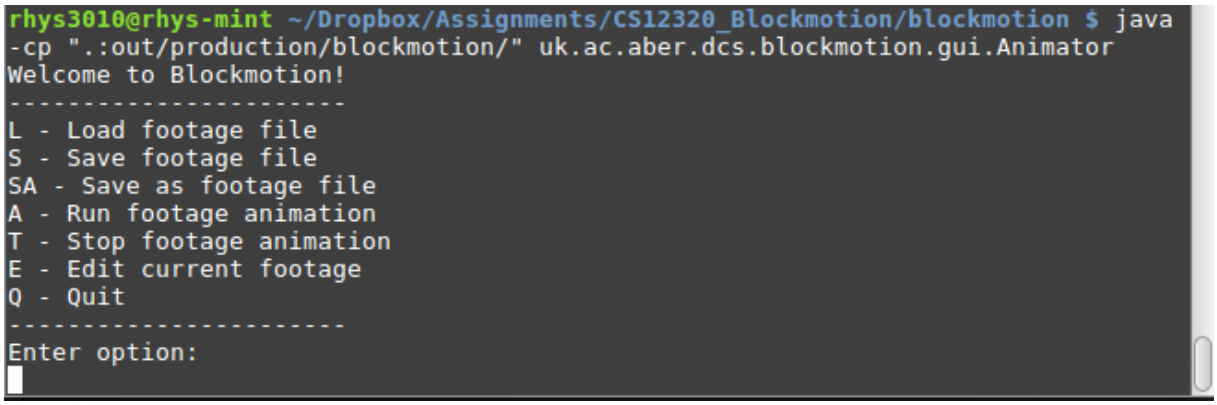

Figure 8: The Main Menu

3.2 Loading and Running Footage

Both these features are fully functional, loading the footage handles any bad input or IO Exceptions. When a footage file is successfully loaded it opens a JavaFx window of the appropriate size, the user can then run the animation and it is displayed on the JavaFx window. Below is the test file 'aber.txt' running.

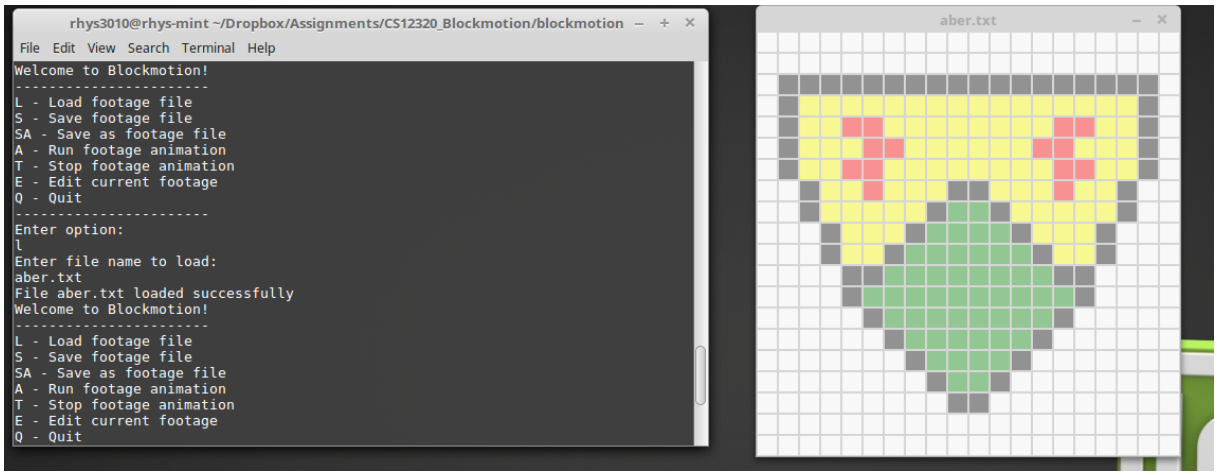

Figure 9: Loading and Running the Footage

3.3 Edit Footage

This feature is fully functional, upon selecting 'E' at the main menu a new menu is entered with several transformation options.

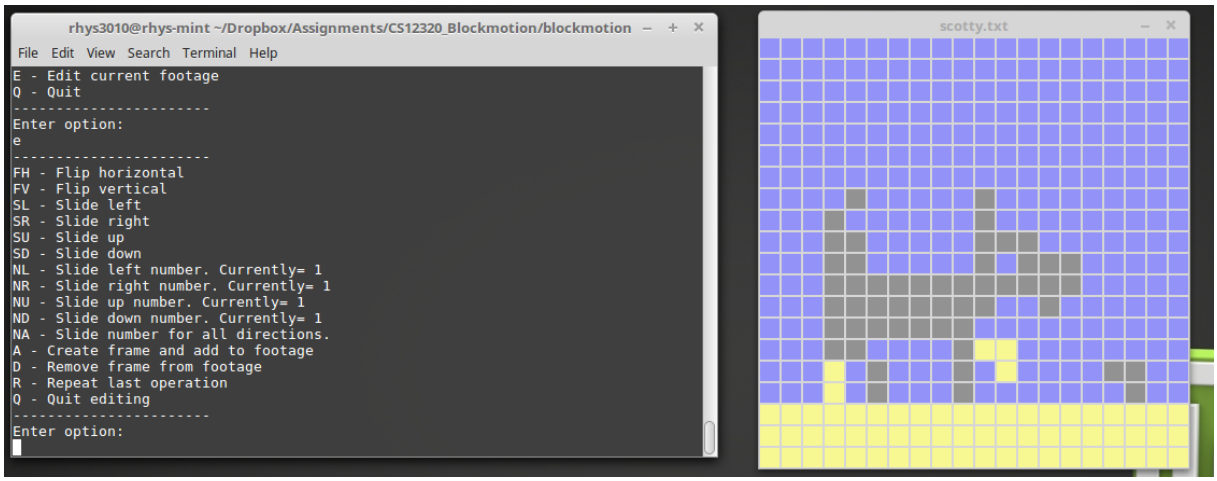

Figure 10: Edit Menu

3.4 Flip Vertical

This feature is fully functional and behaves as expected.

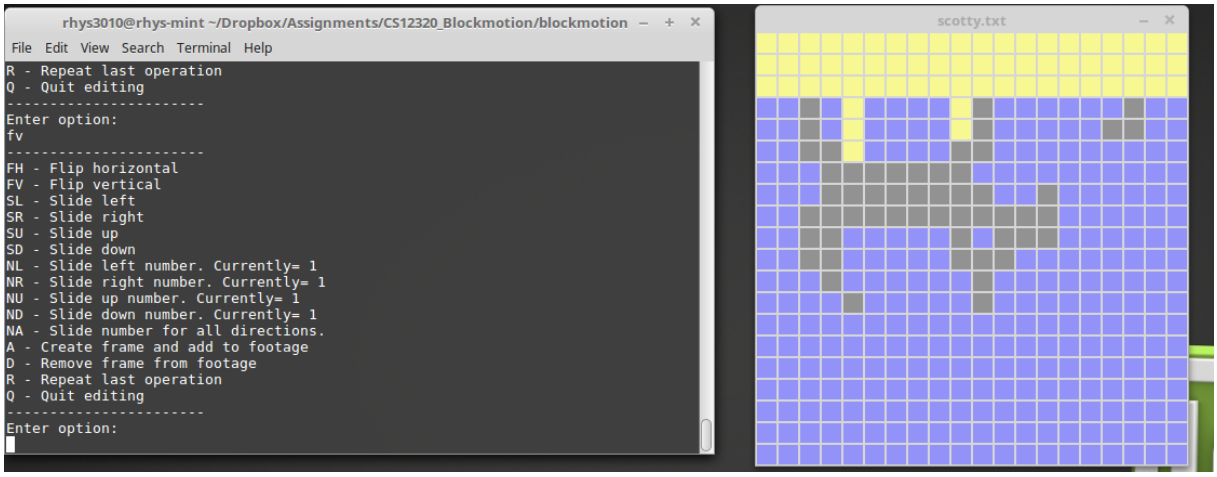

Figure 11: Flipping the Footage Vertically

3.5 Flip Horizontal

This feature is fully functional and behaves as expected.

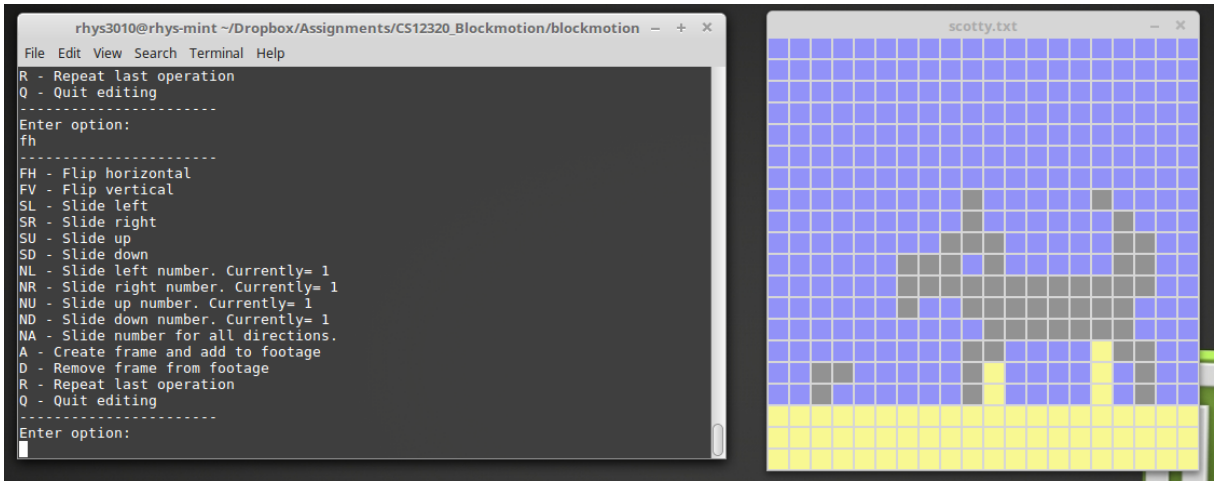

Figure 12: Flipping the footage Horizontally

3.6 Slide Up and Slide Left

Both these features work fully, as seen in the screenshot below the footage was moved up by 1 and to the left by 1. Both features also work independently.

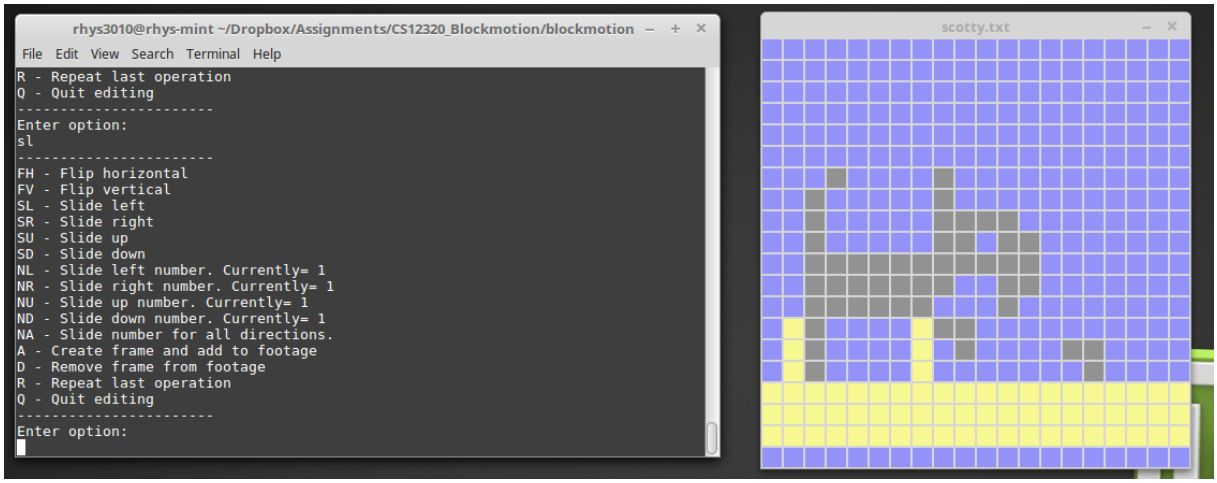

Figure 13: Sliding Up and Sliding Left

3.7 Slide Down and Slide Right

Both these features work as expected and present no issues.

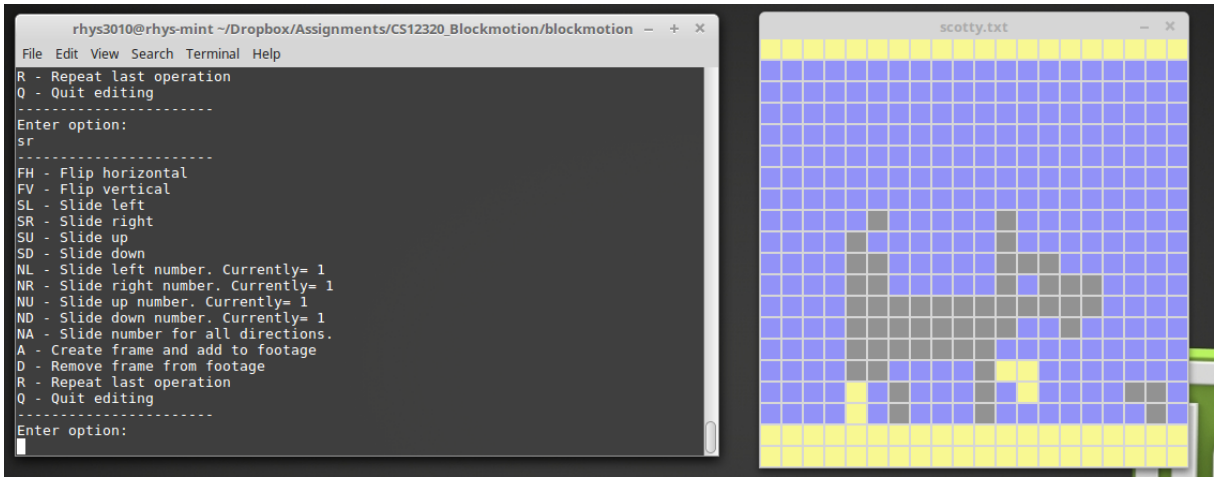

Figure 14: Sliding Down and Sliding Right

3.8 Slide Numbers

All slide numbers can be set and function properly.

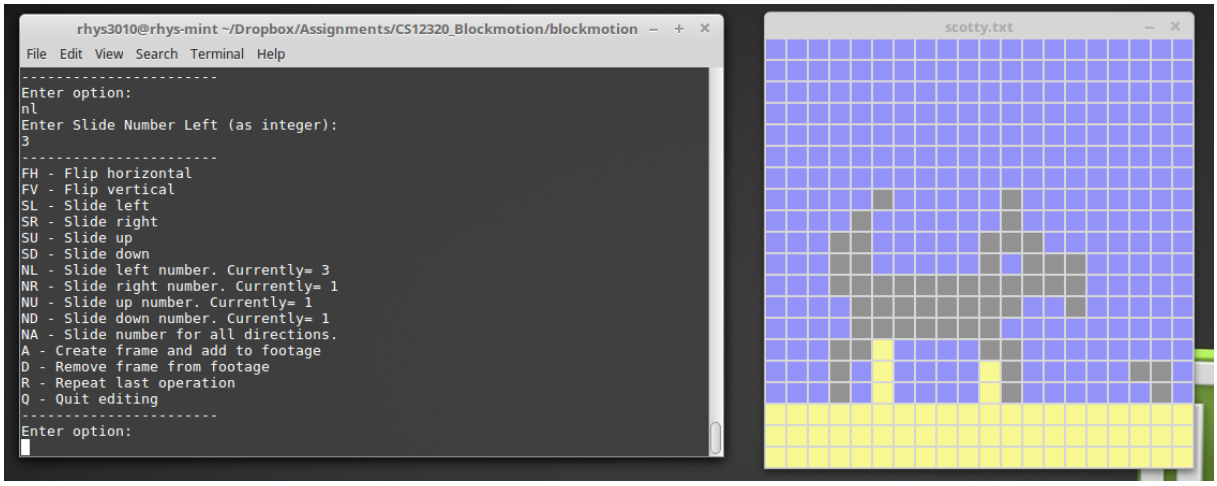

Figure 15: Changing the Slide Left Number

3.9 Wrap Around

When the footage is moved off screen it seamlessly wraps around to the other side, this works both horizontally and vertically. I tested this feature extensively.

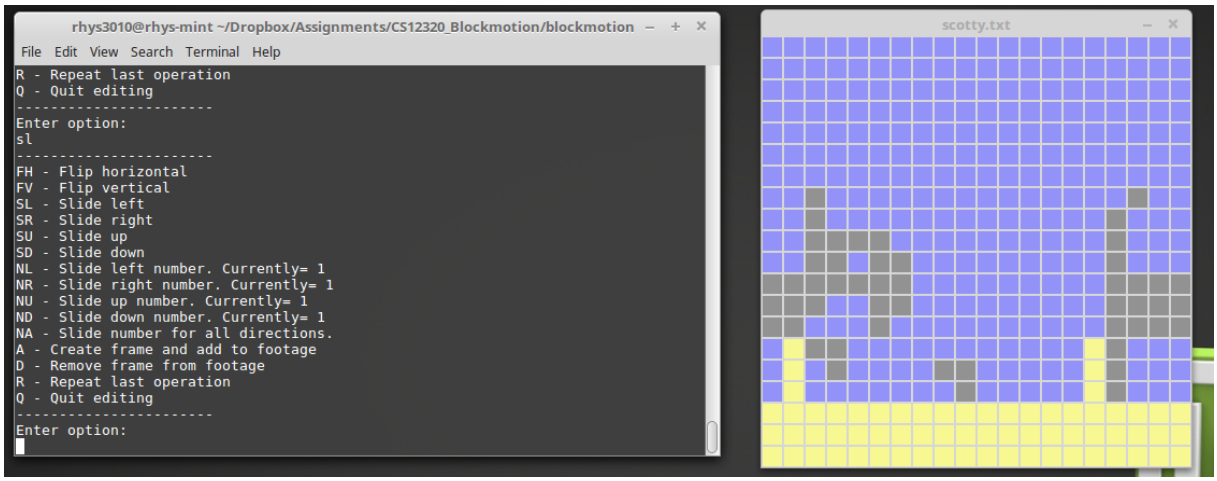

Figure 16: Wrapping Around Horizontally

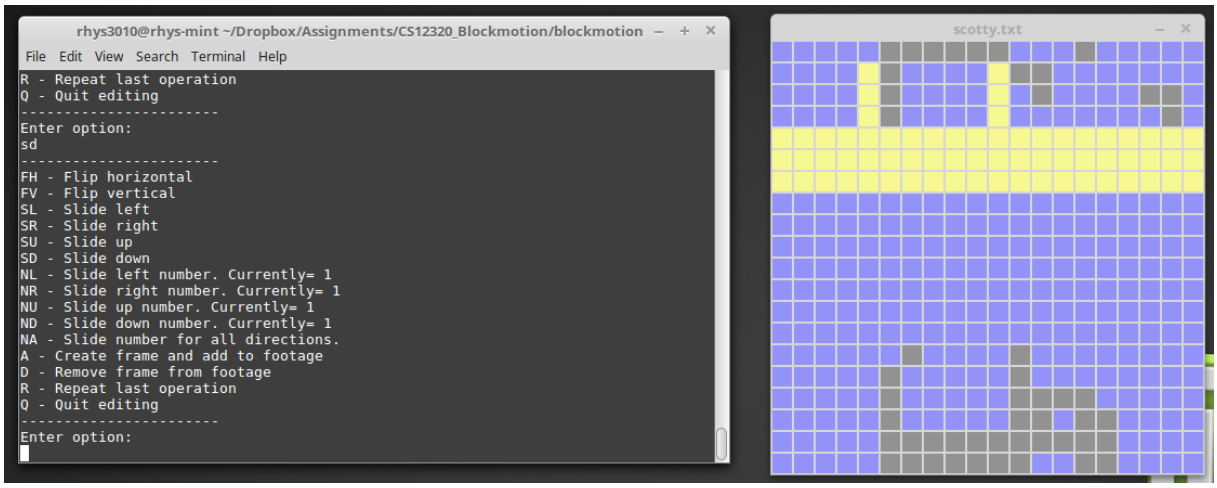

Figure 17: Wrapping Around Vertically

3.10 Redo Last Operation

This feature works as expected and is reset whenever the edit menu is exited.

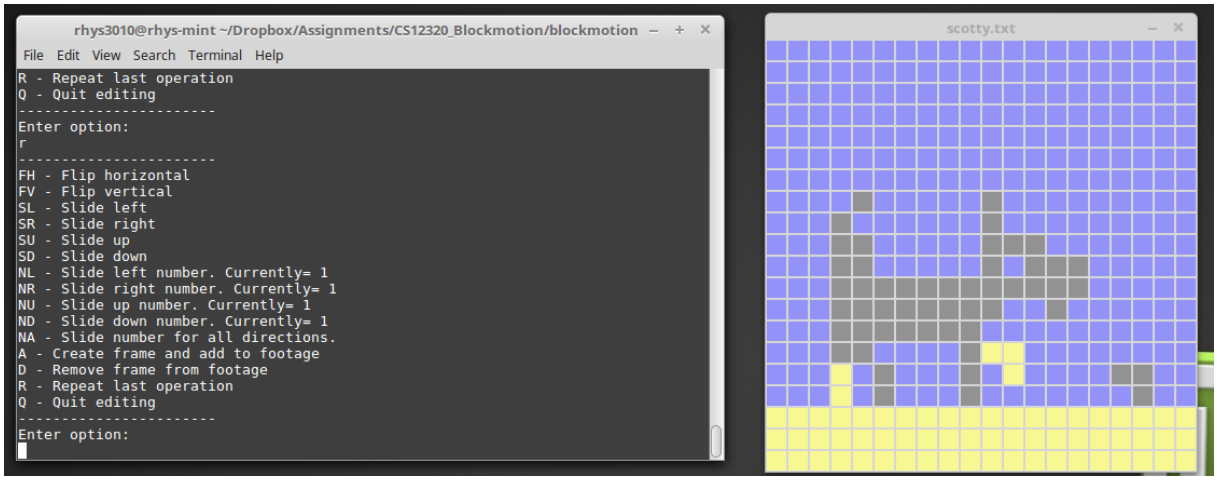

Figure 18: Redo Last Operation

3.11 Change all Slide Numbers

For convenience I also chose to add an option to change all the slide numbers. This feature works as expected.

Figure 19: Changing all Slide Numbers

3.12 Saving and Saving As

Both these features work as expected. In order to test this feature I loaded a test file, saved it under a new name and reloaded it. As the screenshots below show, the user is also presented with a prompt to save the footage if any changes were made.

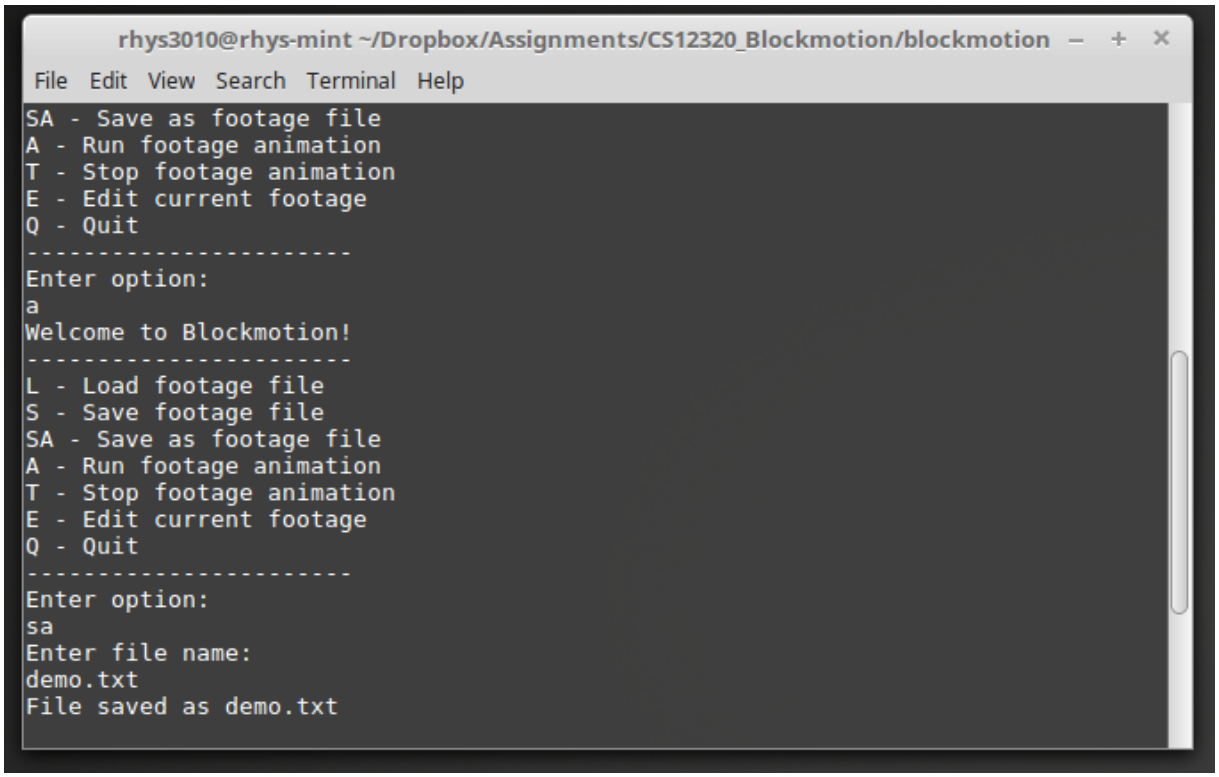

Figure 20: Saving a Footage File

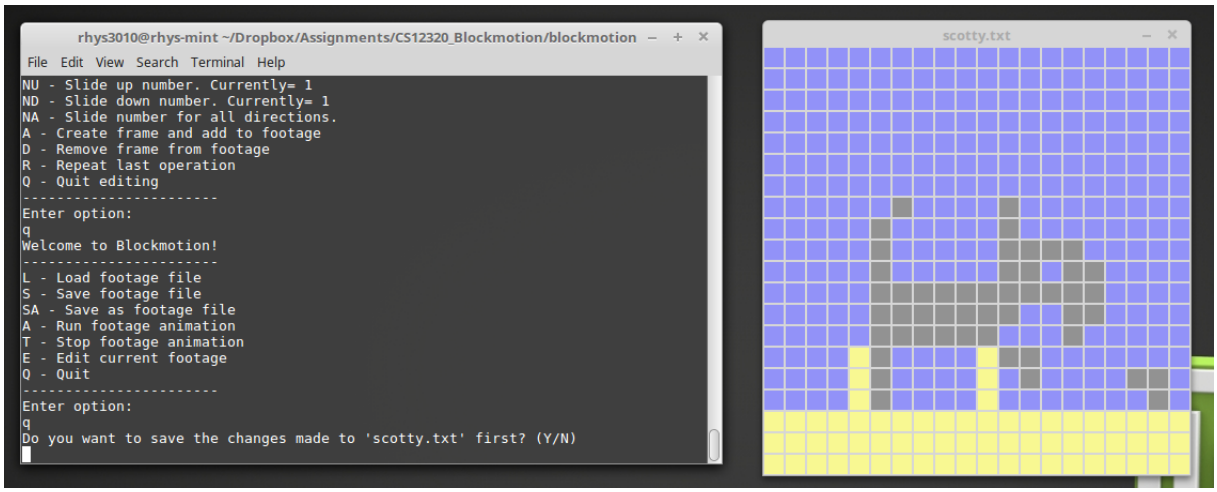

Figure 21: Prompting to Save Changes

3.13 Colours: Red and Green

As an extra feature I added support for two new colours, in order to test these I had to create a new file as test date. I chose to make the University logo, to create the footage I used the conditional formatting feature in Google Sheets so that I could create the frame needed for the text file.

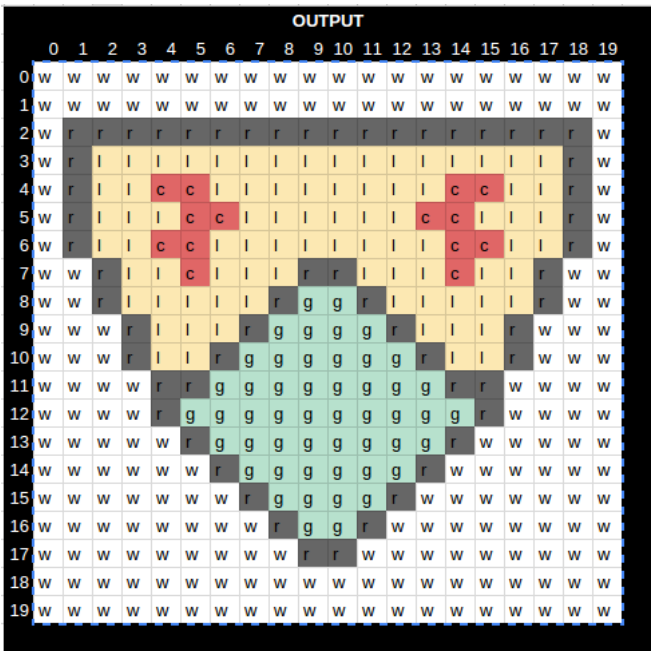

Figure 22: Creating the Footage

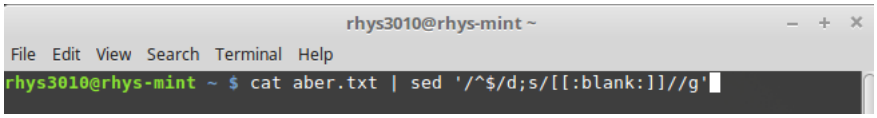

Figure 23: Removing all Whitespace

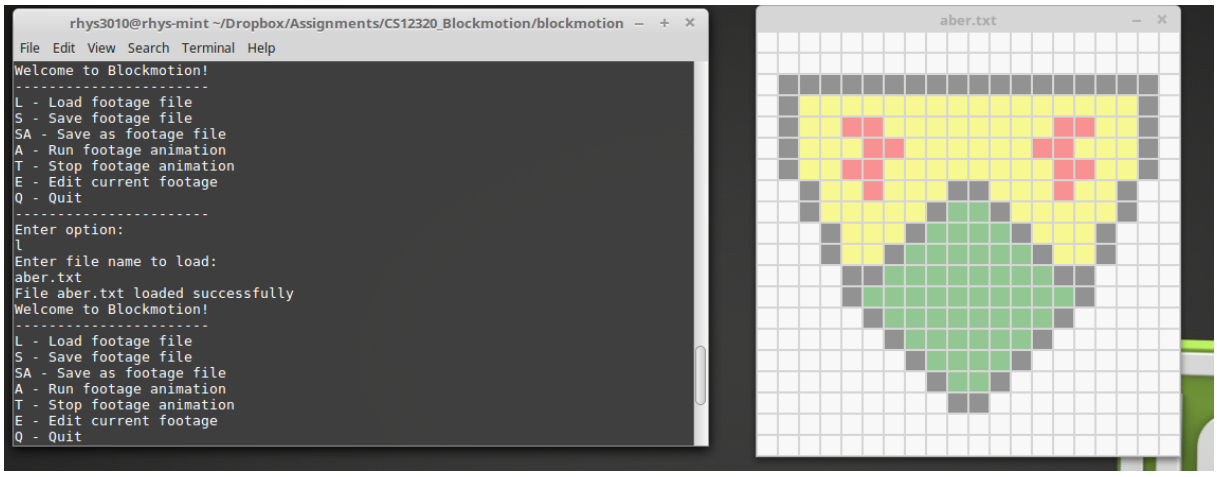

Figure 24: Running the Footage

3.14 Add Frame to Footage

The edit submenu supports the ability to add a frame to the footage without manually editing the footage file. The program reads the user's input line by line in the same way as the footage file (bottom up). In order to test this feature I used the file 'spotty.txt' due to its small dimensions and simple pattern. This feature works fully.

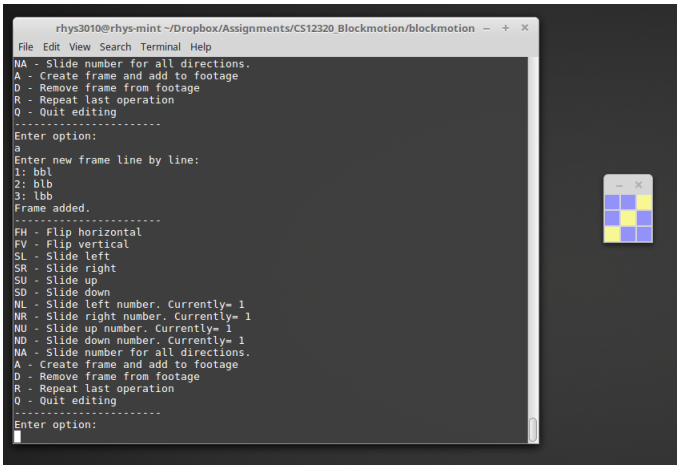

Figure 25: Adding a New Frame

3.15 Remove Frame from Footage

The edit submenu also supports the ability to remove any frame from the footage. Testing this was very straightforward. I simply loaded the 'scotty.txt' test file and removed a few frames, then checked the footage file to verify the frames were deleted.

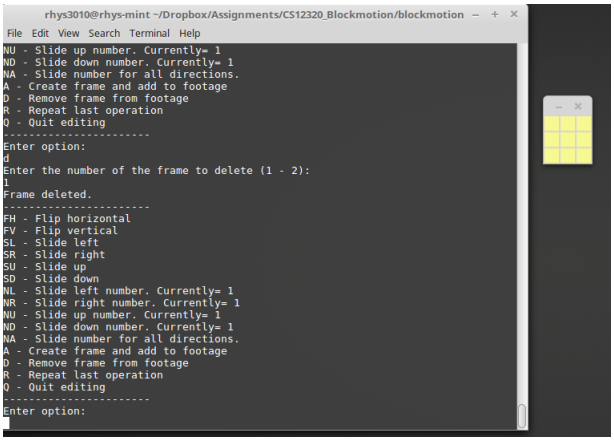

Figure 26: Deleting a Frame

4 Evaluation

4.1 Functional Requirements

In order to begin solving the assignment I familiarised myself with the Interfaces and Animator Class provided in the template, having done this I began following the steps suggested on the Brief. I started by implementing a main menu, to do this I simply ran the menu with a switch statement inside a do-while loop. I then began implementing the Frame and Footage classes, using empty methods to satisfy the interface, I also included the required instance variables. I implemented the load and save features first, they both proved to be relatively straightforward to implement. I used the same practices as in the MUDGame worksheets by catching IO Exceptions and handling bad user input appropriately. The only problem I encountered was that the footage was being loaded upside-down, I quickly realised that the footage had to be loaded from the bottom up. I then implemented the the JavaFx aspect of the program, this was fairly easy as almost all of the required code was provided in the template. I simply replaced the parameter for the 'createGrid()' method with the correct instance variable and made sure to call the relevant methods from the menu. The only difficulty that arose from this stage of implementation was a misunderstanding on my behalf, I spent some time trying to iconify the program but later discovered this was no longer a requirement.

Having successfully implemented all of the basic features of the main menu I then began creating the transformation submenu. I implemented this submenu the same way as the main menu, from a private method called 'runSubMenu()'. Having been given the opportunity to reflect I feel like implementing this menu in the Animator class wasn't the best decision, it made the class very lengthy. Once the edit submenu was fully functional I had to implement each transformation, this was the trickiest part of the Assignment. I first implemented the Transformer interface by creating two classes 'Slide' and 'Flip' I implemented the Flip transformation first by referring to my lecture notes on matrix manipulation and writing the algorithm in pseudocode. Once I was happy with the functionality of both algorithms I implemented them in their respective classes. It was then time to implement the sliding transformation, during my first attempt at writing the algorithm for sliding it seemed very straightforward. I implemented the algorithm that I had written and after testing it for a short time I quickly realised there were many problems, I failed to consider the addition of slide numbers and the footage's behaviour when it went off the screen. Having realised my errors I was able to rewrite and implement the transformation algorithm properly. Implementing the slide numbers was straightforward and simply required the addition of a single method to the Slide class.

The 'redo last operation' feature presented some difficulty to begin with as it would itself become the last operation, after proper planning and in-depth testing I was able to implement it and it is fully functional. The final feature to implement was the support for prompting the user to save the footage file if any changes had been made, the majority of the time spent implementing this feature went to testing, overall it was reasonably straightforward to implement.

4.2 Non-Functional Requirements

As mentioned in the introduction I also implemented additional features that were mostly for convenience. In order to add support for the colours Red and Green I simply added the two colours to the runAnimation() method, I also added the letters 'c' and 'g' to represent the colours. I also added the ability for users to change the slide number for all directions, this option is listed as 'NA' in the edit sub menu, In order to implement this I simply prompted the user for a number and assigned it to all slide numbers. The final two additional features allow support for creation and deletion of frames directly from the menu, these were fairly straightforward to implement. I simply accessed the frames ArrayList within my Footage class and removed or added the relevant frames, the most challenging part of implementing these features was dealing the exceptions and handling the user's input properly.

4.3 Conclusion

Overall I am very happy with the outcome of this project. I would have liked to implement some sort of GUI but as my experience with JavaFx is very limited I felt my time was best spent ensuring the required features are implemented correctly. I feel that I deserve an overall mark of roughly 75% - 80% for the project. My documentation includes all required sections and figures, I have fully implemented all of the required features and made my best attempt to follow best practices. I have thoroughly tested each feature implemented and documented the testing appropriately. Although my additional features aren't very complex I feel like I deserve a small proportion of the marks available for flair. Where I feel I should lose marks are for my use of inheritance when dealing with the transformations, although my implementation doesn't contain any code duplication I still feel the attempt was weaker than it could have been.