Blockmation

Carlos Roldan – car23@aber.ac.uk

5th May 2017

Contents

1 Introduction

This assignment required us to implement a program which produces a small ‘animation suite’ so that users can run a footage. A footage is made up of frames. At the same time these frames are made of little blocks which are squares of colour. The program must allow editing of the footage through transformations. These also include reverse the image, turn it upside down and move the image to the any direction (left, right, up or down). The program also must either save or load the animation.

In my project, I have covered all the brief requirements without much difficulty and have added a GUI (User Graphics Interface) as flair, this was done by using JavaFX and was responsible for the clear majority of all the challenges I faced and consumed most of the programming time. The biggest challenge being managing the implementation of transforming the animation, turn the image upside down and reverse it. As a final product, I was very pleased with the program with only a few issues related with the format of the code.

2 Design

The program is split into 2 packages and their classes. General relationships are represented on next page on figure 1, with more detailed descriptions in subsections; Model and GUI package. Given enough time a restructure would have been highly good to split the main class into 2 main subclasses, but due to learning JavaFX as I was going it turned out messy and is quite a large job to clean up

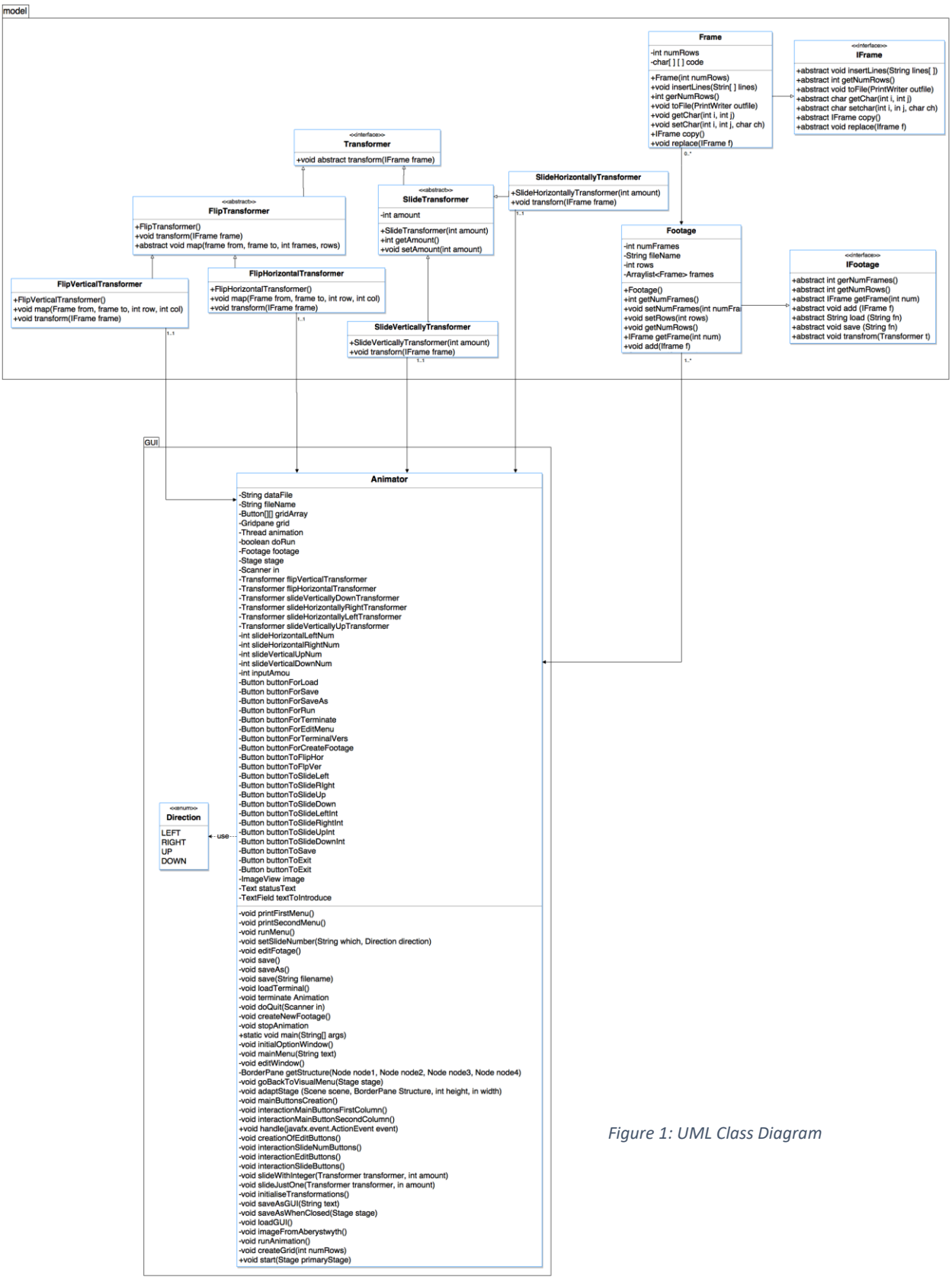

Figure 1: UML Class Diagram

2.3 Package – model

Figure 2 shows all the classes in the Model package with all their instance variables and non-trivial methods, Transformer, IFrame and IFootage are interfaces. FlipTransformer, SlideTransformer are abstract classes.

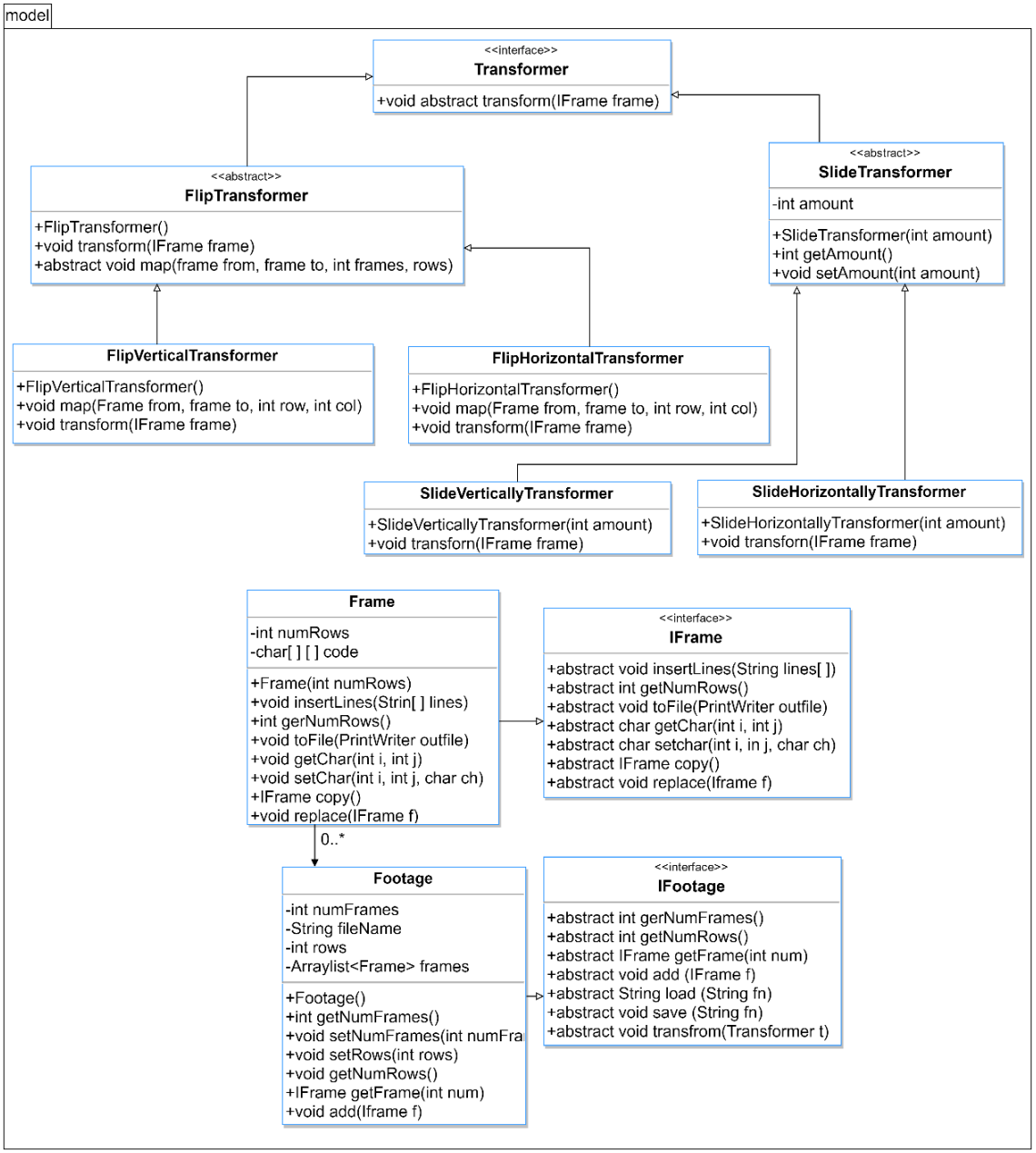

Figure 2: Classes in Model Package

2.3 Sequence Diagram

Figure 3 depicts how a Flip Horizontal process is executed when a footage is transformed. This only includes the horizontal flip. As you can see in figure 3, the sequence starts by a conditional (if there is any footage then...) to perform the transformation. Next, it goes into the Footage class and enters into a For Each loop within the method transform().

Then, it goes FlipHorizontalTransformation class, specifically to the method distribution. Once into this method, a new Frame is created and there are 2 For loops. Then every frame (for each frame loop) is replaced by the new frames.

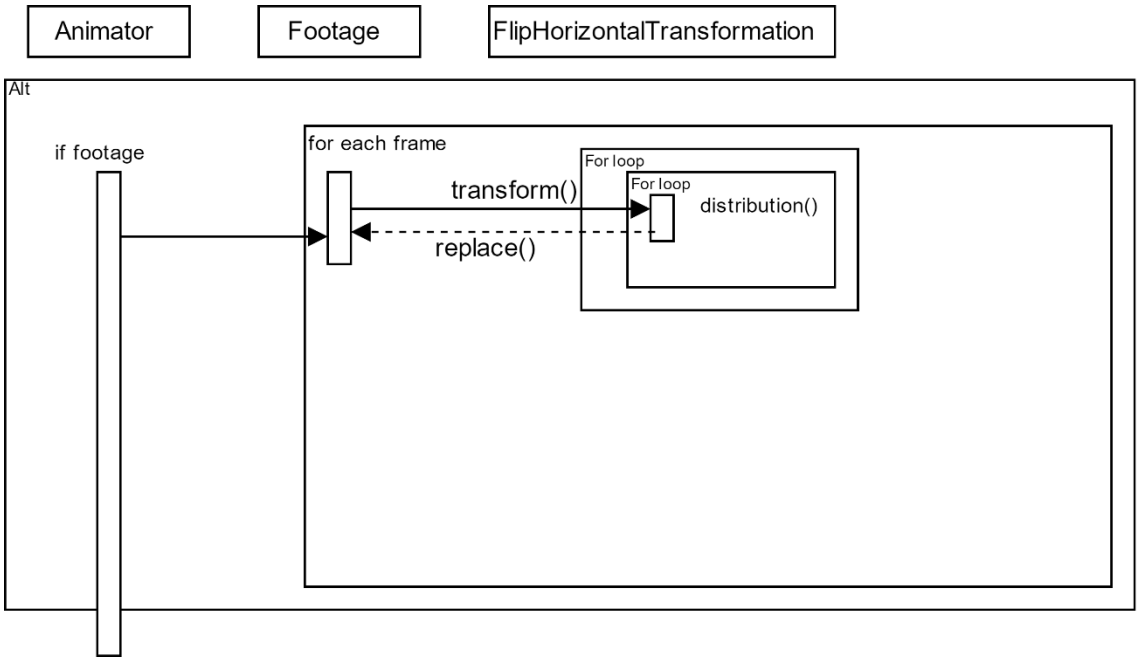

Figure 3: Sequence Diagram from a Horizontal Flip Transformation

2.4 Package – GUI

Figure 4 on next page shows all the classes in the GUI package with all their instance variables and non-trivial methods. Animator is a class where GUI is implemented with JavaFX. Direction is an enum related with the transformation directions which are only 4 constants: LEFT, RIGHT, UP and DOWN.

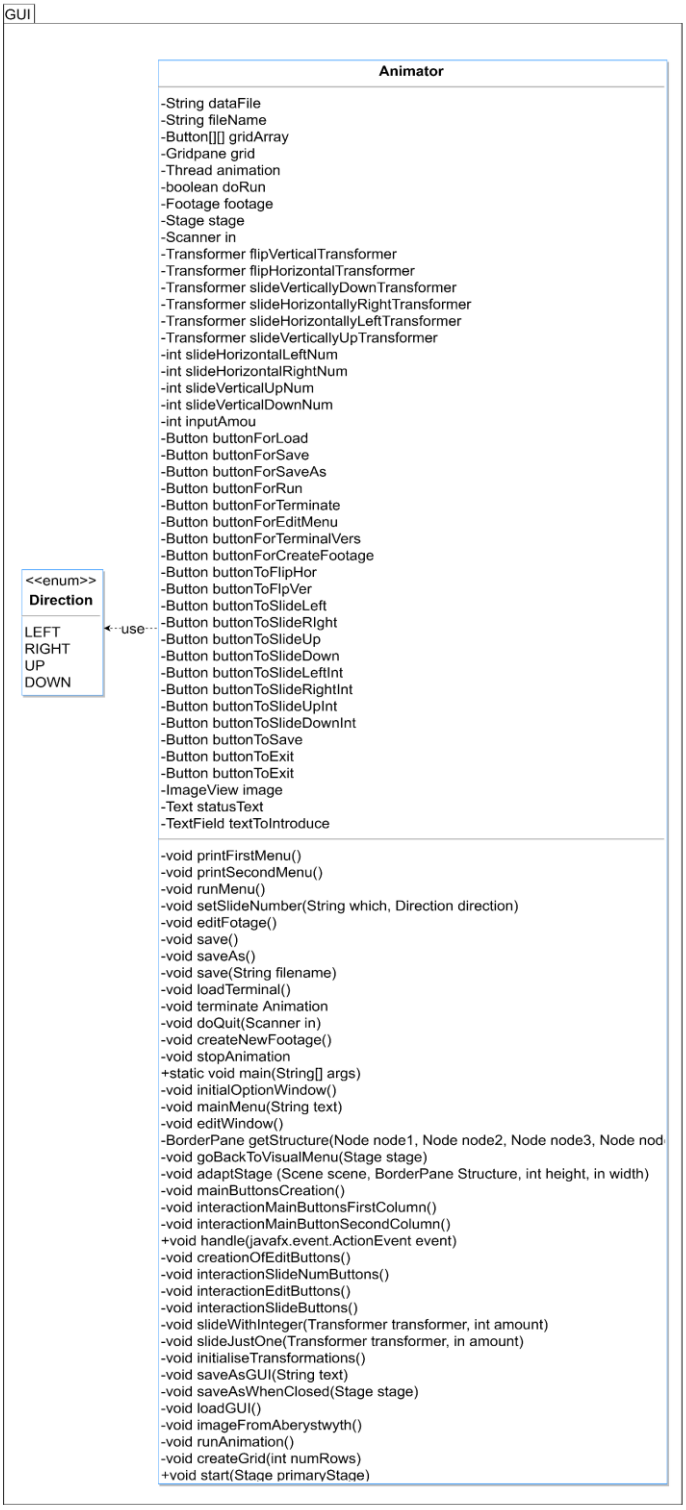

Figure 4: Classes in the GUI package

2.5 Use-Case Diagram

On figure 5 we can visualise the options a user can have within the program. The first two options represent the versions of the program: ***Visual version*** and ***Command Line version*****.** Within either of this two versions the user has the same options*: load a footage, save a Footage, save a footage as a different file name, run the animation, edit the footage* and *create a new footage*.

Within the Edit Footage Option the user can also find more options related with the edition (transformation) of the animation*:* ***Flip the animation horizontally*** or ***vertically*** (this option is extended from an abstract class) and ***slide the footage to any of the 4 basic constant directions*** (left, right, up or down).

Figure 5: Use Case Diagram

3 Testing

Most my testing was done throughout the creation of the program at possible stages where an aspect of the program was complete to the point where an output would be specifically available. Alternatively, I have done a GUI when writing the main functions of the programs such Footage, Frame and the transformations behaviour. The interaction with the program was not just limited to the console, I also implement a graphic version, nonetheless messages about program states used to detect errors would be (mostly logical) output in the console.

3.1 Loading a Footage - functioning

After doing a correctly use of inheritance by implementing the interfaces IFootage and IFrame to Footage and Frame, it was required to work on its respective methods. Writing the method to load the footage it was required to implement another two loading methods in the Animator class. One of them must be GUI (JavaFX) friendly, with a graphical file chooser interface, as you can see on figure 7. The other loading method should to complete the function to load a footage through the command line like on figure 6, where I perform several tests to check the stage where the animation was working correctly.

The output allowed me to run an animation once loaded. By one hand, the first number in the file loaded is the number of frames, on the other hand, the second number is the rows and columns. Then come lines of characters which each of them stands:

- R for dark. Black colour.
- L for light. Yellow colour.
- B for background. Blue colour.

This test also confirmed the footage was loaded as I could transform the animation once it was loaded, independently of being running. This test resulted in a total success.

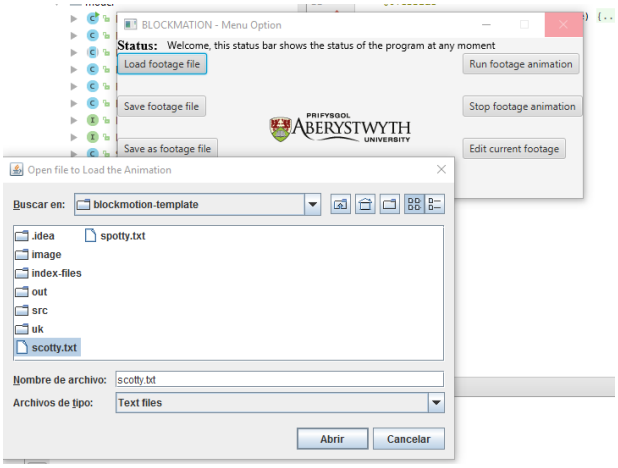

Figure 7: Testing the loading through GUI

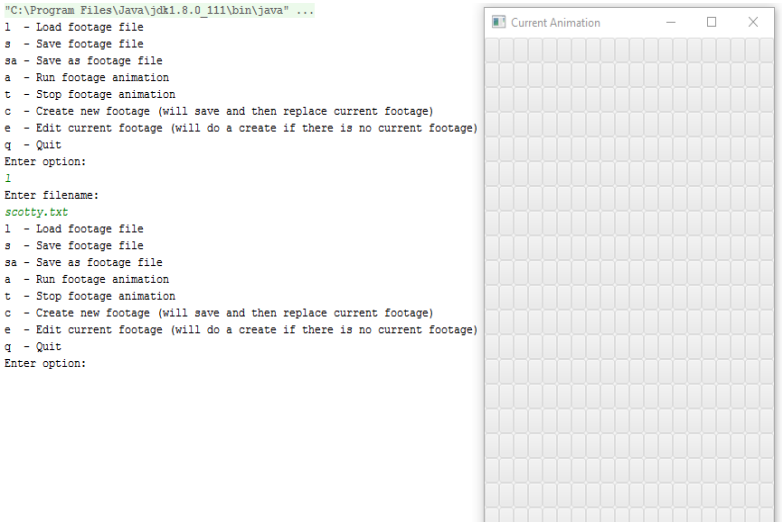

Figure 6:Testing the loading through the Command Line

3.2 Saving a Footage - functioning

After doing a correctly use of inheritance by implementing the interfaces IFootage and IFrame to Footage and Frame, it was required to work on its respective methods. Next, I had to implement 2 saving methods type in the Animator class.

Nonetheless, I decided to call this function when closing the program to give the user the option to save the animation in case he/she forget it to do it.

1. The first saving method was based on saving in the actual animation.
2. The second saving method was based on saving the footage as another animation with a (logically)different file name.

This test confirmed the animation was able to be saved as I could load it over and over again once saved. Also, this test resulted in a total success in both method executions: GUI Version and Terminal Version.

Logically the GUI version required a slightly more effort as an understanding within the area of JavaFX. Figure 8 shows the performance of the Save As method.

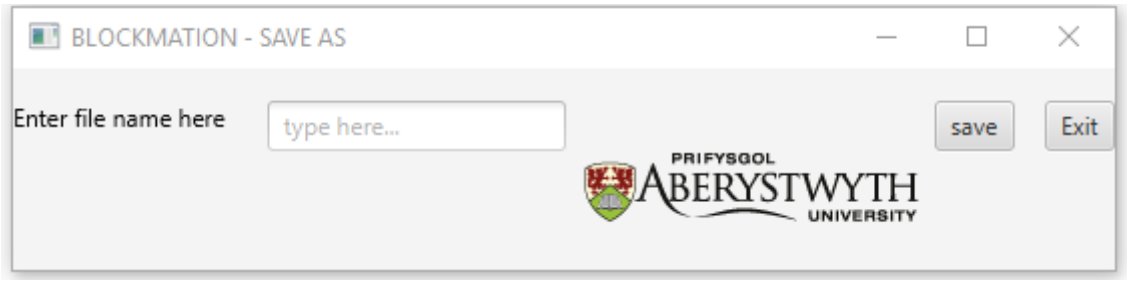

Figure 8:Save As GUI Method

3.3 Running the Animation - functioning

The output when running the animation resulted in a set of frames being displayed working in a functional way. By testing if the animation running was working correctly I alternatively tested the loading function. The animator class (and the running animation method) was GUI, so being especially unfamiliar with JavaFX made a few situations where I had some issues I did not immediately known how to solve.

Nonetheless I achieved this step so that I could run the animation correctly. I also decided to add a tittle when running the animation to give a more professional look. Figure 9 shows the Animation.

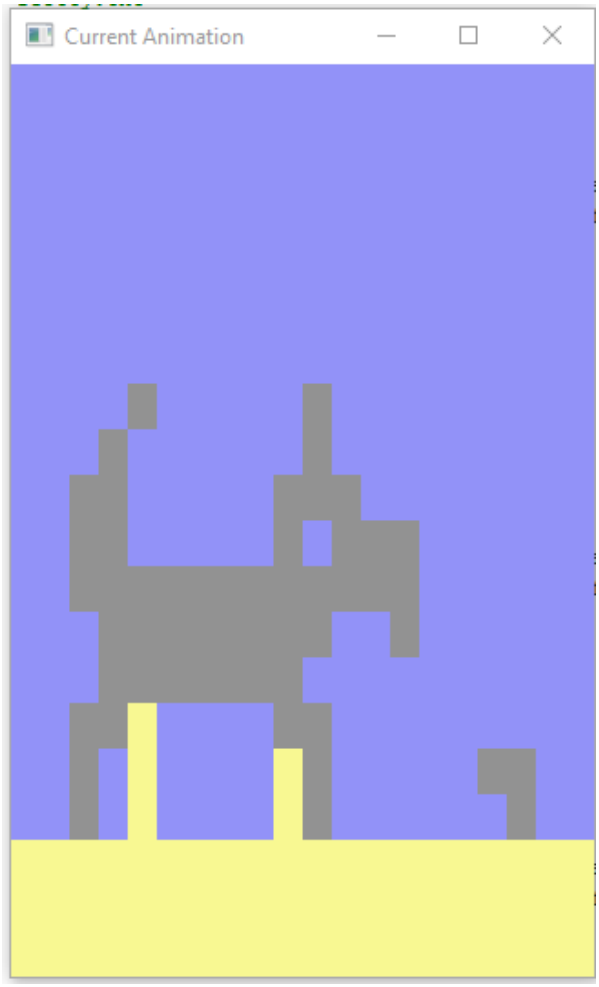

Figure 9: Random frame from the Animation Running

3.4 Transforming the Animation – functioning

Arguably the most complicated function of this assignment is the transforming side, specifically flipping the image (turn it upside downs and reverse it), to test this I not only wanted to see how accurate was the image flipped, I also wanted to check the program did not crash.

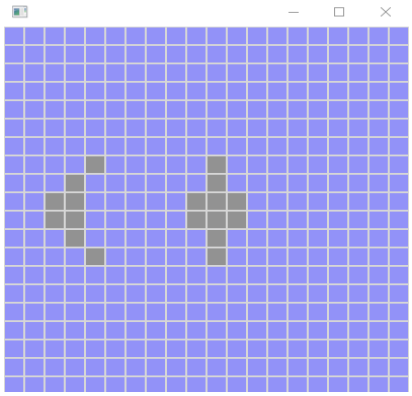

Figure 12: Wrong Output when flipped vertically

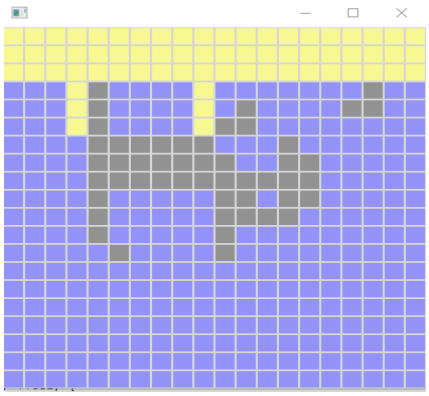

Figure 10: Correct Output when fixed

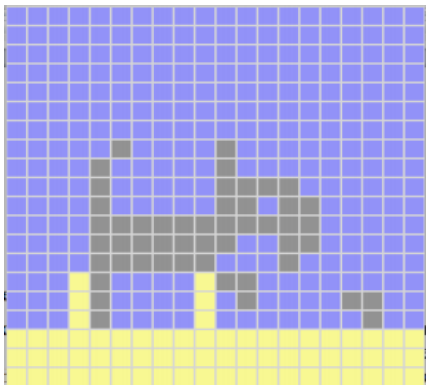

Figure 11: Output without transforming

A good technique said by the lecturer recommended to do the transformations in paper before implementing the code. Therefore, I initially sketch the transformations by hand and I found out that the process was slightly mathematical.

A fault that I had not considered initially was the replacement from the initial frame to the next frame, when setting the new distribution of characters. So that, once when the image was flipped vertically resulted in an undesired output which you can see in figure 12. Once fixed the transformation process we get the desired image conversion, as you can see in figure 10. As additional information, Figure 11 represents the image without having any transformation.

The Edit Menu code for the Command Line version was not suppose a big issue, it was simple due to the accomplishment of an Enum with 4 constants of direction. Figure 13 on next page stands for the performance of the Edit Menu on the terminal version.

Because of the implementation of an additional GUI (Graphical User Interface) I had to write a considered majority of similar methods, such as **save, load, menus, transformations** and more. Thus, the transformation methods oriented to the GUI required softly more determination. By self-learning I introduced buttons, a new stage (different from the main menu) and a text field to take the input from the user in order to perform a slide transformation with the user desired amount, see figure 14 on next page .

```
Enter slide down as an integer
4
fh  - Flip  horizontal
fv  - Flip  vertical
sl  - slideJustOne left
sr  - slideJustOne right
su  - slideJustOne up
sd  - slideJustOne down
nr  - slideJustOne right number. Currently = 4
nl  - slideJustOne left number. Currently = 1
nd  - slideJustOne down number. Currently = 1
nu  - slideJustOne up number. Currently = 1
r   - Repeat last operation
Q   - Quit editing
Enter option:
nu
Enter slide up as an integer
2
fh  - Flip  horizontal
fv  - Flip  vertical
sl  - slideJustOne left
sr  - slideJustOne right
su  - slideJustOne up
sd  - slideJustOne down
nr  - slideJustOne right number. Currently = 4
nl  - slideJustOne left number. Currently = 1
nd  - slideJustOne down number. Currently = 1
nu  - slideJustOne up number. Currently = 2
r   - Repeat last operation
Q   - Quit editing
Enter option:
```

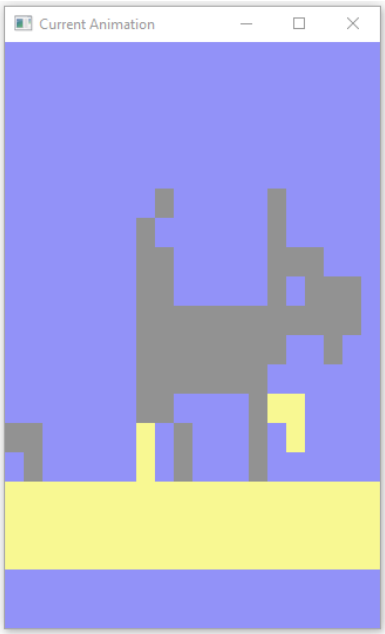

Figure 13: Edit Menu for Terminal Version

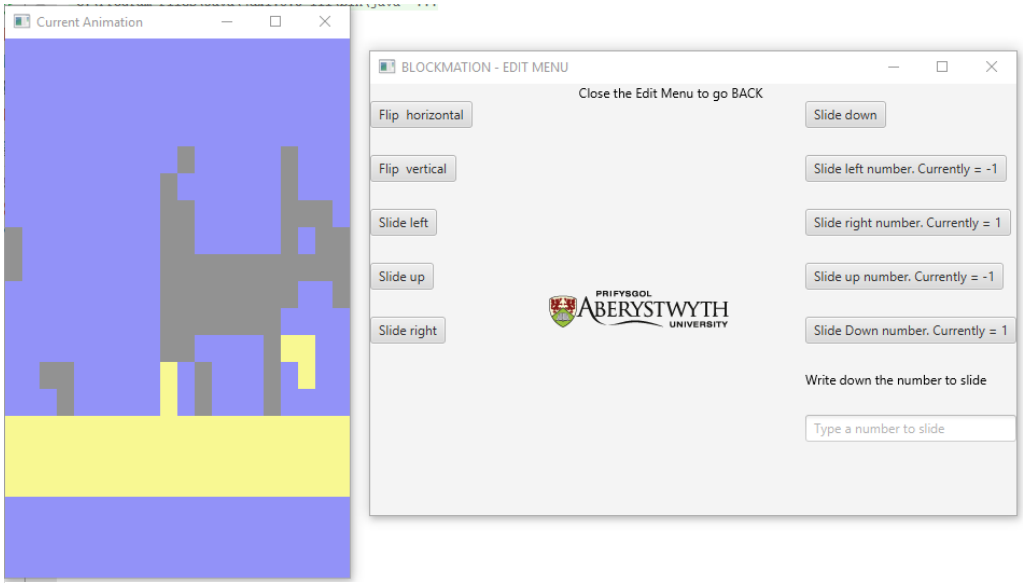

Figure 14: Edit Menu for GUI

3.5 GUI

I have implemented 2 versions within the program: A command line version (default) and a Graphical User Interface. Because I was not too familiar with the JavaFX environment I decided to self-learn by watching tutorials on YouTube. Then, once I felt comfortable with the new understanding based on the events logic of JavaFX I designed an initial option (see figure 15 below) for the user to choose at the beginning of the program between the 2 different versions.

If Choosing the GUI, another Stage was opened with the main menu options, including an image from Aberystwyth University (logo) and a status bar that keeps the user informed about every move. In case of loading a footage, a graphical file chooser will appear for the loading function graphically.

There are 2 more stages: The edit menu, with its functions controlled by more buttons, and the Save As method, which allows the user to write the new desired file name in a text field.

Figure 15: Initial Option Stage

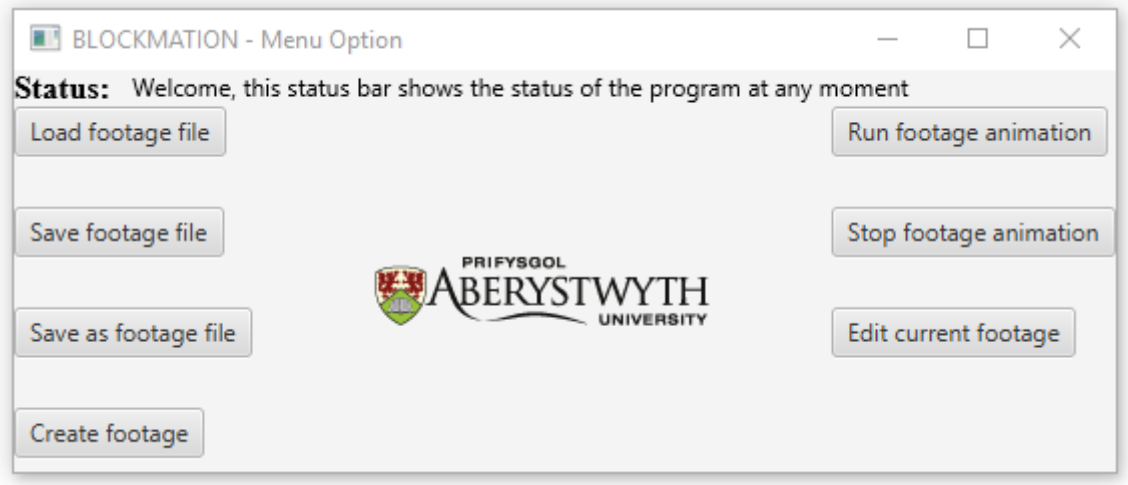

Figure 16: Main Menu Stage

3.6 Memory Usage – Concern

I considered highly important that every method accomplished the rule to not overcome more than 20 lines of code in order to achieve a highly efficient code practise (including code maintainability). As well as not abusing of parameters.

This resulted in a relatively low memory usage as it represents in figure 17 that **does not even exceeds 100MB** and in figure 18 when the animation is running just reaches the 164,2MB.

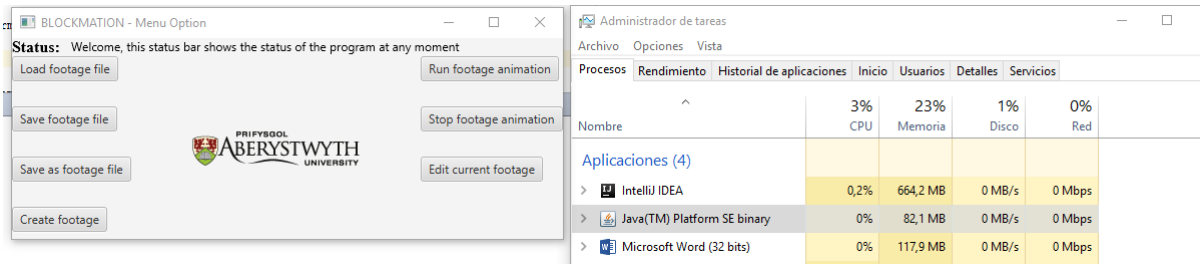

Figure 17: Memory Usage when Running the program in a normal stage

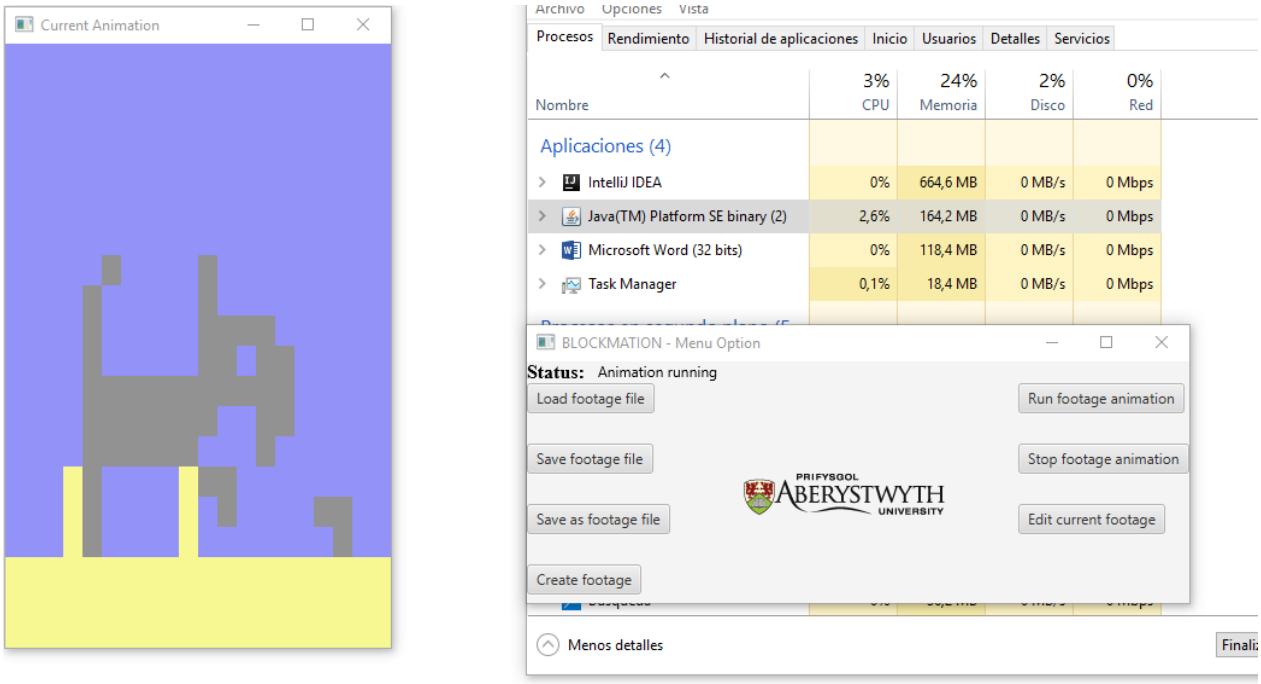

Figure 18: Memory Usage when Running the Animation loaded

4 Evaluation

4.1 Requirements

To begin working on a solution I had to work from the starting point which was analysing problem, which was 3 Interfaces (IFootage, IFrame, Transform) given in the brief which cannot be altered so dictates the workings of the program, so knowing what these 3 Interfaces consists of I produced a class diagram. I first implemented every Interfaces with a class (Footage, Frame, SlideTransformer and FlipTransformer) and then its corresponding methods, which were mostly covered during the academic course. All the required by the brief are fulfilled having been placed correctly in the suited class.

A **Frame** is made out of a two-dimensional array to store the animation code (characters), and the number of Rows.

A **Footage** contains, the number of Rows and a set of Frames to animate.

Transformers are basically a mathematical re-distribution of every Frame code (characters).

At this point, once when the problem analysis was performed and the methods implemented, all that was left was outputting the information in such a way it would be meaningful and easy to understand. It also was the desired time to start with the GUI as part of the flairs.

4.2 Reflection

I am really pleased with the project, as a mark I would say between the range 72%-87%, all the functional requirements are completed and, as flair I feel the GUI is worth a fair proportion of the available 20%. Where I felt I should lose marks are some aspects of the GUI as I was not entirely familiar with JavaFX.

Also in terms of format code, in the class Animator, the GUI could have been in a different class to no overload it. On the documentation side I feel the quality is softly high. I have no pseudo-code examples as looking through my program I could not find anything I felt warranted one. Other than these small flaws I believe the project is to a very high standard.