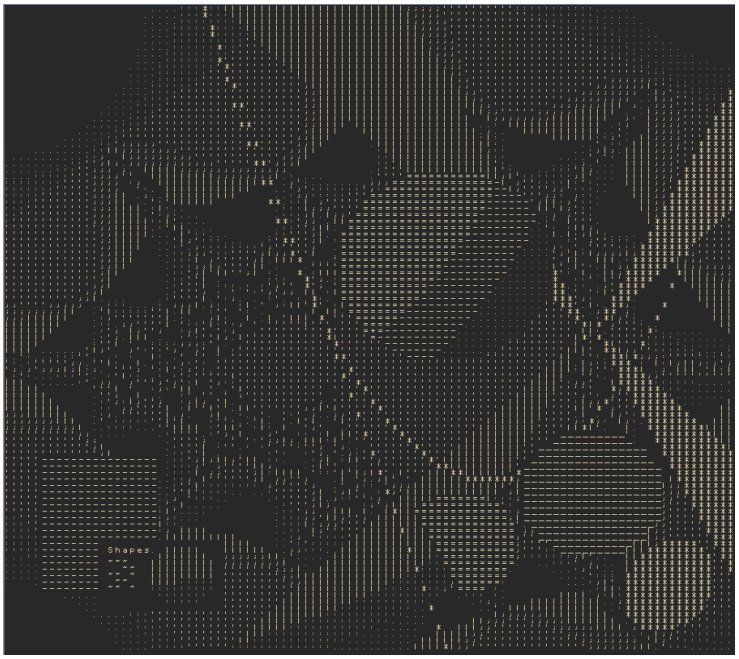
CI401 Introduction to Programming 2020-21
Coursework assignment semester 2
Project technical report.

Project Description

Shapes is an ASCII drawing program with circle, line, curve, rectangle, cross and custom string drawing features. These features are augmented by a selection of other features such as copy and pasting of areas, copy and pasting of “items”, tiling, rotation, mirroring, shading and filling.



There is also a history function and a multithreaded implementation of a Mandelbrot explorer that can be drawn to the screen. The user is able to manipulate the screen using a cursor which can be moved with a selection of practical commands.

The program is able to save the contents of it’s screen to file in a compressed format using a compression algorithm with multithreaded aspects which speed up the rate of compression.

Pictured top-right is a saved file containing all of the drawing functions available in the program; Including, shaded circle, circle, line, curve, semicircle (phase circle), rectangle, cross, oval, freehand, eraser and Mandelbrot. Including also: shaded fill and character fill.

Briefly: how to run the code

The code was written and tested in the following Java environment:
openjdk version "1.8.0_272"
OpenJDK Runtime Environment (build 1.8.0_272-b10)
OpenJDK 64-Bit Server VM (build 25.272-b10, mixed mode)

To run the code, run the Shapes.class class file. This will begin the program and present the user with a list of commands for the program. After reading and understanding these commands the user is free to move the cursor and draw onto the buffer in the way they would like. The Mandelbrot explorer has extra controls for manipulation but these controls are explained elsewhere.

The user can then save the screen to file. There are two ways to do this, firstly typing “save” in the main buffer will save the buffer to savedBuffer.csv. This file acts as a quick-save for the screen and will be overwritten without any prompt next time the “save” command is issued.

If the user would like to save the screen more permanently, then they must enter the file menu by typing “files”. This will present the user with a screen containing representations of the files that have already been saved as labelled thumbnails. Next the user can type “new” which will prompt the user to decide a file name and then save the buffer in the compressed format into this file name. Should the user want to overwrite a file in the file menu, then they can type “save” whilst selecting a file in the menu, In which case they’ll be prompted for a confirmation before proceeding.

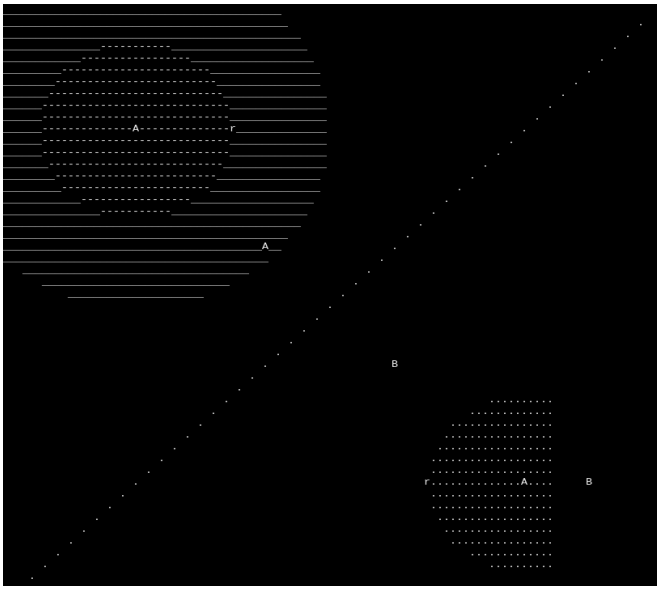
To load a file, the user can either issue the “load” command on the main screen to load savedBuffer.csv, or alternatively they can press the enter key whilst selecting the file in the file menu.

The buffer is represented by a 100x100 grid and so in order for the buffer to present in the correct aspect ratio, it is important to choose the correct font. The ideal font is a square monospace font, meaning the character space of each character is equal to all other characters and the height and width of each individual character space are equal. Alternatively if the height by width ratio of the character space is a whole number then the print method can accommodate by printing extra spaces in-between characters as per the height by width ratio specified in the source in order to pad the output to the correct aspect ratio.

How the code works

Drawing shapes onto the buffer

To display a screen into the terminal, the program uses a two-dimensional array called the buffer to store the values of certain points on the screen. To draw shapes onto the buffer, a function iterates through every point in the buffer and checks whether it meets certain requirements necessary for the function.

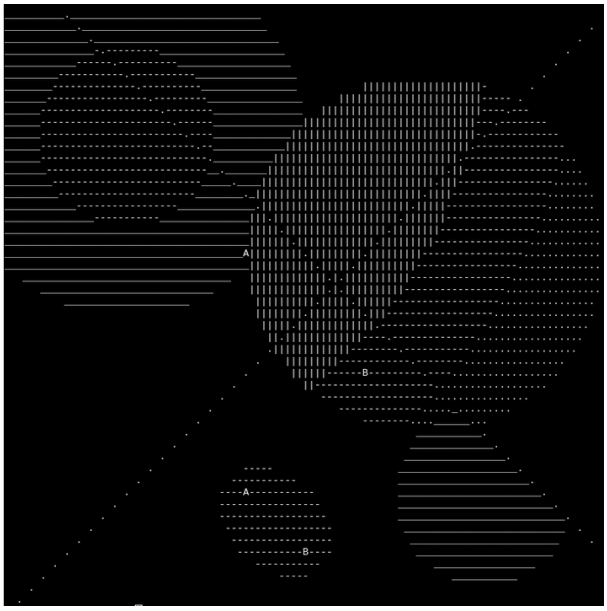


Pictured (right) is a screenshot of some of the first experiments with the functions used in this project to draw shapes onto the screen. All of the equations pictured in this image use Pythagoras as their main calculation.

Reading across the image left to right and downwards, the first letter encountered is A and the next is r. In this case, A represents the origin point of a circle and r represents the radius of the circle. The buffer is iterated through and all the points which have a distance of less than r to A are “coloured” with an arbitrary character.

The next two letters, A and B, are the points used in the calculation for the line in the centre of the image. When iterating through the buffer, the evaluation made is whether the distance for the point to both A and B is equal. This results in a normal-line drawn at the bisector between the points A and B.

The final points r, A and B are a combination of these calculations, where points coloured are within distance r of A, but not on the opposing side of the normal-line drawn at the bisector between A and B. Drawing what in this document will be refereed to as a “phase circle”.



Pictured (left) is a shaded circle and an oval amongst the previously described functions. Shaded circle, similarly to the phase circle, draws a circle but combines this calculation with additional distance calculations for more specific styling of the shape. In this case, distance between 0,0 and the origin point (distance from “light-source”) and distance between the current iterated point and the origin point (distance from “object” alternatively the origin) is calculated for each point and depending on the values of these calculations, colours the point a certain character within the distance r of the origin.

The oval equation checks whether the sum of the distance of a point from the cursor(B) and from the origin(A) is less than (the distance between the origin and the cursor) multiplied by a value “roundness”.

Equations other than the ones here are used in the program, but the basic method for drawing the

equations onto the buffer remains the same for them all. Where iterating through every coordinate, each coordinate is checked for whether it matches the condition specified in an equation.

User control and interacting with the buffer

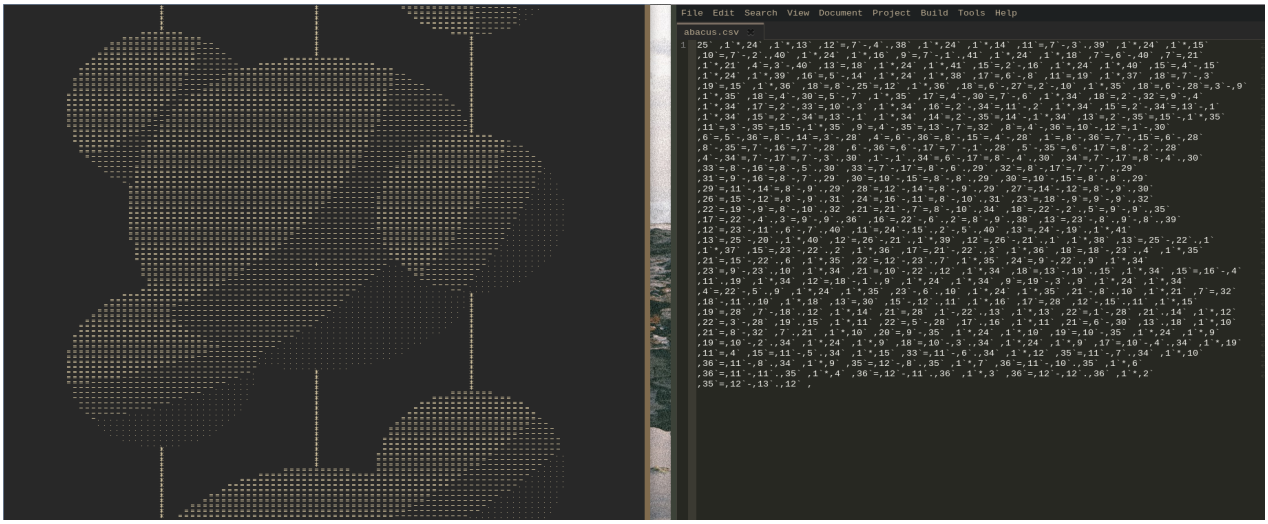
User control requires a cursor for drawing shapes. Illustrated right are three shaded circles with the cursor character drawn in every point that it has passed over, this happened early in the development because the cursor character (represented as '+') was drawn directly onto the buffer, overwriting the information stored at the points the cursor passed. To avoid this, a second buffer, the "work-buffer" was necessary to store temporary changes to the buffer without affecting the information stored in the buffer.

User control sits within a while loop with a boolean condition that maintains it. On every pass of this loop, the work-buffer is initialised (Meaning every character in the buffer is copied into the corresponding location in the work-buffer) and items such as the cursor or a shape are drawn where appropriate into the work-buffer. The work-buffer is then printed to the terminal and progress is paused for user input. Once input is received the input is processed for valid inputs each of which perform some action such as moving the cursor or placing a specific shape.

To draw shapes to the buffer, a three-stage method of initialising the work-buffer, doing work and committing the work takes place. For work to be "committed" to the buffer, the data of the work-buffer is written into the buffer character by character. Changes made in the work-buffer can be discarded during placement control loops (which are sub-loops that sit within the main control loop and are run when a certain input is received in the main loop.) without being committed to the buffer by escaping the placement control loops before any work is committed.

File handling and compression algorithm

The basic concept behind the compression algorithm is that when iterating through the buffer, adjacent pieces of equal information can be represented as a repeat of this piece of information. The original implementation of the compression algorithm would read through every coordinate in the buffer and whilst iterating through, count up whilst the next coordinate was the same as the current. When the next coordinate was not equal to the current, the count and the character that was being counted would be recorded to file.

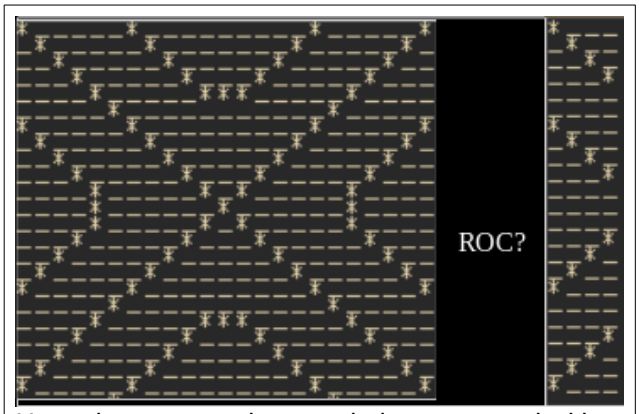


As shown here, iterating left to right downwards through the whole buffer in this way would would reduce adjacent repeats of each character to a number and the character, where each unit of number and character was separated by a comma in a csv file.

However this method was easily broken as a file that contains information arranged in such a way that every other character is different (as large_file.csv in project) will have to encode each character with additional information, resulting in a compressed size that is significantly larger than the input data.

The solution required the compression method to be adapted to compare character groups of different sizes, as previously the size of the compared groups was one. These groups will from now on be referred to as “regions” and the method was named for reference: LESC. LESC aligned all characters on the buffer onto a line (linear) and compared region sizes of increasing size until a pre-determined limiting range was met. (exhaustive search) This method solved the issue caused by large_file.csv and reduced the file sizes of other files such as those with tiled patterns significantly as well.

***



Here, these two regions are being compared with region-overlay-comparison. Consider the region on the left as region (with) index 4 and the right as region (with) index 5.

To compare these two regions, ROC first establishes the dimensions of the two regions. Both of these regions have an index in the buffer, and the index can be converted into a relative coordinate in the buffer at which dimensions can be measured.

Once the dimensions of the two regions are established, then every coordinate in the region with the smallest dimensions is checked against the relative coordinates in the other region. If at any point a character is different, then the result is false. Otherwise the result is true if every compared character is the same.

It can therefore be said in this case, that since every individual character at the corresponding coordinate in region index 5 is the same as in every individual character at the corresponding coordinate in region index 4 then region index 5 can be represented with the information contained in region index 4. Providing region index 4 contains more information than region index 5.

The next stage in the compression algorithm was creating a method that was able to compare regions with two-dimensions. This method can be described as 2DESC or 2 Dimensional Exhaustive Search Compression. Named because the region comparison is two-dimensional and the method runs an exhaustive search of all region sizes it can to find the optimal size. The file sizes using this method were smaller than using the linear (or one-dimensional) method.

The issue with the method lay in the way that the regions where determined as either alike or not alike. In scenarios where the region width was not a factor of the buffer width, there would always be a region at the end of each row of regions in the buffer that did not fit into the buffer. This meant that when the regions where being compared, the region at the edge of the buffer would always not be the same as the one before it because its size was different, even when the pattern is clearly tiled. This interrupted the count every row and meant the compressed files contained more data than needed.

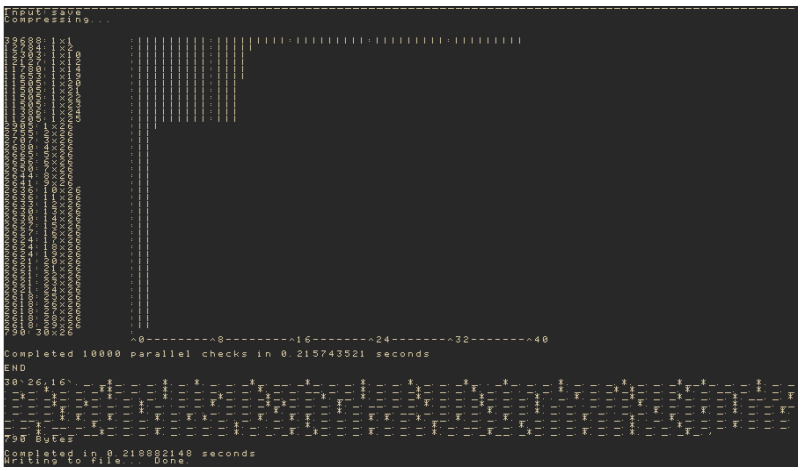
To solve this, a new method called Region Overlay Comparison was used as the comparison condition for determining whether the next region along in the iteration stage was equal. This method, explained left, was able to evaluate regions that overhung the edge of the buffer against regions that did not in a fair way and be able to count past the edge of the buffer.

This is most effective in files created using the tile function, as they can be represented simply by a repeat of a single region. Previously the file would have had to contain information about every row of regions and the compression would have been poorer.

***

The compression scan or compression search works by creating a compression-map. The compression-map is a two-dimensional array of strings where each string represents the compressed buffer that is compressed with a region-size of height and width relative to the height and width of the coordinate in the compression-map.

To speed up the process of compressing with region size, the compression-map is populated by performing the compressions in parallel, achieved using streams. The compression map is then iterated through, checking whether the literal length of each string is shorter than the length of any string it has passed, until the end of the map is reached and the smallest value is determined, the shortest string is then returned and saved to file.



Pictured here is the information output of a compression run on a file. A full compression map is used here. When searching the compression map, “items” are added to the compression graph when a new minimum size is reached.

Once the search is complete the graph is formatted with the scale of the largest value and printed to the terminal. As well as this; the time taken, and the final compressed string is printed to the terminal.

Before region overlay comparison was used as the main method of comparing the regions in the compression stage, a shortcut was possible where the optimal region dimensions could be found by finding the smallest compression size along the first row of the buffer and then checking only the dimensions along the y axis at this smallest x coordinate.

This occurred because each row’s counting was interrupted by a region that differed at the end of each row. Because of this, the pattern in the compression sizes for each row attempted was the same where the features (which could be seen in the compression graph when the code was adapted to record and graph with every item in the compression map) of the pattern scaled proportionally to each other, meaning the scan could skip values that did not have the x dimension of the smallest value in the first row as the dimensions with smallest compression always lay in the same x dimension for each additional y dimension.

This shortcut was no longer completely effective with region overlay comparison for files with certain abnormal features. Features such as large amounts of blank space in a tiled region, which had optimal region dimensions that where skipped in the scan because they did not have x dimensions that matched the optimal x dimension along the first row. Therefore in the final version the code, the compression scan performs a full check of every possible region-size rather than using this shortcut.

***

To decompress these files, a file header containing the dimensions used in the original compression is needed at the start of the file. Without any instruction of how to parse the data in the file, the data contained in each region could not be understood. The read-in method writes each region one at a time into adjacent regions in the buffer, each of an area determined by the dimensions in the header. Without the header, the dimensions that the characters represent are not clear and the data could not easily be written in the correct way.

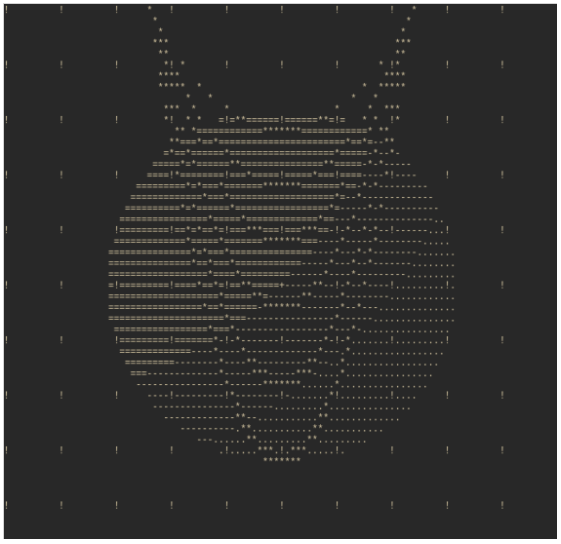
File menu thumbnailer

The thumbnailer component of the file menu works by loading a file into a virtual-buffer (a temporary 2D array with equal dimensions to the buffer).

The virtual-buffer is divided into square regions of equal number to the number of characters needed to make a thumbnail of a desired size. In the image right, each exclamation mark represents the top-left coordinate of each of these regions.

For each of these regions, all the characters are collected to an array which is then processed to find the most frequent character. The most frequent character is then written to the thumbnail coordinate relative to the region coordinate.

Doing this for every region results in a thumbnail of the loaded file with dimensions of height (number of vertical regions) and width (number of horizontal regions).



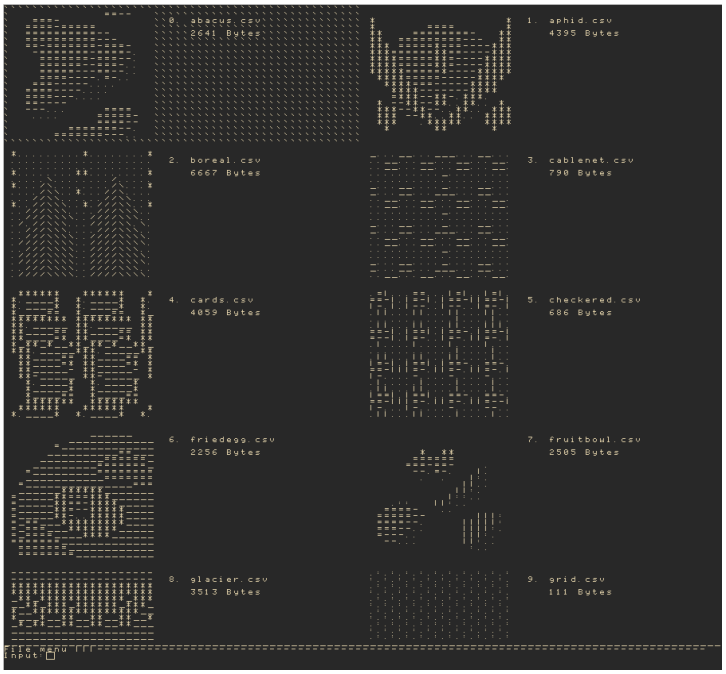
File menu

The file menu gets an indexed version of the files in a directory, then with the styling characteristics specified in the source, calculates the number of “items” (rectangular areas containing file information such as thumbnail, file name and file size) that will fit into the screen.

Then for this number of items, a section of the file index starting at a point determined by the scroll in the menu and ending at this scroll value plus the number of files that fit on the screen is iterated through. For each iteration an item is made of each file.

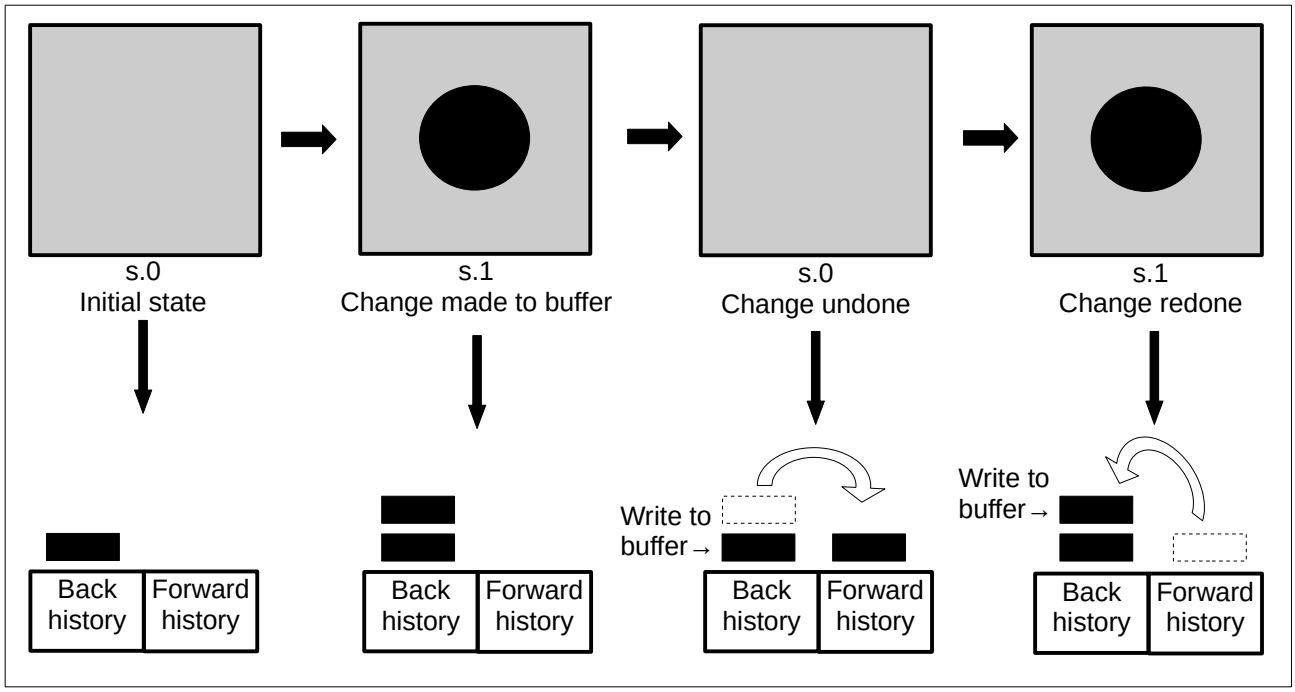
Each item is then drawn onto the buffer, finishing with a menu as pictured right.

There are then navigation commands that allows the menu view to be manipulated in order to display different sections of the file list.



History - undoing/redoing changes

The history method uses two stacks that pass data between each other to achieve the expected behaviour of the undo/redo method, the back-history stack and the forward-history stack. When recording the state of the buffer, a copy of the buffer is saved to the back-history and the forward-history is cleared. Then when navigating the history data is passed between the stacks as shown:

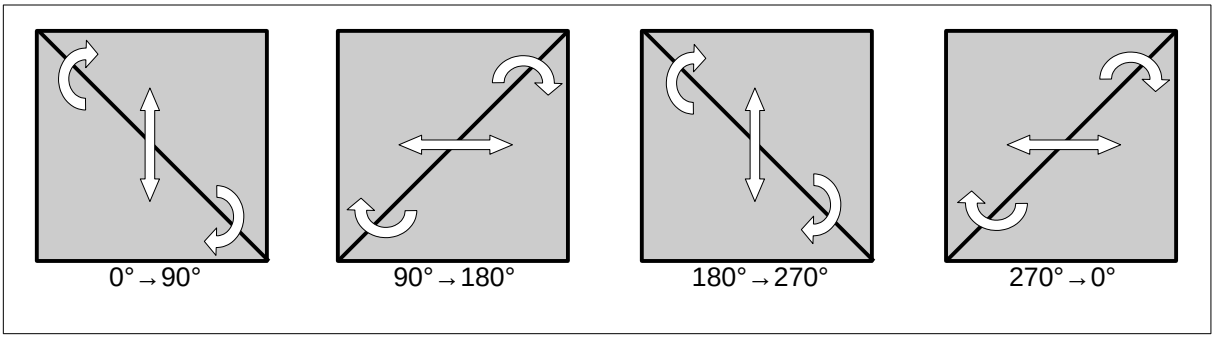


Stepping backwards and forwards through the history is illustrated in the diagram above.

- In the initial state (s.0), a copy of the buffer sits in the back-history stack.
- When a change is made (s.1) to the buffer a copy of the buffer is added to the back-history.
- Undoing a change moves the top-item of the back-history stack to the forward-history stack and the next item in the back-history stack is written to the buffer.
- Redoing this change moves the top-item of the forward-history stack to the back-history stack and writes this new item on the back-history stack to the buffer.

Buffer rotation

Rotating the buffer requires two stages. Firstly the buffer is reflected on a virtual diagonal that flips between a gradient of 1 and -1 for every 90° rotation. Then depending on the angle of the diagonal the buffer is mirrored either vertically or horizontally. Illustrated below is the process of rotating the buffer through 360° clockwise.



0°→90°:	Reflected around \	and mirrored vertically.
90°→180:	Reflected around /	and mirrored horizontally.
180°→270:	Reflected around \	and mirrored vertically.
270°→0°:	Reflected around /	and mirrored horizontally.

Some problems encountered

Testing the read in method was important as a proof of concept when designing the compression method. It was possible to write simple files by hand and allow a method to decode them to check whether it was working as intended.

Pictured above, a hand-written file used in development was decoded into a buffer where the top half was filled with ! characters and the bottom half was filled half with @ characters and half with * characters.

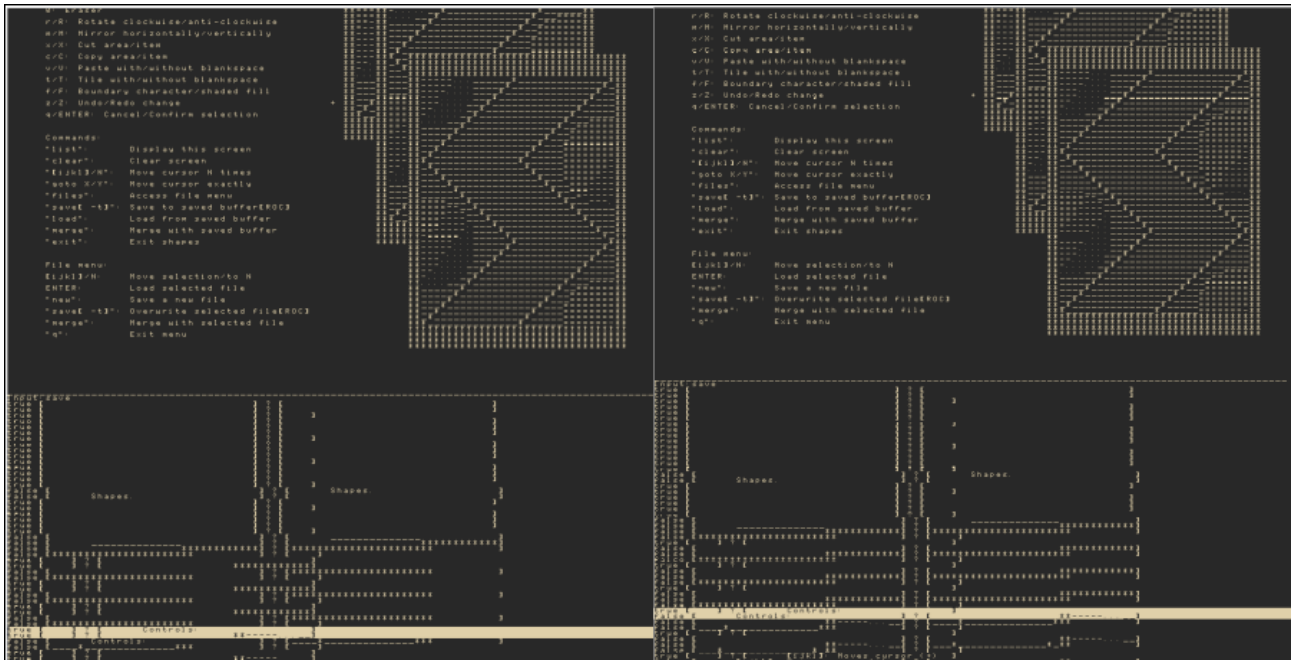
Pictured left, two attempts at saving the buffer into a compressed format that can be used by the read-in method. The compression result is printed to the terminal and can be read manually to determine whether the buffer is saved correctly.

In the top image, the code highlighted in red is responsible for iterating through the regions and comparing them. In this version the debug output revealed that it was not dealing with the final region properly and not recording it, as the expected end item “508: ,” is not present.

Highlighted in the bottom image is the revised code that handles the last item correctly, which can be seen in the respective output which contains the correct final item: “508: ,”

Also note that the format here uses a colon to delimit the repeat value and the repeated character. This was revised later because compressing the buffer when it contained the delimiter characters broke the read-in method. Colon is used more frequently than grave in writing and so it was used as the delimiter instead.

***



At one point, region-overlay-comparison was unable to compress files that did not have an explicitly tiled pattern. To determine why this was, print statements were added so that each comparison could be inspected for problems.

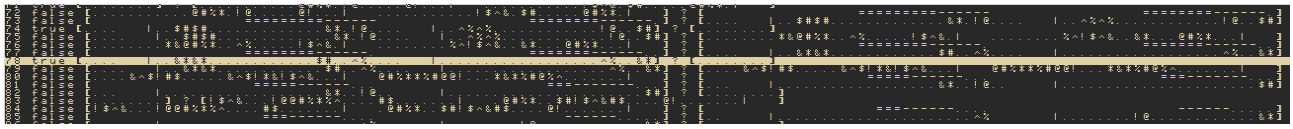
The frame to the left highlights the problem that was uncovered. Because the reference-data-index was set incorrectly, the regions were being compared to the wrong region. The method was evaluating the highlighted regions and coming to the wrong conclusion about their similarities because of the logic error in deciding the reference-data-index.

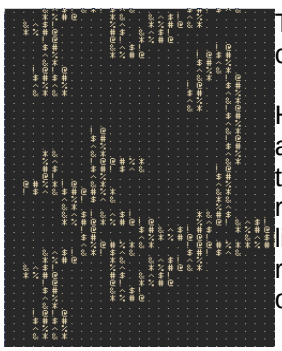
The frame to the right shows the corrected version of the comparison, where the reference data index is correctly changed to the region containing the most information in order to correctly evaluate the next region. By uncovering the logic error here, ROC was able to handle all types of patterns.

```
if (regionOverlayComparison(buffer,height,width,refenceDataIndex,formattedBuffer[refenceDataIndex],index+1,formattedBuffer[index+1])){
    if (formattedBuffer[index+1].length() > formattedBuffer[refenceDataIndex].length()) refenceDataIndex = index+1;
    count++;
}else{
    compressedBuffer.append(count).append("`").append(formattedBuffer[refenceDataIndex]).append(",");
    refenceDataIndex = index+1;
    count = 1;
}
```

This code contains the line needed to check whether the reference-data-index is set to the correct region in the current count.

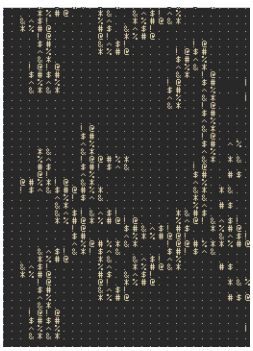
Once the ROC compression was fixed, it was necessary to adapt the read-in method to be able to write regions representing larger areas, into regions with smaller areas correctly. Highlighted below is the debug output highlighting a scenario where writing regions along the edge of the buffer causes problems





The debug output highlighted above helped explain the corruption (right) of a specific buffer section. (left)

Here, two regions with a height of 2 are being compared and are correctly evaluated with ROC. The problem arose because the read-in method was using the dimensions of the larger region to wrap the characters. The smaller section had the literal first 8 characters of the region wrapped around the 2x4 region. The behaviour needed was to write the first 2x4 section of the larger region into the 2x4 region.



What I learned

- Shape algorithms - I learnt how I could draw shapes into a grid by checking every point in the grid for whether it met a certain condition.
- KeyEvents in terminal - I learnt that since the terminal environment does not have any event-based calls, it is not possible to have a listener in terminal, which would have removed the need for pressing enter after every movement in the program.
- Temporary Buffer - I learnt that to store and display temporary changes to the buffer I could create another buffer that copied from the main buffer every time the screen was updated.
- Exception handling - I learnt that to prevent the environment crashing when a problem was encountered I could use catch statements to handle it.
- Instance variables - I learnt about the need for instance variables in program structure.
- File saving - I learnt how to save information from my program to disk.
- Buffer representation - I learnt a basic way to save the buffer to file.
- % operator - I learnt that the % operator can be used to check whether a number is a factor or otherwise what remainder appears with division. **[1]**
- Sharing variables and using methods to affect them - I learnt I could share certain arrays storing coordinates amongst the drawing functions by using a method that set the values of the array before any calculations took place.
- Indexing files in directories - I learnt how to retrieve an array of the files in a directory.
- Random numbers - I learnt how to use Java's random methods to generate a random number within a range in order to display different home-screens.
- Clipboard, copy/paste and class variables - I learnt how I could use a class variable to store a copied section of the buffer and then use methods in the class to re-write it back into the buffer.
- Sorting - Learnt how to use Arrays to sort an array of strings alphabetically.
- Reading a compressed format - I wrote files by hand to test how I would read a file with a certain format back into the buffer.
- Saving compressed format - I learnt how by comparing adjacent characters when iterating through the buffer I could represent the content in a literally smaller way.
- Improving effectiveness of compression - I learnt how I could iteratively increase the width of the adjacent regions of the buffer in order to compress more complex files effectively.
- Comparing 2D regions of the buffer - I learnt that by comparing 2D regions, the representation of the buffer could be even smaller.
- Optimising speed - I learnt that by reducing the number of operations done in every step of the compression I could increase it's speed.
- Recursive method calls - I learnt how I could use recursive method calls to perform a boundary fill of the buffer. **[2]**
- Stack overflows - I learnt how operations like recursive method calls are prone to causing stack-overflows.
- Object Oriented code - I re-wrote my entire project to conform to OO techniques, as up till this point the code did not have any OO characteristics. This included rethinking the class organisation of the code, moving variables into appropriate classes, and creating methods to interact with the variables within each class.
- Thumbnailing and hashmaps - I learnt how by collecting regions of the buffer and then averaging the characters in each region I could create a scaled down representation of the buffer. **[3]**
- Casting - I learnt how to use casting to perform the correct operations on numbers. **[4]**
- Stacks - Using stacks I was able to implement an undo/redo function.
- Stack based boundary fill - To prevent boundary fill from causing stack overflow, it was possible to use stacks to store coordinates in the search. **[5]**
- Lists - I learnt how using lists, I could store coordinates found in a boundary search under certain conditions to a list, then use the information stored in the list.
- Array comparison - In order to prevent the boundary search from checking over the same points forever, I needed to compare arrays to one another using the Arrays class.
- Buffer rotation - By rolling a ruler around an imaginary diagonal line with my hands, I was able to work out how reflection can be used with mirroring in order to make the content of the buffer rotate 90 degrees.
- Adapting pseudo code from Wikipedia - By adapting the pseudocode from Wikipedia's article on Mandelbrot plotting algorithms I was able to implement a Mandelbrot explorer as a drawing method in the program. **[6]**
- Non-integer division - My Mandelbrot implementation had issues because I'd adapted it in such a way that integer division was being performed when it shouldn't have been. By explicitly writing the numbers with decimals the problem was solved. **[7]**
- Parallel processing with Streams - Mandelbrot plotting is a "perfectly parallel task" meaning it is easy to multithread. By creating a mapped integer stream of mapped integer streams of Mandelbrot calculations, I was able to calculate the Mandelbrot in parallel. **[8][9][10]**
- Performing compression scan component of compression in parallel - Using the same technique, I was able to speed up the compression scan by performing every compression in parallel and creating a 2D "compression map" which could be processed.
- Floor and Ceiling math operations - The Math class's floor and ceil methods where used for formatting the compression graph which was used to represent the date from the search stage of the compression.
- Optimising compression time by finding shortcut - I learnt that a shortcut was possible in most scenarios that reduced time significantly.
- StringBuffer - I learnt that the + concatenation operator is slow because Strings in Java are immutable and a new variable has to be made every time it is used and that this was slowing the compression a lot. Therefore I researched StringBuffer and speed increased significantly. **[11]**
- Multithreading read-in method - In order to multithread this task, I needed to convert each coordinate in the buffer into a position in the file data. First, this involved converting the coordinate in the buffer into a region index which related to a region in the data. Then calculating the "region sub-index" which meant calculating the number of characters that would be before the subject character in the virtual-region dimensions in order to determine which character in the determined region in the data was needed to fill the coordinate.
  However because of the complexity of this, it was slower than before so I didn't use it.
- Region Overlay Comparison - I learnt that I could compare the adjacent regions in a different way in order to compress the buffer more effectively.

Overall, I am happy with the final version of my project. I have learnt a lot about how Java works and how OO techniques can be useful. As well as this the logic techniques I've learned will definitely help in solving programming problems that I'll encounter in the future. I think that my approach to testing has let me down, as all of the testing done has been in the process of developing, which has left little room to evidence this testing. Next time, I will put more effort into understanding unit testing and what kind of problems can be tested with unit testing and plan how I will test the code from the start of the project.

I probably learnt the most in this project from the compression algorithm development, I enjoyed solving the problems that it's rule environment created and am very satisfied with what it does. If I was to take it further, I would use another layer of compression to reduce the size and find a faster way of finding the ideal region-size. This could look something like a database of saved files and their ideal region-sizes that the program could consult in order to assign a similarity value between the buffer and a pattern that has been encountered which could be used to reduce the range of dimensions needed to be tested in a scan.

References

[1] GeeksforGeeks (2020) *Java Program to Check if a Given Integer is Odd or Even* Available at: https://www.geeksforgeeks.org/java-program-to-check-if-a-given-integer-is-odd-or-even/

[2] freeCodeCamp (2020) *Boundary Fill Algorithm* Available at: https://www.freecodecamp.org/news/boundary-fill-algorithm/

[3] educative (2020) *How to find the most frequent word in an array of strings* Available at: https://www.educative.io/edpresso/how-to-find-the-most-frequent-word-in-an-array-of-strings/

[4] isnot2bad, Johnson C.J. (2013) *Math not calculating right in Java* Available at: https://stackoverflow.com/a/19072603

[5] enterbios (2014) *Stack overflow error when filling a shape with boundary fill algorithm* Available at: https://stackoverflow.com/a/23031371

[6] Wikipedia (2021) *Plotting algorithms for the Mandelbrot set* Available at: https://en.wikipedia.org/wiki/Plotting_algorithms_for_the_Mandelbrot_set (Accessed: 5 April 2021)

[7] Cronen J.C. (2013) *Calculation returns zero instead of expected result* Available at: https://stackoverflow.com/a/19230167

[8] Wikipedia (2021) *Embarrassingly parallel* Available at: https://en.wikipedia.org/wiki/Embarrassingly_parallel/

[9] Wiktor L.W. (2014) *Filling a Multidimensional Array using a Stream* Available at: https://stackoverflow.com/questions/26050530/filling-a-multidimensional-array-using-a-stream/26979389#26979389/

[10] Smith M.S. (2016) 'streams.pdf'. *CI228:Object-oriented programming in Java*. University of Brighton. Unpublished.

[11] Castorina N.C. (2018) *Java String Concatenation: Which Way Is Best?* Available at: https://redfin.engineering/java-string-concatenation-which-way-is-best-8f590a7d22a8/