Algorithm visualiser designed to demonstrate and display the process behind a variety of sorting algorithms and to play an audible ‘tone’ corresponding to the value of the current data being sorted.

The user will be able to select the algorithm to sort by as well as the method used to ‘shuffle’ the data accessed via drop down menus. The user will be able to control sliders to determine the time between each step in the sort; the size of the array to be sorted; the pitch of the tone played. The user, through the use of buttons, will be able to start and pause the sort; step through the individual steps of the sort; reset the array back to the original state; mute the sound of the tones; select a random sort.

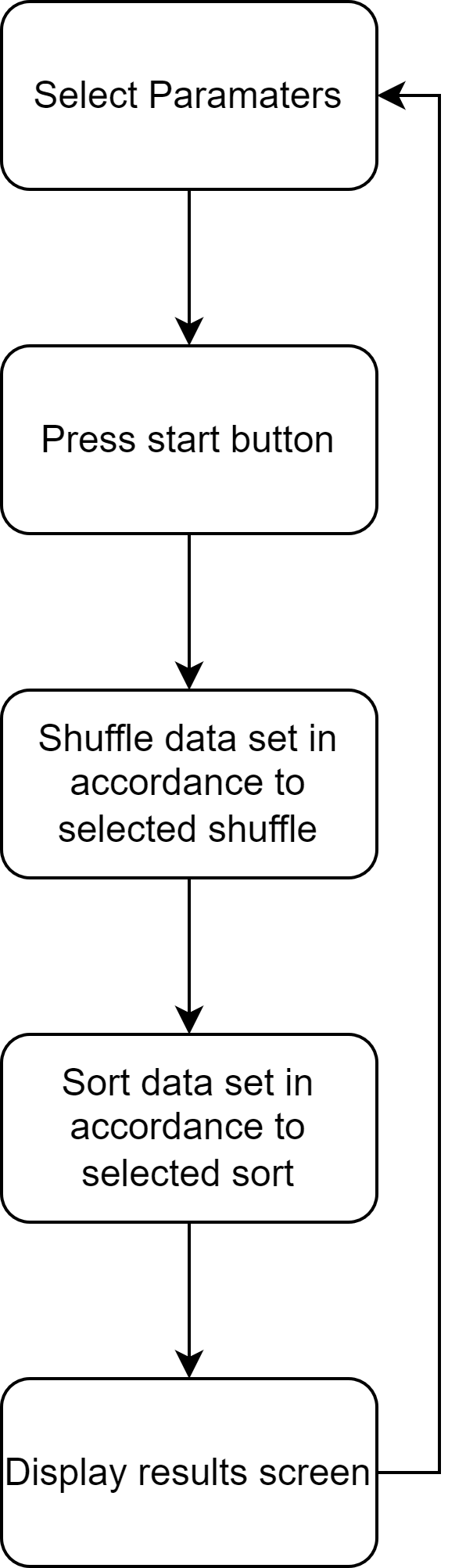
Objectives

1. Display the current state of the array, representing each element as a bar with height corresponding to the value of the data element
2. Display a results screen containing statistics relating to the sort – sort type, number of elements in array, time taken for sort, number of comparisons made, number of swaps made
3. Create a control/settings panel containing:

* 6 buttons to start, pause, step through and reset the sort as well as mute the tones and select a random sort.
* 3 sliders to change the time between sorting steps; change the array size; change the pitch of the tones played
* Drop down menu to select the method used to shuffle the array
* Drop down menu to select the sorting algorithm used
* Information panel to display percentage of array sorted; time taken so far; comparisons made; swaps made

1. Implementing a variety of sorting algorithms

Overall program loop



Initial shuffling of data

In order for the data to be correctly sorted, it first must be shuffled. To accomplish this I have taken the decision to give the user the choice to either reverse the order of the array or to create a ‘truly shuffled’ array in which every permutation is equally likely.

To reverse the array I increment through from the start to the midpoint, swapping each element with its equivalent from the end point.

To truly shuffle the array I decided upon a variation of the ‘Fisher-Yates shuffle’. In the original shuffle:

* You have a list of numbers arranged in order from 1 to N (number of elements)
* Select a random number between 1 and the remaining number of elements, X
* Remove the number in the X position of the list and copy it to the end of a separate list
* Repeat the previous 2 steps until all numbers have been moved to the new list

Whereas in a more modern approach devised by Richard Durstenfeld, the algorithm I chose for the shuffle, has been proven more efficient when performed by a computer as instead of counting the amount of remaining numbers in the list every iteration, the number to be moved is instead swapped with the last unaltered number at the end of the list each iteration. In doing so the time complexity of the algorithm has been reduced from O(n^2) to O(n).

Rendering

For the rendering I am using the libGDX application framework in which uses the Lightweight Java Game Library (LWJGL) as the backend. I have chosen this platform due to the ease of use of its shape renderer component which renders simple shapes such as rectangles in batches to speed up performance. Due to the vast majority of the application being displayed to the user consisting of vertical rectangles I felt this to be an acceptable decision combing ease of use with reliable performance.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

An example piece of code using the shape renderer to render a simple rectangle at position (50, 100) with a size of 200 x 200

Sorting algorithms

Settings panel