Sorting Algorithms in Unity3D

Buchbauer Benjamin, BSc

b.buchbauer@student.tugraz.at

February 18, 2017

Contents

1. Introduction
2. Sorting Algorithms
   1. QuickSort
   2. MergeSort
   3. HeapSort
   4. GnomeSort
   5. RadixSort
3. Visualization in Unity3D
   1. Functionality / Structure???
   2. Swapping Elements
   3. Coding Window
   4. Snippets
4. Conclusion and Future Work
5. **Introduction**

In this study we want to visually outline sorting algorithms either one after another or several at the same time. Furthermore we want to write our own code in runtime to try to sort the elements and compare with different sorting algorithms. The platform we use is Unity3D. It is a cross-platform game engine and used to develop video games for PC, mobile devices, websites and consoles.

TODO: add something from project plan

The remainder of this report is organized as follows. Section 2 discusses several sorting algorithms we have implemented in this project. Visualization of our sorting algorithms is presented in Section3, which also includes some code snippets. Section 5 provides final conclusions and directions for future work.

1. **Sorting Algorithms**

In this section we will discuss the functionality and characteristics of several sorting algorithms which we want to visualize.

**2.1 QuickSort**

QuickSort is a quick, recursive and efficient 'divide and conquer' algorithm. However it is not stable, which means that the relative order of equal elements is not preserved.

In the beginning the list is divided into two sublists. In order to do this quicksort chooses a pivot element from the original list. All elements, which are smaller than the pivot element, are put into the left sublist and all the bigger elements are put into the right sublist. Equal elements can be put into either sublist. After that quicksort can start to recursively sort the sublists and reapply the above steps.

Time complexity:

Best case: O(n log n)

Worst case: O(n²)

Space complexity: O(log n)

**2.2 MergeSort**

Similar to QuickSort it is a comparison-based sorting algorithm, which is based on the 'divide and conquer' principle. However in contrast to the former it is a stable algorithm.

First the unsorted list is divided into sublists which consist of one element. The sublists are then recursively merged to generate new sorted sublists. This procedure will be repeated until there is only 1 list remaining which is the sorted list.

Time complexity:

Best case: O(n log n)

Worst case: O(n log n)

Space complexity: O(n)

**2.3 HeapSort**

HeapSort is an in-place comparison-based and instable sorting algorithm. It uses a special data structure called Heap, which is a binary tree-based data structure that satisfies the heap property. If A is a parent node of B, then the value of node A is either bigger (if we use a max heap) or smaller (min heap) than the value of node B. The same ordering is applied across the whole heap. In our case we use a max heap.

In the beginning we build a heap out of our unsorted array which means that the biggest element will be on the first position. We then go on and swap this element with the last element and reduce our heap size by one. Next we have to rebuild our heap because it is possible that the new first position element doesn't satisfy the heap condition. The aforementioned steps are repeated until the heap size is 1. In the end we have a sorted array.

Time complexity:

Best case: O(n log n) TODO: korrekte zeichen der o-notation

Worst case: O(n log n)

Space complexity: O(1)

**2.4 GnomeSort**

GnomeSort is a very simple and stable comparison-based sorting algorithm. It starts at the second element and always compares the current element with the previous one. If they are in the correct order it will move one position ahead, otherwise it will swap the elements and go one step back. After that the comparison procedure is started again and everything is repeated until the end of the array is reached.

Time complexity:

Best case: O(n) TODO: korrekte zeichen der o-notation

Worst case: O(n²)

Space complexity: O(1)

**2.4 RadixSort**

RadixSort is a non comparison based, stable and out-of-place sorting algorithm which is based on BucketSort or CountingSort. It is assumed that the unsorted data only consists of keys with characters of an ending alphabet. For example strings with characters from a to z, or decimal numbers (base 10). RadixSort can be implemented on the basis of least significant digit sort, which starts from the least digit and moves towards the most significant digit, and most significant digit sort, which works the other way around.

For the former we start by taking the least significant digit of each key. Next we group the keys based on that digit by also taking the original order of keys into consideration, which is what makes it a stable sort. Repeat the steps for each more significant digit. The grouping process is done by using BucketSort or CountingSort.

Time complexity:

Worst case: O(wn) where n is the size of keys and w is the length the keys.

Space complexity: O(w+n)

1. **Visualization in Unity3D**

In the previous section we have briefly explained various sorting algorithms we will use in our project and in this section we will discuss our approach to visualize the algorithms.

We will use the cross-platform game engine Unity which enables the realization of computer games and interactive 3D applications. By default Unity3D uses MonoDevelop as development environment. It supports the scripting languages UnityScript, Boo and C#. In our case we will use C#.

**3.1 Structure**

Our project consists of two different scenes in Unity3d. The first one is the initial starting point where one can select the number of elements which will be sorted.

TODO: add help button/text in start scene (description, available functions)???

The second one is our main scene where we can see the elements, choose the sorting algorithm, execute code in the coding window and add a new set of elements.

First of all an array with the size of the desired number of elements and with randomly arranged GameObjects of different size is created. Furthermore we have made a prefab called 'SortingBox' that consists of the elements and a container which is used to represent the area of a single set of elements. This area can be selected to apply a specific sorting algorithm or to execute any given code at runtime. Hence to start the sorting procedure one has to select the area and click on the desired algorithm button on the right side. To execute code one also starts by selecting the area. After that the desired code is to be entered and the execution is started by simply clicking outside of the coding window.

By choosing a size and clicking the 'Add' button on the top right corner further so-called 'SortingBoxes' are added. The camera distance can be changed by altering the vertical slider on the left side.

**3.2 Swapping Elements**

In this subsection we talk about our approach to visualize the swapping of elements.

After a sorting algorithm is selected and executed we keep track of the array element swaps and put the according order into a queue. We then pass the queue to a separate script which starts the swapping visualization. The elements, which switch positions, will start to move and change color. The way elements are swapped depends on the algorithm used. We have to distinguish in-place and out-of-place algorithms.

1. In-place algorithms:

In our case that would be QuickSort, HeapSort and GnomeSort. For these three sorting algorithms we use the same movement script. We pass the queue and take the first two elements out of it. For the visualization we use a simple method called 'RotateAround' which rotates a game object around an axis passing through a point in a certain angle. Hence we have to calculate a particular rotation point, which is simply half the distance between both of the elements. We then let the method run for every frame until their destination points are reached. After that we take out the next two elements, calculate their rotation point and continue until we have arrived at the end of the queue.

1. Out-of-place algorithms:

For the out-of-place algorithms RadixSort and MergeSort (there are also some in-place variants for MergeSort) we use seperate movement scripts for each of them because of their respectively special behaviour. For RadixSort we have two phases. First the grouping by digit, where our elements are moved up towards groups, and second the sorting of buckets, where they are moved back down to the original position. We use the basic method 'MoveTowards', which moves our element to specific coordinates, for each process.

In the course of visualizing MergeSort we use a combination of direct swapping to another elements position and moving elements in straight lines upwards and back to original positions, which should express the sorting that takes place, when two arrays are merged.

**3.3 Coding window**

In our project we also have the possibility to enter and execute code at runtime. That means we can get information about any desired element and swap them. In best case we can even insert an own created algorithm and compare its performance against the provided algorithms.

To do this we provide some functions to access the elements:

* size(x)