Fall 2020

The Use of Virtual Manipulatives in Teaching Sorting Algorithms

Olivia Witanowska
ow0714@bard.edu

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_f2020

Part of the Graphics and Human Computer Interfaces Commons

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License.

Recommended Citation
https://digitalcommons.bard.edu/senproj_f2020/25

This Open Access is brought to you for free and open access by the Bard Undergraduate Senior Projects at Bard Digital Commons. It has been accepted for inclusion in Senior Projects Fall 2020 by an authorized administrator of Bard Digital Commons. For more information, please contact digitalcommons@bard.edu.
The Use of Virtual Manipulatives in Teaching Sorting Algorithms

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Olivia Witanowska

Annandale-on-Hudson, New York
December, 2020
This project builds an online tool for users to practice tracing sorting algorithms on a virtual set of cards. Sorting algorithms are crucial to computer science. Visualization of such algorithms can aid in cementing the concepts. The purpose of a virtual manipulative is to create a similar effect of a physical manipulative but on a digital platform so that it is more accessible than its physical alternative. This project is motivated by the educational adjustments needed to be made during the 2020 Covid-19 pandemic. By building off of a previous Deck of Cards project, an algorithm to check a user’s work is implemented to allow users to receive immediate feedback on their tracing of a sorting algorithm at any point in time.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 History of Using Manipulatives</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Virtual vs Physical</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Related Work</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1 Sorting Algorithms</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Choosing to Use Cards</td>
<td>7</td>
</tr>
<tr>
<td>3 Design</td>
<td>9</td>
</tr>
<tr>
<td>3.1 A Virtual Deck of Cards</td>
<td>10</td>
</tr>
<tr>
<td>3.1.1 Notable Women in Computing</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Choosing a Sorting Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Checking User’s Progress</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1 Tracking Cards</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 Generators and Pausing Progress</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Multi-platform Program</td>
<td>17</td>
</tr>
<tr>
<td>4 Results</td>
<td>19</td>
</tr>
<tr>
<td>4.1 Sorting Using the Program</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Checking Work</td>
<td>21</td>
</tr>
<tr>
<td>4.2.1 Program Limitations</td>
<td>25</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>27</td>
</tr>
</tbody>
</table>
5.1 Summary .................................................................................. 27
5.2 Future Work ........................................................................... 28
5.3 Experimentation ...................................................................... 28

Appendices ................................................................................. 31

A Appendices .............................................................................. 31
A.1 index.html Code ..................................................................... 31
A.2 deckGraphics.css Code .......................................................... 39
A.3 deck.js Code ........................................................................ 47

Bibliography ................................................................................ 65
Acknowledgments

I would like to thank Keith O’Hara for being my advisor, for aiding me in this project and for all that he has taught me in these two and a half years.

Thank you to all my friends who made my time at Bard better than I ever thought it could be through adventures and late night study sessions.

To Patrycja and Jason, thank you for guiding me through these years and watching my plant collection grow.

To my family who helped me get this far, thank you for all the visits and home-cooked meals.

Without all of you, I would never have been able to do this.
Manipulatives are a popular tool for teaching simple mathematical ideas such as shapes and arithmetic, but their effectiveness reaches farther than just teaching basic math ideas. A manipulative’s purpose is to give students a visualization and hands-on representation of a concept, therefore it can be applied to concepts beyond math. Topics in computer science are particularly difficult to conceptualize as they are mainly taught symbolically using code. Data structures and algorithms, two essential parts of programming, are particularly complicated to explain in words and some find visual aids helpful in understanding these ideas. Having a hands-on activity of how these structures are created and sorted can allow the student to have a better understanding of what the computer does and how it does it.

This project creates a tool that allows users to explore common sorting algorithms. Users are be able to access the tool at any time via its url [https://sorting-with-cards.glitch.me/](https://sorting-with-cards.glitch.me/). Upon connecting, the user can choose any of the sorting algorithms presented to have its corresponding code appear on screen. Using a technique used by many Data Structures instructors, this program uses cards to represent elements stored in an array and for variable names. The user can move cards with variable names over the assumed element it stores. Once they are finished with a step they may check their work.
using the "Check" button which reveals the current information stored in each active variable. When the tracing is completed, the program congratulates the user. For the code behind the project, users may view it on Glitch at [https://glitch.com/edit/#!/sorting-with-cards](https://glitch.com/edit/#!/sorting-with-cards).

Figure 1.0.1. User Interface for Sorting with Cards
2 Background

2.1 History of Using Manipulatives

History shows that visualizations may have been used as early as 7500-3000 BC. Artifacts found at archaeological sites show tokens made of clay were used to keep track of goods. The shape and amount of tokens would symbolize the type and amount of an item [1]. By definition, manipulatives are "objects (such as blocks) that a student is instructed to use in a way that teaches or reinforces a lesson" [2]. In a similar way, these clay tokens used by ancient civilizations are some of the earliest examples of visualization. The tokens were used as a way to visualize and connect shape to the quantity of a specific good similar to how modern day colored blocks can represent values and shapes.

Today, manipulatives are commonly used in early mathematics education. Some of the most basic items are used to reinforce ideas such as: pattern blocks to identify shapes, color tiles for counting, and a clock for telling and writing time [4]. Research has shown that students can gather valuable knowledge when they can actively create an understanding of mathematical concepts through the use of tangible objects. Through experience they can develop a connection between the physical and mental idea of the lesson. It’s important that they learn at their own pace with the appropriate tools so they are capable of using these manipulatives the way they are intended to be used [5].
2.1.1 Virtual vs Physical

While physical manipulatives are common, they have limitations. An interactive digital program could prove to be more advantageous and convenient than a physical one. Both forms have their own restrictions but especially in a time such as now during the COVID-19 pandemic the demand for virtual manipulatives has increased. With restrictive settings and social distancing, physical manipulatives have become less available. A more accessible approach would be to use a virtual visualizer that allows users to learn from the comfort of their home or anywhere else that they take their device. The benefit of this tool being digital is that it can be taken anywhere as long as there is wifi, making it convenient to some users as many people already carry some sort of smart device and have access to the internet. A potential downside of a virtual tool is making it too automatic. It is vital that it has the right balance of automation and need for user-input. A manipulative should be manipulated by the learner, not the computer. Therefore, the program should require the user to do the steps in order for them to learn.

2.2 Related Work

There are already a handful of visualizers, both physical and virtual, that are created for various computer science topics. For sorting algorithms, the Comparison Sorting Algorithms webpage is an online visualizer that allows users to select amongst different methods and watch as the computer sorts an array right in front of them [6]. They can modify the array size and the speed at which the algorithm sorts it, but users do not have the option of sorting the array on their own. This virtual visualizer lacks the creativity and independence manipulatives commonly give users to learn from experience. Another alternative visualizer is Python Tutor [7][8]. While having Python in its name, the program also interprets Java, C, C++, JavaScript, and Ruby. This program allows users to input code and watch as the visualizer explains what is happening during each line as seen
in Figure 2.2.1. Similarly to the previous program, Python Tutor does not allow for the user to do any of the work, but is more user friendly by allowing the code to be modified.

Researchers found that there is more to a manipulative than just focusing on visuals and/or user input. The study conducted used their program, The JDSL Visualizer, and analyzed the effect it had on students completing an assignment. The program created data structures based on the inputted code and then tested the functionality of those data structures. An observation that they made during their research was that some students were in fact not using the visualizer in it’s intended way, therefore creating a new category in their results. They concluded that the layout of the user-interface was just as important as explaining how to use the program \[9\]. Without direction a manipulative may not be as effective as it was intended to be.

An example of a more hands-on visualizer is a deck of cards used as a physical manipulative by Data Structures instructors to teach various methods of sorting. Each student or group of students can be given a deck of cards to use as a manipulative, where the cards represent the values stored in an array. The setback of this card method is that it requires a teacher or someone who has knowledge on the method to check the students technique in sorting. The manipulative’s purpose is not to show the students all the cards at once,
but to have them individually "reveal" the cards just like a computer would access the objects in that array one by one. The experience is to show how and when certain data is accessed in an array and why sorting on a computer can be different than a person who is visually sorting by looking at all values at once.

### 2.2.1 Sorting Algorithms

An important part of learning about algorithms is learning about sorting. Instructors usually ask students to trace what happens in the code either by hand or some other method so that they may understand how the computer moves around elements. With sorting, it's important to carefully keep track of all the variables and understand what is happening as the computer works through the array. Each sorting method has its benefits and setbacks. Arguably, some of the most common methods taught are merge sort, quicksort, selection sort and insertion sort. These are the methods that are focused on for this project.

Both insertion and selection sort methods sort as they move down the array. As it moves down the array, insertion sorts by moving cards of lesser value back up the array until there are no more elements left to be sorted. Selection sort selects the first value in the array to swap and moves down the array to find the smallest value. Once it reaches the end of the array, it swaps the first value with the lowest value. The process repeats by starting with the next value in the array until it reaches the end.

Quick and merge sort are both divide and conquer algorithms, often using recursion. For merge sort, the algorithm recursively splits the array until it can no longer do so. Merge sort's `merge()` merges two array halves after each half is sorted. For quick sort, the algorithm uses a pivot point to compare values. Quick sort's `partition()` sorts all values lower than the pivot to the left and all values greater to the right. Once the lower and the higher meet, the pivot is inserted into this position. The algorithm then recursively calls `partition` on the left and right side until the array is sorted.
2.2. RELATED WORK

2.2.2 Choosing to Use Cards

Each of the sorting algorithms use an array of different values to sort. In a computer, these values are stored in an array, but are not immediately accessed until they are called on. Using cards is a great way to replicate the function of an array with values because cards have a front and a back. When the back is showing, the holder does not see the value on the other side. A card’s “value” can be accessed by flipping it over. Cards can also be shuffled and arranged in a order like an array. Elements exchanged in code can be done by hand as well by simply swapping cards. Additionally, cards don’t take up much space due to their size, making them a great manipulative in a classroom.
We aim to create an accessible tool to aid for those seeking to understand and practice tracing sorting algorithms. Since the original Deck of Cards program was written using HTML, CSS, and JavaScript, this program is written in these languages as they allow it to be hosted on a website with the aid of the developer tool Glitch. It replicates an array of elements using a deck of cards with additional cards used for any variables found in the sorting algorithms.

By modifying the starting position of the deck, a random selection of eight cards are laid out like they would be in an array with the variable cards found below. The amount of cards put in the array can be modified to have more or less by changing the max value of cardsOut within the code. Each card has the properties rank, suit, and position. Cards containing the Jack, Queen, and King have ranks 11, 12, and 13 respectively. The cards laid out must be treated like an array, therefore are stored in an array and sorted in order of their x-position. Variable cards are similarly sorted for easy identification in their own global array. Through the commands found in file Deck.js, users can click to flip and drag any of the cards on the user interface. Should any card from the main array be moved, the array keeping track of active cards re-sorts them by x-position.
A drop-down menu allows for users to select a sorting method, making a text box appear with the desired code. Currently there are four sorts available: insertion, selection, merge and quick sort. The text box is updated with every selection from the drop-down menu. To be able to increment through the sorting algorithm in JavaScript with the user, a generator function is used. For every step where a variable is updated, the generator stops until the user presses the check button allowing the generator to increment and yield the new values on screen. To check the users work, variable card positions are checked to see if they are within range of the card containing the value of the true answer. Using conditionals, the generator calls to methods to calculate distance between the users answer and the correct answer, and whether or not they have sorted when needed. Until the correct answer is given, the generator loops and yields the correct answer to the current step. Changing the sorting method of choice ends a generator and restarts a new one for the desired method.

3.1 A Virtual Deck of Cards

The digital deck of cards program that is modified for this project was originally found on GitHub made by Juha Lindstedt who goes by the username ”pakastin”. The newer version of his project is hosted on the website [https://deck.of.cards/](https://deck.of.cards/). It has a multiplayer feature that generates a link for users to use and communicate with, as seen in Figure 3.1.1. For the purposes of the manipulative project, the older, simplified version of the deck of cards without the multiplier feature is used. The website allows for users to flip, shuffle, and sort the cards by rank or suit. Two other features that can be used are the fan, as seen in Figure 3.1.2 and poker buttons which can fan out the cards or set up five random cards like in the game of Poker.
3.1. A VIRTUAL DECK OF CARDS

Figure 3.1.1. The latest version of Deck of Cards

Figure 3.1.2. The older version of Deck of Cards, with cards fanned out
3.1.1 Notable Women in Computing

To continue the theme of computer science, the original "face" images displayed on the cards have been replaced with the Notable Woman in Computing card deck. Ada Lovelace is considered the world’s first computer programmers. She created plans for how computations can be performed on the Analytic Engine, the first general computer. Through years, women like Lovelace, and their contributions to computer science have been forgotten and in more recent years women in this field have been underrepresented. By including them in this project, we hope to remind people of these women and their important role in computer science.

Figure 3.1.3. Layout out of the deck with the first three cards in the array facing up to reveal the "faces"

3.2 Choosing a Sorting Algorithm

The program allows for the user to choose amongst a couple of sorting methods: insertion sort, selection sort, merge sort and quick sort. Using HTML and CSS, a drop-down menu
was created to display the available options. The user is able to select any of these options like a button.

```javascript
function selSort () {
  checkOtherOptions();
document.getElementById("sel").style.display = "block";
changeSort("sel");
checkProg();
}
```

The above method is called upon when selection sort is chosen from the drop-down menu. Every sorting choice has a corresponding method. Once a sorting method is chosen, its code appears on screen for the user to see. Should the user want a different method, they can re-select another from the drop-down. Every time the sorting algorithm is changed the program checks to make sure any other code on screen is hidden before displaying the new code. The method calls to `changeSort()` and passes along the new active sorting algorithm. The program currently does not have a check algorithm for merge and quicksort, therefore these two options are commented into the program for future work.

```javascript
function changeSort (sort) {
  // if a new sort is chosen
  if (sort != activeSort) {
    if (activeSort != null) {
      currentGen.return();
document.getElementById("check").innerHTML = ";
    }
    var generators = {
      "merge": mergeGen(),
      "quick": quickGen(),
      ins: insGen(),
sel: selGen()
    };
    activeSort = sort;
    if (sort in generators) {
      currentGen = generators[sort];
      // console.log(currentGen);
    }
  }
}
```

It is important that the function `changeSort()` is called first as it is responsible for updating the global variable containing the information on which generator method is currently active for checking. Finally, it finishes by calling to `checkProg()`.
### 3.3 Checking User’s Progress

Part of this program’s goal is to help the user in a way that isn’t just blindly moving cards. Using HTML’s button feature, a button was created at the bottom left of the screen that is used to check the user’s work. It’s purpose is to help guide the user into selecting the right value (or card, for this purpose) for the right variable at the correct time. Every time the user has moved the variable cards above the corresponding values in the array, they can check their guess for the step by pressing the "check" button.

#### 3.3.1 Tracking Cards

The method `checkProg()` begins by going through the active array of cards and sorting them based on their x-value positions. Sorting cards based on position helps keep track of where the cards are moved in the array by the user. The array `cardOrder` is stored as a global variable so it can be accessed by the generator methods.

```javascript
function checkProg () {
  var cards = [];
  for (var i = 0; i < activeDeck.length; i++) {
    cards.push(activeDeck[i]);
  }
  // Array to store cards ordered by x position
  cardOrder = [];
  for (var i = 0; i < activeDeck.length; i++) {
    var lowestX = cards[0];
    for (var j = 0; j < cards.length; j++) {
      if (lowestX.x > cards[j].x) {
        lowestX = cards[j];
      }
    }
    cardOrder.push(lowestX);
    cards.splice(cards.indexOf(lowestX), 1);
  }
  var genValue = currentGen.next().value;
  // print variable values
  document.getElementById("check").innerHTML = JSON.stringify(genValue);
}
```

After the position of the cards are sorted, the program calls to the generator corresponding to the selected sorting method. The returned values are those stored in the variables used at a specific position in the sorting algorithm. Using a conversion method, the returned JavaScript object is converted into a JSON string and printed for the user to see.
3.3. CHECKING USER’S PROGRESS

3.3.2 Generators and Pausing Progress

To check a user’s progress, a generator is used for "checkpoints" in the sorting algorithm. By carefully sectioning the algorithms, a generator stops after specific lines in the code. By separating parts of the code, the program can carefully check the user’s choices and pausing if it is incorrect. For example, in selection sort the generator selGen() pauses at every line that updates a variables value. For the purposes of this manipulative, the user does not need to move the "n" variable card but should be aware of its value.

```javascript
function* selGen() {
    var n = cardOrder.length;
    i = 0;
    yield 0;
    while (i < n) {
        // Check i card
        while (checkICard(cardOrder, i) == false) {
            yield { i: i, j: j, min: min }; // Check i card
            yield { i: i, j: j, min: min }; // Check i card
        }
        var min = i;
        // Check i and min card
        while (checkMinCard(cardOrder, min) == false ||
            checkICard(cardOrder, i) == false) {
            yield { i: i, j: j, min: min }; // Check i card
            yield { i: i, j: j, min: min }; // Check i card
        }
        for (var j = i + 1; j < n; j++) {
            // Check j card
            while (checkICard(cardOrder, i) == false ||
                checkJCard(cardOrder, j) == false ||
                checkMinCard(cardOrder, min) == false) {
                yield { i: i, j: j, min: min }; // Check i card
                yield { i: i, j: j, min: min }; // Check i card
            }
            if (cardOrder[j].rank < cardOrder[min].rank) {
                // Check all variable cards
                while (checkICard(cardOrder, i) == false ||
                    checkJCard(cardOrder, j) == false ||
                    checkMinCard(cardOrder, min) == false) {
                    yield { i: i, j: j, min: min }; // Check i card
                    yield { i: i, j: j, min: min }; // Check i card
                }
            }
            minRank = cardOrder[min].rank;
        }
    }
}
```
For every line of code that modifies the value of a variable, the program calls a series of methods that compares the user’s answer to the actual card value from the generator by calculating distance between card x-position values. If the generator wants to check the user’s answer for the value of variable \(i\) the cardOrder array and the actual value of the variable would be passed to the method below.

```javascript
function checkICard(a, i) {
    distI = Math.abs(a[i].x - varDeck[0].x);
    if (distI < 30) {
        return true;
    }
    return false;
}
```

The method calculates the distance between the x-position of the variable card \(i\) and the card in the variable array that is the real value of \(i\). The distance needs to be less than half the width of a card or else it fails to be within distance of the answer. If any of the checkCard() methods fail at any point, the program catches and waits for the user to correct their answer. By checking multiple variable cards at once the program insures that the user has not moved other variable cards into the wrong positions.

In addition to checking variable cards, the generator waits for the user to swap cards at the appropriate moment. For selection and insertion sort, the isSorted() method checks to see if all the cards up to the position \(i\) are sorted.

```javascript
function isSorted(a, i) {
    for (var el = 0; el < i + 1 && el < a.length - 1; el++) {
        if (a[el].rank > a[el + 1].rank) {
            return false; // hasn't been sorted
        }
    }
    return true;
}
```
Once the algorithm is done running, the message changes from the value of the variables to the words “Complete: good job!”. With this, the user has successfully finished sorting the array.

3.4 Multi-platform Program

A huge benefit of using the original Deck of Cards program is that it allows for this modified version to have multi-platform capabilities. Allowing the program to be used on multiple devices is a huge advantage as it doesn’t limit it to a computer and can allow tablet and phone users to participate as well. Most of the program’s visual elements should be programmed in respect to the screen resolution therefore allowing proper viewing of the program once it’s loaded.
3. DESIGN
4.1 Sorting Using the Program

The final appearance of the program resembles the original Deck of Cards but includes new card visuals as well as new interactive buttons. The cards are aligned faced down to represent an array and the screen is clear of code until an option is selected. Below the array the user can find the variable cards. As seen in Figure 4.1.1 located at the top right corner of the screen is a drop-down menu. The user can select from any of the available sorting algorithms. The menu currently contains insertion, selection, merge and quick sort. Once the desired sorting method is selected, it appears on the screen. Figure 4.1.2 is an example of what the code for selection sort looks like after it has been selected. For merge and quick sort, only the merging and the partitioning methods appear as this is where the algorithms differentiate from the other sorting methods. This allows users to focus on code responsible for merging and partitioning the array.
Figure 4.1.1. The drop-down menu is located at the top right of the screen, when hovered over the sorting options are revealed

Figure 4.1.2. Code for selection sort appears on screen after it has been selected from the drop-down menu
4.2 Checking Work

As the user traces the code they should move variable cards to the value it contains. Cards can be flipped at the appropriate time during tracing. Cards that are not being accessed in the code at a given moment should remain face down. Once a variable card is moved, the user presses the ”Check” button located in the bottom left corner. After pressing the button, a text box appears at the bottom right of the screen containing the information about each variable’s value at that moment. Figures 4.2.1 and 4.2.2 show the early tracing of selection sort and insertion sort respectively. Currently, merge sort and quick sort do not have a check algorithm programmed. If the user tries to press the button while doing merge or quick sort, a zero appears.

![Sorting with Cards](image)

Figure 4.2.1. An example of checking work during selection sort
If the user’s answer was wrong, the algorithm halts and continues to display the real answer for the user to correct their choice. Once the user corrects their answer, they press the check button once before tracing the next step in the code. It is important that the user does not press it a second time until they have finished the next step as the algorithm interprets this as the user’s answer.
When the user reaches a line of code that requires them to exchange values, they must swap cards as demonstrated in Figure 4.2.4 or the program stops from progressing and
waits until the user swaps the cards. As before, the user should only press the check button once after they make a correction. From there they can continue to trace like before until the sorting is complete and they are notified by the text box “Complete: good job!” as seen in Figure 4.2.5. The user can flip over all the cards to reveal that the array has been properly sorted.

Figure 4.2.5. Selection sort method has completed, user has reached the end
4.2. CHECKING WORK

At any given time the user has the option of selecting a new sorting algorithm for the drop-down menu. When a new method is chosen the code displayed changes and the generator responsible for checking the user’s work resets. Therefore, every time a user changes a method they must start from the beginning. To reshuffle the cards, the page must be refreshed.

4.2.1 Program Limitations

There are some limitations to the program as it does not account for every possible moment a user makes. It is crucial that the user keeps the array of cards relatively organized in a row. The program identifies and orders the cards in use based on their x positions, rather than their y position. Should a user scramble around the cards, the program is still able to order them but it may become difficult for the user to identify which card is in which position in the array. Additionally, there is no algorithm that adds more cards to the initial array. Any cards pulled out of the deck to the side of the array are not added to the array being sorted.
5

Conclusion

5.1 Summary

The original idea for this project was to visualize graph algorithms such as breadth-first search (BFS), depth-first search (DFS) or minimum spanning tree (MST) on graph data structures and evaluate its effectiveness when used for learning. While the project has changed over the course of the year, it still has revolved around the idea of visualizing a concept in computer science. Instead of BFS and DFS, the final program focused on merge, quick, insertion, and selection sort. It took a physical manipulative used by professors and created a virtual solution for a more convenient and accessible tool. By limiting what is done by the program, the user learns to understand the concept at their own pace with guidance from the computer. It preserves the idea that the manipulative is an object that the user manipulates to help connect visualizations to concepts. The project has potential to develop into much more, to expand its capabilities and to be integrated into classrooms or used by self-learners.
5.2 Future Work

To build upon the program, it would be beneficial to develop a highlighting function for the code being displayed on screen. Every time the learner checks their work and proceeds to another step in the sorting algorithm, the section of code highlighted would change to represent which step is currently being done. This can improve the understanding of what specific step is changing the value of a variable, answering a conditional statement, breaking out of a loop, etc.

More sorting algorithms can be added to allow users to learn even more ways of sorting data. Currently, merge and quick sort are already added to the drop-down menu and the generators list, but there are no generators implemented for them. The goal is to maintain an “interactive” experience with the user. By making the program more automatic, it loses its original purpose as a manipulative that awaits user input. It should await user input and require their effort in doing the sorting on their own. A button can be created such that when the user hovers over it, it reveals an informative paragraph on how to properly use the visualizer as it currently does not explain to users that it does not sort the array on its own.

Another addition that can change the usage of the tool greatly is if a multiplayer function was added to the program. The creator of the Deck of Cards program has a new version that allows for multiplayer, but has not yet released the needed code to allow others to also implement it in their projects. Allowing for multiple users to work on this manipulative together would be beneficial as they can check each other’s work and help each other understand any wrong moves that they made during tracing.

5.3 Experimentation

Due to the emergence of COVID-19 a lot of initial plans for the project had to be re-examined. The proposal was to study the effectiveness of two manipulatives, one virtual and one physical. The study would be conducted on Bard students and would consist
of a short test on graph sorting algorithms such as breadth-first search and depth-first search. The control group would be those who would have no access to either manipulative during the examination while the other groups will have access to a virtual or physical manipulative. Each test-taker will take a before and after questionnaire to gather general information about the test taker and to gather their thoughts on the test itself. The analysis will take into consideration of whether the test taker has previous knowledge of the topic, how well they did, and how useful they found the manipulative (if applicable).
5. CONCLUSION
Appendix A
Appendices

A.1 index.html Code

This is a modified index.html file from Deck of Cards that contains the necessary JavaScript and text element information. The initial positioning of the cards has been modified as well as for the new cards that have been added. The code also contains the required methods for checking user progress.

```html
<!DOCTYPE html>
<html>
<head>
<title>Sorting with Cards</title>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link rel="stylesheet" href="/deckGraphics.css" />
<style>
.choice {
  background-color: #FFFFFF;
}
</style>
</head>
<body>
<pre>
<!--
Original Design by:
Juha Lindstedt

Modifications Made by:
Olivia Witanowska
o.witanowska@gmail.com

Source code for deck of cards and checking sorting algorithms
-->
<!DOCTYPE html>
<html>
<head>
<title>Sorting with Cards</title>
<meta charset="utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link rel="stylesheet" href="/deckGraphics.css" />
<style>
.choice {
  background-color: #FFFFFF;
}
</style>
</head>
<body>
<pre>
```
APPENDIX A. APPENDICES

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++) aux[k] = a[k];
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else a[k] = aux[i++];
```
<div id="ins" style="display: none; background-color:white;" class="choice">
  <pre><code class="application/java">
public static void sort(Comparable[] a) {
  int n = a.length;
  for (int i = 0; i < n; i++) {
    int min = i;
    for (int j = i + 1; j < n; j++) {
      if (less(a[j], a[min])) min = j;
    }
    exch(a, i, min);
  }
}</code></pre>
</div>

<div id="sel" style="display: none; background-color:white;" class="choice">
  <pre><code class="application/java">
public static void sort(Comparable[] a) {
  int n = a.length;
  for (int i = 1; i < n; i++) {
    for (int j = i; j > 0 && less(a[j], a[j-1]); j-- ) {
      exch(a, j, j-1);
    }
  }
}</code></pre>
</div>

<div id="quick" style="display: none; background-color:white;" class="choice">
  <pre><code class="application/java">
private static int partition(Comparable[] a, int lo, int hi) {
  int i = lo;
  int j = hi + 1;
  Comparable v = a[lo];
  while (true) {
    while (less(a[++i], v)) if (i == hi) break;
    while (less(v, a[--j])) if (j == lo) break;
    if (i >= j) break;
    exch(a, i, j);
  }
  exch(a, lo, j);
  return j;
}</code></pre>
</div>
int j = partition(a, lo, hi);
sort(a, lo, j-1);
sort(a, j+1, hi);
}

After you select a sort and move the variable cards, press check.

<script src="/deck.js"></script>

function checkOtherOptions () {
  var choices = document.getElementsByClassName("choice");
  for (var i = 0; i < choices.length; i++) {
    if (choices[i].style.display === "block") {
      choices[i].style.display = "none";
    }
  }
}

function mergeSort () {
  checkOtherOptions ();
  document.getElementById("merge").style.display = "block";
  changeSort("merge");
}

function insSort () {
  checkOtherOptions ();
  document.getElementById("ins").style.display = "block";
  changeSort("ins");
  checkProg();
}

function selSort () {
  checkOtherOptions ();
  document.getElementById("sel").style.display = "block";
  changeSort("sel");
  checkProg();
}

function quickSort () {
  checkOtherOptions ();
  document.getElementById("quick").style.display = "block";
  changeSort("quick");
}

var $container = document.getElementById("container");
A.1. INDEX.HTML CODE

```javascript
degk.mount($container);
degk.shuffle();

var cardsOut = 0;

degk.cards.forEach(function(card, i) {
    // card.setSide(Math.random() < 0.5 ? 'front': 'back');
    card.setSide('back');
card.enableFlipping();
card.enableDragging();

    if (cardsOut < 8 && card.suit < 4) {
        // explode
        cardsOut++;
        activeDeck.push(card);
        card.animateTo({
            delay: 1000 + cardsOut * 2, // wait 1 second + i * 2 ms
            duration: 500,
            ease: "quartOut",
            x: (cardsOut % 8) * 80 - 240, //Math.random() * window.innerWidth - window.innerHeight / 2,
            y: 100 //Math.floor(i / 7) *120-150, //Math.random() * window.innerHeight - window.innerHeight / 2
        });
    } else if (card.suit == 4 && card.rank > 3) {
        // move the jokers down
        varDeck.push(card);
        card.setSide("front");
card.disableFlipping();
card.animateTo({
            delay: 1000 + i * 2, // wait 1 second + i * 2 ms
            duration: 500,
            ease: "quartOut",
            x: -window.innerWidth / 2 + (card.rank - 3) * 60, //Math.random() * window.innerWidth - window.innerHeight / 2,
            y: 250 //Math.random() * window.innerHeight - window.innerHeight / 2
        });
    } else {
        card.setSide("back");
        card.animateTo({
            delay: 1000 + i * 2, // wait 1 second + i * 2 ms
            duration: 500,
            ease: "quartOut",
            x: -window.innerWidth / 2 + 50, //Math.random() * window.innerWidth - window.innerHeight / 2,
            y: 100 //Math.random() * window.innerHeight - window.innerHeight / 2
        });
    }
});
```

// Sorting variable cards in varDeck based on position
var n = varDeck.length;
for (var i = 1; i < n; i++) {
    var el = varDeck[i];
    var j = i - 1;
    while (j >= 0 && varDeck[j].x > el.x) {
        varDeck[j + 1] = varDeck[j];
        j = j - 1;
    }
    varDeck[j + 1] = el;
}

```
// Function checks to see if variable card "i" is in proper position
function checkICard (a, i) {
    distI = Math.abs(a[i].x - varDeck[0].x); // varDeck[0] is == to i variable card
    if (distI < 30) {
        return true;
    }
    return false;
}

// Function checks to see if variable card "j" is in proper position
function checkJCard (a, j) {
    distJ = Math.abs(a[j].x - varDeck[1].x); // varDeck[1] is == to j variable card
    if (distJ < 30) {
        return true;
    }
    return false;
}

// Function checks to see if variable card "min" is in proper position
function checkMinCard (a, min) {
    distMin = Math.abs(a[min].x - varDeck[3].x); // varDeck[3] is == to min variable card
    if (distMin < 30) {
        return true;
    }
    return false;
}

// Function checks to see if the proper sorting was done in tracing
function isSorted (a, i) {
    for (var el = 0; el < i + 1 && el < a.length - 1; el++) {
        if (a[el].rank > a[el + 1].rank) {
            return false; // hasn't been sorted
        }
    }
    return true;
}

// Generator that checks work for the selection sort method
function* selGen () {
    var n = cardOrder.length;
    i = 0;
    yield 0;
    while (i < n) {
        // Check i card
        while (checkICard(cardOrder, i) == false) {
            yield { i: i, j: j, min: min }; // Check i and min card
        }
        yield { i: i, j: j, min: min }; // Check i card
        var min = i;
        while (checkMinCard(cardOrder, min) == false ||
            checkICard(cardOrder, i) == false) {
            yield { i: i, j: j, min: min }; // Check i card
        }
        yield { i: i, j: j, min: min }; // Check i card
        for (var j = i + 1; j < n; j++) {
A.1. INDEX.HTML CODE

```javascript
// Check j card
while (
    checkICard(cardOrder, i) == false ||
    checkJCard(cardOrder, j) == false ||
    checkMinCard(cardOrder, min) == false
) {
    yield { i: i, j: j, min: min }; 
}
yield { i: i, j: j, min: min }; 

if (cardOrder[j].rank < cardOrder[min].rank) {
    min = j;  
    // Check all variable cards
    while (
        checkICard(cardOrder, i) == false ||
        checkJCard(cardOrder, j) == false ||
        checkMinCard(cardOrder, min) == false
    ) { 
        yield { i: i, j: j, min: min }; 
    }
    yield { i: i, j: j, min: min }; 
}
}

var minRank = cardOrder[min].rank;  
while ( isSorted(cardOrder, i, minRank) == false ||
    (cardOrder[i].rank == minRank) == false ) { 
    yield { i: i, j: j, min: min }; 
    i++; 
}
yield { Complete: "good job!" }; 

} // Generator that checks work for the insertion sort method

function* insGen(){
    var n = cardOrder.length;  
yield 0; 
for ( var i = 1; i < n; i++) {  
    // Check i variable card
    while ( checkICard(cardOrder, i) == false ) {  
        yield { i: i, j: j}; 
    }
    yield { i: i, j: j}; 
}

    for ( var j = i; j > 0 && (cardOrder[j].rank < cardOrder[j-1].rank); j-- ) {  
        // Check j and i variable card
        while ( checkJCard(cardOrder, j) == false || checkICard(cardOrder, i) == false ) {  
            console.log('stuck')
            yield { i: i, j: j}; 
        }
    yield { i: i, j: j}; 
    console.log('here');  
    //exch(a, j, j-1);  
    // Check to see if user exchanged cards
    while (cardOrder[j-1].rank > cardOrder[j].rank){
        console.log(cardOrder[j-1])
        console.log(cardOrder[j])
        yield(i:i , j:j ); 
    }
    yield(i:i , j:j ); 
    yield(i:i , j: j-1 );
}
```

APPENDIX A. APPENDICES

// Check to see if previous cards are sorted
while ( isSorted ( cardOrder, i-1 ) == false ){
    yield { i: i, j: j }; // Check to see if previous cards are sorted
    yield { i: i, j: j }; // Check to see if previous cards are sorted
    yield { Complete: "good job!" }; // Check to see if previous cards are sorted
}

// Function that checks the selected sort to see if it matches the current sorting
function changeSort ( sort ) {
    if ( sort != activeSort ) {
        if ( activeSort != null ){
            currentGen.return ();
            document.getElementById ( "check" ).innerHTML = "";
        }
        var generators = {
            "merge": mergeGen (),
            "quick": quickGen (),
            ins: insGen (),
            sel: selGen ()
        };
        activeSort = sort;
        if ( sort in generators ){
            currentGen = generators [ sort ];
            console.log ( currentGen );
        }
    }
}

// Stores order of the "active" cards
var cardOrder = [];

// Function that sorts the active deck and stores it in cardOrder, then calls the current active generator
function checkProg () {
    var cards = [];
    for ( var i = 0; i < activeDeck.length; i++ ) {
        cards.push ( activeDeck [ i ]);
    }
    cardOrder = [];
    for ( var i = 0; i < activeDeck.length; i++ ){
        var lowestX = cards [ 0 ];
        for ( var j = 0; j < cards.length; j++ ){
            if ( lowestX.x > cards [ j ].x ){
                lowestX = cards [ j ];
            }
        }
        cardOrder.push ( lowestX );
    }
    var genValue = currentGen.next ().value;
    document.getElementById ( "check" ).innerHTML = JSON.stringify ( genValue );
}
</script>
</body>
</html>
A.2. DECKGRAPHICS.CSS CODE

A.2 deckGraphics.css Code

This file is the modified version of the CSS program from the original Deck of Cards called example.css. It creates all the cards in the deck as well as fitting the proper card images onto them. It also is responsible for formatting any other new graphics such as the background, text boxes, and buttons.

```css
/* Original Design by:
* Juha Lindstedt
* Modifications Made by:
* Olivia Witanowska
* o.witanowska@gmail.com
* Graphics information */

html, body {
  height: 100%;
  background-color: #EEEEEE; /* #45a173 */
  color: #333;
  font-family: "Open Sans", sans-serif;
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
  overflow: hidden;
  -webkit-text-size-adjust: 100%;
  -ms-text-size-adjust: 100%;
  text-size-adjust: 100%;
}

.progressbtn {
  position: absolute;
  left: 0;
  bottom: 0;
  background-color: #DEDEDE /*#f5f5f5*/;
  color: grey;
  padding: 15px 32px;
  font-size: 16px;
  border-radius: 8px;
}

.choice {
  position: fixed;
  background-color: white;
}

/* Dropdown code obtained from w3schools*/
```
APPENDIX A. APPENDICES

/* Dropdown Button */
.dropbtn {
  background-color: #999999; /* #4CAF50 */
  color: white;
  padding: 16px;
  font-size: 16px;
  border: none;
}

/* The container <div> - needed to position the dropdown content */
dropdown {
  position: relative;
  display: inline-block;
}

/* Dropdown Content (Hidden by Default) */
dropdown-content {
  display: none;
  position: absolute;
  background-color: #fff;
  min-width: 160px;
  box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);
  z-index: 1;
}

/* Links inside the dropdown */
dropdown-content a {
  color: black;
  padding: 12px 16px;
  text-decoration: none;
  display: block;
}

/* Change color of dropdown links on hover */
dropdown-content a:hover { background-color: #ddd; }

/* Show the dropdown menu on hover */
dropdown:hover .dropdown-content { display: block; }

/* Change the background color of the dropdown button when the dropdown content is shown */
dropdown:hover .dropbtn { background-color: #646464 /* #3e8e41 */ ;}

#container {
  position: fixed;
  top: calc(50% + 1.5rem);
  left: 50%;
  -webkit-transform: translate3d(-50%, -50%, 0);
  -moz-transform: translate3d(-50%, -50%, 0);
  -o-transform: translate3d(-50%, -50%, 0);
  transform: translate3d(-50%, -50%, 0);
}

#topbar {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  background-color: #334d41;
  padding: 0.25rem;
  text-align: center;
}

#topbar button {
  background: none;
  border: 1px solid #fff;
```css
A.2. DECKGRAPHICS.CSS CODE

117    outline: none;
118    margin: 0.1rem;
119    padding: 0.4rem;
120    font: inherit;
121    font-size: 0.75rem;
122    line-height: 1;
123    color: #fff;
124    cursor: pointer;
125 }
126  #topbar button:hover {
127      background-color: #fff;
128      color: #444;
129  }
130  .message {
131      position: fixed;
132      top: 2.5rem;
133      left: 0;
134      width: 100%;
135      padding: 1rem 0.5rem;
136      font-size: 0.5rem;
137      text-align: center;
138  }
139  .card {
140      position: absolute;
141      display: inline-block;
142      left: -1.9375rem;
143      top: -2.75rem;
144      width: 3.875rem; /* 11.625rem; */
145      height: 5.5rem; /* 16.5rem; */
146      background-color: #fff;
147      -webkit-border-radius: 4px;
148      border-radius: 4px;
149      -webkit-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);
150      box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);
151      cursor: default;
152      will-change: transform;
153  }
154  .card:before , .card:after {
155    position: absolute;
156    font-size: 1.5rem;
157    text-align: center;
158    line-height: 0.7rem;
159    font-family: "Ubuntu Condensed", sans-serif;
160    white-space: pre-line;
161    width: 0.55rem;
162    letter-spacing: 0.1rem;
163  }
164  .card:before {
165    top: 2.25rem;
166    left: 1.2rem;
167  }
168  .card:after {
169    top: 2.25rem;
170    left: 1.2rem;
171    /* Old card modifications for icons on cards */
172    -webkit-transform: rotate(180deg);
173    -moz-transform: rotate(180deg);
174    -o-transform: rotate(180deg);
175    -ms-transform: rotate(180deg);
176    transform: rotate(180deg);
177  }
178  .card .face {
```
.card .back {
  position: absolute;
  background-image: url("/assets/back.png");
}

.card .joker {
  color: #000;
}

.card.joker.rank4, .card.joker.rank5, .card.joker.rank6,
.card.joker.rank7, .card.joker.rank8, .card.joker.rank9,
.card.joker.rank10, .card.joker.rank11 {
  width: 2.5rem; /* 11.625rem; */
  height: 2.5rem; /* 16.5rem; */
}

.card.joker:before, .card.joker:after: {
  width: 2.5rem; /* 11.625rem; */
  height: 2.5rem; /* 16.5rem; */
}

.card.hearts, .card.diamonds, .card.joker.rank3 {
  color: #d40000;
}

.card.joker.rank1:before, .card.joker.rank2:before, .card.joker.rank1:after, .card.joker.rank2:after, {
  /* content: "J\a O\a K\a E\a R"; */
  letter-spacing: 0;
  font-size: 0.4rem;
  line-height: 0.4rem;
  padding: 0.15rem 0.05rem;
  width: 0.5rem;
}

.card.rank1:before, .card.rank1:after {
A.2. DECKGRAPHICS.CSS CODE

```css
.card.rank2:before, .card.rank2:after {
    /* content: "2"; */
}
.card.rank3:before, .card.rank3:after {
    /* content: "3"; */
}
.card.rank4:before, .card.rank4:after {
    /* content: "4"; */
}
.card.rank5:before, .card.rank5:after {
    /* content: "5"; */
}
.card.rank6:before, .card.rank6:after {
    /* content: "6"; */
}
.card.rank7:before, .card.rank7:after {
    /* content: "7"; */
}
.card.rank8:before, .card.rank8:after {
    /* content: "8"; */
}
.card.rank9:before, .card.rank9:after {
    /* content: "9"; */
}
.card.rank10:before, .card.rank10:after {
    /* content: "10"; */
}
.card.rank11:before, .card.rank11:after {
    /* content: "J"; */
}
.card.rank12:before, .card.rank12:after {
    /* content: "Q"; */
}
.card.rank13:before, .card.rank13:after {
    /* content: "K"; */
}
```
APPENDIX A. APPENDICES

.card.joker.rank10:after,  
.card.joker.rank11:before,  
.card.joker.rank11:after{
    left: 0rem;  
    top: 1rem;  
    font-size:1rem;  
} /*The following lines create variable cards */  
.card.joker.rank4:before,  
.card.joker.rank4:after {
    content: "i";  
}

.card.joker.rank5:before,  
.card.joker.rank5:after {  
    content: "j";  
}

.card.joker.rank6:before,  
.card.joker.rank6:after {  
    content: "k";  
}

.card.joker.rank7:before,  
.card.joker.rank7:after {  
    content: "min";  
}

.card.joker.rank8:before,  
.card.joker.rank8:after {  
    content: "n";  
}

.card.joker.rank9:before,  
.card.joker.rank9:after {  
    content: "lo";  
}

.card.joker.rank10:before,  
.card.joker.rank10:after {  
    content: "mid";  
}

.card.joker.rank11:before,  
.card.joker.rank11:after {  
    content: "hi";  
}  
} /*The following paste images onto the cards */  
.card.spades.rank1 .face {
    background-image: url("/assets/Final_SA.jpg");  
}

.card.spades.rank2 .face {  
    background-image: url("/assets/Final_S2.jpg");  
}

.card.spades.rank3 .face {  
    background-image: url("/assets/Final_S3.jpg");  
}

.card.spades.rank4 .face {  
    background-image: url("/assets/Final_S4.jpg");  
}

.card.spades.rank5 .face {  
    background-image: url("/assets/Final_S5.jpg");  
}

.card.spades.rank6 .face {  
    background-image: url("/assets/Final_S6.jpg");  
}

.card.spades.rank7 .face {  
    background-image: url("/assets/Final_S7.jpg");  
}

.card.spades.rank8 .face {  
    background-image: url("/assets/Final_S8.jpg");  
}

.card.spades.rank9 .face {
A.2. DECKGRAPHICS.CSS CODE

```css
background-image: url("/assets/Final_S9.jpg");
}
.
card.spades.rank10 .face {
  background-image: url("/assets/Final_S10.jpg");
}
.
card.spades.rank11 .face {
  background-image: url("/assets/Final_SJ.jpg");
}
.
card.spades.rank12 .face {
  background-image: url("/assets/Final_SQ.jpg");
}
.
card.spades.rank13 .face {
  background-image: url("/assets/Final.SK.jpg");
}
.
card.hearts.rank1 .face {
  background-image: url("/assets/Final_HA.jpg");
}
.
card.hearts.rank2 .face {
  background-image: url("/assets/Final_H2.jpg");
}
.
card.hearts.rank3 .face {
  background-image: url("/assets/Final_H3.jpg");
}
.
card.hearts.rank4 .face {
  background-image: url("/assets/Final_H4.jpg");
}
.
card.hearts.rank5 .face {
  background-image: url("/assets/Final_H5.jpg");
}
.
card.hearts.rank6 .face {
  background-image: url("/assets/Final_H6.jpg");
}
.
card.hearts.rank7 .face {
  background-image: url("/assets/Final_H7.jpg");
}
.
card.hearts.rank8 .face {
  background-image: url("/assets/Final_H8.jpg");
}
.
card.hearts.rank9 .face {
  background-image: url("/assets/Final_H9.jpg");
}
.
card.hearts.rank10 .face {
  background-image: url("/assets/Final_H10.jpg");
}
.
card.hearts.rank11 .face {
  background-image: url("/assets/Final_HJ.jpg");
}
.
card.hearts.rank12 .face {
  background-image: url("/assets/Final_HQ.jpg");
}
.
card.hearts.rank13 .face {
  background-image: url("/assets/Final_HK.jpg");
}
.
card.clubs.rank1 .face {
  background-image: url("/assets/Final_CA.jpg");
}
.
card.clubs.rank2 .face {
  background-image: url("/assets/Final_C2.jpg");
}
.
card.clubs.rank3 .face {
  background-image: url("/assets/Final_C3.jpg");
}
.
card.clubs.rank4 .face {
  background-image: url("/assets/Final_C4.jpg");
}
```
A.3. DECK.JS CODE

This file contains the Deck function that is responsible for the cards and all of their properties. It gets called upon in index.html to create the deck of cards. The Deck() function must be passed the boolean true so that the generated card deck includes jokers. The variable cards added in deckGraphics.css use the joker suit to differentiate them from the other four suits.

```javascript
use strict;

var Deck = (function () {
  /*
   * Author:
   * Juha Lindstedt
   *
   * Functions for creating deck and changing card properties
   */
  use strict;
  html {
    font-size: 125%;
  }
```

var ticking;
var animations = [];

function animationFrames(delay, duration) {
    var now = Date.now();

    // calculate animation start/end times
    var start = now + delay;
    var end = start + duration;

    var animation = {
        start: start,
        end: end
    };

    // add animation
    animations.push(animation);

    if (!ticking) {
        // start ticking
        ticking = true;
        requestAnimationFrame(tick);
    }

    var self = {
        start: function start(cb) {
            // add start callback (just one)
            animation.startcb = cb;
            return self;
        },
        progress: function progress(cb) {
            // add progress callback (just one)
            animation.progresscb = cb;
            return self;
        },
        end: function end(cb) {
            // add end callback (just one)
            animation.endcb = cb;
            return self;
        }
    };

    return self;
}

function tick() {
    var now = Date.now();

    if (!animations.length) {
        // stop ticking
        ticking = false;
        return;
    }

    for (var i = 0, animation; i < animations.length; i++) {
        animation = animations[i];
        if (now < animation.start) {
            // animation not yet started..
            continue;
        }
        if (!animation.started) {
            // animation starts
            animation.started = true;
            animation.startcb && animation.startcb();
        }
    }
}
A.3. DECK.JS CODE

```javascript
// animation progress
var t = (now - animation.start) / (animation.end - animation.start);  
animation.progresscb && animation.progresscb(t < 1 ? t : 1);  
if (now > animation.end) {  
// animation ended
animation.endcb && animation.endcb();  
animations.splice(i--, 1);  
continue;  
}  
requestAnimationFrame(tick);  
}

// fallback
window.requestAnimationFrame || (window.requestAnimationFrame = function (cb) {
    setTimeout(cb, 0);
});

var style = document.createElement('p').style;
var memoized = {};  
function prefix(param) {
    if (typeof memoized[param] !== 'undefined') {
        return memoized[param];
    }  
    if (typeof style[param] !== 'undefined') {
        memoized[param] = param;
        return param;
    }  

    var camelCase = param[0].toUpperCase() + param.slice(1);
    var prefixes = ['webkit', 'moz', 'Moz', 'ms', 'o'];
    var test;
    for (var i = 0, len = prefixes.length; i < len; i++) {
        test = prefixes[i] + camelCase;
        if (typeof style[test] !== 'undefined') {
            memoized[param] = test;
            return test;
        }
    }
}

var has3d;
function translate(a, b, c) {
    typeof has3d !== 'undefined' || (has3d = check3d());
    c = c || 0;
    if (has3d) {
        return 'translate3d(' + a + ' + , ' + b + ' + , ' + c + ')';
    } else {
        return 'translate(' + a + ' + , ' + b + ')';
    }
}

function check3d() {
    // I admit, this line is stolen from the great Velocity.js!
    // http://julian.com/research/velocity/
    var isMobile = /Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera Mini/i.test(navigator.userAgent);
    ```
if (!isMobile) {
  return false;
}

var transform = prefix('transform');
var $p = document.createElement('p');
document.body.appendChild($p);
$p.style[transform] = 'translate3d(1px,1px,1px)';
has3d = $p.style[transform];
has3d = has3d != null && has3d.length && has3d !== 'none';
document.body.removeChild($p);
return has3d;

function createElement(type) {
  return document.createElement(type);
}

var maxZ = 52;

function _card(i) {
  var transform = prefix('transform');

  // calculate rank/suit, etc..
  var rank = i % 13 + 1;
  var suit = i / 13 | 0;
  var z = (52 - i) / 4;

  // create elements
  var $el = createElement('div');
  var $face = createElement('div');
  var $back = createElement('div');

  // states
  var isDraggable = false;
  var isFlippable = false;

  // self = card
  var self = { i: i, rank: rank, suit: suit, pos: i, $el: $el, mount: mount, unmount: unmount, setSide: setSide };
A.3. DECK.JS CODE

```javascript
// add drag/click listeners
addListener($el, 'mousedown', onMousedown);
addListener($el, 'touchstart', onMousedown);

// load modules
for (module in modules) {
    addModule(modules[module]);
}

self.animateTo = function (params) {
    var delay = params.delay;
    var duration = params.duration;
    var _params =
        x === undefined ? self.x : _params.x;
    var y = _params.y === undefined ? self.y : _params.y;
    var _params =
        rot === undefined ? self.rot : _params.rot;
    var ease$$ = params.ease;
    var onStart = params.onStart;
    var onProgress = params.onProgress;
    var onComplete = params.onComplete;

    var startX, startY, startRot;
    var diffX, diffY, diffRot;

    animationFrames(delay, duration).start(function () {
        startX = self.x || 0;
        startY = self.y || 0;
        startRot = self.rot || 0;
        onStart && onStart();
    }).progress(function (t) {
        var et = ease$$['cubicInOut'](t);
        diffX = x - startX;
        diffY = y - startY;
        diffRot = rot - startRot;
        onProgress && onProgress(t, et);
        self.x = startX + diffX * et;
        self.y = startY + diffY * et;
        self.rot = startRot + diffRot * et;

        $el.style[transform] = translate(self.x + 'px', self.y + 'px') + (diffRot ? 'rotate(' + self.rot + 'deg) : ');
    }).end(function () {
        onComplete && onComplete();
    });
};

// set rank & suit
self.setRankSuit = function (rank, suit) {
    var suitName = SuitName(suit);
    $el.setAttribute('class', 'card ' + suitName + ' rank' + rank);
};

self.setRankSuit(rank, suit);

self.enableDragging = function () {
    // this activates dragging
    if (isDraggable) {
        // already is draggable, do nothing
        return;
    }
};
```
self.enableFlipping = function () {
  if (isFlippable) {
    // already is flippable, do nothing
    return;
  }
  isFlippable = true;
};

self.disableFlipping = function () {
  if (!isFlippable) {
    // already disabled flippable, do nothing
    return;
  }
  isFlippable = false;
};

self.disableDragging = function () {
  if (!isDraggable) {
    // already disabled dragging, do nothing
    return;
  }
  isDraggable = false;
  $el.style.cursor = '';
};

return self;

function addModule(module) {
  // add card module
  module.card && module.card(self);
}

function onMousedown(e) {
  var startPos = {};
  var pos = {};
  var starttime = Date.now();

  e.preventDefault();

  if (e.type === 'mousedown') {
    startPos.x = pos.x = e.clientX;
    startPos.y = pos.y = e.clientY;
    addListener(window, 'mousemove', onMousemove);
    addListener(window, 'mouseup', onMouseup);
  } else {
    startPos.x = pos.x = e.touches[0].clientX;
    startPos.y = pos.y = e.touches[0].clientY;
    addListener(window, 'touchmove', onMousemove);
    addListener(window, 'touchend', onMouseup);
  }

  if (!isDraggable) {
    // is not draggable, do nothing
    return;
  }

  // move card
function onMousemove(e) {
  if (!isDraggable) {
    // is not draggable, do nothing
    return;
  }
  if (e.type === 'mousemove') {
    pos.x = e.clientX;
    pos.y = e.clientY;
  } else {
    pos.x = e.touches[0].clientX;
    pos.y = e.touches[0].clientY;
  }

  // move card
  $el.style[transform] = translate(Math.round(self.x + pos.x - startPos.x) + 'px', Math.round(self.y + pos.y - startPos.y) + 'px') + (self.rot ? 'rotate(' + self.rot + 'deg) : '');
}

function onMouseup(e) {
  if (isFlippable && Date.now() - starttime < 200) {
    // flip sides
    self.setSide(self.side === 'front' ? 'back' : 'front');
  }
  if (e.type === 'mouseup') {
    removeListener(window, 'mousemove', onMousemove);
    removeListener(window, 'mouseup', onMouseup);
  } else {
    removeListener(window, 'touchmove', onMousemove);
    removeListener(window, 'touchend', onMouseup);
  }
  if (!isDraggable) {
    // is not draggable, do nothing
    return;
  }

  // set current position
  self.x = self.x + pos.x - startPos.x;
  self.y = self.y + pos.y - startPos.y;
}

function mount(target) {
  // mount card to target (deck)
  target.appendChild($el);
  self.$root = target;
}

function unmount() {
  // unmount from root (deck)
  self.$root && self.$root.removeChild($el);
  self.$root = null;
}

function setSide(newSide) {
  // flip sides
  if (newSide === 'front') {
    if (self.side === 'back') {
      $el.removeChild($back);
    }
  } else {
    if (self.side === 'front') {
      $el.appendChild($back);
    }
  }
}
APPENDIX A. APPENDICES

389     } else {
390         if (self.side === 'front') {
391             $el.removeChild($face);
392         }
393         self.side = 'back';
394         $el.appendChild($back);
395         $el.setAttribute('class', 'card');
396     }
397 }
398 }
399 }
400 }
401 }
402 }
403 function SuitName(suit) {
404     // return suit name from suit value
405     return suit === 0 ? 'spades' : suit === 1 ? 'hearts' : suit === 2 ? 'clubs' :
406         suit === 3 ? 'diamonds' : 'joker';
407 }
408 function addListener(target, name, listener) {
409     target.addEventListener(name, listener);
410 }
411 }
412 function removeListener(target, name, listener) {
413     target.removeEventListener(name, listener);
414 }
415 }
416 var ease = {
417     linear: function linear(t) {
418         return t;
419     },
420     quadIn: function quadIn(t) {
421         return t * t;
422     },
423     quadOut: function quadOut(t) {
424         return t * (2 - t);
425     },
426     quadInOut: function quadInOut(t) {
427         return t < 0.5 ? 2 * t * t : -1 + (4 - 2 * t) * t;
428     },
429     cubicIn: function cubicIn(t) {
430         return t * t * t;
431     },
432     cubicOut: function cubicOut(t) {
433         return --t * t * t + 1;
434     },
435     cubicInOut: function cubicInOut(t) {
436         return t < 0.5 ? 4 * t * t * t : (t - 1) * (2 * t - 2) * (2 * t - 2) + 1;
437     },
438     quartIn: function quartIn(t) {
439         return t * t * t;
440     },
441     quartOut: function quartOut(t) {
442         return 1 - --t * t * t;
443     },
444     quartInOut: function quartInOut(t) {
445         return t < 0.5 ? 8 * t * t * t * t : 1 - 8 * --t * t * t;
446     },
447     quintIn: function quintIn(t) {
448         return t * t * t;
449     },
450     quintOut: function quintOut(t) {
55 return 1 + --t * t * t * t * t;
56 }
57 quintInOut: function quintInOut(t) {
58 return t < 0.5 ? 16 * t * t * t * t * t : 1 + 16 * --t * t * t * t * t;
59 }
60 );
61 var flip = {
62     deck: function deck(_deck) {
63         _deck.flip = _deck.queued(flip);
64     }
65     }
66         function flip(next, side) {
67             var flipped = _deck.cards.filter(function (card) {
68                 return card.side === 'front';
69             }).length / _deck.cards.length;
70             _deck.cards.forEach(function (card, i) {
71                 card.setSide(side ? side : flipped > 0.5 ? 'back' : 'front');
72             });
73             next();
74         }
75     }
76 var sort = {
77     deck: function deck(_deck2) {
78         _deck2.sort = _deck2.queued(sort);
79     }
80     }
81         function sort(next, reverse) {
82             var cards = _deck2.cards;
83             cards.sort(function (a, b) {
84                 if (reverse) {
85                     return a.i - b.i;
86                 } else {
87                     return b.i - a.i;
88                 }
89             });
90             cards.forEach(function (card, i) {
91                 card.sort(i, cards.length, function (i) {
92                     if (i === cards.length - 1) {
93                         next();
94                     }
95                 }, reverse);
96             });
97         }
98         }
99     },
100     card: function card(_card2) {
101         var $el = _card2.$el;
102         _card2.sort = function (i, len, cb, reverse) {
103             var z = i / 4;
104             var delay = i * 10;
105             _card2.animateTo({
106                 delay: delay,
107                 duration: 400,
108                 x: -z,
109                 y: -150,
110                 rot: 0,
111                 onComplete: function onComplete() {
112                 }
113             });
114         }
115     }
116 }
117 };
$
.el.style.zIndex = i;
}
});

_card2.animateTo(
  delay: delay + 500,
  duration: 400,
  x: -z,
  y: -z,
  rot: 0,
  onComplete: function onComplete() {
    cb(i);
  }
});

function plusminus(value) {
  var plusminus = Math.round(Math.random()) ? -1 : 1;
  return plusminus * value;
}

function fisherYates(array) {
  var rnd, temp;
  for (var i = array.length - 1; i; i--) {
    rnd = Math.random() * i | 0;
    temp = array[i];
    array[i] = array[rnd];
    array[rnd] = temp;
  }
  return array;
}

function fontSize() {
  return window.getComputedStyle(document.body).getPropertyValue( 'font-size' ).slice( 0, -2);
}

var ____fontSize;

var shuffle = {
  deck: function deck(_deck3) {
    _deck3.shuffle = _deck3.queued(shuffle);
    function shuffle(next) {
      var cards = _deck3.cards;
      ____fontSize = fontSize();
      fisherYates(cards);
      cards.forEach(function (card, i) {
        card.pos = i;
        card.shuffle(function (i) {
          if (i === cards.length - 1) {
            next();
          }
        });
      });
      next();
    }
    return function (next) {
      var i = cards.length - 1;
      cards.forEach(function (card, i) {
        card.pos = i;
        card.shuffle(function (i) {
          if (i === cards.length - 1) {
            next();
          }
        });
      });
      next();
    }
  }
};

APPENDIX A. APPENDICES
A.3. DECK.JS CODE

```javascript
579            });
580        });
581        return;
582    }
583
584    card: function card(_card3) {
585        var $el = _card3.$el;
586
587    _card3.shuffle = function (cb) {
588        var i = _card3.pos;
589        var z = i / 4;
590        var delay = i * 2;
591
592    _card3.animateTo({
593        delay: delay,
594        duration: 200,
595
596    x: plusminus(Math.random() * 40 + 20) * ___fontSize / 16,
597    y: -z,
598    rot: 0
599    });
600    _card3.animateTo({
601        delay: 200 + delay,
602        duration: 200,
603
604    x: -z,
605    y: -z,
606    rot: 0,
607
608    onStart: function onStart() {
609        $el.style.zIndex = i;
610    },
611
612    onComplete: function onComplete() {
613        cb(i);
614    }
615    });
616    }
617    };}
618
619    var __fontSize;
620
621    var poker = {
622        deck: function deck(_deck4) {
623    _deck4.poker = _deck4.queued(poker);
624
625    function poker(next) {
626        var cards = _deck4.cards;
627        var len = cards.length;
628
629    __fontSize = fontSize();
630
631    cards.slice(-5).reverse().forEach(function (card, i) {
632        card.poker(i, len, function (i) {
633            card.setSide('front');
634            if (i === 4) {
635                next();
636
637            }
638        });
639    });
640
641    },
642```
card: function card(_card4) {
    var $el = _card4.$el;

    _card4.poker = function (i, len, cb) {
        var delay = i * 250;

        _card4.animateTo({
            delay: delay,
            duration: 250,
            x: Math.round((i - 2.05) * 70 * __fontSize / 16),
            y: Math.round(-110 * __fontSize / 16),
            rot: 0,
            onStart: function onStart() {
                $el.style.zIndex = len - 1 + i;
            },
            onComplete: function onComplete() {
                cb(i);
            }
        });
    }
}

var intro = {
    deck: function deck(_deck5) {
        _deck5.intro = _deck5.queued(intro);
        function intro(next) {
            var cards = _deck5.cards;
            cards.forEach(function (card, i) {
                card.setSide('front');
                card.intro(i, function (i) {
                    animationFrames(250, 0).start(function () {
                        card.setSide('back');
                    });
                    if (i === cards.length - 1) {
                        next();
                    }
                });
            });
            card: function card(_card5) {
                var transform = prefix('transform');
                var $el = _card5.$el;

                _card5.intro = function (i, cb) {
                    var delay = 500 + i * 10;
                    var z = i / 4;
                    $el.style[transform] = translate(-z + 'px', '-250px');
                    $el.style.opacity = 0;
                    _card5.x = -z;
                    _card5.y = -250 - z;
                    _card5.rot = 0;
                    _card5.animateTo({
                        delay: delay,
                        duration: 1000,
                    });
                };
            }
        }
    }
};
A.3. DECK.JS CODE

```javascript
var _fontSize;

var fan = {
  deck: function deck(_deck6) {
    _deck6.fan = _deck6.queued(fan);
    
    function fan(next) {
      var cards = _deck6.cards;
      var len = cards.length;
      
      _fontSize = fontSize();
      
      cards.forEach(function (card, i) {
        card.fan(i, len, function (i) {
          if (i === cards.length - 1) {
            next();
          }
        });
      });
    },
    
    card: function card(_card6) {
      var $el = _card6.$el;
      
      _card6.fan = function (i, len, cb) {
        var z = i / 4;
        var delay = i * 10;
        var rot = i / (len - 1) * 260 - 130;
        
        _card6.animateTo({
          delay: delay,
          duration: 300,
          x: -z,
          y: -z,
          rot: 0
        });
        _card6.animateTo({
          delay: 300 + delay,
          duration: 300,
          x: Math.cos(deg2rad(rot - 90)) * 55 * _fontSize / 16,
          y: Math.sin(deg2rad(rot - 90)) * 55 * _fontSize / 16,
          rot: rot,
        });
      },
    },
  }
};
```

```javascript
onStart: function onStart () {
  $el.style.zIndex = i;
},

onProgress: function onProgress(t) {
  $el.style.opacity = t;
},

onComplete: function onComplete () {
  $el.style.opacity = '';
  cb && cb(i);
}
```
onStart: function onStart() {
    $el.style.zIndex = i;
},

onComplete: function onComplete() {
    cb(i);
}
});
});
});

function deg2rad(degrees) {
    return degrees * Math.PI / 180;
}

var __fontSize;

var bysuit = {
    deck: function deck(_deck7) {
        _deck7.bysuit = _deck7.queued(bysuit);
    }

    function bysuit(next) {
        var cards = _deck7.cards;
        __fontSize = fontSize();
        cards.forEach(function (card) {
            card.bysuit(function (i) {
                if (i === cards.length - 1) {
                    next();
                }
            });
        });
    }

    card: function card(_card7) {
        var rank = _card7.rank;
        var suit = _card7.suit;
        _card7.bysuit = function (cb) {
            var i = _card7.i;
            var delay = i * 10;

            _card7.animateTo(
                delay: delay,
                duration: 400,
                x: -Math.round((6.75 - rank) * 8 * __fontSize / 16),
                y: -Math.round((1.5 - suit) * 92 * __fontSize / 16),
                rot: 0,
                onComplete: function onComplete() {
                    cb(i);
                }
            );
        }
    }

    function queue(target) {
        var array = Array.prototype;
        var queueing = [];
    }
A.3. DECK.JS CODE

```javascript
// A.3. DECK.JS CODE

const target = {};

// Initialize queue and queued properties
target.queue = queue;
target.queued = queued;

// Return the target
return target;

// Function queued
function queued(action) {
  return function () {
    var self = this;
    var args = arguments;
    queue(function (next) {
      action.apply(self, array.concat.apply(next, args));
    });
  }
}

// Function queue
function queue(action) {
  if (!action) {
    return;
  }
  queueing.push(action);
  if (queueing.length === 1) {
    next();
  }
}

// Function next
function next() {
  queueing[0](function (err) {
    if (err) {
      throw err;
    }
    queueing = queueing.slice(1);
    if (queueing.length) {
      next();
    }
  });
}

// Function observable
function observable(target) {
  target || (target = {});
  var listeners = {};
  target.on = on;
target.one = one;
target.off = off;
target.trigger = trigger;
return target;
}

// Function on
function on(name, cb, ctx) {
  listeners[name] || (listeners[name] = []);
  listeners[name].push({ cb: cb, ctx: ctx });
}

// Function one
function one(name, cb, ctx) {
  listeners[name] || (listeners[name] = []);
  listeners[name].push({
    cb: cb, ctx: ctx, once: true
  });
```
function trigger(name) {
    var self = this;
    var args = Array.prototype.slice(arguments, 1);
    var currentListeners = listeners[name] || [];
    currentListeners.filter(function (listener) {
        listener.cb.apply(self, args);
        return ! listener.once;
    });
}

function off(name, cb) {
    if (!name) {
        listeners = {};
        return;
    }
    if (!cb) {
        listeners[name] = [];
        return;
    }
    listeners[name] = listeners[name].filter(function (listener) {
        return listener.cb !== cb;
    });
}

function Deck(jokers) {
    // init cards array
    var cards = new Array(jokers ? 63 : 52);
    var $el = createElement('div');
    var self = observable({ mount: mount, unmount: unmount, cards: cards, $el: $el });
    var $root;
    var modules = Deck.modules;
    var module;
    // make queueable
    queue(self);
    // load modules
    for (module in modules) {
        addModule(modules[module]);
    }
    // add class
    $el.classList.add('deck');
    var card;
    // create cards
    for (var i = cards.length; i--; i--) {
        card = cards[i - 1] = _card(i - 1);
        card.setSide('back');
        card.mount($el);
    }
}
return self;

function mount(root) {
  // mount deck to root
  $root = root;
  $root.appendChild($el);
}

function unmount() {
  // unmount deck from root
  $root.removeChild($el);
}

function addModule(module) {
  module.deck && module.deck(self);
}

Deck.animationFrames = animationFrames;
Deck.ease = ease;
Deck.modules = {bysuit: bysuit, fan: fan, intro: intro, poker: poker, shuffle:
  shuffle, sort: sort, flip: flip};
Deck.Card = _card;
Deck.prefix = prefix;
Deck.translate = translate;

return Deck;

});()
Bibliography


