Airplane Cabin Boarding Methods

Adam Zolan
Kennesaw State University

1 INTRODUCTION

We will create an airplane cabin boarding simulator program to test various methods of loading passengers in a queue. The passengers in this queue each have a goal to reach their assigned seat in an “airplane cabin”-style data structure, while also obeying the rules of queueing alongside the other passengers. The behavior of each single passenger is defined by a simple Finite State Machine (FSM) which progresses linearly to describe the passenger’s current short-term goal. The simulator will create all passengers, load each passenger into the initial queue based on a specified sorting algorithm, provide an “airplane cabin” set of end goals for all passengers, then execute the behavior of each passenger as they try to reach their target seat. The simulator will also measure the average amount of time each passenger spends trying to reach their seat. These statistics will be examined across hundreds of different iterations of the simulator in order to find out which queueing algorithms are the most efficient. The efficiency of these algorithms will be compared against those measured by the algorithms’ original proposer, Jason H. Steffen.

2 BASIC GOALS AND DESIGN

The ultimate goal of this simulation will be to emulate the real-world problems involved with queueing and boarding passengers onto an airplane, and measure how long it takes for a single passenger to maneuver and solve these problems through a basic decision-making process. We will outline a single passenger’s goals by using the following real-world obstacles:

1. The passenger is assigned a target seat that is located somewhere within the airplane’s multiple rows of seats.
2. The passenger will wait in a queue before boarding the airplane.
3. The passenger must move through the aisle in the airplane, with limited space for movement, until the target row is reached.
    - This includes waiting for any other passengers ahead of this passenger to finish any tasks before progressing. The passenger should not physically occupy the same space as another passenger.
4. The passenger must spend time stowing a bag before sitting down.
5. The passenger will sit down in the assigned seat and perform no more actions.

Fig 1. Passenger basic goals

For the sake of this simulation, we will define the airplane cabin to be a single lane of space , with rows of seats placed on both sides (port and starboard) of the aisle. The aisle itself has an entry point at the front of the airplane's cabin. This entry point is where passengers leave the waiting queue and enter the cabin aisle. Each row of seats has a designated row number, and each seat within the row has a designated seat number. Once all passengers have achieved their final goal of sitting in their assigned seat, the simulation will end.

Because passenger waiting time is the primary statistic to be measured, we will create a system that prioritizes simple tracking of how long it takes a single passenger automaton to achieve a "success" state. The goals of a single passenger progress in a linear fashion. This means that we can define the behavior of a passenger as a simple Finite State Machine, where the passenger changes their state directly in order as follows:

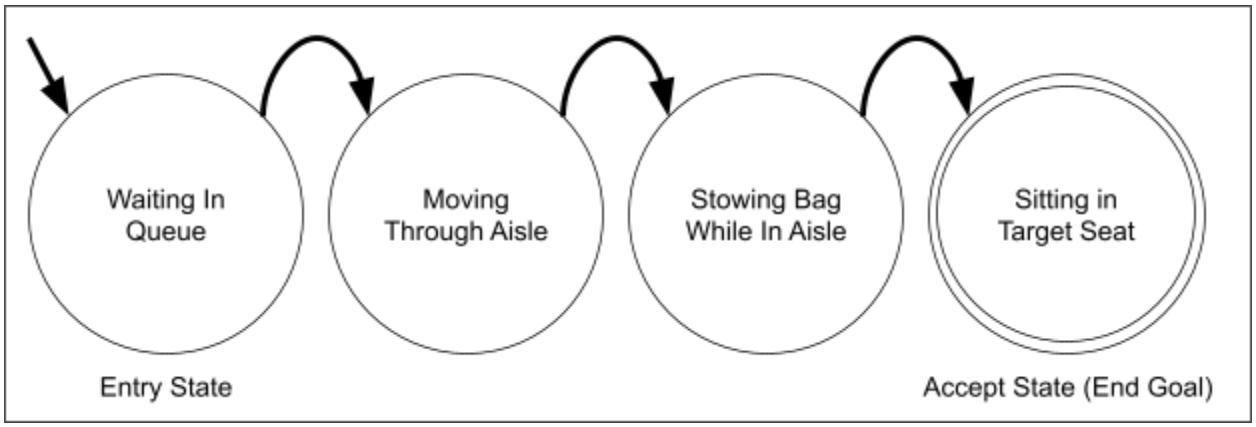

Fig 2. Passenger Finite State Machine

Additionally, we will define a set of conditions that dictate when a passenger may transition from one state to the next in the State Machine:

1. From "Waiting In Queue" to "Moving Through Aisle"
    1. Passenger must be at the front of the queue
    2. There must be an occupiable space at the beginning of the cabin aisle to move into
2. From "Moving Through Aisle" to "Stowing Bag While In Aisle"
    1. Passenger must reach the row of seats that its target seat is located in
3. From "Stowing Bag While In Aisle" to "Sitting in Target Seat"
    1. Passenger must have finished spending all allotted stow time

Fig 3. Passenger State Transition Conditions

One major benefit to a linear progression of turns like this is that, because the passenger never regresses to a previous state, and it is only ever concerned with the actions of other passengers ahead of them in the turn order, then there is no chance that one passenger's state will conflict with another's and cause a deadlock-like situation. Passengers are only ever potentially concerned with the actions and states of the passengers ahead of them in the queue and the aisle; a passenger will never need to make a decision based on the needs of a passenger behind them in the queue.

One important aspect of the passenger's progression is the movement into, through, and out of the cabin's aisle. Basically, there must be some structure in place such that passengers have limited physical space within the airplane cabin's aisle. This space is limited such that if a passenger is stationary within the aisle, the passenger immediately behind them must wait until they are no longer stationary in order to move down the aisle. The exact structure of this simulated physical space will be detailed later in Section 3.

We will also need to make sure this system is capable of measuring a large number of these passengers equally. Instead of tracking each passenger in real time, we will implement a "turn" system as the means of time measurement within the simulation. This will follow rules similar to a board game, where each player performs some action on their turn individually, before focus is shifted to the next waiting player. We will use the same mentality for handling how and when passengers make their decisions; by convention, every passenger in the waiting queue will be given a "turn" that they will use to examine their state, see their available options, and make a decision based on their state and options.

Passengers will be processed in order from the front of the waiting queue to the back; also note that this turn order is constant throughout the simulation, and is independent of each passenger's state. Once all passengers make some decision and action on their turn, control is then passed back to the first passenger, and the cycle of decision making starts again. When this happens, a global timer will be incremented, signifying the total amount of turns that have passed while the simulation has been running. When iterating over all passengers, if a passenger has reached their final end goal state, their turn should be automatically skipped.

3 DATA STRUCTURE DESIGN

Based on our design of the Passenger in Section 2, we can simply define a Passenger as a class that contains the basic information about a Passenger, including its state, target seat, stowing time, and overall lifetime. The initial stowing time of the passenger can be a preset number of turns, or set to be a random integer for more dynamic simulations.

```
class Passenger
  int id;
  PassengerState state;
  int targetRow;
  int targetSeat;
  int stowingTime;
  int lifetime;
```

Fig 4. Passenger class pseudocode

In the actual implementation of the simulator, the passenger's behavior is implemented in the "main" method. This behavior can also be implemented in methods of the Passenger class. However, using the "main" method approach allowed for easier global control and access to all Passengers and their decisions.

Next is to design the data structures that describe the physical spaces that the passengers will occupy. We will first load all passengers into a Queue structure, which is already a common data structure in most programming languages. After this ready queue, passengers must move throughout the aisle of the cabin, which is associated with the rows of seats within the airplane. To design this airplane cabin, we will again return to the board game analogy. We can define the cabin's aisle as a sequence of "spaces" that a passenger may occupy, similar to spaces on a game board. Along these aisle spaces, we can define occupiable seat spaces for each seat in each row as well.

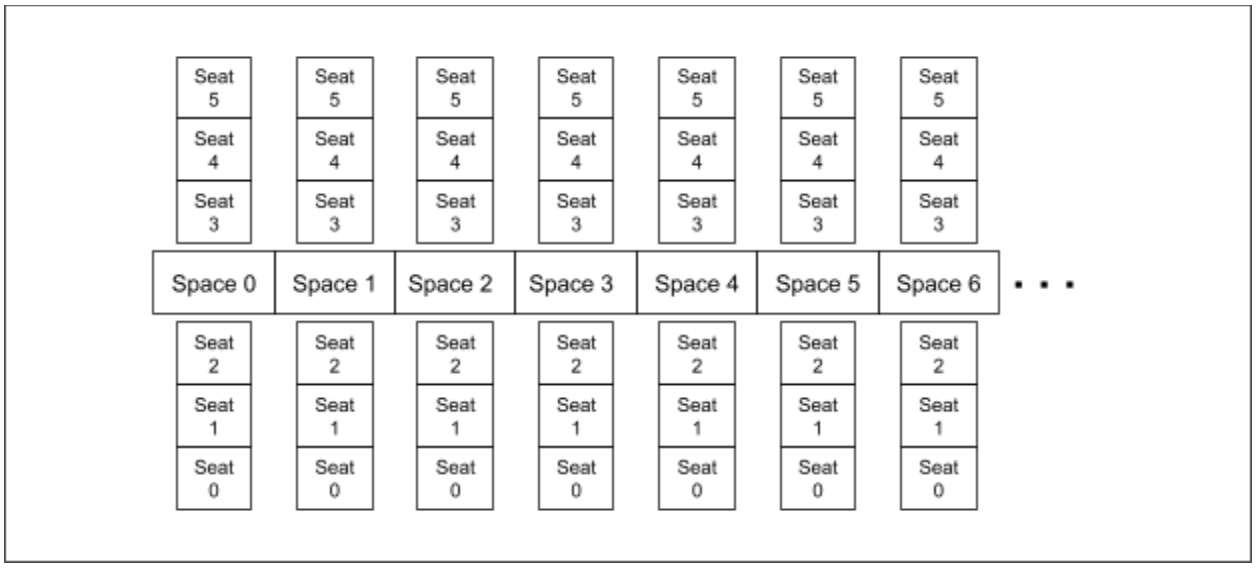

Fig 5. Diagram of the Airplane Cabin structure

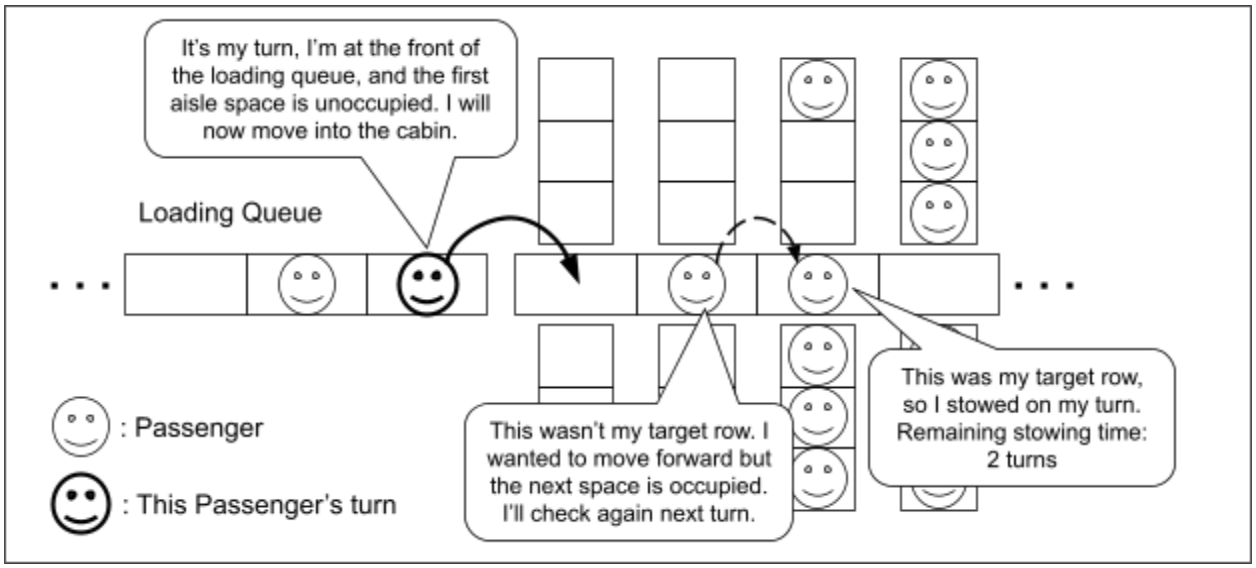

Fig 6. Example snapshot of passenger behavior within the Loading Queue and the Airplane Cabin. Note the various decisions that passengers made this turn, depending on their current row and next aisle space.

Following the diagram in Fig 6., the passenger at the front of the loading queue can occupy Space 0 if it is unoccupied. Passengers may then spend one turn to move forward one space, assuming the next space is unoccupied. When a passenger reaches their target row, they will be spending any number of turns stowing their bag while occupying an aisle space. This will cause a passenger in the previous space to wait for the stowing passenger to finish, which may cause the passenger behind them to wait as well, and so on. Passengers must be able to query the space ahead of them to see if it is unoccupied; depending on its status, it may or may not be possible for the passenger to progress into or through the aisle.

```
class OccupiableSpace
  boolean Occupied;
  Passenger* Occupant;

class SeatSpace extends OccupiableSpace
  int RowNumber;
  int SeatNumber;

class AisleSpace extends OccupiableSpace
  int RowNumber;
  List<SeatSpace> PortSeats;
  List<SeatSpace> StarboardSeats;
```

Fig 7. OccupiableSpace classes pseudocode

Note that the length of the aisle is dynamic, and it determines the number of rows of seats in the airplane. We can use a sequential list of AisleSpaces with dynamic size to represent this

structure. Finally, we can define a parent simulation class to hold all data associated with this version of the simulator:

```
class SimulatedAirplane
  int NumPassengers;
  List<Passenger> AllPassengers;
  Queue<Passenger> LoadingQueue;
  List<AisleSpace> AisleWithSeats;
```

Fig 8. Basic simulation parent class pseudocode

4 PASSENGER QUEUEING ALGORITHMS

The next step to this organization is to choose the order in which passengers are pushed into the loading queue, which determines the order in which passengers will board the airplane. Passengers are typically queued in a certain order based on the location of their assigned seat. The main concern with choosing a queueing method is selecting one that minimizes the amount of time that passengers spend waiting before taking their seat. The primary cause of this waiting is the fact that passengers who are stowing will block the aisle, preventing other passengers from moving. For some methods, it is not uncommon that a single stowing passenger who is stopped in the aisle may cause all other non-seated passengers, including those in the loading queue, to wait for their stow time to end.

Due to this, the most obvious goal would be to find a queueing algorithm that minimizes the amount of time passengers spend waiting behind stowing passengers. This simulation implements six boarding methods which are described in Jason H. Steffen's paper, "Optimal boarding method for airline passengers." [1]. They are as follows:

1 Back-to-Front

Passengers are loaded into the boarding queue based on their seat's proximity to the back row of the airplane cabin. For example, passengers whose seats lie in the back row are loaded first, followed by those in the second-to-last row, followed by those in the third-to-last row, and so on.

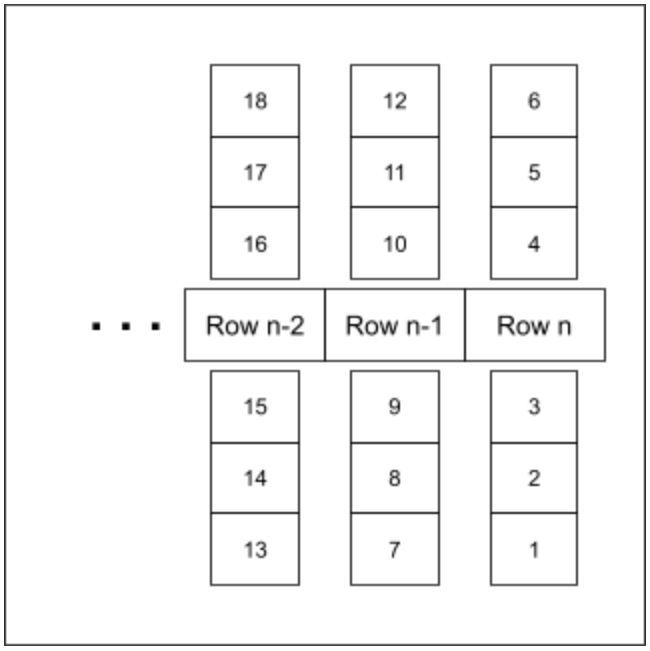

Fig 9. Back-to-Front Queueing. The number on each seat represents that passenger's position in the loading queue.

2 Front-to-Back

Similar to Back-to-Front, passengers are placed in the loading queue in order of their seat's distance to the front of the airplane cabin. For example, passengers whose seats are in the first row are loaded first, followed by those in the second row, then third, and so on.

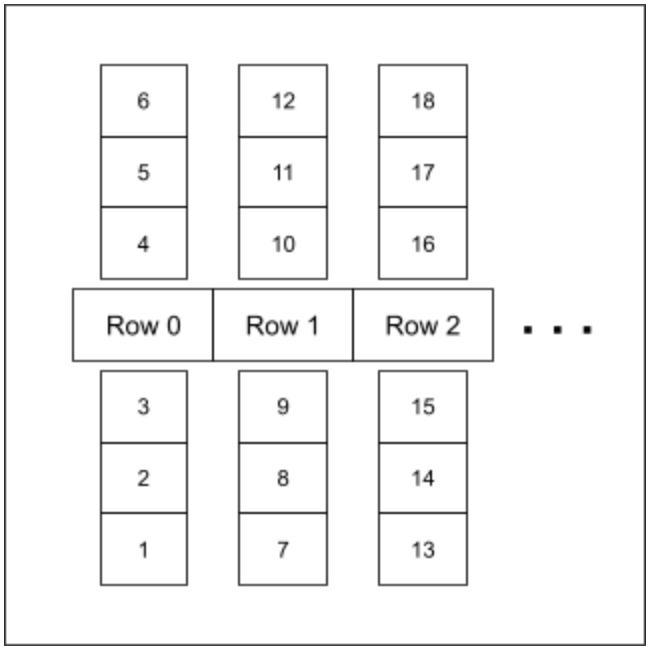

Fig 10. Front-to-Back Queueing

3 Random

Passengers enter the loading queue in a random order, without regard to their assigned seat's position.

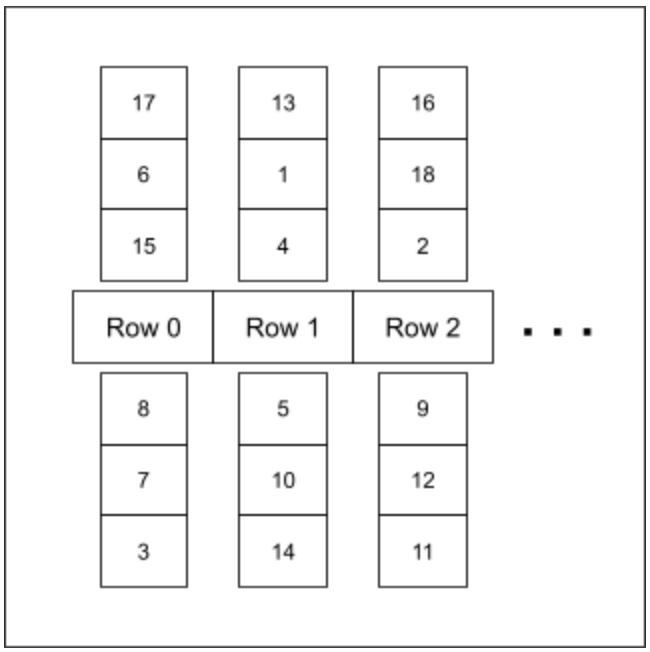

Fig 11. Random Queueing (One potential queueing order)

4 Window-Middle-Aisle

Passengers whose assigned seats are a "window seat" (the seats farthest from the aisle on either side of the aisle) are loaded into the queue first, followed by those with "middle" seats, then those with "aisle" seats. Within these groups of "window," "middle," and "aisle," passengers are placed into the loading queue randomly. This method is meant to minimize the number of times seated passengers must get up and move into the aisle to allow another passenger to reach their seat (an obstacle that is not covered in this simulation, deemed a "seat shuffle" by Youtube user CGP Grey in his video [3] on these methods). Note that this algorithm can be applied to any number of seats in the row; The passengers with seats farthest from the aisle are loaded first, then those whose seats are one seat away from those "window" seats are loaded, then those with seats that are two seats away, and so on.

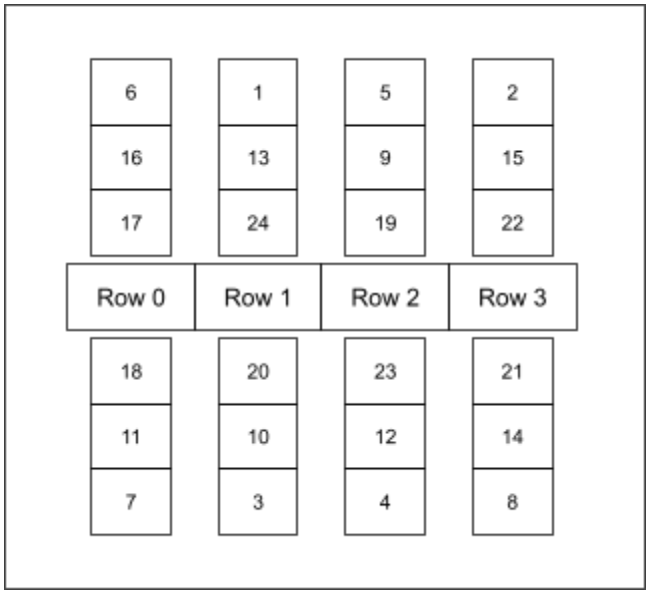

Fig 12. Window-Middle-Aisle Queueing (With 4 rows)

5 Steffen Perfect

The theoretical “optimal” boarding method proposed by Steffen in his paper [1]. Passengers are loaded in a very specific order; CGP Grey describes the method as “back to front, in alternating rows, in alternating sides, window to aisle.” [3] This method not only eliminates seat shuffling, but it also nearly eliminates passengers being stuck behind stowing passengers, given small stow times. The biggest downside to this method, however, is that implementing its precise queueing order in a real-world scenario is very often not feasible.

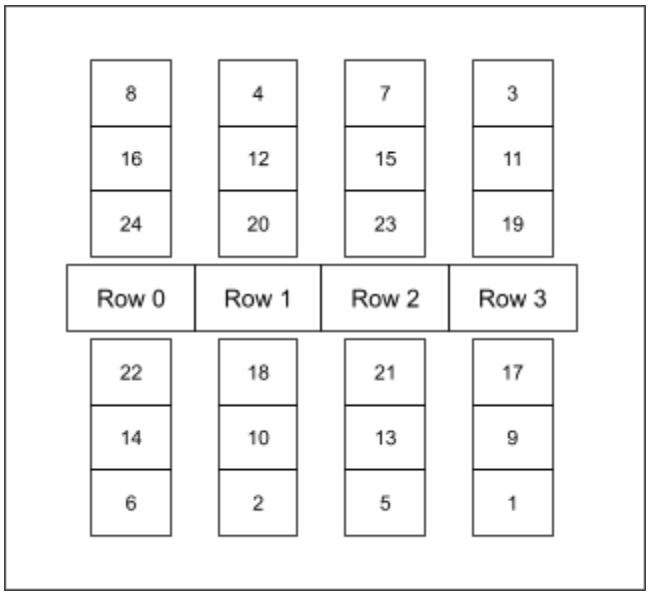

Fig 13. Steffen Perfect Queueing (With 4 rows)

6 Steffen Modified

An alternative method that is somewhat similar to the Steffen Perfect method, and also much easier to enforce in a real-world scenario. Passengers are separated into four boarding groups, with the first group being those with seats on one side of the cabin in every even-numbered row, followed by those with seats in even-numbered rows on the opposite side of the cabin. After this, passengers with seats in odd-numbered rows on the first side are loaded, then finally those with seats in odd-numbered rows on the opposite side. This method is theorized to be easier to carry out in a real scenario since passengers sitting next to each other can typically enter the queue in the correct order beforehand, and because passengers who sit next to each other in the same row are often in the same party.

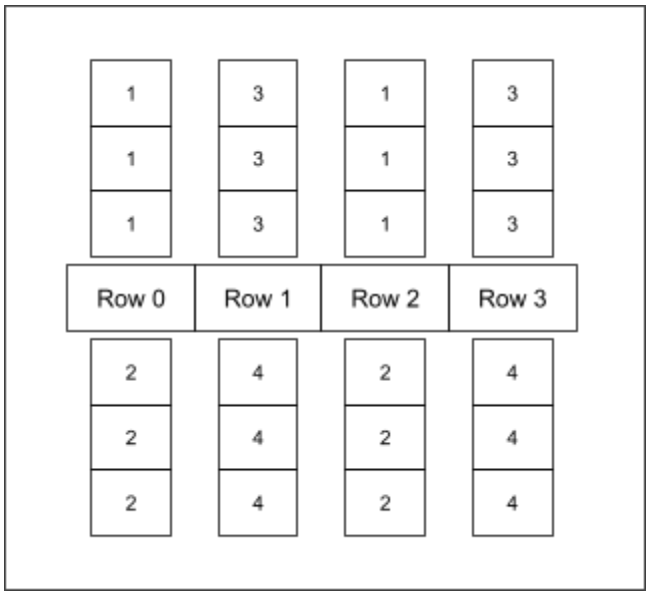

Fig 14. Steffen Modified Queueing (With 4 rows. Number on seat represents the boarding group order)

For this simulator, each of these queueing methods was fully implemented using C++. Given an initial number of passengers, each method is able to create a given number of passengers in order based on the chosen algorithm. Each algorithm is dynamically written such that the passenger seat assignment is independent of the number of seats on either side of the aisle; passengers will be placed in the loading queue based only on whichever queueing algorithm is used.

5 EXPERIMENT AND ANALYSIS

One extra feature of the simulator is the ability to provide command line arguments to each simulation that is run. This allows for much easier and more dynamic testing across all queueing algorithms. For this experiment, each queueing algorithm was run with 594 iterations, with each

iteration scaling from 6 to 600 passengers created. The number of rows was set to floor(n / 6) in order to accommodate these passengers. Additionally, the potential stow time of each passenger was a random integer between 1 and 20 in each iteration. From all iterations, I chose to record the number of passengers and the average passenger wait time to measure how efficient the algorithm was on that iteration. Here are the results from those 594 iterations:

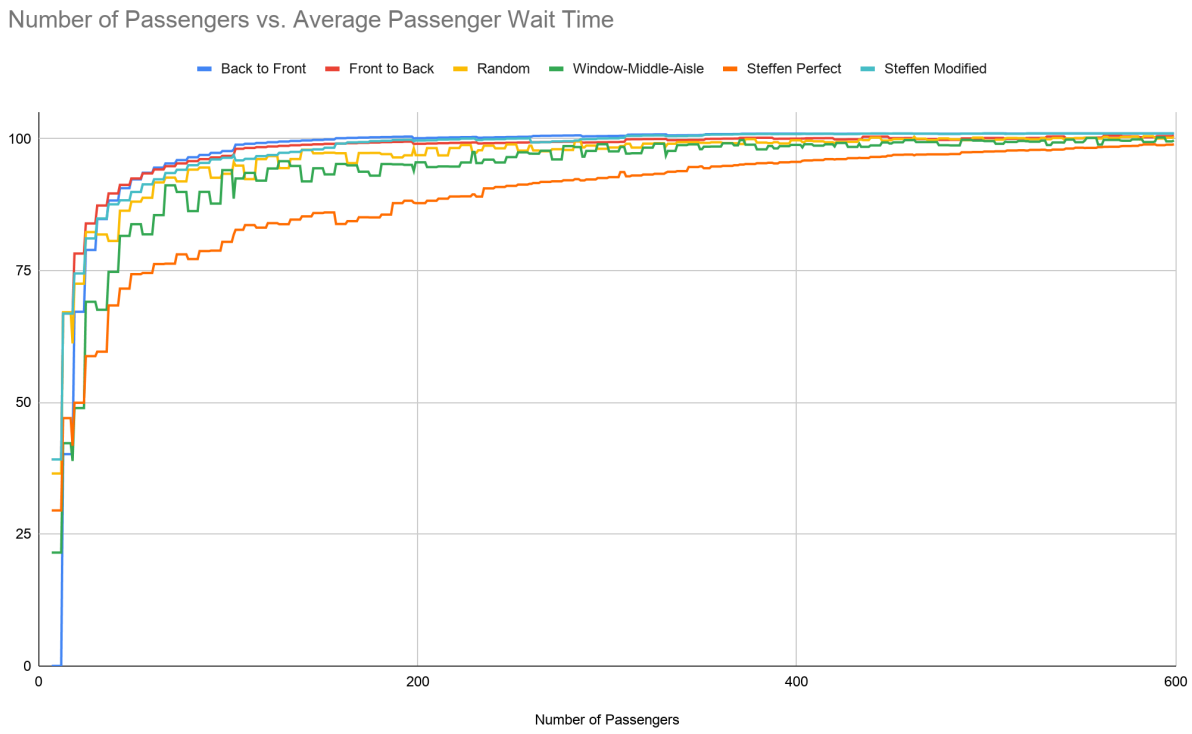

Fig 15. Results from 594 iterations

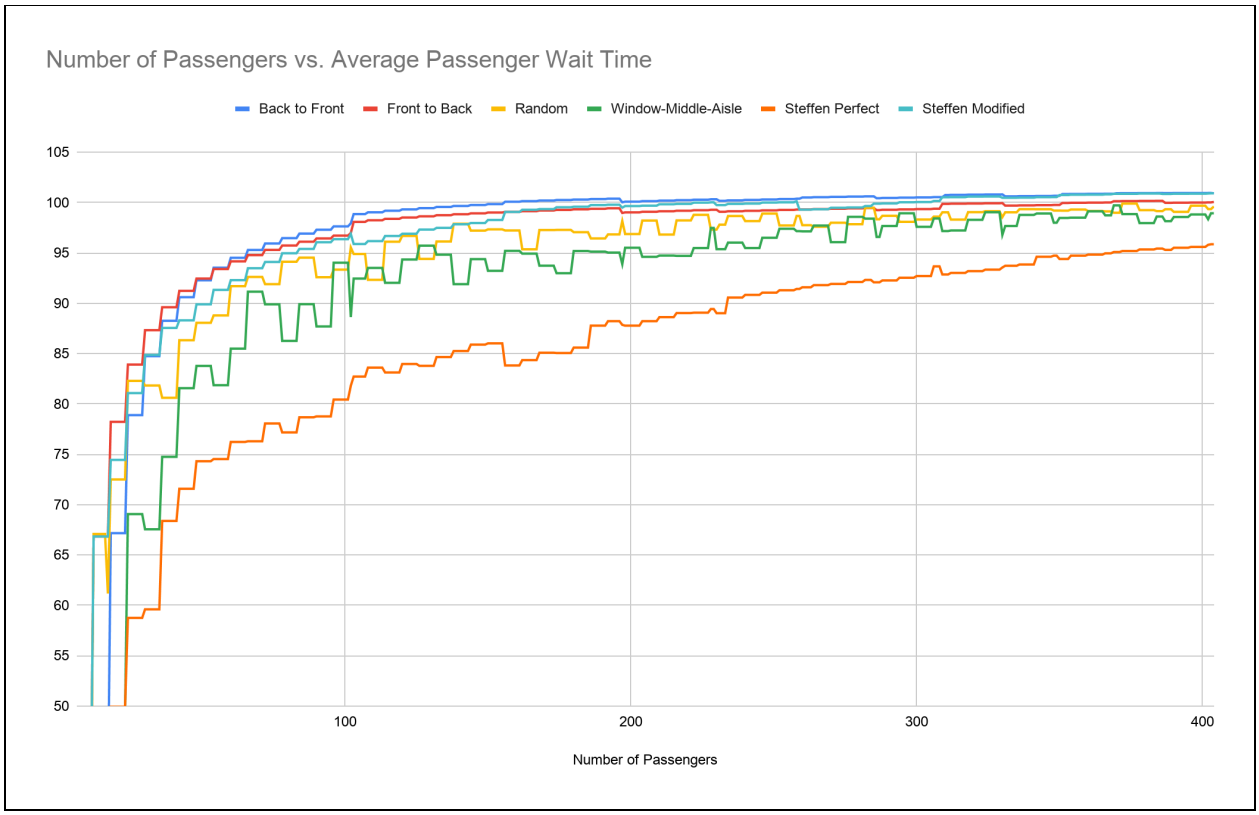

Fig 16. Results from 594 iterations (zoomed)

It can be seen from these results that the Steffen Perfect algorithm was consistently the most efficient algorithm across all runs when compared to the other methods. This was expected, as Steffen showed that even in real-world testing [2], his theoretical algorithm was proven to save the most time for all passengers overall.

Second to Steffen Perfect was actually the Window-Middle-Aisle algorithm, which I did not fully expect. This version of the simulator does not yet implement “seat shuffling,” so I did not think that the Window-Middle-Aisle method’s strengths would be useful. I can speculate that the algorithm had overall good performance since it greatly reduced the number of passengers that would potentially have a seat located past a stowing passenger within the aisle, due to passengers being loaded in large boarding groups.

Another surprising result was the performance of the Steffen Modified algorithm. It actually had consistently higher wait times than both the Random and Window-Middle-Aisle methods; I expected it to at least perform better than Random queueing.

Lastly, the most inefficient algorithms across all iterations were the Back-to-Front and Front-to-Back methods. This was not surprising, as Steffen’s simulations in his initial paper [2] provided the same result: these two methods will be much less efficient than Random sorting on virtually every iteration.

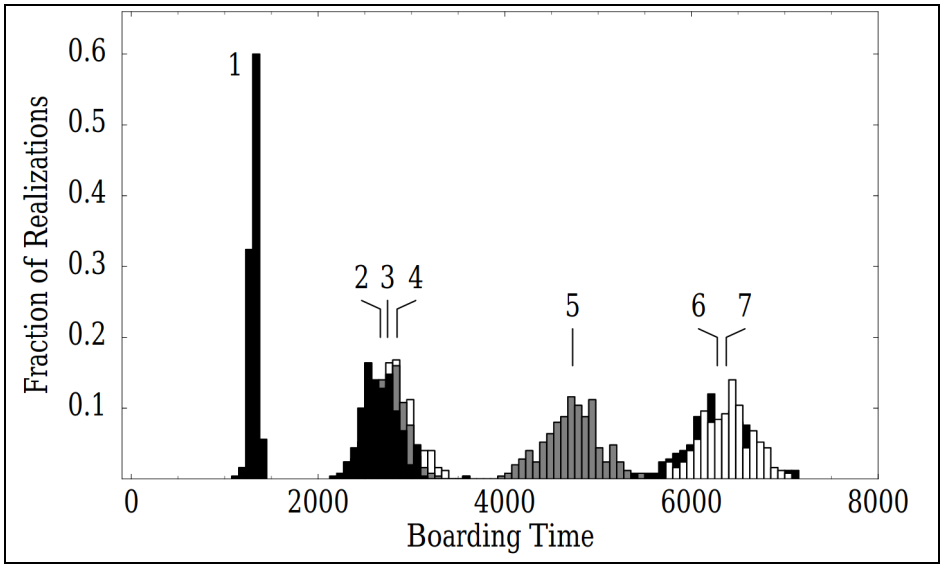

Fig 17. Results from simulations in Steffen's original paper.

Besides Steffen Modified, the results are quite similar to those achieved in this experiment. Here are the methods displayed in Fig 17:

1) Steffen Perfect
2) Steffen Modified
3) Window-Middle-Aisle
4) Random
5) Back to Front (with boarding groups)
6) Back to Front
7) Front to Back

6 CONCLUSION

The results from my simulations were reasonably consistent with Steffen's original findings. The few anomalies in my results can likely be attributed to my simulator not being fully complete. I suspect that fully implementing the "seat swapping" case would have a measurable impact on the results if this experiment were to be performed again. Overall, I would consider the simulator, the implementations of the six queueing algorithms, and the results that they produced all to be quite successful.

I really enjoyed getting the chance to create this simulator from scratch. I'm especially glad that I was able to fulfill all of the primary goals that I set out for myself in my initial project proposal. Designing everything, especially the structure of the airplane and its internal data structures, implementing the FSM behavior for every passenger, and making sure there were no conflicts in the decision making were all very tough challenges to me at first. However, once I got my footing and created something that was dynamic and easily accessible, implementing the six queueing algorithms became a somewhat straightforward task.

7 REFERENCES

[1] Steffen, Jason H. “Optimal Boarding Method for Airline Passengers.” Journal of Air Transport Management 14.3 (2008): 146–150. Crossref. Web.

[2] Steffen, Jason H, and Jon Hotchkiss. “Experimental Test of Airplane Boarding Methods.” Journal of Air Transport Management, Jan. 2012, pp. 64–67. Web.

[3] CGP Grey. “The Better Boarding Method Airlines Won't Use.” *YouTube*, 4 Feb. 2019, https://youtu.be/oAHbLRjF0vo