

**Schnelles Boarding**

**Simulation mit zellulären Automaten**

A picture containing symbol, pixel

Description automatically generated

**Master-Seminar SoSe 2023**

|  |  |
| --- | --- |
| vorgelegt von: | Jolan Eggers  1219436  Arrode 9  33790 Halle Westfalen  jolan.eggers@gmail.com |
| Studiengang: | Optimierung und Simulation  HSBI  1. Semester |
| Erstgutachter: | Prof. Dr. rer. nat., Dipl.-Math.  Svetozara Petrova |
| Zweitgutachter: |  |
| Ort und Datum | Bielefeld, 22.06.2023 |

# Inhaltsverzeichnis

[Inhaltsverzeichnis](#_Toc138344818)

[Abbildungsverzeichnis](#_Toc138344819)

[1 Einleitung und Motivation 1](#_Toc138344820)

[2 Stand der Technik 2](#_Toc138344821)

[2.1 Zellulärer Automaten 2](#_Toc138344822)

[2.2 Conway's Game of Life 4](#_Toc138344823)

[2.2.1 Aufbau und Regeln 4](#_Toc138344824)

[2.2.2 Implementierung in Python 4](#_Toc138344825)

[2.3 Einsatzgebiete Zellulärer Automaten 6](#_Toc138344826)

[2.3.1 Biologie 6](#_Toc138344827)

[2.3.2 Physik 6](#_Toc138344828)

[2.3.3 Epidemiologie 6](#_Toc138344829)

[2.3.4 Verkehrsmodellen 6](#_Toc138344830)

[3 Zielsetzung 7](#_Toc138344831)

[4 Variantenuntersuchung 8](#_Toc138344832)

[4.1 Aufbau des Vorgängermodells (?) 8](#_Toc138344833)

[4.1.1 Spielfeld 8](#_Toc138344834)

[4.1.2 Zustände 8](#_Toc138344835)

[4.1.3 Nachbarschaft 9](#_Toc138344836)

[4.1.4 Schritte 10](#_Toc138344837)

[4.1.5 Randbedingungen 10](#_Toc138344838)

[4.1.6 Startkonfiguration 10](#_Toc138344839)

[4.1.7 Erste Simulation 11](#_Toc138344840)

[4.2 Erweitern des Modells 12](#_Toc138344841)

[4.2.1 Zustandsraum 13](#_Toc138344842)

[4.2.2 Realistische Daten 14](#_Toc138344843)

[5 Finale Lösung 17](#_Toc138344844)

[5.1 Statistische Auswertung 17](#_Toc138344845)

[5.2 Quellcode 20](#_Toc138344846)

[6 Zusammenfassung und Ausblick 21](#_Toc138344847)

[7 Literatur **Error! Bookmark not defined.**](#_Toc138344848)

[8 Eigenständigkeitserklärung 22](#_Toc138344849)

# Abbildungsverzeichnis

[Abbildung 1: Conway's Game of Life showSpace() 5](#_Toc138343043)

[Abbildung 2:Conway's Game of Life - Gleiter Schrittfolge 6](#_Toc138343044)

[Abbildung 3: Conway's Game of Life in Conway's Game of Life (Bradbury, 2012) 6](#_Toc138343045)

[Abbildung 4: Wahrscheinlichkeitsverteilung verschiedener Boardingverfahren 17](#_Toc138343046)

[Abbildung 5: Normalverteilung der verschiedenen Boardingverfahren 18](#_Toc138343047)

[Abbildung 6: Steffen Perfect Boardingverfahren 18](#_Toc138343048)

[Abbildung 7: Front to Back Boarding 19](#_Toc138343049)

# Einleitung und Motivation

Zelluläre Automaten beschreiben ein System von Spielern, im Falle dieser Arbeit Flugzeugpassagieren, die sich nach exakt gleichen Regeln verhalten, die auf ihre Nachbarschaft angewandt werden (Scholz, 2013). Ein einfaches Beispiel wäre dabei, ein eindimensionaler Gang, z.B. den Mittelgang eines Flugzeugs, der mit Passagieren gefüllt ist. Jeder Passagier sieht genau einen Meter nach vorne, dies wird als die „Nachbarschaft“ bezeichnet und soll sich solange fortbewegen, dieser den nächsten Passagier sieht. Diese einfache Regel würde (logischerweise) nicht ausreichen, um den gesamten Boarding-Prozess eines Flugzeugs zu beschreiben, da ein Passagier so weder seinen Sitz finden, sein Gepäck verstauen oder anderen Personen ausweichen würde.

Die nächste Frage, die sich stellt ist, wieso ein solches Modell (überhaupt) benötigt wird. Das Boarding eines Flugzeugs ist bei Kurzstreckenflügen mit ca. 10 bis 20 Minuten für ca. 30-50% der Standzeit verantwortlich (Eiselin, 2016). Da bei einen längeren Aufenthalt durch Gate-Gebühren, Personal, erhöhtem Treibstoffverbrauch etc. höhere Kosten entstehen, gilt es diesen zu vermeiden, bzw. zu reduzieren. Des Weiteren geht es dabei nicht darum eine theoretisch optimale Methode zu finden, sondern eine Methode, die eine gute Balance zwischen Geschwindigkeit und Zufriedenheit, bzw. Stress bei den Passagieren findet. Diese beiden Ziele stehen bei vielen Methoden im Konflikt, da bspw. bei der effizientesten Methode zuerst abwechselnd alle Fensterplätze von hinten nach vorne belegt werden, dann alle mittleren Sitze und dann alle Gang-Sitzplätze. Nicht nur, müsste bei dieser Methode jeder Passagier in der richtigen Reihenfolge vor dem Gate warten, sein Gepäck gleichzeig verstauen, sowie seinen Ziel-Sitzplatz bereits vor Augen haben, es werden auch Passagiere, die hinterher zusammen sitzen beim Boarding getrennt. Dies kann vor allen Dingen bei Familien mit Kindern zu Problemen führen. Vielmals verhalten sich Menschen nicht rational und wollen als ersten im Flugzeug sitzen, obwohl die Gesamtgeschwindigkeit dadurch reduziert wird. Hinzu kommen die vielen individuellen Faktoren, wie z.B. Geschwindigkeit, Kooperationsbereitschaft, Gepäckgröße usw., die eine einfache Antwort auf diese Problem verhindern.

Ziel dieser Arbeit ist es also zunächst ein Modell des Flugzeug-Boarding Prozesses aufzubauen und daraufhin verschiedene Strategien basierend auf Simone Göttlichs „Schnelles Boarding leichtgemacht: Eine Simulationsstudie mit zellulären Automaten“ zu analysieren.

# Stand der Technik

## Zellulärer Automaten

„Zelluläre Automaten wurden erstmalig im Jahr 1940 von Stanislaw Marcin Ulam vorgestellt.“ (Kretzer, 2010) (Später wurde, basierend auf diesem Konzept, von seinem Kollegen John von Neumann, die Von-Neumann-Architektur entwickelt, auf der alle modernen Computer basieren.) Da aus einem Satz einfacher Regeln ein komplexes Gesamtverhalten emergieren kann, bieten sich zelluläre Automaten des Weiteren ideal an, um künstliches Leben, bzw. Evolution und Selbstreproduktion zu simulieren (Zitat).

Im Folgenden werden nach (Scholz, 2013) die 6 Größen beschrieben anhand dessen sich ein System von Zellulären Automaten beschrieben werden kann.

**Spielfeld:** Das Spielfeld, beschreibt die Umgebung in der die Simulation abläuft. Dabei kann das Spielfeld beliebig viele Dimensionen, mit beliebiger Größe besitzen. Meistens handelt es sich jedoch um ein 1-3 dimensionales „Spielbrett“ mit endlicher Größe und diskreten Schritten, sodass das Spielfeld als eine n-Dimensionale Matrix beschrieben werden kann. Jedes Feld, also jeder Eintrag dieser Matrix kann dabei genau einen Zustand einnehmen. Dieses „Feld“ wird auch als Zelle bezeichnet. Ein Beispiel für ein 4x4 Spielfeld zum Zeitpunkt t würde dabei folgendermaßen aussehen:

**Zustände:** Die Zustandsmenge definiert alle möglichen Zustände, die eine Zelle theoretisch einnehmen könnte, beispielsweise:

Ein Spielfeld besitzt somit theoretische Gesamtzustände, wobei i und j die Größe des Spielfelds definieren.

**Nachbarschaft:** Die Nachbarschaft einer Zelle beschreibt die Umgebung einer Zelle aufgrund dessen sich ihr Verhalten errechnet. Um den Zustand einer Zelle zum Zeitpunkt zu berechnen, werden die Informationen der Nachbarschaft benötigt:

a\*b ist dabei die Größe der Nachbarschaft, wobei im generellen von einer symmetrischen Verteilung um eine Zelle ausgegangen wird. Wenn also der Zustand aus der oben gegebenen Matrix berechnen werden soll, müssten, bei einer Nachbarschaftsgröße von 3\*3, folgende Zustände mit einbezogen werden:

**Schritte:** Die Schrittfunktion, hier S genannt berechnet den nächsten Zustand einer Zelle basierend auf ihrer Nachbarschaft. Die interne Berechnung dieser Funktion kann dabei beliebig komplex sein, muss allerdings deterministisch bleiben, sodass bei einer Nachbarschaft von a\*b und q Zuständen, höchstens verschiedene Ausgänge Möglich sind. Ein einfaches Beispiel einer solchen Funktion wäre die Bedingung, dass eine Zelle 1 wird, genau dann, wenn die Summe ihrer Nachbarschaft 7 ist:

Im Beispiel oben würde sein, da lediglich 5 Zellen aus der Nachbarschaft = 1 sind. Der Schritt wird gleichzeitig, nicht der Reihe nach (!), auf jede Zelle angewandt.

**Randbedingungen:** Bei jedem endlichen Spielfeld müssen Randbedingungen aufgestellt werden, da die oben beschriebene Schritt Berechnung am Rand, bspw. für nicht ausführbar wäre, da der Zustand von benötigt werden würde, was nicht definiert ist. Eine Mögliche Lösung wäre hier alle Randzellen statisch zu definieren, sodass die Schrittberechnung nur auf den inneren Zellen Erfolgen muss.

**Startkonfiguration:** Die Startkonfiguration beschreibt den Gesamtzustand zum Startzeitpunkt. Da es sich in den meisten Fällen um ein deterministisches System handelt, wird aufgrund des Startzustand jeder weitere Zustand eindeutig beschrieben.

## Conway's Game of Life

### Aufbau und Regeln

Um ein Verständnis für die Funktionsweise und den Aufbau zellulärer Automaten zu bekommen wird zunächst Conways Game of Life in Python implementiert. Bei Conway’s Game of Life handelt es sich um ein zweidimensionales System, mit einem Spielfeld unendlicher Größe und zeitlich und räumlich diskreten Schritten, bei dem jede Zelle entweder lebendig oder tot sein kann. Die Nachbarschaft hat eine Größe von 3\*3, allerdings wird der eigene Zellzustand nicht mitbetrachtet, sodass 8 Zustände in die Berechnung eines einzelnen Zellschritts einfließen. Für eine Zelle gelten die folgenden beiden Regeln, um den nächsten Zustand zu berechnen:

Wenn eine Zellegenau 3 Nachbarn hat, wird diese im nächsten Zustand leben

Wenn eine Zelle weniger als 2 lebende Nachbarn oder mehr als 3 Nachbarn hat stirbt diese („Einsamkeit“ und „Überbevölkerung“).

Wenn keiner dieser Fälle Eintritt, bleibt die Zelle unverändert.

### Implementierung in Python

Für die Ausgabe wird die Bibliothek „pygame“ verwendet, sodass die Zellen grafisch dargestellt werden können.

Die Variable “cellState = np.zeros((envHeight, envWidth))” wird als Matrix von Nullen initialisiert, sodass der Zustand jeder Zelle hier gespeichert werden kann:

Es wird eine Funktion „showSpace()“ definiert, die den Zustand der Matrix visuell ausgibt indem Kästchen mit einer lebenden Zelle schwarz und mit einer toten Zelle weiß gefärbt werden. Mithilfe der Funktion „initEnv()“ wird die Matrix mit einem Initialzustand gestartet. Dazu werden im ersten Schritt die Felder zunächst einzeln beschrieben:

def initEnv():  
 cellState[3][1] = 1  
 cellState[1][2] = 1  
 cellState[3][2] = 1  
 cellState[2][3] = 1  
 cellState[3][3] = 1

Werden nun die Funktionen „initEnv()“ und „showSpace()“ nacheinander aufgerufen, ist folgendes Fenster zu erkennen:

A picture containing square, rectangle, building, line

Description automatically generated

Abbildung 1: Conway's Game of Life showSpace()

Diese „Gebilde“ ist eine einfache Form eines sogenannten Gleiters. Ein Gleiter ist eine Struktur aus Zellen, die sich mithilfe der gesetzten Regeln eigenständig fortbewegen kann. Um eine solche Bewegung nun zu realisieren, wird die Funktion „nextStep()“ eingeführt in der die Regeln auf die Zellen in einem Zeitschritt angewendet werden. Dazu wird zunächst der cellState in eine zweite Variable kopiert, „cellStateCache“, auf der die Änderungen angewandt werden, sodass es nicht zu Änderungen kommt, die sich gegenseitig überschreiben. Danach wird über jede Position der Matrix iteriert und die Regeln angewandt. Dazu wird zunächst ermittelt wie viele der Nachbarn leben, indem die umliegenden Zellen aufaddiert werden:

neighbourAliveCount=cellState[x1][y+1]+cellState[x][y+1]+cellState[x+1][y+1]+

cellState[x-1][y]+cellState[x+1][y]+cellState[x-1][y-1]+cellState[x][y-1]+cellState[x+1][y-1]

Daraufhin können die beiden Regeln implementiert werden, die bestimmen, ob Zellen geboren werden oder sterben:

if neighbourAliveCount == 3: # new cell gets born  
 cellStateCache[x][y] = 1  
if neighbourAliveCount < 2 or neighbourAliveCount > 3: # cells dies  
 cellStateCache[x][y] = 0

Als letztes wird der cellStateCache in den CellState zurück kopiert. Wird die Funktion „nextStep()“ nun jede Sekunde ausgeführt kann man den zuvor beschriebenen Gleiter beobachten:

A picture containing line, text, diagram, crossword puzzle

Description automatically generated

Abbildung :Conway's Game of Life - Gleiter Schrittfolge

Dabei ist zu erkenne, dass die Struktur nach 4 Zyklen wieder im Ausgangszustand endet, allerdings jeweils eine Zelle nach rechts bzw. unten verschoben ist, sodass sich das Gebildet kontinuierlich weiterbewegt. Es wird dabei zunächst von einem unendlich großen Raum ausgegangen, sodass keine weiteren Randbedingungen benötigt werden.

Da Conway’s Game of Life Turing komplett ist lässt sich jede erdenkliche Berechnung, die auf „normalen“ Computern zu tätigen ist, auch in Conway’s Game of Life realisieren. Das Flugzeug des Titelblatts ist dabei auch eine mögliche Startkonfiguration. Eine Demonstration für die zu erreichende Komplexität ist, dass selbst Conway‘s Game of Life in Conway‘s Game of Life simuliert werden kann:

A picture containing symmetry, line, pattern, parallel

Description automatically generated

Abbildung 3: Conway's Game of Life in Conway's Game of Life (Bradbury, 2012)

## Einsatzgebiete Zellulärer Automaten

### Biologie

### Physik

### Epidemiologie

### Verkehrsmodellen

# Zielsetzung

Ziel dieser Arbeit ist es zunächst ein Modell des Flugzeug-Boarding Prozesses aufzubauen und daraufhin verschiedene Strategien basierend auf Simone Göttlichs „Schnelles Boarding leichtgemacht: Eine Simulationsstudie mit zellulären Automaten“ zu analysieren. Zunächst wird dabei versuch die beschriebene Vorgehensweise Schritt für Schritt in Python zu implementieren. Wenn die Ergebnisse dieses Papers validiert werden konnten, wird versucht darüber hinaus eine komplexere Simulation zu erzeugen, indem die bereits schon im Paper angesprochenen Schwächen versucht werden zu reduzieren. Einer der grundlegenden Schwächen ist dabei die Simplifizierung des Sitzwechsel Vorgangs, da lediglich mit einer Wartezeit simuliert wird und nicht mit einer realen Bewegung der Personen. Vielmals kann ein solcher Sitzwechsel Vorgang allerdings weitreichende Folgen haben, da es häufig dazu kommt, dass nicht genügend Platz im Gang ist, damit zwei Personen den Sitz verlassen ohne, dass sich die gesamte Reihe an dahinter stehenden Leuten zurück bewegt. So könnte bspw. nicht mitsimuliert werden, ob eine andere Geometrie des Innenraums zu einem besseren Boarding Prozess führt. Dies ist auch der geringen räumlichen Auflösung geschuldet. Bei dem Spielbrett der Quelle handelt es sich um eine Zellengröße von 0,5x0,5m, sodass ein Sitz auch nicht mit 0,6 oder 0,3 m simuliert werden kann. Darüber hinaus wird zwar mit einer Wahrscheinlichkeitsverteilung beim Gepäck verstauen gerechnet, allerdings können die Passagiere keine individuellen Eigenschaften besitzen, wie bspw. verschiedene Größen, Geschwindigkeiten oder Familienangehörige. Ziel ist es also ein Modell aufzubauen und statistisch zu validieren, dass in diesen Kategorien eine höhere Auflösung bietet.

# Variantenuntersuchung

Die oben beschriebenen Bedingungen werden nun Schritt für Schritt auf den Boarding Prozess eines Flugzeugs übertragen, sodass eine erste Simulation aufgebaut werden kann.

## Aufbau des Vorgängermodells (?)

### Spielfeld

Beim Spielfeld handelt es sich um ein zweidimensionales Feld, welches zunächst mit der Größe 12x7 initialisiert wird. Dabei ist zu beachten, dass es sich um kein homogenes Spielfeld wie im „Game of life“ handelt, da sich die zellulären Automaten am Ende anders verhalten sollen, je nachdem, ob sich diese gerade im Gang, zwischen den Sitzen oder auf einem Sitz befinden. Diese Verhalten soll im Nachhinein noch beliebig angepasst werden können. Deswegen wird neben der Matrix „cellState“ eine Variable „environment“ mit der gleichen Dimension erstellt, die für jede Zelle die Umgebung beschreibt.

### Zustände

**Globale Zustände**

Die Zustandsmenge Q beschränkt sich in diesem Fall auf die beiden Zustände

.

Da neben dem cellState allerdings auch noch die Umgebung definiert werden muss wird eine zweite Zustandsmenge für das environment aufgestellt:

.

Diese beiden Zustandsmengen lassen sich zwar zu einer kombinieren, allerdings ist für die saubere Ausarbeitung eine Trennung geeigneter.

**Interne Zustände**

Es ist davon auszugehen, dass bereits in der Arbeit von Simone Göttlich mit internen Zuständen der Zellen gearbeitet wurde, da in Abschnitt 8.2.5 „Algorithmus zellulärer Automat“ in Schritt 2 davon gesprochen wird bei der passenden Sitzreihe links oder rechts abzubiegen. Dies spricht dafür, dass jeder Zelle von Anfang an ein Sitzplatz zugewiesen wird, der intern gespeichert wird. Damit ist im Gegensatz zu Conways Game of Life der nächste Globale Zustand nicht alleine aus dem Globalen Zustand zu bestimmen, sodass zwei gleich aussehende Spielfelder in einem anderen nächsten Zustand resultieren können.

Es ist zwar durchaus möglich den internen Zustand einer Zelle in der gesamt-Zustandsmatrix zu kodieren:

allerdings wird dies schnell unübersichtlich, bzw. führt zu einer exponentiell wachsenden Zustandsraumbeschreibung, wenn neben dem Alter noch Faktoren, wie bspw. Zielsitzplatz, Größe, Gewicht, Verspätung, etc. eine Rolle spielen sollen. Deshalb bietet es sich an jede aktive Zelle als ein einzelnes Objekt zu beschreiben und die notwendigen Informationen dort zu speichern. Wenn alle Informationen einer Zelle intern gespeichert werden, wäre theoretisch auch keine Zustandsmenge mehr notwendig, da auch die Position mit in der Datenstruktur des Objekts gespeichert werden kann, allerdings würde dies die Ermittlung der Nachbarschaft einer Zelle erschweren. Deshalb wird die Zustandsmenge weiterhin verwendet, allerdings wird in ihr nicht nur kodiert, ob ein Passagier in der Zelle vorhanden ist, sondern auch welcher Passagier auf der Zelle steht, indem jeder Passagier mit einem Index versehen wird:

Alternativ könnten hier auch für jeden Passagier über alle anderen Passagiere iteriert werden (ineffizient) oder Datenstrukturen wie bspw. spatial hash maps verwendet werden, um die Bestimmung der Nachbarn einfacher zu gestalten.

### Nachbarschaft

Auf die Nachbarschaft einer Zelle werden im nächsten Abschnitt die Regeln angewandt. Die Größe der Nachbarschaft beschreibt wie groß der Einflussbereich der Umgebung auf eine Zelle ist, also „wie weit eine Zelle sehen kann“. Um die Programmierung möglichst einfach zu halten wird hier, genau wie bei „Conways Game of life“ eine Umgebung mit einer Größe von 3x3 Pixeln gewählt.

### Schritte

Um aus der Zustandsmatrix , den internen Zuständen und dem Zustand der Umgebung zum Zeitpunkt den nächsten Zeitschritt zu generieren, müssen gewisse Regeln definiert werden. Da ohnehin schon interne Zustände definiert wurden, lohnt es sich ebenfalls den nächsten Schritt intern in einem Agent zu berechnen:

### Randbedingungen

Da sich der nächste Zustand einer Zelle aus den Zellen ihrer Nachbarschaft ergibt, muss festgelegt werden was passiert, wenn die Nachbarschaft an einer oder mehreren Seiten endet. Konkret kann dies zu Problemen führen, wenn über alle Felder einer Matrix iteriert wird und die Nachbarschaft wie folgt ausgerechnet wird:

*for* x *in range*(0, envWidth):  
 *for* y *in range*(0, envHeight):  
 neighbor = cell[x - 1][y + 1] + cell [x][y + 1] + cell [x + 1][y + 1]

Hier ist der Zustand aus den Indizes x=-1 oder x=envWidth+1 gefordert, der nicht definiert ist. Um das Problem für den Fall der Flugzeugsimulation zu lösen, wird zunächst der gesamte Bereich mit einer Wand umzogen, also einem Zustand der sich nicht ändern kann, folglich auch keiner Berechnung bedarf, sodass nur noch von 1 bis envWidth-1 iteriert werden muss:

*for* x *in range*(1, envWidth-1):  
 *for* y *in range*(1, envHeight-1):  
 neighbor = cell[x - 1][y + 1] + cell [x][y + 1] + cell [x + 1][y + 1]]

### Startkonfiguration

Als letzte Bedingung fehlen die Startbedingung, um eine Simulation zum Laufen zu bringen. Da sowohl Startkonfigurationen, als auch Schritt-Regeln später geändert und untersucht werden sollen, wird hier nur auf eine Möglichkeit eingegangen. In der „initEnv()“ Funktion werden Sitzplätze wie folgt beschrieben:

*def* initEnv():  
 *for* x *in range*(envWidth):  
 *for* y *in range*(envHeight):  
 *if* y != 3:  
 *if* x % 2 == 1:  
 environment[x][y] = 1

So ist auf jeder zweiten Reihe ein Sitzplatz, außer auf der Mittellinie. Wenn nun für , die Umgebung auf 2 (Wand) gesetzt wird, erhält man folgende Umgebung:

A picture containing pattern, square, rectangle, screenshot

Description automatically generated

Abbildung : Erstes Flugzeug-Spielbrett

Die erste Zelle im Gang benötigt eine besondere Regel, damit Passagiere der Simulation hinzugefügt werden können. Die Passagiere können entweder nacheinander auf den ersten Platz „teleportiert“ werden, sobald dieser frei ist oder der Gateway zum Flugzeug kann mitsimuliert werden und die Simulation startet mit allen Passagieren bereits in der Simulation.

### Erste Simulation

Für eine erste Simulation werden alle Randbedingungen so einfach wie möglich gehalten. Dazu wird alle 3 Schritte ein neuer Passagier an der Position (1,4) initialisiert, mit dem bekannten Ziel nextSeatX, nextSeatY:

entities.append(Entity(index=1, target=(nextSeatX, nextSeatY), position=(1, 4)))

entitiesToCellState()

Da es für eine erste Simulation zu möglich wenig Problemen kommen soll, wurde die optimale Strategie der Quelle gewählt, die „reverse Pyramid-Boarding“-Strategie, bei der zunächst von hinten die am Fenster gelegenen Sitze aufgefüllt werden , daraufhin die Sitzreihe weiter innen usw.:

A picture containing screenshot, square, pattern, rectangle

Description automatically generated

Abbildung : Erste Boarding-Simulation

Da es bei dieser Strategie zu keinen Kollisionen kommt und alle Passagiere bereits in der korrekten Reihenfolge bereitstehen, wird das Tauschen von Sitzplätzen oder das Verstauen von Gepäck zunächst nicht mit modelliert.

Übrig bleibt hier also nur noch die interne Logik der Passagiere zu beschreiben:

Ein Passagier soll so lange den mittleren Gang nach rechts durch gehen, bis dieser einen Pixel vor seinem Ziel-Sitz steht, also im Gang vor dem Sitz, dann soll diese Reihe so weit wie möglich nach oben, bzw. unten durchgegangen werden, bis die Y-Koordinate des Ziel-Sitzes der eigenen Y-Koordinate entspricht, woraufhin ein letzter Schritt nach rechts unternommen wird, um den finalen Sitzplatz einzunehmen:

*def* nextStep(*self*, cellStateInternal):  
 x = *self*.position[0]  
 y = *self*.position[1]  
 *if* y == *self*.target[1]:  
 *if* x < *self*.target[0]:  
 *self*.position = (*self*.position[0] + 1, *self*.position[1])  
 *elif* x == *self*.target[0] - 1:  
 *if self*.target[1] > y:  
 *self*.position = (*self*.position[0], *self*.position[1] + 1)  
 *if self*.target[1] < y:  
 *self*.position = (*self*.position[0], *self*.position[1] - 1)  
 *elif* cellStateInternal[x + 1][y] == 0 *and* environment[x + 1][y] != 2:  
 *self*.position = (*self*.position[0] + 1, *self*.position[1])

## Erweitern des Modells

Da in der Quelle bereits beschrieben wurde, das eine der ersten zu verbessernden Abläufe das Verhalten zwischen zwei oder mehreren Passagieren beim Tauschen von Sitzplätzen ist, wird zunächst hier angegriffen: Das in der Quelle angewandte Verfahren beruht darauf eine Wartezeit für jeden Vorgang, basierend auf der Poisson-Verteilung, , zu berechnen. Dabei gehen nützliche Informationen über das tatsächliche Verhalten der einzelnen Passagiere verloren, wie z.B., ob es zu einem übermäßigen Gedrängel kommt was durch eine andere Gang-Geometrie vermindert werden könnte. Um ein solches Verhalten zu simulieren bedarf es einer höheren räumlichen und möglicherweise auch zeitlichen Auflösung. Im (Paper) wird von einem Sitz und Gang-Maß von 0,5x0,5m ausgegangen, also einer Auflösung von . Da noch nicht klar ist welche Auflösung letztendlich am besten für eine schnelle Simulation geeignet ist, werden Zeit-, sowieso Raumauflösung variabel gestaltet, wobei mit Werten von und einem Intervallschritt von einer Sekunde gestartet wird.

### Zustandsraum

Die Änderung der Umgebung gestaltet sich trivial, da eine Matrix geringerer Auflösung lediglich hochskaliert werden muss (solange man nicht neues Detail hinzufügen möchte, wie bspw. eine andere Gang-Form oder abgerundete Sitze):

[[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2. 0. 1. 0. 1. 0. 1. 0. 1. 2.]

[2. 0. 1. 0. 1. 0. 1. 0. 1. 2.]

[2. 0. 0. 0. 0. 0. 0. 0. 0. 2.] 🡪

[2. 0. 1. 0. 1. 0. 1. 0. 1. 2.]

[2. 0. 1. 0. 1. 0. 1. 0. 1. 2.]

[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]]

[[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 2. 2.]

[2. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 2. 2.]

[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]

[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]]

environmentUpscale = np.zeros((envWidth\*2, envHeight\*2))  
*for* x *in range*(envWidth\*2):  
 *for* y *in range*(envHeight\*2):  
 environmentUpscale[x][y]=environment[*int*(x/2)][*int*(y/2)]

|  |  |
| --- | --- |
| A picture containing square, pattern, screenshot, rectangle  Description automatically generated  Abbildung 6: Spielbrett Auflösung "1" | *A picture containing pattern, square, rectangle, screenshot  Description automatically generated*  Abbildung 7: Spielbrett Auflösung "2" |

Die Änderung des Zustandsraums der Passagiere gestaltet sich bereits etwas schwieriger, da ein Passagier nun nicht mehr eine Position im Raum einnimmt, sondern mehrere Felder. Deshalb wird in der Zustandsmatrix der Passagiere nur der Mittelpunkt der jeweiligen Passagiere gespeichert und die Größe der einzelnen Passagiere im internen Zustand des Passagiers. Dies muss in der Berechnung des nächsten Schritts beachtet werden, da nun die Umgebung nicht mehr aus 8 Zellen sondern aus Zellen.

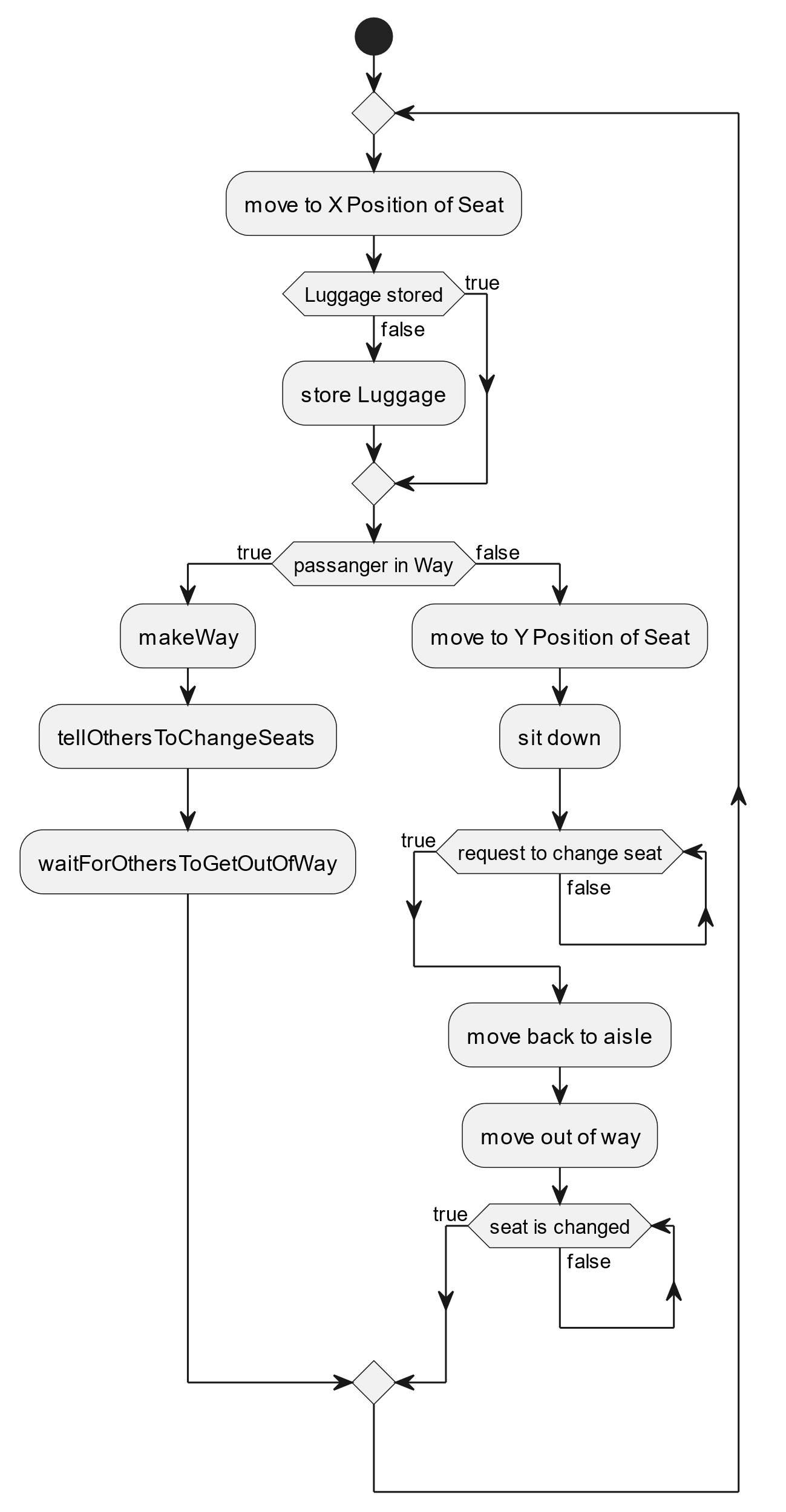
### Passagier-Agent

|  |  |
| --- | --- |
| **Variable** | **Erklärung** |
| index | Name der Zelle (Platz in der Liste aller Passagiere) |
| target | x,y – Koordinaten des Ziels (Sitzplatz) in m |
| position | x,y – Koordinaten der derzeitigen Position in m |
| speed | Maximale Geschwindigkeit in m/s |
| diameter | Durchmesser der Person in m |
| seated | True/False: Gibt an, ob man auf dem Sitzplatz ist |
| luggageStored | True/False: Gibt an, ob das Gepäck bereits verstaut ist |
| luggageStoreTime | Zeit, die zum Gepäck verstauen benötigt wird in s |
| changeSeatFor | Index der Person für die man den Sitz wechseln soll |
| awaitingSeatChangeFor | Liste von Personen auf die gewartet wird, um den Sitz zu wechseln |
| state | Interner state (int), dient zur Koordination der Schritte |

Jedem Passagier liegt nun ein Agenten-basiertes Modell zugrunde, das die Verhaltensweise des Passagiers beschreibt. Dabei hat ein Passagier folgende Interne Zustände:

Die Zustände werden dabei als Instanzattribute der Klasse „Entity“ gespeichert. Alle Zustände besitzen Standartwerte, können aber auch beim initialisieren überschrieben werden. Zusätzlich enthält diese Klasse alle internen Methoden, die notwendig sind, um einen nächsten Schritt auszuführen. Die grobe Schrittabfolge wird dabei im folgenden Aktivitätsdiagramm „Abbildung 8: Entity Aktivitätsdiagramm“ dargelegt.

Nachdem ein Passagier initialisiert wurde, wird zunächst die Funktion „moveToX“ aufgerufen, die den Passagier solange in X-Richtung nach vorne oder nach hinten bewegt, bis der Passagier an seiner Sitzreihe angekommen ist. Dabei besteht diese Funktion aus Unterfunktionen, sodass bspw. darauf geachtet werden kann, ob ein Passagier den Weg blockiert oder ob der Passagier vor sich dazu auffordert zurückzutreten, um Platz zu machen. Auf diese Funktionen wird separat nochmal eingegangen (?).

Als nächstes wird überprüft, ob der Passagier bereits sein Gepäck verstaut hat, falls nein, bleibt die Person so lange auf der Stelle stehen, bis das Gepäck verpackt ist. Daraufhin wird überprüft, ob Passagiere den Weg auf den eignen Sitzplatz versperren, also ob bereits Passagiere in der eigenen Reihe sitzen, dessen Sitzplatz näher am Gang sind, als der eigene. Wenn dies der Fall ist, tritt die Person zunächst selber zurück, woraufhin den Passagieren, die im Weg sind, gebeten werden den Sitz zu wechseln („tellOthersToChangeSeats“). Bei dem oder den Passagieren, die zuvor im Weg gesessen haben wird nun die Abfrage „request to change seat“ wahr, sodass der oder die Passagiere zurück in den Gang und daraufhin aus dem Weg gehen. Daraufhin kann Passagier 1 den Sitz einnehmen, indem die Schleife wieder von vorne beginnt (move To X – move to Y Position – sit down). Die Passagiere, die Platz gemacht haben kehren ebenfalls zum eignen Sitz zurück, indem auch hier die Schleife von vorne beginnt.

Um klar abzugrenzen in welchem Zustand jedes Objekt sich befindet, wird jeder Aktivität in „Abbildung 8: Entity Aktivitätsdiagramm“ ein state zugeordnet. So ist bspw. „move to X Position of Seat“ state 0, “store Luggage” state 1 usw. Jede Funktion kann den state ändern, sodas theoretisch jede Funktion zu jeder anderen Funktion springen könnte. Als Beispiel ein Ausschnitt des Codes:

Abbildung : Entity Aktivitätsdiagramm

*if self*.state == 4:  
 *self*.makeWay(cellStateInternal, environmentInternal, entities)  
*if self*.state == 5:  
 *self*.tellOthersToChangeSeats(cellStateInternal, environmentInternal, entities)  
*if self*.state == 6:  
 *self*.waitForOthersToGetOutOfWay()

Bevor eine Person den nächsten Schritt macht, muss sichergestellt werden, dass es zu keiner Kollision kommt. Dafür wird die Funktion „collisionAtPoint(x,y)“ eingeführt, die zurück gibt, ob es am abgefragten Punkt zu einer Kollision kommt. Mögliche Antworten dieser Funktion wären bspw. ,dass an Punkt x,y im Umkreis des eigenen Durchmessers bereits eine Wand oder ein Sitz ist. Wenn eine Person im Weg steht, wird der Name (index) der Person verwendet. Bei Personen unterscheidet sich das Vorgehen allerdings minimal, da Passagiere nicht nur über die eine Zelle in der Matrix definiert werden, sondern auch über ihren Durchmesser. So wird zunähst eine Liste mit allen Personen in der Nachbarschaft erstellt, woraufhin durch diese Liste iteriert wird und überprüft wird, ob der Abstand kleiner als die halbe Summer der Durchmesser ist. All diese möglichen Kollisionen werden in einer Liste zurückgegeben. Wenn die Liste leer ist, kann die Person den nächsten Schritt vollziehen. Wenn ein Passagier herausfinden will, ob ein anderer Passagier auf dem Pfad zum eignen Sitz im weg steht/sitzt, wird die Funktion für mehrere Punkte auf dem Vektor zwischen einem selber und dem Ziel ausgeführt und die Ergebnisse kombiniert und gefiltert, sodass eine Liste aller Personen zurückgegeben wird, die zwischen einem selbst und dem Ziel stehen. Diese Information kann im nächsten Schritt dazu verwendet werden die Passagiere aufzufordern den Platz zu wechseln. Als Beispiel eine Funktion, die in 11 Schritten alle Kollisionen zwischen dem Passagier und ihrem Ziel berechnet (das Summenzeichen dient hier als das Anhängen des Ergebnisses an eine Liste).

A picture containing screenshot, rectangle, colorfulness, square

Description automatically generated

Abbildung : Kollision zweier Passagiere

Ein Sitzwechsel-Vorgang läuft dabei folgenderweise ab:

**A picture containing screenshot, colorfulness, font, graphics

Description automatically generated**

Abbildung : Sitzwechsel-Vorgang

### Starten der Simulation

Um die Simulation starten zu können werden die benötigten Parameter über eine csv Datei in das Programm geladen. Im Folgenden ein Ausschnitt dieser Daten:

index,targetX,targetY,positionX,positionY,diameter,speed  
1, 1.9, 1.25, 0.75, 2.24, 0.3, 1  
2, 1.9, 1.75, 0.75, 2.24, 0.3, 1  
3, 1.9, 0.75, 0.75, 2.24, 0.3, 1

Es werden für jeden Passagier Index, Ziel-Sitzplatz, Startposition und Geschwindigkeit übergeben. Es könnten auch weitere der oben beschriebenen inneren Zustände übergeben werden. Im Ablauf der Simulation werden diese Passagiere der Reihe nach der Simulation hinzugefügt, sobald an der gegebenen Position genug Platz vorhanden ist. Der interessante Teil ist dabei die Generierung der Zielposition, die am Ende darüber entscheiden um welche Boarding Strategie es sich handelt. Dafür wird in einem separaten Skript die csv Datei gemäß dem Passenden Verfahren erstellt. Wenn es sic um ein Verfahren handelt bei dem nicht jeder Sitz fest vergeben ist, wie z.B. das random-Boarding, wird bei jedem Aufrufen des Skripts eine neue Sitzordnung generiert.

### Realistische Daten

Damit die Simulation möglich realistisch gestaltet werden kann, muss diese mit möglichst realistischen Parametern initialisiert werden. Die maßgeblichen Parameter der Umgebung sind dabei der Sitzabstand, die Sitzbreite, sowie Sitztiefe und der Gangbreite.

Am Beispiel eines Lufthansa Airbus A321 lauten diese wie folgt:

|  |  |
| --- | --- |
| Sitzabstand | 0,79-0,89m |
| Sitzbreite | 0,43-0,44m |
| Sitztiefe |  |
| Gangbreite |  |

[201812\_A321-200.pdf](file:///C:\Users\jolan\Downloads\201812_A321-200.pdf)

[Sitzabstand und Sitzbreite in der Economy Class (fairliners.com)](https://www.fairliners.com/sitzabstand.html)

Da über die Auflösung eine genauere Unterteilung erreicht werden kann, ist nicht, wie zuvor eine Zelle für einen Sitz reserviert, sodass die Maße in Metern gerechnet werden können und die entsprechende Anzahl an Zellen über die Auflösung ausgerechnet werden kann.

# Finale Lösung

## Statistische Auswertung

Um das Modell zu validieren wurde ein Python Programm geschrieben, das zuerst eine neue, zufällige, aber auf dem Schema basierende, Sitzanordnung erzeug und daraufhin die Simulation startet. Danach wird das Ergebnis dieser Simulation in einer csv Datei gespeichert und Schritt 1 wiederholt. Alle Metaparamter, wie bspw. die Größe des Flugzeugs oder die Sitzanordnung, aber auch die Initialisierung der einzelnen Passagiere, wurden über alle Verfahren gleich gelassen. Konkret wurde dabei mit folgenden Parametern gearbeitet:

|  |  |
| --- | --- |
| diameter | 0,2 m |
| speed | 1 m/s |
| storeLuggageTime | 2 s |
| Sitzbreite | 0,5 m |
| Sitztiefe | 0,5 m |
| Sitzabstand | 0,5 m |
| Gangbreite | 0,5m |
| Sitzkonfiguration | 3 Sitze rechts, 3 Sitze links |
| Anzahl Sitze | 6 Sitze/Reihe \* 11 Reihen = 66 Sitze |

A picture containing text, screenshot, diagram, plot

Description automatically generated

Abbildung : Wahrscheinlichkeitsverteilung verschiedener Boardingverfahren

A picture containing text, plot, line, diagram

Description automatically generated

Abbildung : Normalverteilung der verschiedenen Boardingverfahren

An der Wahrscheinlichkeitsverteilung der verschiedenen Verfahren ist bereits, ohne Überraschung, ein klarer Gewinner zu erkennen: Steffen Perfect. Bei diesem Verfahren gibt es mit n Passagieren auch n Boarding Gruppen. Beim Boarding wird jeweils eine Reihe Platz gelassen, sodass jeder sein eigenes Gepäck verstauen kann, ohne dass andere auf einen warten müssen und es wird von hinten nach vorne, bzw. von außen nach innen geboarded, sodass es zu keinen Sitzwechsel Vorgängen kommt. Die finale Anordnung ist in „Abbildung 10: Steffen Perfect Boardingverfahren“ zu erkennen. Da bei dieser Versuchsreihe eine Gepäck-Verstauzeit von 2 Sekunden verwendet wurde, ist dieses Verfahren, als einziges, komplett deterministisch, mit einer Standartabweichung .

A screenshot of a game

Description automatically generated with medium confidence

Abbildung : Steffen Perfect Boardingverfahren

Am langsamsten hingegen ist das Verfahren „frontToBack“. Dabei wird das Flugzeug in drei Gruppen geboardet, wobei zuerst die vorderen Passagiere das Flugzeug betreten, dann die mittleren und zum Schluss die Passagiere im hinteren Teil des Flugzeugs. Dieses Verfahren ist aus der oben getroffenen Auswahl das langsamste, da nur ein kleinstmöglicher Teil an Passagieren zu seinem Sitz kommt und die gesamte Masse an Leuten warten muss, wenn eine Person seinen Platz verlässt oder sein Gepäck verstaut, siehe „Abbildung 11: Front to Back Boarding“.

A screenshot of a game

Description automatically generated with medium confidence

Abbildung 14: Front to Back Boarding

Wieso wird dieses Verfahren also modelliert? Obwohl das „Front to Back“ Verfahren praktisch das ineffizienteste Boarding Verfahren ist, kommt es dennoch häufig zum Einsatz. Zum einen bildet sich dieses, oder jedenfalls ein ähnlicher Ablauf, auf natürliche Weise von selbst, wenn das Flugzeug wieder verlassen wird, da zunächst alle vorderen Reihen einzeln geleert werden, bevor die hinteren Passagiere ihre Plätze verlassen können und zum anderen wird es bei Passagieren der ersten Klasse mit Absicht eingesetzt, da diese das Flugzeug zumeist als erstes betreten dürfen, aber die erste Klasse im vorderen Teil eines Flugzeugs lokalisiert ist.

Ein effizienteres Boarding Verfahren, laut Literatur () und den eigenen Versuchen, ist das komplett zufällige Boarding, welches in Europa vorwiegend verwendet wird. Aufgrund der vielen zufälligen Faktoren hat dieses Verfahren allerdings mit die höchste Standartabweichung, sodass ein Erfolg nicht garantiert ist.

Die beiden Erfolgreichsten, realistischen Verfahren sind „window-middle-aisle“, bei dem in drei Gruppen von außen nach innen geboarded wird und „steffen Modiefied“, bei dem nach dem gleichen Ablauf wie bei „steffen Perfect“ geboarded wird, allerdings in 4, statt in n Gruppen.

In der oben gezeigten Versuchsreihe ist „window-middle-aisle“ der klare Sieger, allerdings ist dies dem Fakt geschuldet, dass der Sitzwechselprozess bei der gewählten Simulation negativer gewichtet wird, als das Gepäck verstauen. Da bei „window-middle-aisle“ keine Sitzwechsel erforderlich sind, ist dieses Verfahren im Vorteil.

Welchen Effekt die Gepäck-Verstauzeit auf das Ergebnis haben kann ist im folgenden Diagramm zu erkennen (Abbildung 12: Gepäck Verstauzeit vs Gesamtzeit). Dabei wurden beispielhaft die beiden Verfahren „window-middle-aisle“ und „back to Front“ miteinander verglichen. Es ist zu erkennen, dass eine erhöhte Verstauzeit das Verfahren „back to Front“ negativer beeinflusst, als „window-middle-aisle“. So kann es sein, dass die Verfahren ihre Rangliste ändern, wenn die Simulation mit verschiedenen Parametern initialisiert wird, allerdings scheinen die meisten Verfahren mit der Zeit eher zu divergieren.

A picture containing text, line, plot, diagram

Description automatically generated

Abbildung : Gepäck Verstauzeit vs Gesamtzeit

## Quellcode

Der Code dieses Projekt, inklusive dieser Dokumentation und den Daten der statistischen Auswertung ist auf GitHub unter <https://github.com/JolanEggers/CellularAutomata/> einzusehen. Um die Hauptsimulation zu starten müssen die notwendigen Bibliotheken installiert werden und die Datei „main.py“ ausgeführt werden. In der Kommandozeile lässt sich dies mit dem Befehel „python main.py“ bewerkstelligen, vorausgesetzt Python ist installiert und man befindet sich im Projektordner.

# Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Modell des Flugezug-Boaradingprozesses aufgebaut und in durch eine höhere räumliche, sowie zeitliche Auflösung verbessert. Des Weiteren wurde das Verhalten der einzelnen Passagiere realistischer gestaltet, indem jedem Passagier zu einem Agenten umgeschrieben wurde, das das Verhalten der Person bestimmt, sodass die Passagiere bspw. kommunizieren können, um den Sitzplatz zu wechseln. Das Flugzeugmodell ist variabel aufgebaut, sodass Sitzanzahl, Sitzgröße, Flugzeugform, aber auch die räumliche oder zeitliche Auflösung geändert werden können. Auch der Agent ist parametrisierbar, bzw. modular aufgebaut, sodass individuelle Eigenschaften wie Geschwindigkeit oder Größe angepasst werden können.

An dieser Stelle könnten zukünftig weitere Verhaltensmuster hinzugefügt werden, bspw., eine komplexere Pfadfindung, bei der Passagieren einander ausweichen können, wenn der Gang breit genug ist. So kann mithilfe dieses Modell bspw. auch eine gute Ganggröße ermittelt werden bei der die Passagiere ihren Platz möglichst schnell den Platz erreichen, sodass nicht nur Boardingverhalten, sondern auch Flugzeugdesign mit einbezogen werden können. Eine Erweiterung im 3D-Raum wäre hier ebenfalls denkbar.

Obwohl das Modell qualitativ mit den vorangegangenen Studien übereinstimmt, wäre für eine genauere Validierung bzw. Parameteranpassung der gleiche Boarding Prozess, bspw. „window-middle-aisle“ simuliert und in der Realität ausgeführt werden um möglich das Verhalten der einzelnen Passagiere abzugleichen und ein genaueres Modell zu bekommen. Auch muss eine gute Balance zwischen Performance und Auflösung gefunden werden. Zum einen hat der Code an einigen Stellen noch Optimierungsbedarf, bspw. bei der Kollisionsdetektierung. Des Weiteren sind zelluläre Automaten im allgemeinen gut parallelisierbar, sodass einige Segmente auf der GPU ausgeführt werden können um die CPU zu entlasten (hier ist idR. ca. eine Größenordnung an Performance zu gewinnen).

Abschließend könnte dieses Programm mit anderen ihrer Art verglichen werden oder über eine automatische Art der Boarding Verfahren nachgedacht werden.

# Eigenständigkeitserklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende schriftliche Ausarbeitung selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bielefeld, 22.06.2023

Ort, Datum, Unterschrift