OST
Ostschweizer
Fachhochschule

Entwicklung einer Animationsbibliothek für SimPy

Studienarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2021

Autoren: David Kühnhanss, Moritz Schiesser
Betreuer: Marc Sommerhalder, Prof. Dr. Andreas Rinkel

Zusammenfassung

Ausgangslage

Simulationen werden verwendet, um reale Situationen unter variierenden Bedingungen nachzubilden und damit Verhaltensweisen zu untersuchen und vorherzusagen. SimPy ist ein schlankes Python Framework für diskrete Ereignissimulation und wird zum Beschreiben von Simulationen verwendet. JupyterLab bietet die Möglichkeit sogenannte Notebooks zu erstellen. Notebooks stellen eine webbasierte interaktive Programmierumgebung zur Verfügung in welcher unter anderem SimPy Simulationen geschrieben und verwendet werden können. Zur Verifikation der Ergebnisse einer solchen Simulation ist es wünschenswert eine zur Simulation passende Animation zur Hand zu haben. Momentan bietet SimPy noch keine integrierte Möglichkeit eine Simulation zu visualisieren und zu animieren.

Ziel

Vorbereitend auf eine Folgearbeit, welche eine Animationsbibliothek implementiert, wird in dieser Arbeit die Evaluation und Überprüfung von Technologien für die Visualisierung und die Animation von Simulationen durchgeführt. Ebenfalls existiert ein Architekturvorschlag für die Umsetzung einer Animationsbibliothek für SimPy.

Ergebnisse

Recherche zu SimPy und Jupyter führen zu einem klaren Bild der Situation. Die Integration in Jupyter, das Aufbereiten der Simulation für die Animation sowie das Animieren selbst sind der Fokus. Für diese Anforderungen werden Lösungsvorschläge ausgearbeitet und diskutiert. Zur Erleichterung der Vergleichbarkeit werden Funktionalitäten definiert, an welchen sich die Recherchen orientieren.

Bei den Recherchen handelt es sich um Prototypen für die Integration in Jupyter und Prototypen für die Animation, deren Ziel es ist eine Referenzsimulation zu visualisieren.

Die Arbeit definiert eine grundlegende Architektur zur Einbettung einer animierten Simulation in das Jupyter Ökosystem. Diese Architektur ist erweiterbar und modular. Eine Folgearbeit kann auf den bestehenden Prototypen aufbauen und auf Implementationsvorschläge zurückgreifen. Auf dieser Basis kann die Entwicklung der Animationsbibliothek durchgeführt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Ausgangslage

1.1 Aufgabenstellung

SimPy ist ein schlankes Python Framework für diskrete Ereignissimulation und beinhaltet von Haus aus Funktionalitäten im Bereich Prozessdefinition und Steuerung, Interprozessinteraktionen und geteilten Ressourcen. Im Vergleich zu anderen Simulationsframeworks fehlt jedoch der Bereich Animation. In dieser Arbeit soll daher die Grundlage für eine Folgearbeit geschaffen werden, in der eine Animationsbibliothek für SimPy in Jupyter entwickelt wird. Die Arbeit umfasst ein vorgängiges Recherchieren von Animationsmöglichkeiten, sowie die Evaluation von Technologiekandidaten. Im Rahmen von Prototypen sollen verschiedene Lösungsansätze erforscht und validiert werden. Mit den daraus gewonnenen Erkenntnisse soll ein Architekturvorschlag erstellt werden, welcher zusammen mit den dokumentierten Prototypen eine Grundlage für die Folgearbeit bietet.

1.2 Zielsetzung

Vorbereitend auf eine Folgearbeit, welche eine Animationsbibliothek implementiert, wird in dieser Arbeit die Evaluation und Überprüfung von Technologien für die Visualisierung und die Animation von Simulationen durchgeführt. Ebenfalls existiert ein Architekturvorschlag für die Umsetzung einer Animationsbibliothek für SimPy.

Kapitel 2

Stand der Technik

Dieses Kapitel verschafft einen Überblick über die im Verlauf der Arbeit begutachteten und verwendeten Technologien.

2.1 Vorausgesetzte Technologien

2.1.1 JupyterLab

JupyterLab ist Teil von Project Jupyter[1] wird verwendet, um Notebooks zur Verfügung zu stellen. Ein Notebook[2] ist eine webbasierte interaktive Programmierumgebung. Notebooks verwenden die von Jupyter bereitgestellten Kernels zum Ausführen von Code. Ein Notebook besteht aus mehreren Zellen, in einer Zelle steht beispielsweise ausführbarer Code oder beschreibender Text. An der Fachhochschule Ost werden solche Notebooks unter anderem in Übungen verwendet. Im Rahmen dieser Arbeit wird JupyterLab 3 oder höher verwendet.[1]

2.1.2 SimPy

Zum Implementieren von Simulationen wird SimPy in der Version 4 oder höher verwendet.

2.1.3 Python

Da SimPy vorausgesetzt ist, wird für die Integration der Animation und der Simulation Python verwendet. Im Rahmen dieser Arbeit wird Version 3.9 oder höher verwendet.

2.2 Bestehende Lösungen

2.2.1 Salabim

Salabim[3] ist ein in Python geschriebenes Framework zur Simulation und Animation von Modellen. Salabim zieht seine Stärken aus der engen Integration der Funktionalität zur Simulation und Animation. So werden sowohl für Simulationskomponenten als auch für Animationskomponenten Klassen der Bibliothek verwendet. Salabim ist eine standalone Applikation und eignet sich deshalb nicht zur Integration mit JupyterLab.

[1] https://jupyter.org/
[2] https://jupyter-notebook.readthedocs.io/en/stable/
[3] https://www.salabim.org/

2.3 Technologiekandidaten

Die Visualisierung wird als Teil eines Notebooks in JupyterLab dargestellt. Die in den folgenden Abschnitten beschriebenen Technologien sind Kandidaten für die Visualisierung einer Simulation.

2.3.1 bqplot

Im Rahmen der Evaluation von Möglichkeiten zur Integration von Livediagrammen in der Simulation wurde bqplot[4] untersucht und eingesetzt. Die Bibliothek bqplot ist eine Extension für Jupyter und erlaubt es Daten, welche in Python vorhanden sind, in Echtzeit als Diagramme abzubilden.

Die Architektur von bqplot war der entscheidende Hinweis, wie Daten einfach und sauber zwischen Python und JavaScript innerhalb eines Jupyter Notebooks übertragen werden können. Bqplot benutzt zum Erstellten der Diagramme D3.

2.3.2 JavaScript Visualisierungsbibliotheken

JupyterLab ist eine Browseranwendung, deshalb kann die Visualisierung mit Browsertechnologien wie JavaScript umgesetzt werden. Alle erwähnten Technologien sind Open Source und können unter den entsprechenden Lizenzbedingungen eingesetzt werden.

2.3.2.1 D3.js

D3[5] wird oft als Standard für Daten Visualisierung mit JavaScript beschrieben, hat eine grosse Community (99k GitHub Stars) und wird aktiv unterhalten. D3 bietet eine Schnittstelle um Daten und das Document Object Model (DOM) zu verbinden, unterstützt Debugging und bietet solide Performance. Mit D3 können Daten in einer beliebigen vom DOM unterstützten Form dargestellt und animiert werden. [2]

Die Popularität von D3 zeigt sich auch dadurch, dass Alternativen darauf aufbauend, zusätzliche oder vereinfachte Verwendungsmöglichkeiten zur Verfügung stellen, wie beispielsweise Vega[6]. [3]

2.3.2.2 Chart.js

Chart.js[7] wird hauptsächlich für Diagramme verwendet, hat deshalb entsprechende Einschränkungen und eignet sich nicht für die Visualisierung von Simulationen. Diese Bibliothek könnte zum Darstellen von Livediagrammen verwendet werden.

2.3.2.3 three.js

Three.js[8] ist eine JavaScript 3D Bibliothek mit einer grossen Community (75k GitHub Stars) die aktiv unterhalten wird. Three.js basiert auf WebGL und profitiert deshalb von einer Hardwareunterstützung durch die Grafikkarte.

2.3.2.4 PixiJS

PixiJS[9] beansprucht für sich der schnellste und flexibelste 2D WebGL Renderer zu sein. PixiJS verfügt über eine grosse Community (35k Github Stars) und wird aktiv unterhalten.

[4] https://github.com/bqplot/bqplot
[5] https://d3js.org/
[6] https://github.com/vega/vega
[7] https://www.chartjs.org/
[8] https://threejs.org/
[9] https://pixijs.com/

Kapitel 3

Erarbeitung von Konzepten und Technologien

Dieses Kaptiel beschreibt die während der Arbeit erstellen Prototypen. Prototypen wurden erstellt um komplexe Konzepte zu verstehen und auf Basis der implementierten Grundlage zu diskutieren. Des Weiteren berücksichtigen die Prototypen in der Arbeitsgruppe diskutierte Ideen und beinhalten erste Varianten für deren Umsetzung.

3.1 Anforderungen an Prototypen

Die Prototypen sind auf die folgenden beschriebenen Anforderungen ausgelegt. Die Ideen und die Voraussetzungen haben sich im Verlauf der Arbeit weiterentwickelt deswegen ist beispielsweise der Use Case “Agent verfolgen” nur im PixiJS Prototyp implementiert und nicht alle Anforderungen sind vollständig in jedem Prototypen umgesetzt.

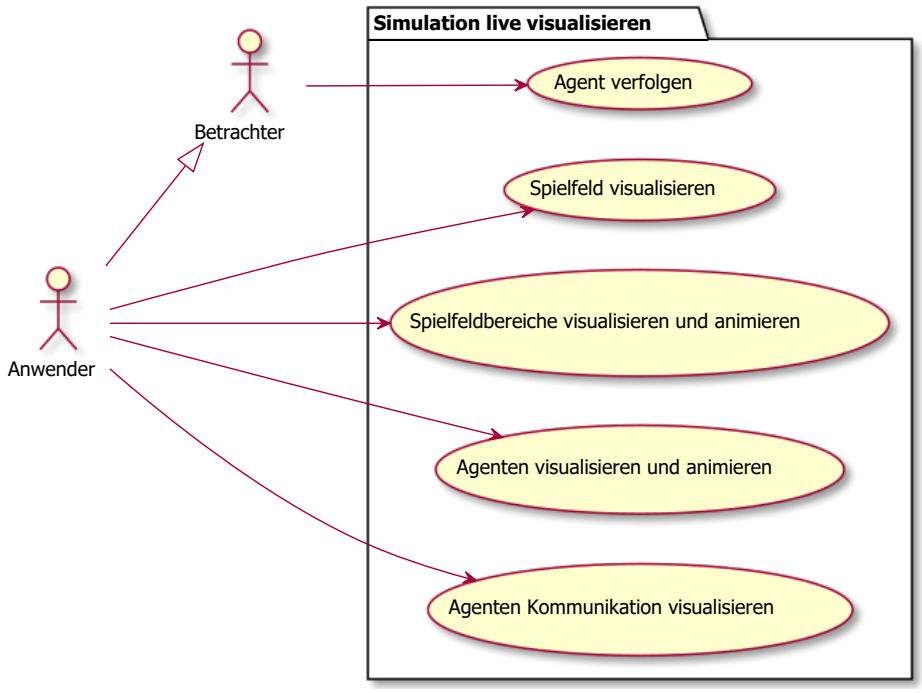

Abbildung 3.1: Prototypen Use Cases

Bei diesen Use Cases handelt es sich um Ideen und Vorschläge und noch nicht um spezifische Anforderungen für eine major Version. Wie und in welchem Umfang diese Use Cases in einer major Version vorhanden sein sollen, ist im Abschnitt Major Version Features[1] des Kapitels Ausblick und Vision beschrieben.

3.1.1 Spielfeld visualisieren

Standardablauf: Der Anwender der Animationsbibliothek möchte eine fertige oder angefangene Simulation visualisieren, die mit SimPy in einem Jupyter Notebook vorhanden ist. Dafür muss ein Spielfeld definiert werden, auf welchem die Simulation visualisiert wird. Der Anwender erstellt ein Standardspielfeld oder passt die Grösse seinen Bedürfnissen an.

3.1.2 Spielfeldbereiche visualisieren

Vorbedingungen:

- Ein Spielfeld ist definiert.

Standardablauf: Die Simulation benötigt mehrere Bereiche, also Orte an welchen sich ein Agent aufhalten kann, dazu teilt der Anwender das Spielfeld in mehrere Bereiche ein. Visuelle und strukturelle Eigenschaften wie Farbe und Türen des Bereichs werden vom Anwender definiert. Implizit werden Wände dort erstellt, wo keine Türen definiert sind.

3.1.3 Agenten visualisieren und animieren

Vorbedingungen:

- Spielfeld verfügt über mindestens einen Bereich.

Standardablauf: Der Anwender definiert in der Simulation Agenten und deren visuelle Eigenschaften, also Farbe, Form oder Bild. Der Agent wird vom Anwender einem Bereich zugewiesen und Änderungen müssen live

[1] 7.1 Major Version Features

in der Visualisierung animiert werden. Der Anwender definiert für die verschiedenen Zustände des Agenten ein visuelles Verhalten, beispielsweise eine tanzende Bewegung.

Alternativer Ablauf:

- Der Anwender weist den Agenten keinem Bereich zu, deshalb wird der Agent nicht dargestellt.
- Der Anwender definiert keinen Zustand auf dem Agenten, dann wird kein oder ein simulationsweiter Standardwert als visuelles Verhalten verwendet, der zu Beginn als Eigenschaft des Spielfelds definiert wurde.

3.1.4 Agenten Kommunikation visualisieren

Vorbedingungen:

- Ein Spielfeld mit Bereichen und darauf platzierten Agenten existiert.

Standardablauf: Der Anwender definiert für jeden Agenten andere Agenten mit welchen eine Kommunikation dargestellt wird. Im Verlauf der Simulation ändert der Anwender die Abhängigkeiten und erwartet, dass sie live animiert werden.

3.1.5 Agent verfolgen

Vorbedingungen:

- Eine fertige Simulation mit mindestens einem Agenten ist vorhanden und wird ausgeführt.

Standardablauf: Der Betrachter wählt einen Agenten aus, den er verfolgen möchte. Die visuelle Darstellung wird ständig so angepasst, dass der ausgewählte Agent der Mittelpunkt der Darstellung ist.

3.2 Referenzsimulation

Die in dieser Arbeit durchgeführten Recherchen und Prototypimplementationen sind alle auf dieselbe simple Simulation ausgelegt.

Diese Simulation erhebt keinen Anspruch auf Korrektheit und nahen Bezug zur Realität, und auch nicht darauf eine gut konzipierte Simulation zu sein. Das primäre Ziel dieser Simulation ist, möglichst viele Aspekte einer Agentenbasierten Simulation abzudecken und somit eine Grundlage für Demonstration und Tests zu bieten. Die Simulation ist in SimPy als real-time Simulation[2] implementiert. Die Referenzsimulation ist als Teil eines Notebooks des D3 Prototyp implementiert und im Appendix[3] vorhanden.

3.2.1 Ausgangslage

Ein Lokal einer kleinen Stadt hat an einem beliebigen Abend eine Anzahl von Besuchern und Besucherinnen. Diese Besucher:innen tanzen gerne, konsumieren gerne Getränke an der Bar, quatschen gerne miteinander und müssen - nachdem sie genug getrunken haben - zur Toilette.

Dieser Nachtclub ist in 4 Bereiche unterteilt: einen Eingangsbereich, eine Tanzfläche, einen Barbereich und einen Toilettenbereich. Die Musik auf der Tanzfläche wechselt regelmässig, und nicht alle Besucher:innen möchten zu allen Musikstilen tanzen. Gewisse Besucher:innen ziehen also Schlager oder Techno vor, andere tanzen nur zu Discostampfer. Die Bedienung an der Bar ist zwar bemüht möglichst schnell zu arbeiten, allerdings kann es doch zu Wartezeiten kommen. Ebenso im Toilettenbereich wo nur eine kleine Anzahl an Kabinen zur Verfügung steht.

[2] https://simpy.readthedocs.io/en/latest/topical_guides/real-time-simulations.html
[3] Appendix: d3_danceclub/danceclub_combination.ipynb

3.2.2 Eigenschaften eines Besuchers

Anmerkung: Die Besucher heissen in der Implementation 'Guest'.

Aus der oben beschriebenen Situation lassen sich entsprechend folgende Eigenschaften ableiten:

state:

Zustand / aktuelle Aktivität des Guest. Element aus [WAITING, TALKING, DANCING, ON_TOILET].

location:

Standort des Guest, also einer der 4 Bereiche [ENTRY, BAR, DANCEFLOOR, TOILET].

liked_music_styles:

Musikstile zu welchen die Guests gerne tanzen. Auswahl aus [SCHLAGER, TECHNO, DISCOSTAMPFER].

known_peers:

Andere Guests welche der Guest kennt.

current_peers:

Andere Guests mit welchen der Guest aktuell interagiert.

drink_level:

Füllungsgrad des aktuellen Getränks.

bladder_level:

Füllungsgrad der Blase des Guest.

3.2.3 Verhaltensregeln

Das definierte Verhalten der Guests wird - wie für Simulationen in SimPy üblich - als Endlosschleife ausgeführt [4]. Die Simulationsumgebung von SimPy übernimmt dabei das Ausführen und Pausieren dieser Routine, und ebenfalls die Terminierung der Routine nach definierter Zeit.

Aus der oben beschriebenen Situation lassen sich folgende Regeln ableiten:

1. Falls `bladder_level > 90` wird Guest sich in die Warteschlange vor der Toilette stellen.
2. Falls `drink_level > 0` wird Guest einen Schluck aus seinem Getränk nehmen.
3. Ist aktuell aufgelegter Musikstil in `liked_music_styles` enthalten, beginnt Guest zu tanzen.
4. Ist `state == DANCING`, und andere Guests mit Aktivität `DANCING` sind in `known_peers` enthalten, beginnt Guest mit diesen zu interagieren, also mit diesen zu tanzen. Diese Guests werden dann in `current_peers` eingefügt.
5. Falls `state == DANCING` und aktueller Musikstil nicht in `liked_music_styles` enthalten, hört Guest auf mit den `current_peers` zu tanzen und verlässt die Tanzfläche. Guest wechselt zu `location BAR` und hat nun den `state WAITING`.
6. Falls `drink_level <= 0` reiht sich Guest in die Warteschlange der Bar ein, um einen neuen Drink zu holen.
7. Falls `location == BAR` und andere Guests in `known_peers` ebenfalls im Barbereich sind, beginnt Guest mit diesen zu interagieren, folglich haben beide Guests nun `state = TALKING`.

3.2.4 Eigenschaften eines Bereichs

Aus der oben beschriebenen Situation lassen sich entsprechend die folgenden Eigenschaften für die Bereiche ableiten:

Gemeinsame Eigenschaften

location_type:

Typ des Bereichs, Element aus [ENTRY, BAR, DANCEFLOOR, TOILET].

capacity:

Kapazität des Bereichs, die Anzahl Guests welche sich im Bereich aufhalten können.

Dancefloor

music:

Aktuell abgespielter Musikstil.

Kapazitäten

Dancefloor: 15

Entry: 200

Bar-Bedienung: 2

Toiletten-Kabine: 1

3.2.5 Weitere Agenten und Simulationselemente

DJ

Das DJ-Simulationsobjekt ist zuständig für das regelmässige Wechseln des Musikstils auf dem DANCEFLOOR.

3.3 Prototypen zur Datenübertragung

Eine grosse Herausforderung zu Beginn dieser Arbeit war, Simulationsdaten von Jupyter Notebooks in die Animationsbibliothek zu übertragen. Python Daten, welche in der Zelle eines Notebooks existieren, müssen so bereitgestellt werden, dass sie in JavaScript verarbeitet werden können. Die erste Implementation deklarierte JavaScript Code direkt als Strings in Python. Dieser Code konnte dann als HTML gerendert, und entsprechend ausgeführt werden.

Folgendes Beispiel gibt einen Eindruck:

```
from IPython.core.display import display, HTML
data = 0
display(HTML('''
  <script type="module">
    import * as d3 from "https://cdn.skypack.dev/d3@7";
    let svg = d3.select("#d3_diagram");
    let numberOfItemsInRow = parseInt(%d, 10);
    ...
  </script>
''' % data))
```

Dies ist kein stabiler, skalierbarer Weg um grosse Animationslogiken deklarieren zu können.

Die nächste, wohl wichtigste Evolutionsstufe war, eine Extension für Jupyter zu schreiben, welche tieferen Zugang zu Jupyter Kernel und den darin vorhandenen Möglichkeiten hat.

Um die nachfolgenden Erläuterungen und Beschreibungen nachvollziehen zu können, muss erst die Grundlage von Jupyter und Jupyter Extensions geschaffen werden. Die im Rahmen dieser Arbeit untersuchten Möglichkeiten zur Implementation setzen als Grundlage jeweils ein “Custom Widget” auf Basis von ipywidgets ein [5]. Bei einer Widget-Extension dieser Art übernehmen Jupyter, und insbesondere ipywidgets die Implementation zur Synchronisation von Python und Web-Frontend. Dieser Austausch geschieht über Kommunikationskanäle in denen Messages in Form von JSON bidirektional ausgetauscht werden können. Die Dokumentation von ipywidgets stellt folgende Übersicht zur Verfügung:

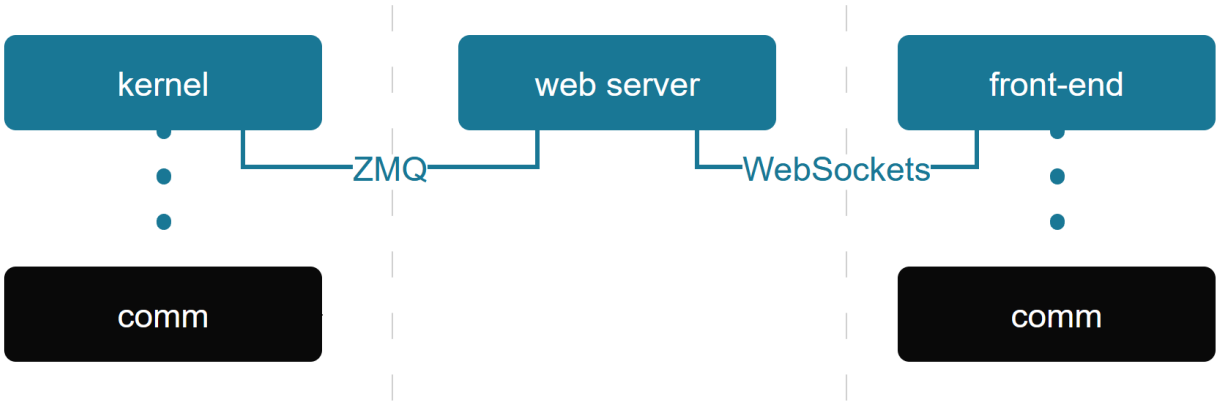

Abbildung 3.2: Jupyter Notebook `comms`

[6]

Wenn nun also ein synchronisiertes Property eines Widgets im Pythoncode der Notebookzelle verändert wird, wird damit der oben beschriebene Weg zur Datenübertragung ausgelöst. Dies schafft für die Entwicklung von komplexeren Bibliotheken viel Komfort. Sehr leicht können nun Daten, welche von der Simulation produziert werden, an die benötigte Stelle in der Visualisierungslogik übertragen werden.

3.3.1 Performance

Die Performance der Implementation - wie in jedem Projekt - ist auch hier ein wichtiger Aspekt. Während dieser Arbeit sind verschiedene Herausforderungen bezüglich der Performance aufgetreten, was wiederum die Stabilität und Zuverlässigkeit der Implementation beeinträchtigte.

Die Anforderungen welche diese Arbeit an die Kommunikationskanäle stellt, bringen eine naive Implementation mit Nutzung der Übertragungsfähigkeiten der Widgets jedoch schnell an ihre Grenzen. Der Kernel von JupyterLab hat ein Limit für die Anzahl solcher Messages welche in einem gewissen Intervall gesendet und empfangen, also bidirektional verarbeitet werden können. Per Default ist dieses Limit bei 3000 Messages in 3 Sekunden. Dieses Limit ist insofern nachvollziehbar, als die Serialisierung und Deserialisierung von JSON Messages sehr CPU intensiv ist. Folgendes Sequenzdiagramm stellt den obigen Ablauf aus einer anderen Perspektive dar:

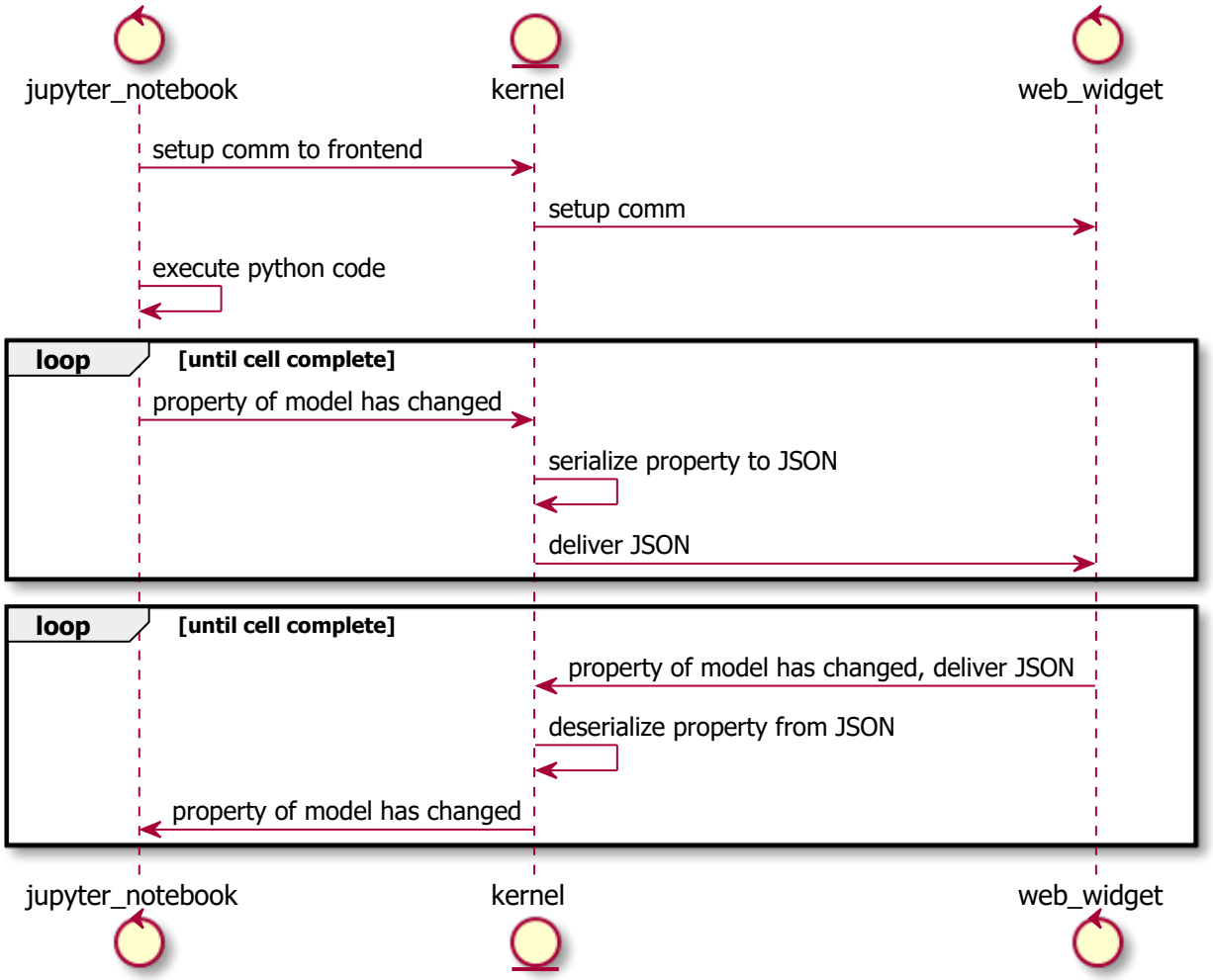

Abbildung 3.3: Jupyter Notebook Datentransfer

Bei jeder Änderung eines Properties in Python, und bei jeder Änderung eines Properties im Web-Frontend, wird also der Vorgang des Sendens einer Message ausgelöst. Im Rahmen dieser Arbeit existiert effektiv nur eine unidirektionale Nachrichtenübermittlung vom Python 'Backend' zum Web-Frontend. Das Web-Frontend hat keine interaktiven Elemente welche die Ausführung in Python beeinflussen. Entsprechend ist der oben dargestellte, zweite Loop im Sequenzdiagramm im Kontext dieser Arbeit nur theoretisch.

3.3.1.1 Bottlenecks in frühen Architekturentwürfen

Das Architekturdesign welches im Rahmen dieser Arbeit zustande kam, wurde iterativ entwickelt. Die nachfolgenden Beschreibungen zeigen die Evolution welche geschah, und den Einfluss welche die Performancebedenken hatten. Wie zuvor beschrieben sendet also Jupyter bei jeder Änderung an einem Property in Python eine Nachricht an das Web-Frontend. Die erste, naive Implementation hatte den Ansatz, den gesamten relevanten Datenstamm bei jeder Änderung eines Agentenobjekts oder Ressourcenobjekts neu zu bilden, und diesen als Änderung dem Web-Frontend zu übergeben. Allerdings hatte diese Implementation nur zum Ziel, einen vertikalen Durchstich als Prototypen zu erreichen, und konzentriert sich nicht auf die Optimierung der Ressourcenverwendung. Trotzdem wird bei diesem Ansatz schnell klar, dass die Durchsatzrate von JupyterLab schnell überschritten wird, sofern die Agentenobjekte in Anzahl genügend gross, und / oder

schnell mutierend sind. Der erste Prototyp mit D3.js[4] demonstriert, dass die Kommunikation der Änderungen in der Simulation erfolgreich ans Web-Frontend übergeben werden kann. Dieser Prototyp wurde mit wenigen, langsam mutierenden Agenten demonstriert und durchgeführt. Offensichtlich ist jedoch, dass die Serialisierung des gesamten Datenstamms keine optimale Lösung ist. Regelmässig werden bereits bekannte und somit redundante Daten übermittelt, der effektive Informationsgehalt pro Nachricht ist also klein.

Property based Updates

Die Weiterentwicklung dieses Prototyps bringt das Konzept von 'Property-based Updates' mit sich, um diese Ineffizienz zu eliminieren. Das Konzept setzt voraus, dass konsistente Datenzustände sowohl im Backend in Python, als auch im Web-Frontend in TypeScript gehalten werden. Die Synchronisation von Änderungen geschieht hierbei auf Basis einzelner Eigenschaften von einzelnen Objekten. Diese `EntityUpdates` haben folgende Struktur:

```
{
    "id": "id",
    "property": "property-mutated",
    "value": "new-value"
}
```

Der Informationsgehalt pro Nachricht ist mit diesem Konzept signifikant höher als zuvor. Folgendes Ablaufdiagramm gibt einen Überblick:

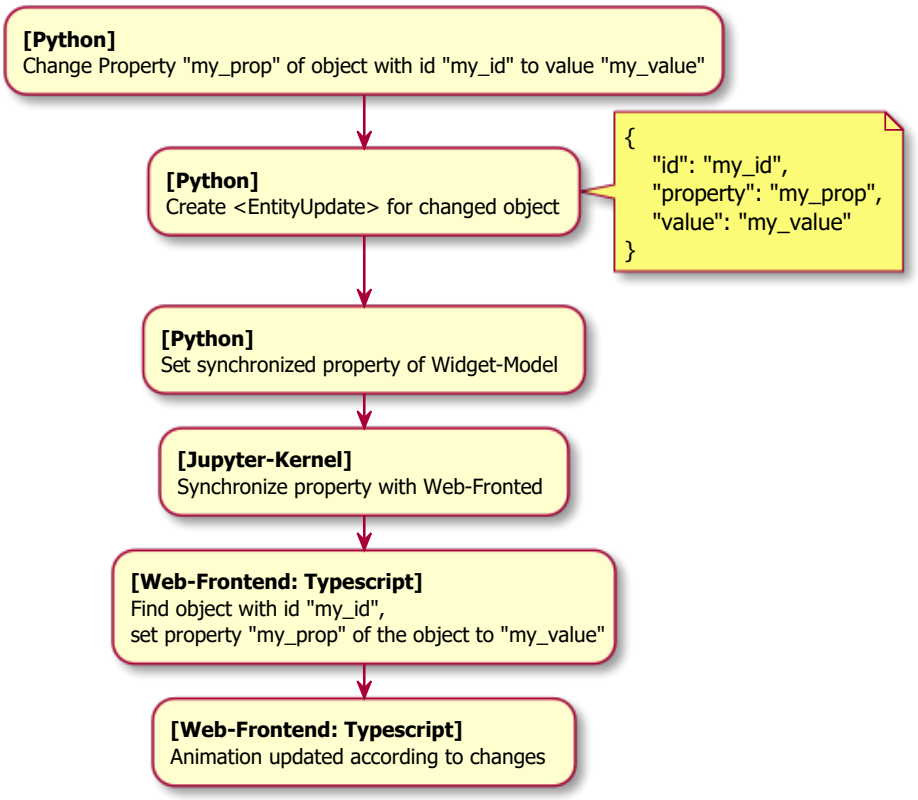

Abbildung 3.4: Property based Update

Wie bereits beim ersten Prototypen funktioniert dieser Ansatz sehr gut für Simulationen mit wenigen Agenten, oder Agenten welche wenig mutieren. Jedoch bereits bei Simulationen mit wenigen hundert Agenten wird

[4] 3.4 D3.js Prototyp

auch hier die Nachrichtenkapazität von JupyterLab überschritten.

Property based Updates mit Message Bundles

In Gesprächen mit der Arbeitsgruppe welche diese Arbeit durchführt und betreut wurde die Anforderung geäussert, dass auch Simulationen mit einer grossen Zahl von Agenten simuliert werden können sollen. Diese Anforderung führte zum nächsten Evolutionsschritt des Softwaredesigns, dem Konzept von `EntityUpdate Bundles`. Dabei werden eine definierte Anzahl Nachrichten gruppiert, und dann gemeinsam als Nachrichtenbündel serialisiert. Dieses serialisierte Nachrichtenbündel fungiert dann als 'Briefumschlag' für alle enthaltenen einzelnen "Property Updates". Entsprechend muss JupyterLab effektiv nur eine Nachricht übertragen. Folgendes Ablaufdiagramm gibt einen Überblick:

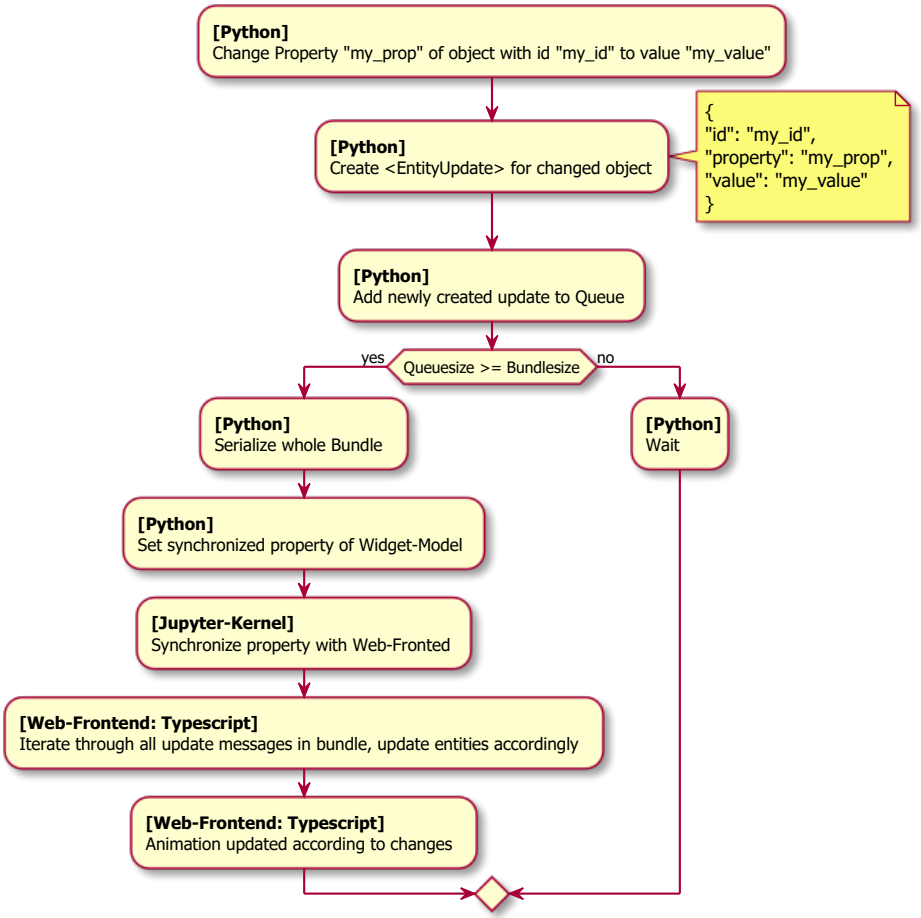

Abbildung 3.5: Property based Update mit Bundles

Dieses Konzept hat grossen Erfolg darin, die effektive Anzahl Nachrichten welche JupyterLab verarbeiten muss zu senken, der Faktor der Verbesserung ist dabei die definierte Grösse des Nachrichtenbündels. Wird eine Nachrichtenbündelgrösse von 5 festgelegt, können anstelle von 3000 msg/3s nun 15'000 msg/3s übertragen werden, bei einer Nachrichtenbündelgrösse von 10 sind es bereits 30'000 msg/3s.

Obwohl erfolgreich, lässt sich bei diesem Ansatz schnell ein Problem feststellen: Nachrichten werden lange nicht, oder am Ende der Simulation sogar niemals übertragen. Dies liegt daran, dass Nachrichtenbündel nicht in garantiert vorhersehbarer Zeit gefüllt werden können. Sind also am Ende der Simulation noch die letzten 3 Nachrichten in der Queue, werden sie dem Web-Frontend nie mitgeteilt.

Multi-Threaded Property based Updates mit Message Bundles

Um dieses Problem zu lösen, muss Multithreading eingesetzt werden. Ein designierter Thread ("Message-emitter thread") bearbeitet dabei periodisch die aktuelle Queue welche durch den Main-Thread oder Simulationsthread gefüllt wird. Somit kann zusätzlich zur Bedingung `QueueSize >= BundleSize` eine zeitliche Bedingung hinzugefügt werden. Diese Bedingung besagt, dass (sofern aktuell Nachrichten in der Queue sind) mindestens alle `x` Millisekunden ein Nachrichtenbündel übertragen werden soll, unabhängig von der Grösse ebendieses.

Das zuvor betrachtete Ablaufdiagramm ändert sich entsprechend wie folgt:

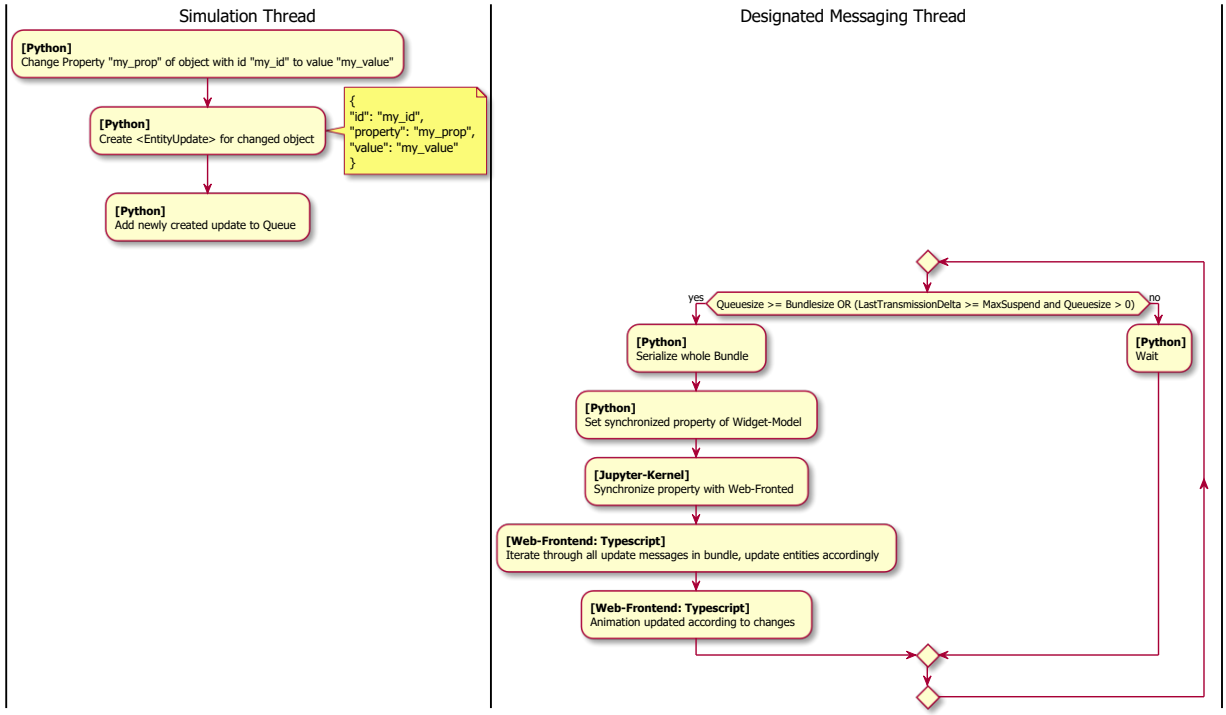

Abbildung 3.6: Property based Update mit Bundles und maximaler suspend time

Die Implementation eines naiven Ansatzes dieser Logik in Python ist an sich trivial. Dafür verantwortlich sind 2 Faktoren. Die temporär suspendierten Nachrichten werden in einer Behälterstruktur gesammelt welche mit garantiert atomaren Funktionen manipuliert werden kann, entsprechend müssen keine Thread-Synchronisierungskonzepte mit Locks eingesetzt werden [7]. Weiter ist es so, dass Python aufgrund des "Global Interpreter Locks" pro Prozess nur einen der aktiven Threads ausführen kann [8].

Obwohl dank dieser beiden Punkte einiger Overhead durch Thread Synchronisation vermieden werden kann, haben Threads an sich trotzdem einen Overhead. Werden nun also die Nachrichtenbündel zu gross, dauert die Serialisierung ebendieser sehr lange - während dieser Zeit ist die Simulation blockiert und kann keine Aktionen durchführen. Um dies zu verbessern ist es also nötig, die Grösse der Nachrichtenbündel, und die Maximale Suspendierungsdauer sensibel zu wählen. In der Realität zeigt sich, dass Änderungen an der Anzahl der Agenten, oder der Häufigkeit von Mutationen die idealen Werte dieser beider Parameter stark beeinflussen. Um für jeweilige Simulationen die besten Werte zu finden, werden wohl mehrere Versuche durch den Anwender dieses Projekts nötig sein. Weiter ist es so, dass die Serialisierungsdauer, und damit verbunden die maximal mögliche Desynchronisation des Web-Frontends relativ zur effektiven Simulation direkt von der eingesetzten Hardware abhängen.

Dynamische Skalierung der Bündelgrössen

In Versuchen hat sich gezeigt, dass die Referenzsimulation - vermutlich auch weil diese nicht von Simulationsexperten implementiert wurde - besonders zu Beginn der Simulation eine hohe Anzahl von Eigenschaftsmutationen aufweist. Somit sind zu Beginn sehr grosse Nachrichtenbündel ideal, später aber zeigt sich, dass Nachrichtenbündel nahezu überflüssig sind. Um dieser Realität gerecht zu werden, wurde damit experimentiert die Bündelgrösse dynamisch zu skalieren. Dabei wurde, falls wiederholt die maximale Anzahl Nachrichten in einem Bündel übertragen wurden, die Bündelgrösse mit einem Faktor erweitert, und umgekehrt, falls wiederholt Bündel mit weniger als der möglichen Grösse übertragen wurde, die Bündelgrösse verkleinert. Dieser Ansatz - obwohl zumindest teilweise erfolgreich - kann jedoch nicht als allgemeingültig angesehen werden. Es wird vermutet, dass der Erfolg dieses Ansatzes stark von der spezifischen Implementation der Simulation abhängt. Sollten bei Folgearbeiten weiterhin Performanceprobleme auftreten, ist vorstellbar, dass dieser Ansatz weiterverfolgt werden kann, es müssen allerdings zuvor simulationsagnostische Testbedingungen geschaffen werden, um diese Hypothese zu überprüfen.

3.3.1.2 Fazit zur Performance in Jupyter

Sehr grosse Simulationen (Anzahl Agenten > 5000) stellen für die aktuelle Architektur der Nachrichtenübermittlung eine Herausforderung dar. Es ist vorstellbar, dass eine Implementierung dieses Ansatzes den Anwendenden die Möglichkeit gibt, die Art der Nachrichtenübermittlung zu konfigurieren, sodass für die jeweilige Simulation die möglichst ideale Zusammensetzung eingesetzt werden kann.

Die aktuellen Limits der Performance können im entsprechenden Kapitel[5] eingesehen werden.

3.3.1.3 Performance im Web-Frontend

In den vorhergehenden Abschnitten werden die Limits der aktuellen Architektur in Zusammenhang mit JupyterLab behandelt. Ebenfalls ist darin von “sehr grossen Simulation” die Rede, also Simulationen mit mehreren Tausend Agenten. Diese Agenten müssen im Frontend ebenfalls dargestellt werden können. Die anfänglich eingesetzte Javascript Bibliothek D3.js[6] rendert die definierten Elemente als SVG oder mittels HTML5 Canvas, und hat ebenfalls Limits. So können ‘nur’ rund 10’000 HTML Elemente dargestellt werden, bevor die Performance merklich degradiert. [9]

Andere Libraries, konkret PixiJS[7] oder three.js[8], setzen für die Darstellung von dynamischen Objekten auf die Hardwareunterstützung durch Grafikkarten. Durch den Einsatz dieses Konzepts erreichen sie deutlich höhere Kapazitäten bei der Darstellung von Objekten.

3.4 D3.js Prototyp

Dieser Prototyp ist eine funktionsfähige JupyterLab Extension, welche ein ipywidget[9] `Custom Widget` zur Verfügung stellt. Der Prototyp zeigt auf, wie eine bestehende Simulation - in diesem Fall die Nachtclub Referenzsimulation - welche in einem Jupyter Notebook implementiert ist, mit Verwendung von D3 visualisiert und animiert wird. Für die Integration von JupyterLab und Animation wurde eine Vorgängerversion des im Kapitel Prototypen zur Datenübertragung[10] beschriebenen Prototypen verwendet. [10]

3.4.1 Implementierte Funktionalitäten

3.4.1.1 Visualisierung und Animation

Die naive Implementation des Python-Teils der Extension ermöglicht, dass durch die Simulation ausgelöste Änderungen an das `Custom Widget` übertragen werden und im JavaScript behandelt werden können. Bei jeder Änderung wird der gesamte Zustand der Simulation als JSON übertragen. Beim initialen Aufruf

[5] 4.1 Qualitätsziele
[6] https://d3js.org/
[7] https://pixijs.com/
[8] https://threejs.org/
[9] https://ipywidgets.readthedocs.io/en/latest/index.html
[10] 3.3 Prototypen zur Datenübertragung

des `Custom Widget` werden das Spielfeld und die Bereiche erstellt und als SVG bestehend aus SVG `rect` Elementen[11] im Notebooks dargestellt. Anschliessend werden die einzelnen Agenten ausgelesen. Falls der Agent unbekannt ist oder sich verändert hat, wird er neu als SVG `circle` Element[12] visualisiert.

Im laufenden JupyterLab kann das existierende danceclub_combination.ipynb[13] Notebook gestartet werden und mit Klick auf den `start` Button wird die Simulation gestartet und die Animation beginnt.

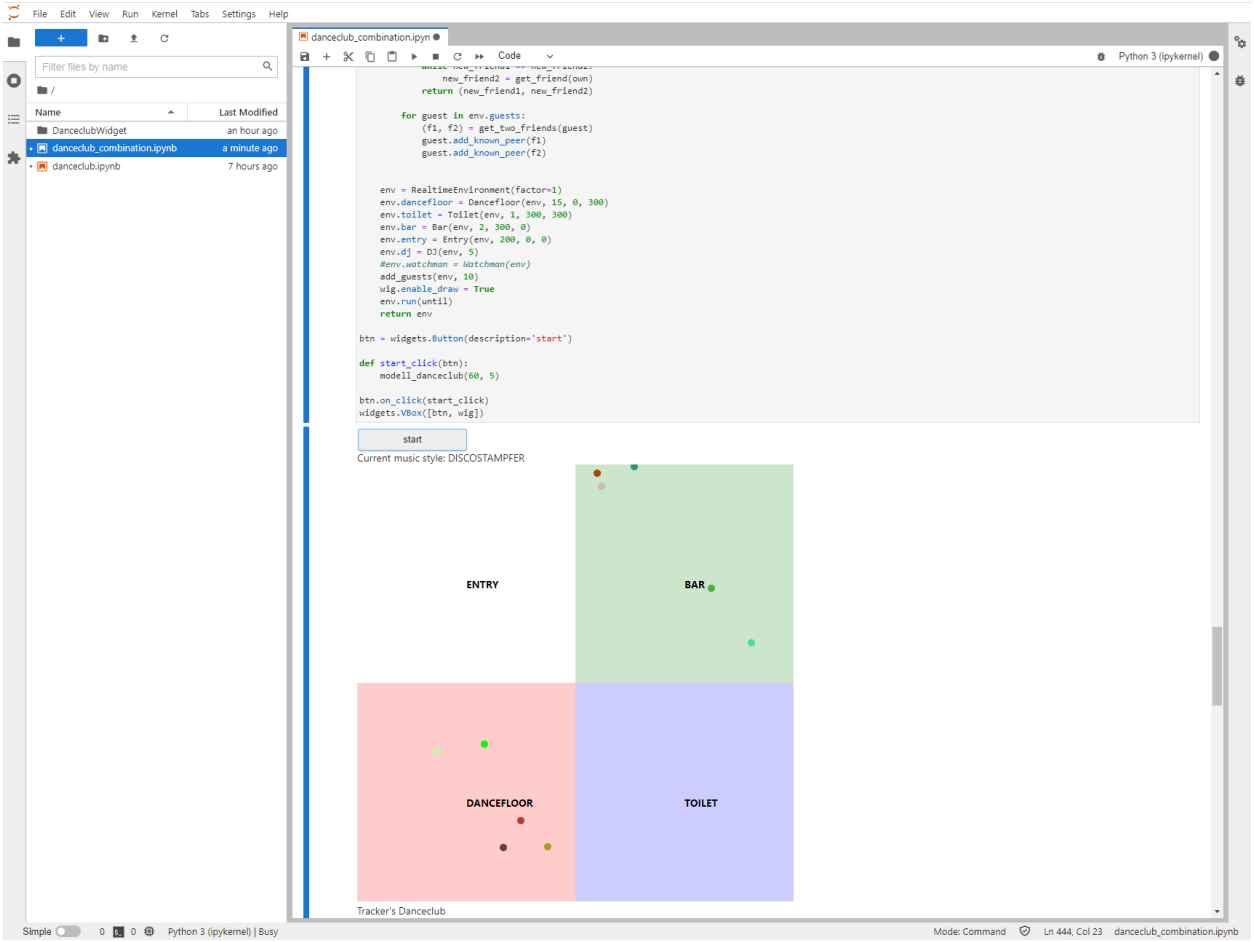

Abbildung 3.7: D3 Prototyp: Visualisierte Simulation

Eigenschaften wie die Höhe oder Breite des Spielfelds und der Bereiche sind als Teil der Simulationsobjekte definiert und werden für die Visualisierung ausgelesen. Das Erstellen des Spielfelds sowie das Definieren dessen Eigenschaften müssen im Notebook wie folgt vorgenommen werden:

```
1 import DanceclubWidget as dc
2 self.wig = dc.Club()
3 self.wig.value = "Danceclub"
4 self.wig.height = 600
5 self.wig.width = 600
```

[11] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/rect
[12] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/circle
[13] Appendix: d3_danceclub/danceclub_combination.ipynb

Im JavaScript werden die Werte des Spielfelds mit der Verwendung der Funktion `drawField`, welche Teil der in diesem Prototyp implementierten Funktionalität ist, visualisiert:

```
animtations.drawField(
  {
    cells: cells,
    height: this.model.get('height'),
    width: this.model.get('width'),
  },
  'svgcontainer'
);
```

Im Verlauf der Simulation entstehen Situationen in welchen die Agenten miteinander kommunizieren. Die Kommunikation wird als SVG `line` Element[14] in der Farbe des Agenten von welchem die Kommunikation ausgeht dargestellt.

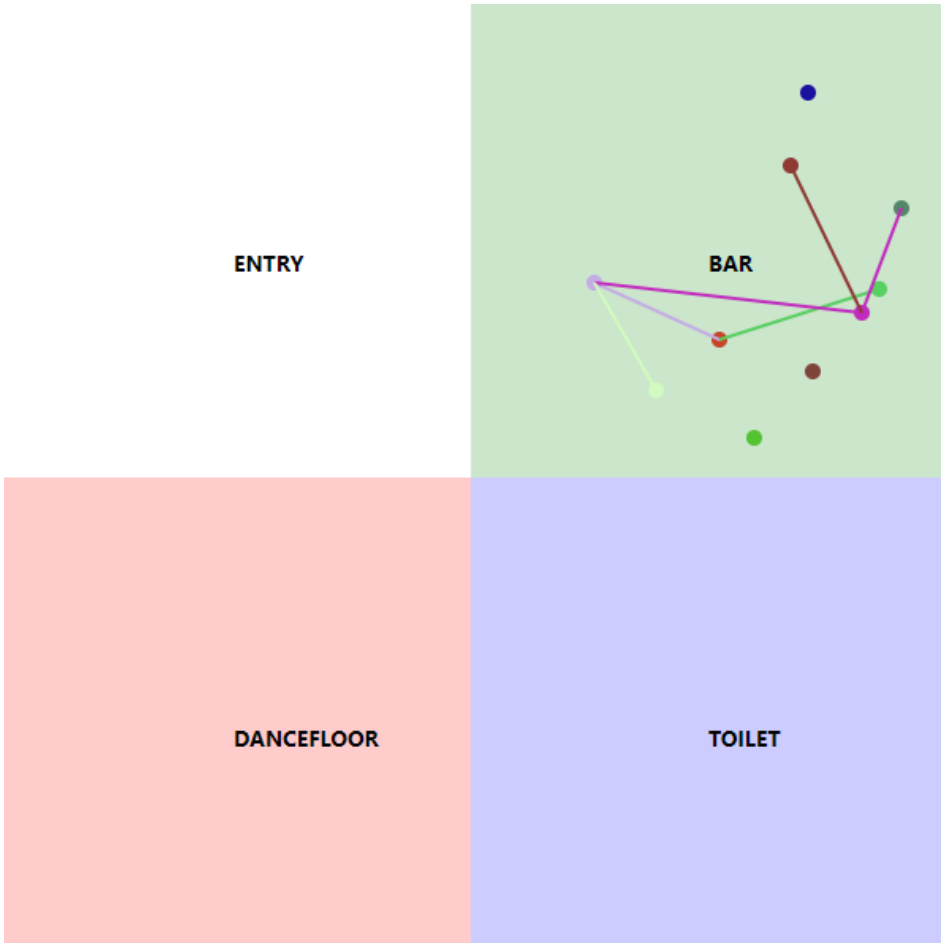

Abbildung 3.8: D3 Prototyp: Visualisierung der Kommunikation

Der Status des Agenten beeinflusst sein Verhalten, in diesem Fall wird anhand der `location` eines Agenten unterschiedliches Verhalten abgebildet. So bewegen sich Agenten, welche sich im Bereich `DANCEFLOOR` befinden 8-förmig, als würden sie tanzen. Befinden sich die Agenten hingegen im Bereich Bar, wird die Kommunikation dargestellt.

[14] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/line

3.4.1.2 Einschränkungen

Der Prototyp funktioniert, hat aber noch folgende Bugs:

- Eine Simulation kann mit Verwendung des `start` Button nur einmal gestartet werden, für einen zweiten Start muss die Seite neu geladen werden.
- Die Kommunikation wird nicht immer korrekt dargestellt, teilweise bleiben Kommunikationslinien zu lange sichtbar.

3.4.1.2.1 Performance

Während der Entwicklung des Prototyps sind Fragen bezüglich Performance aufgekommen, und die Anforderung für eine sehr grosse Anzahl Agenten wurde gestellt. Da D3 ohne Hardwareunterstützung durch die Grafikkarte auf dem DOM operiert, sind gewisse Einschränkungen vorhanden und das Visualisieren von mehr als 10'000 Elementen ist nicht mehr in der gewünschten Qualität möglich. Für eine flüssige Visualisierung und Animation bei einer hohen Anzahl Elemente, eignen sich Visualisierungsbibliotheken mit einer Hardwareunterstützung durch die Grafikkarte, wie beispielsweise threejs[15], PixiJS[16], D3FC[17] oder Stardust [18] besser als D3 [11] [9] [12].

3.4.2 Fazit

Dieser Prototyp hat bestätigt, dass ein `Custom Widget` sich eignet, um eine vorhandene Simulation in einem Jupyter Notebook im Browser zu visualisieren. D3 ist intuitiv anzuwenden und verfügt über die Funktionalitäten die zur Implementation benötigt werden. Aufgrund der Performanceeinschränkungen eignet sich D3 aber nur für eine Visualisierung mit einer nicht zu hohen Anzahl Agenten.

3.5 three.js Prototyp

Der three.js[19] Prototyp ist ein TypeScript Projekt[20] mit welchem die Möglichkeiten und Funktionalitäten von three.js untersucht wurden. Das Projekt verwendet esbuild[21] und npmjs[22]. Der Prototyp ist nicht in ein Jupyter Notebook eingebunden. Bewegungen von Agents werden in diesem Prototyp durch UI Elemente ausgelöst.

3.5.1 Implementierte Funktionalitäten

3.5.1.1 Spielfeld und Spielfeldbereiche visualisieren

Das Spielfeld wird als Rechteck mit einer Breite und Höhe definiert. Für die Spielfeldbereiche muss eine Position mit x / y Koordinaten, Breite und Länge angegeben werden.

[15] https://threejs.org/
[16] https://pixijs.com/
[17] https://github.com/d3fc/d3fc
[18] https://github.com/stardustjs/stardust
[19] https://threejs.org/
[20] https://gitlab.ost.ch/moritz.schiesser/sa_simpy_animationlib/-/tree/master/research/threejs_investigations
[21] https://esbuild.github.io/
[22] https://www.npmjs.com/

Abbildung 3.9: Visualisierung des Spielfelds und der Bereiche mit three.js

Die Eigenschaften eines Bereichs wie Name und Farbe werden entsprechend visualisiert.

3.5.1.2 Agenten visualisieren und animieren

In diesem Prototyp werden die Agenten als 3D-Model eines Hundes dargestellt. [13] Im Verlauf der Entwicklung wurden zuerst Kugeln zur Darstellung für die Agenten verwendet, als Kugeln können auf einer high-end Maschine 5000 Agenten gleichzeitig und flüssig animiert werden. Die Darstellung der Agenten als 3D-Model eines Hundes ist nicht performanceoptimiert und deshalb wurde keine hohe Anzahl an gleichzeitig animierten Agenten erreicht.

Die Agenten können mit dem Button `Move Agents` verschoben werden. Sobald sich die `location` eines Agenten während der Simulation verändert, würde diese Funktionen ausgelöst werden.

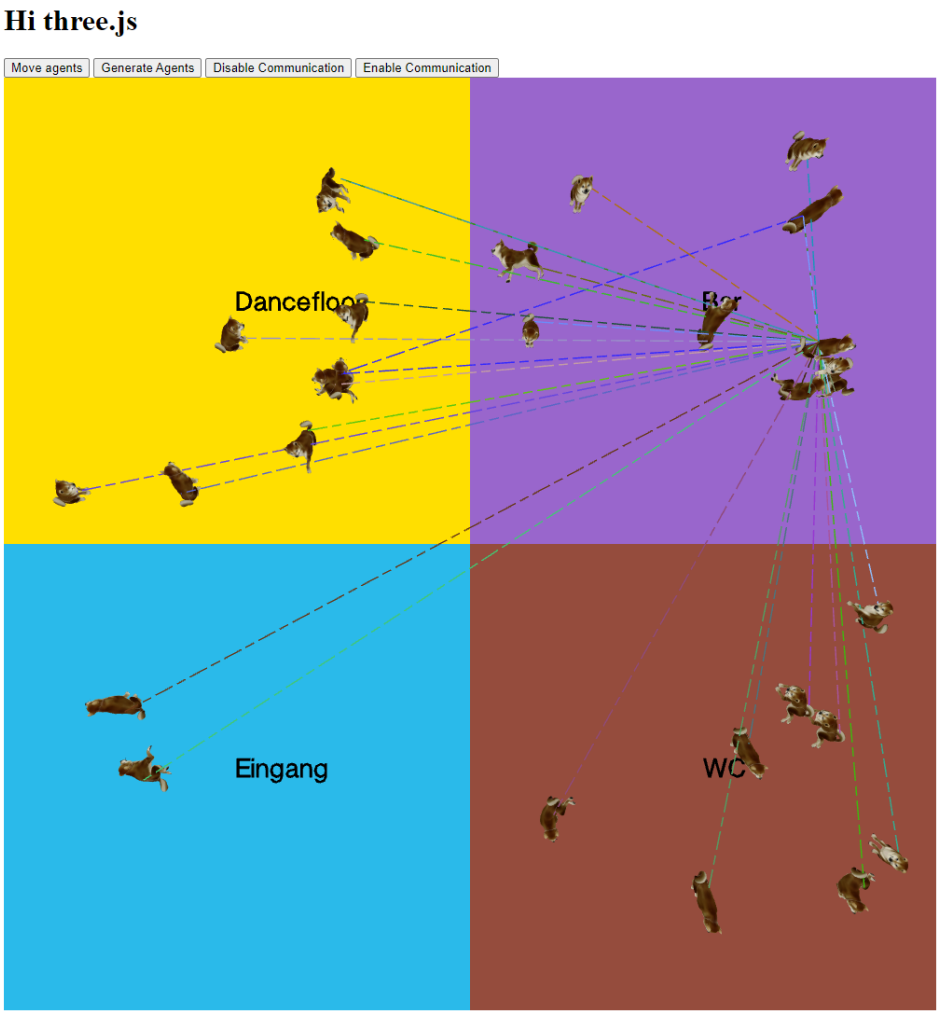

Abbildung 3.10: Visualisierung der Agenten mit three.js

Die Seitenansicht ermöglicht das Eintauchen in die Szene.

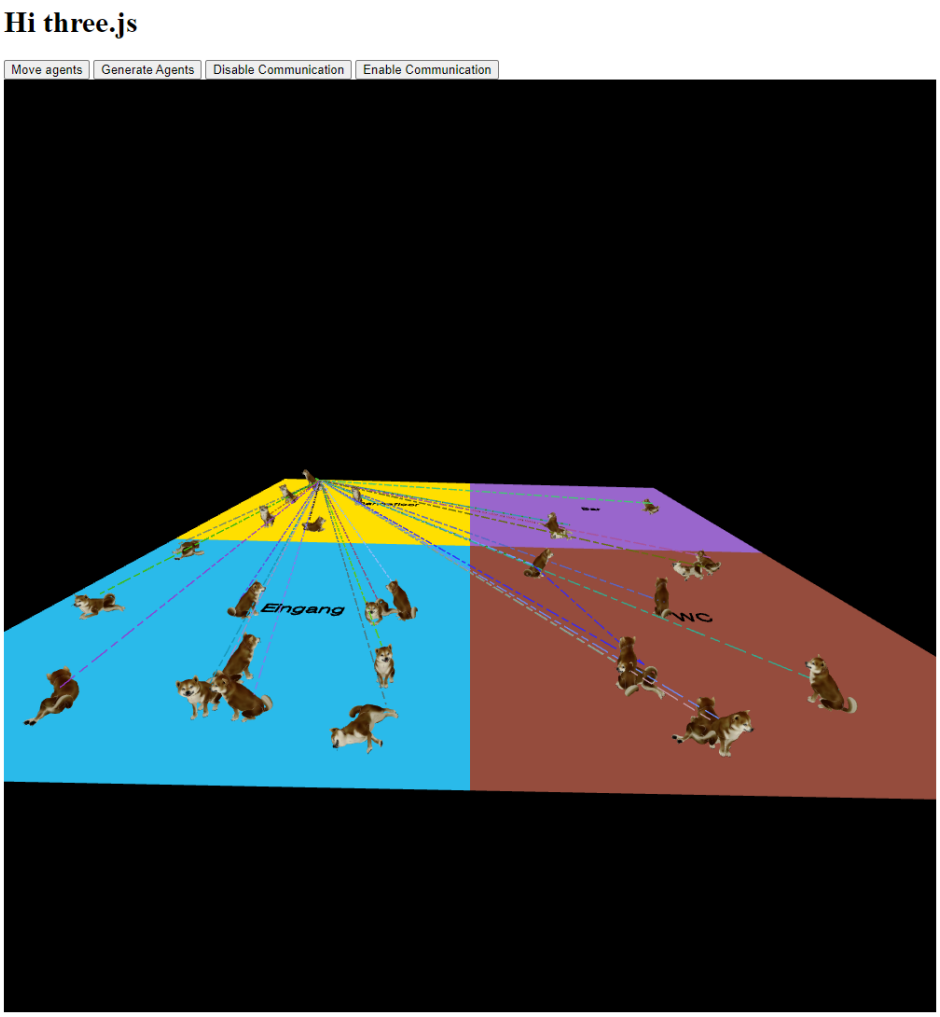

Abbildung 3.11: Visualisierung der Agenten in Seitenansicht mit three.js

Das 3D-Model verfügt über Animationen die abgespielt werden, solche Animationen können zum Abbilden des Zustands des Agenten verwendet werden. Die Kamera lässt sich in diesem Beispiel verschieben, was ermöglicht das Spielfeld aus verschiedenen Perspektiven zu betrachten. Zusätzlich kann auch zur Mitte gezoomt werden, um das Geschehen genauer zu betrachten.

3.5.1.3 Agenten Kommunikation visualisieren

Wie auf den Bildern oben ersichtlich, wird die Kommunikation zwischen Agenten als gestrichelte Linie dargestellt. Eine Besonderheit dieses Prototyps ist, dass sich die Linie fliessend in die Richtung des referenzierten Agenten bewegt. Falls beide Agenten sich gegenseitig referenzieren, werden 2 Linien erstellt, welche sich in die jeweilige Richtung bewegen. Im momentanen Prototypen überlappen sich diese Linien.

3.5.2 Fazit

Three.js ist für die 3D-Entwicklung ausgelegt, und entsprechend steil ist die Lernkurve. Um three.js professionell einzusetzen, müssen verschiedene 3D Konzepte wie Kameras, Szenen, renderLoop, Geometrien, Materialien erarbeitet werden. Die Visualisierung wird im 2D Raum dargestellt, das heisst ein Grossteil der Features von three.js wird nie benötigt.

Three.js wird interessant, falls die Visualisierung in 3D gemacht wird. Das führt aber dazu das ein Anwender

zusätzliche Konzepte kennen muss, dazu gehört unter anderem, dass für jedes Element neu die z-Achse definiert werden muss, dass Bereiche als 3D Modell, Würfel oder andere geometrische Form definiert werden, dass je nach Darstellung eine visuelle Verbindung zwischen Bereichen definiert werden muss.

Three.js ist eine geeignete, performante Alternative zu D3.

3.6 PixiJS Prototyp

Der PixiJS Prototyp ist ein eigenständiges TypeScript Projekt welches die Möglichkeiten und Fähigkeiten von PixiJS erforscht. Die animierten Daten werden dabei nicht von einer echten Simulation erstellt, sondern direkt in TypeScript synthetisiert oder definiert. Unter anderem werden dafür auch zufällige Agents erstellt. Der Prototyp ist nicht in ein Jupyter Notebook eingebunden. Bewegungen von Agents werden in diesem Prototyp durch UI Elemente ausgelöst.

PixiJS ist eine JavaScript Bibliothek welche die Vorteile und Leistungskapazitäten von WebGL nutzt. Damit erreicht PixiJS sehr gute Performance im Allgemeinen und hohe Werte in der Anzahl der gleichzeitig renderbaren Elemente. Auf leistungsstarker Hardware können damit über 200'000 Elemente gleichzeitig bei über 60 FPS dargestellt werden [14].

Der Einsatz von PixiJS stellt sicher, dass dieses Projekt nicht durch Leistungsengpässe der JavaScript Bibliothek beeinträchtigt wird. Die API welche PixiJS anbietet ist intuitiv und schnell verständlich. Unter anderem deshalb ist der Prototyp welcher mit PixiJS geschrieben wurde am weitesten fortgeschritten.

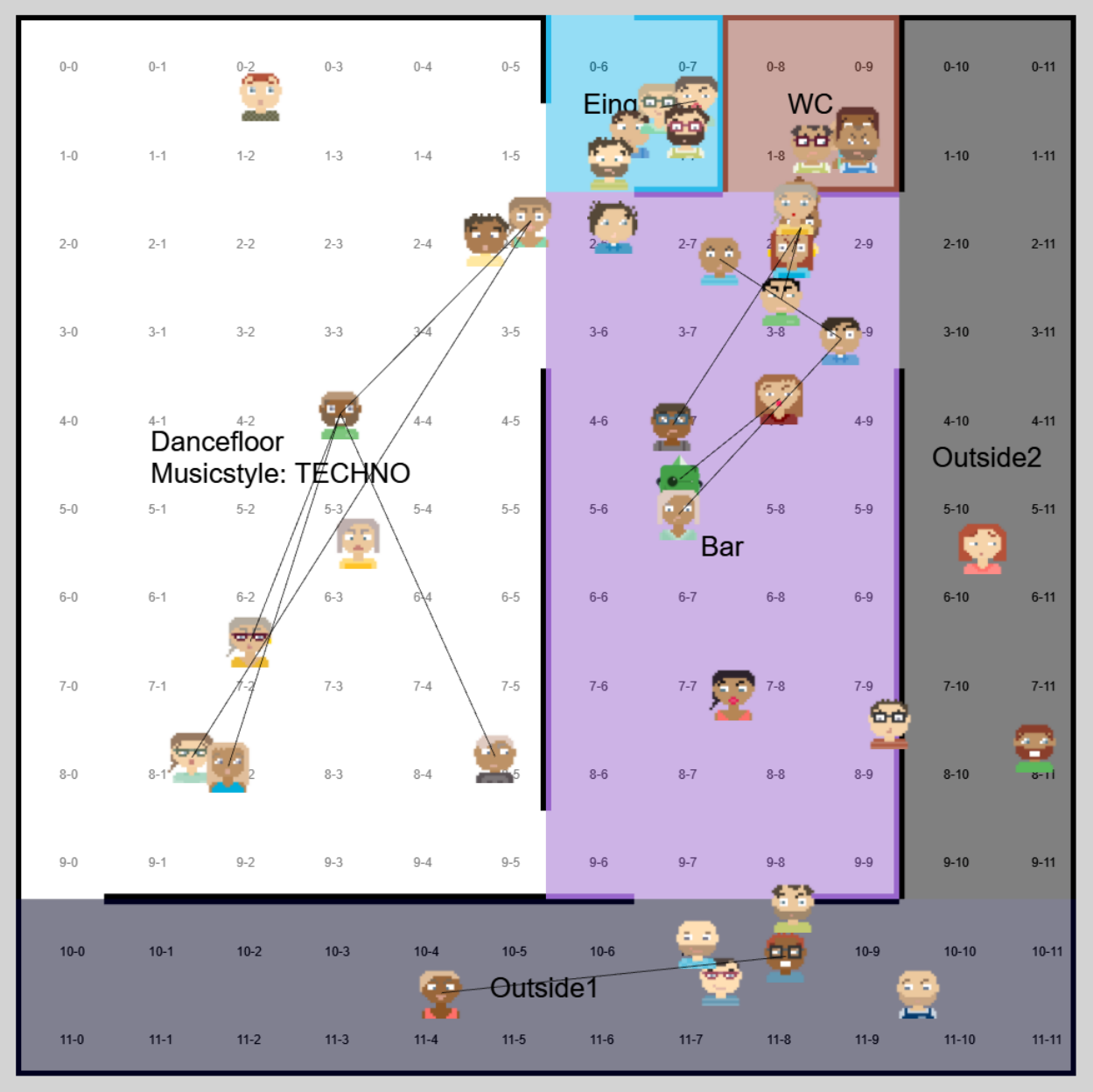

Abbildung 3.12: Screenshot des mit PixiJS entwickelten Prototypen

3.6.1 Implementierte Funktionalitäten

3.6.1.1 Wände und Wanddurchbrüche / Türen

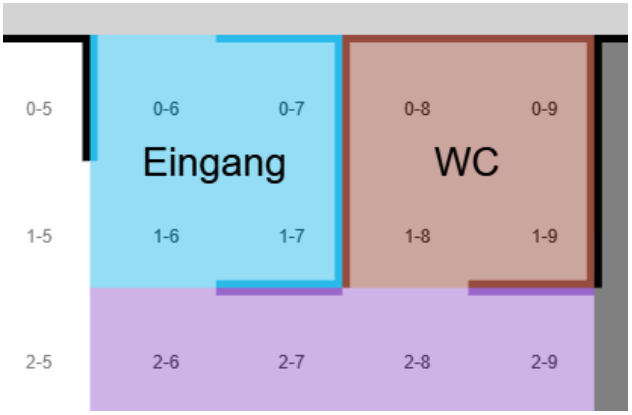

Abbildung 3.13: Wände und Wanddurchbrüche / Türen mit PixiJS

Der Prototyp kann aufgrund der definierten Durchgänge eines Bereichs implizit die Wände ebendieses erkennen und darstellen. Wände werden dabei als stärkere Farbvariation der Hintergrundfarbe dargestellt.

3.6.1.2 Eindeutige Avatare für alle Agents

In frühen Iterationen dieses Prototyps wurden Agenten als eingefärbte Kreise auf dem Spielfeld dargestellt. Die Aktionen und Bewegungen eines konkreten Agenten nachzuvollziehen, war dementsprechend schwierig. Um diesen Umstand zu verbessern, generiert der Prototyp Avatare, wobei der definierbare Name der Agenten als Seed für die Generierung verwendet wird. Diese Avatare werden mit der Bibliothek DiceBear[23] erstellt und können als SVG von PixiJS verarbeitet und dargestellt werden.

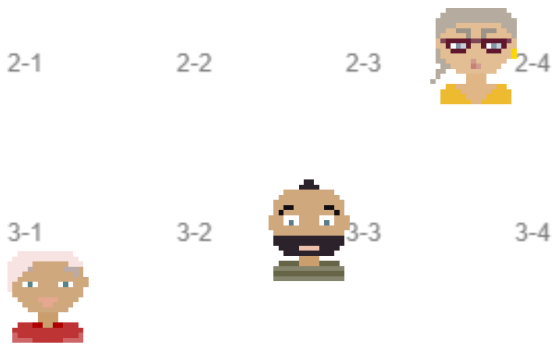

Abbildung 3.14: Eindeutige Avatare für Agents mit PixiJS

Im Rahmen dieses Prototyps wurden die Agents, ihr Geschlecht und ihre Namen zufällig generiert.

3.6.1.3 Panning und Zooming innerhalb des Playgrounds

Bereits bei einer mittelgrossen Anzahl von Agenten ist es teilweise so, dass die Agents sich auf kleinem Raum drängen. Um dabei trotzdem einen guten Überblick behalten zu können, wurde für diesen Prototyp

[23] https://avatars.dicebear.com/

die Möglichkeit des Vergrösserns (Zooming) und des Hin- und Herschiebens (Panning) bei Vergrösserung implementiert.

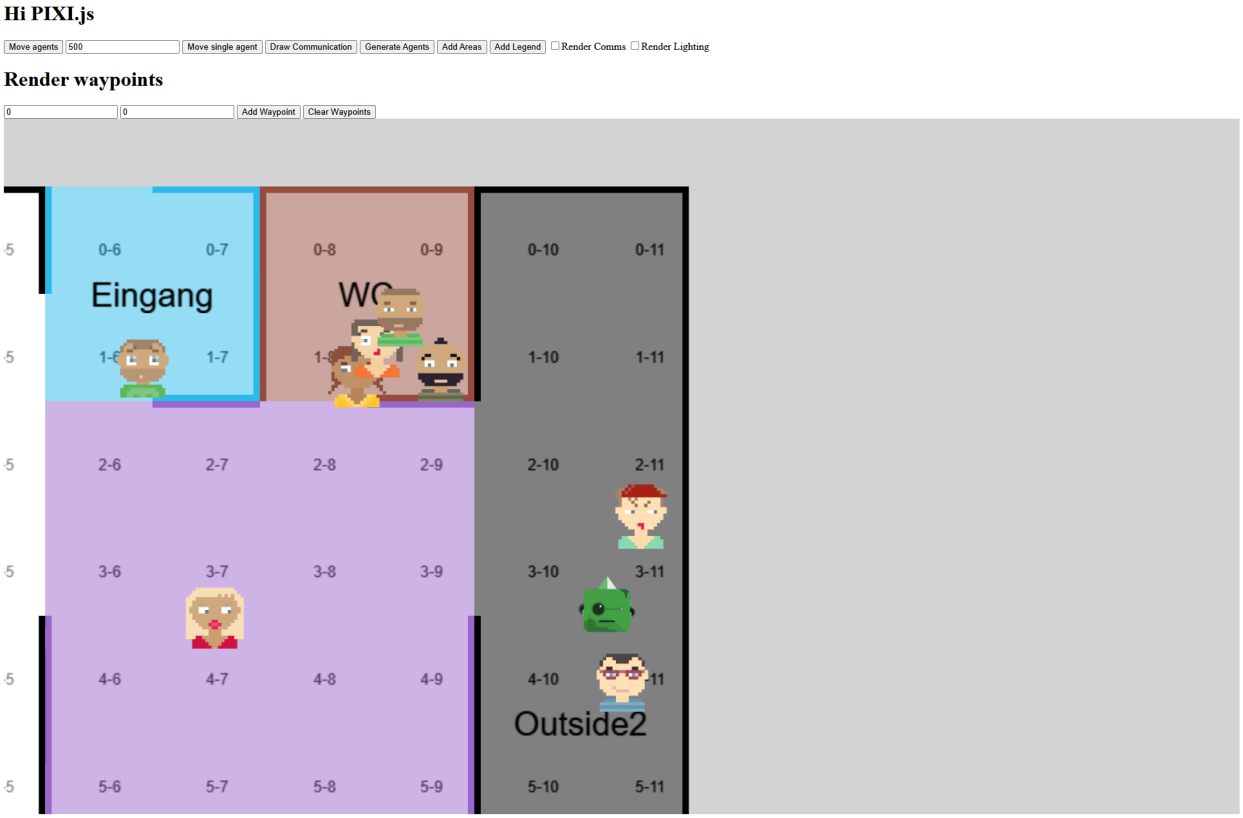

Abbildung 3.15: Vergrösserte Ansicht des Playground mit PixiJS

3.6.1.4 Verfolgen von Agents

Gegebenenfalls sind Betrachterinnen und Betrachter insbesondere an einem einzelnen, konkreten Agent interessiert. Dieser Prototyp unterstützt das Verfolgen von einzelnen Agents auf beliebiger Zoomstufe. Somit kann immer die unmittelbare Umgebung eines einzelnen Agents im Überblick behalten werden.

Abbildung 3.16: Tooltip eines Agents mit PixiJS

3.6.1.5 Kollisionsfreie Bewegung von Agents

Wenn sich Agents von einem Bereich in den anderen bewegen, respektieren sie die Wände und wählen einen Pfad, mit dem sie durch die Durchgänge in den nächsten Bereich gehen.

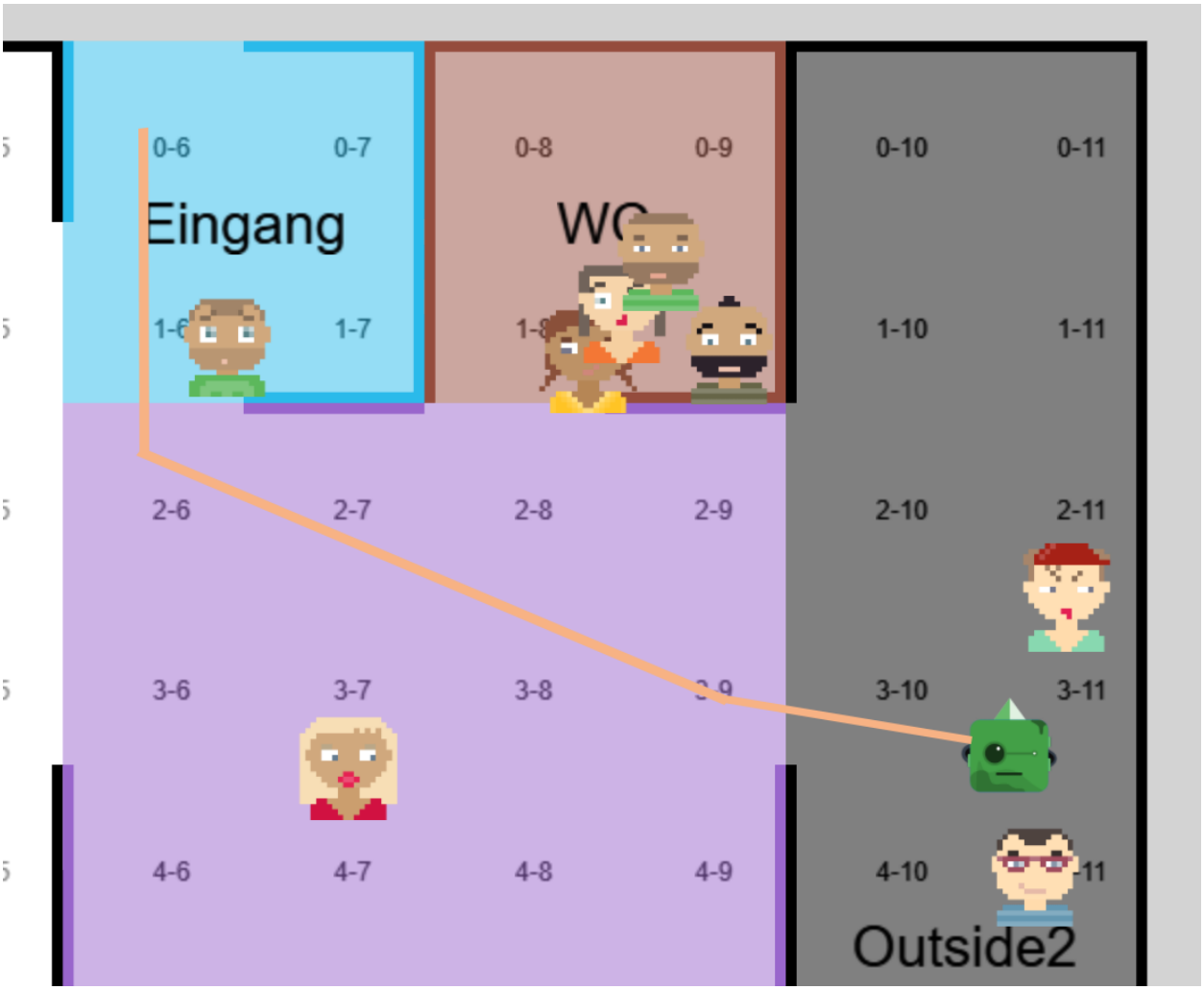

Abbildung 3.17: Pfadfindung des grünen Roboters in Beige mit PixiJS

3.6.1.6 Hohe Anzahl von Agents

Obwohl bei diesem Prototyp keine konkreten Performanceoptimierungen vorgenommen wurden, sind, können mehrere Tausend unterschiedlich aussehende Agenten dargestellt werden - zumindest ohne die Darstellung der Kommunikationslinien.

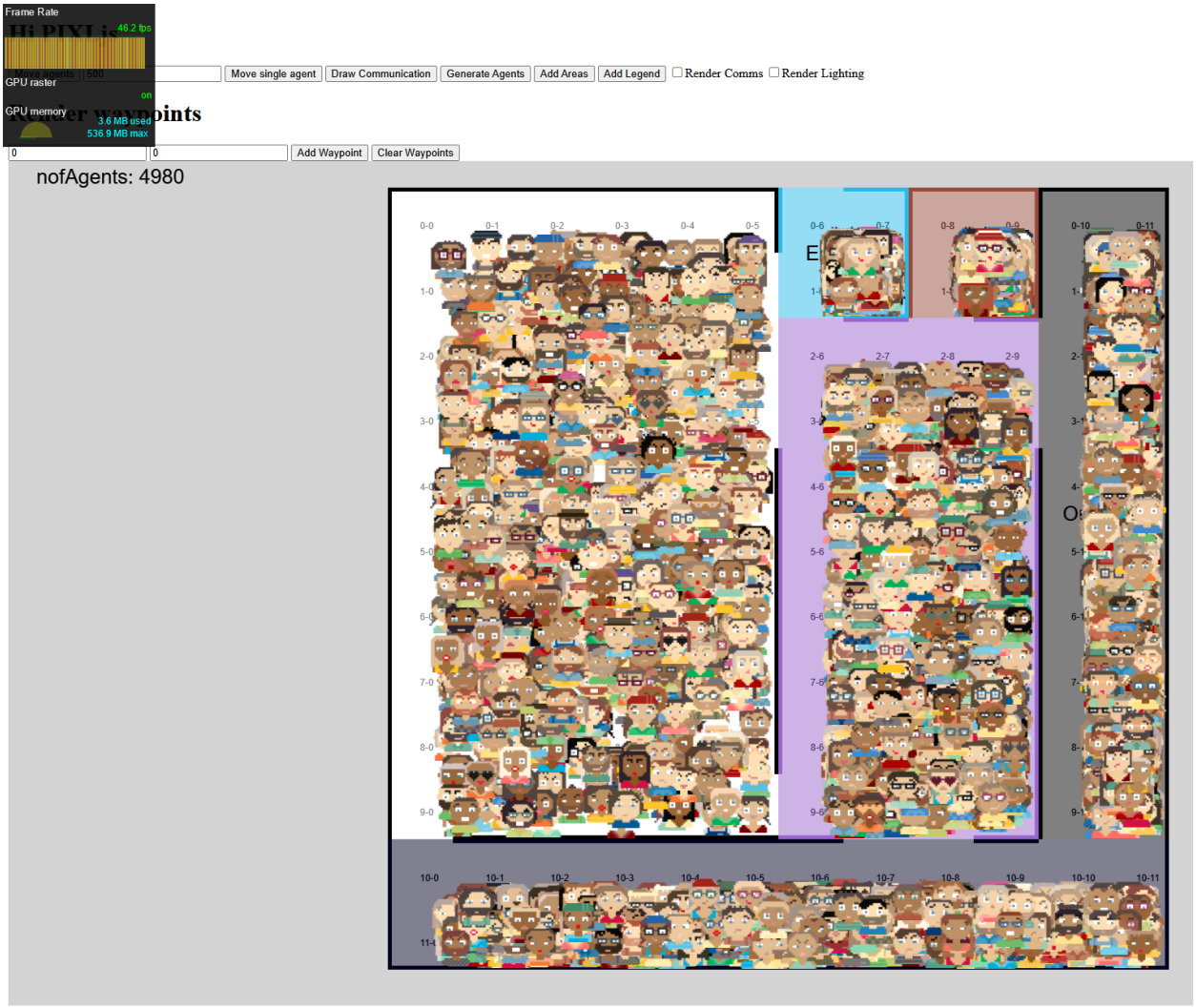

Abbildung 3.18: 4980 Agenten ohne Kommunikationslinien: 46 FPS mit PixiJS

Aktiviert man nun aber die Kommunikationslinien leidet die Performance.

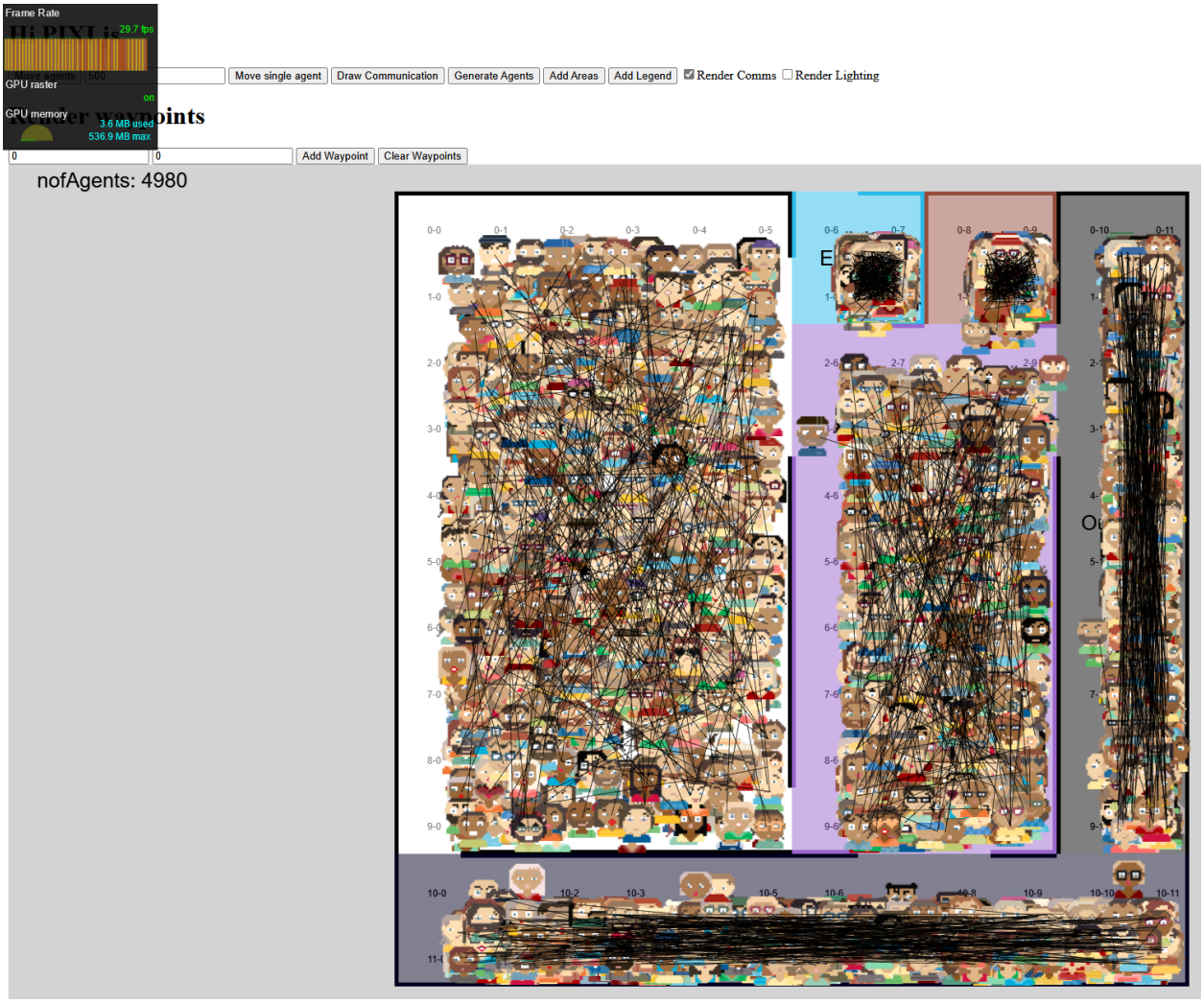

Abbildung 3.19: 4980 Agenten mit Kommunikationslinien: 29 FPS mit PixiJS

Dies liegt daran, dass die Kommunikationslinien die sich bewegenden Agents verfolgen. Diese “mitverfolgenden” Kommunikationslinien sind nicht zwingend eine Kernfunktionalität dieses Projekts. Zudem ist die Implementation der Linien sicherlich naiv. Eine reifere Implementation könnte hier Optimierungen vornehmen.

3.6.1.7 Zusätzliche, experimentelle Features

PixiJS hat ein grosses Ökosystem an zusätzlichen Filtern und Libraries. Um diese - oder ein Teil davon - kennenzulernen, wurde als experimentelles Features die Beleuchtung des Dancefloors implementiert. Auch diesen zusätzlichen grafischen Layer kann PixiJS problemlos darstellen.

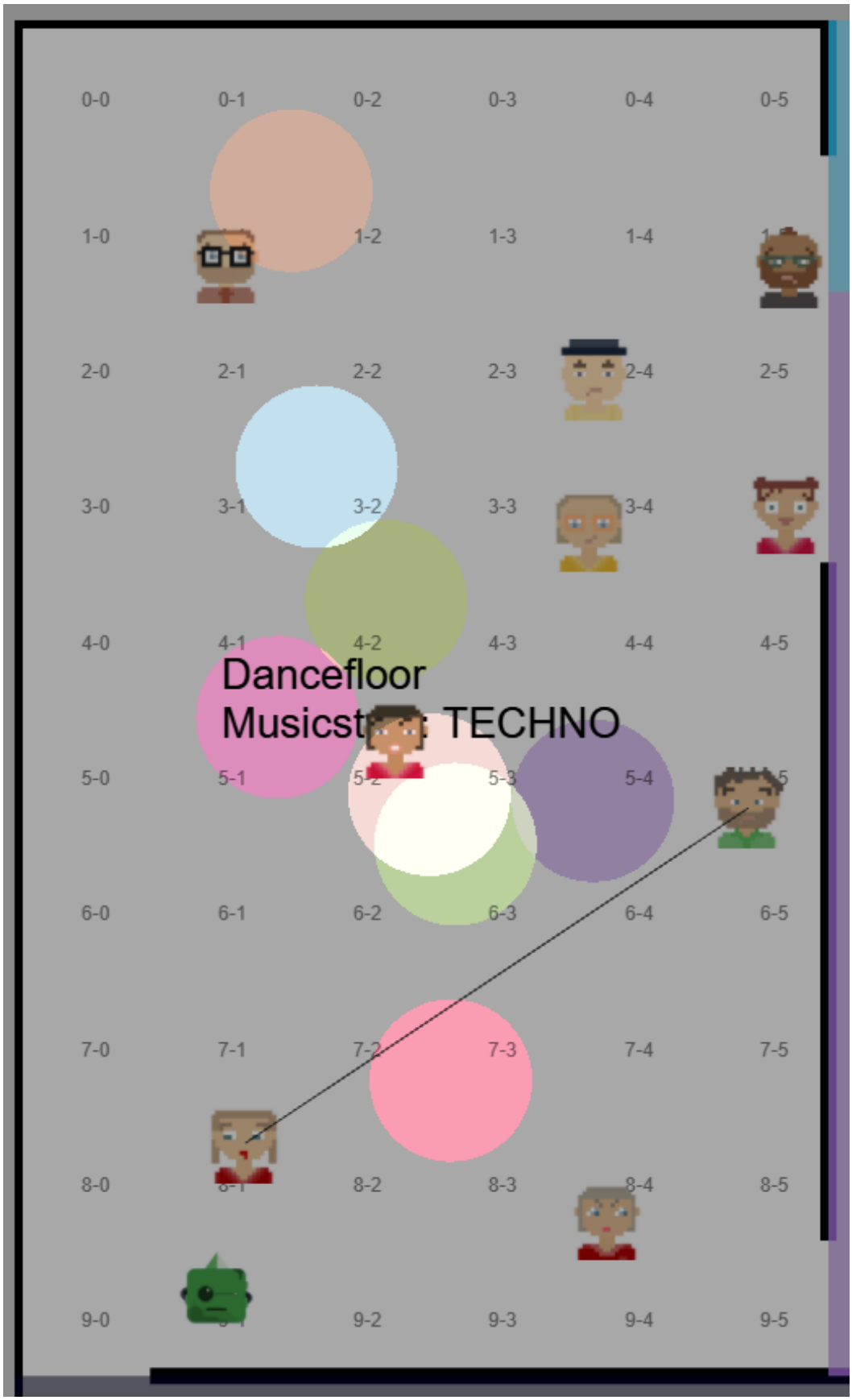

Abbildung 3.20: “Discobeleuchtung” auf dem Dancefloor mit PixiJS

Anmerkung: Dieses Feature ist keine Kernfunktionalität des Projekts und sehr spezifisch auf das Beispiel eines Nachtclubs bezogen

3.6.2 Fazit

PixiJS eignet sich für die Anforderungen dieses Projekts sehr gut. Die gut verständliche und konsequente API, die exzellente Leistung und leichte Integration der Bibliothek bieten eine grossartige Basis für die Fortführung dieses Projekts.

3.7 Erkenntnisse und Fazit

Die Prototypen haben bewiesen, dass die Anforderungen mit den gewählten Technologien umsetzbar sind. Weiter geben die Spezialitäten der jeweiligen Technologie, beispielsweise die Verwendung von 3D-Modellen, fliessenden Linien oder Beleuchtung des Dancefloors, einen Ausblick was mit der entsprechenden Technologie alles umgesetzt werden könnte.

Während der Entwicklung der Prototypen hat sich bestätigt, dass sich Animationen ideal mit der Verwendung eines `Custom Widget` von ipywidgets in JupyterLab integrieren lassen. Die Architektur wurde entscheidend durch die Prototypen beeinflusst und basiert auf den Erkenntnissen aus den Prototypen.

Im Verlauf der Entwicklung des ersten Prototyps wurde die Anforderung für eine hohe Anzahl darstellbarer Elemente gestellt. Deshalb wurden die Möglichkeiten zur Performanceoptimierung der Datenübertragung durch die JupyterLab Extension erforscht. Ebenfalls wurden zusätzliche Visualisierungstechnologien mit Hardwareunterstützung durch Grafikkarten untersucht. Dabei hat sich PixiJS als geeignet herausgestellt um die Anforderungen bezüglich Funktionalität und Performance umzusetzen.

Die Prototypen dienten oft als Inspirationsquelle und Diskussionsgrundlage für zusätzliche Funktionalitäten, beispielsweise sind die unterschiedlichen Arten und Zustände der Prototypen für die Visualisierung auf Feedback und Ideen aus der Arbeitsgruppe zurückzuführen. Die Prototypen haben dadurch nebst den Abklärungen von Möglichkeiten auch dazu beigetragen, dass die Arbeitsgruppe die tatsächlichen Anforderungen an die Animationsbibliothek besser versteht. Die besprochenen Ideen und Funktionalitäten sind im Kapitel Vision und Ausblick[24] zusammengefasst.

[24] 7 Vision und Ausblick

Kapitel 4

Architekturentwurf der Integration in JupyterLab

Die aus den Prototypen gewonnenen Erkenntnisse haben die Architektur und den Aufbau der Bibliothek massgebend beeinflusst. Die Architektur wurde mehrfach überarbeitet und hat sich durch das Ändern von Anforderungen oder der Visualisierungsbibliothek oft geändert.

4.1 Qualitätsziele

4.1.1 Trennung von Simulations- und Animationscode

Die Bibliothek wird mit der Prämisse entwickelt, dass Elemente und Code zur Animation nicht (oder zumindest möglichst wenig) Einfluss auf den Aufbau der Simulation haben. Dies hat unter anderem den Effekt, dass eine bestehende Simulation gut um eine Animation ergänzt werden kann.

4.1.2 Leistung

Simulationen mit bis zu 5000 Agenten werden auf einer high-end Maschine mit dedizierter Grafikkarte ohne merkbare Verzögerungen animiert. Das kann mit entsprechenden Tests überprüft werden, die entweder manuell oder automatisiert durchgeführt werden.

4.1.3 Wartbarkeit

Code & Architektur Qualität: Die Code & Architektur Qualität muss möglichst hoch sein, um Bugs wenn möglich zu vermeiden oder ohne Nebenwirkungen zu beheben. Diese hohe Qualität trägt auch dazu bei, die Einstiegshürde für weitere Contributors zu diesem Open-Source-Projekt zu verringern. Die Codequalität kann durch das Definieren von Guidelines, automatisierten Tests und mit einer statischen Codeanalyse gewährleistet werden. Die Architektur wird im Team und mit den Betreuern besprochen.

Modularer Aufbau & Modifizierbarkeit: Die Bibliothek ist möglichst modular aufgebaut, sodass Teile ohne zusätzlichen Aufwand ausgetauscht und angepasst werden können. Die Modularität trägt dazu bei, dass Erweiterungen und Anpassungen mit wenig zusätzlichem Aufwand vorgenommen werden können. Das muss entsprechend in der Architektur berücksichtigt werden. Damit dies eingehalten wird, wird die Architektur im Team besprochen.

4.1.4 Benutzerfreundlichkeit

Die Bibliothek sollte so benutzerfreundlich sein, dass die Einstiegshürde möglichst klein ist. Vollständige und strukturierte Benutzerinstruktionen tragen entscheidend dazu bei. Für jedes Major Release müssen

entsprechende Instruktionen vorhanden sein. Des Weiteren ist es wichtig, das für mögliche Konfigurationen sinnvolle Standarddefinitionen vorhanden sind, sodass nicht erfahrene Benutzer sich auf diese Standarddefinitionen verlassen können. Für erfahrene Benutzer muss eine Möglichkeit bestehen die Standarddefinitionen anzupassen.

4.1.5 Zuverlässigkeit

Neue Versionen der Bibliothek sollen, wenn möglich rückwärts kompatibel sein. Breaking Changes müssen als Teil der Benutzerinstruktionen dokumentiert sein und wo nötig existieren Migrationsanleitungen. Damit wird gewährleistet, dass Anwender der Bibliothek möglichst reibungslos auf neue Versionen wechseln können.

4.2 Architektur der JupyterLab Extension

Die Prototypen haben eindeutig aufgezeigt, dass die Animationsbibliothek als JupyterLab Extension in Jupyter integriert werden soll. Die JupyterLab Extension heisst `JupyterSimulationPlayground` und ist eine Widget-Bibliothek.

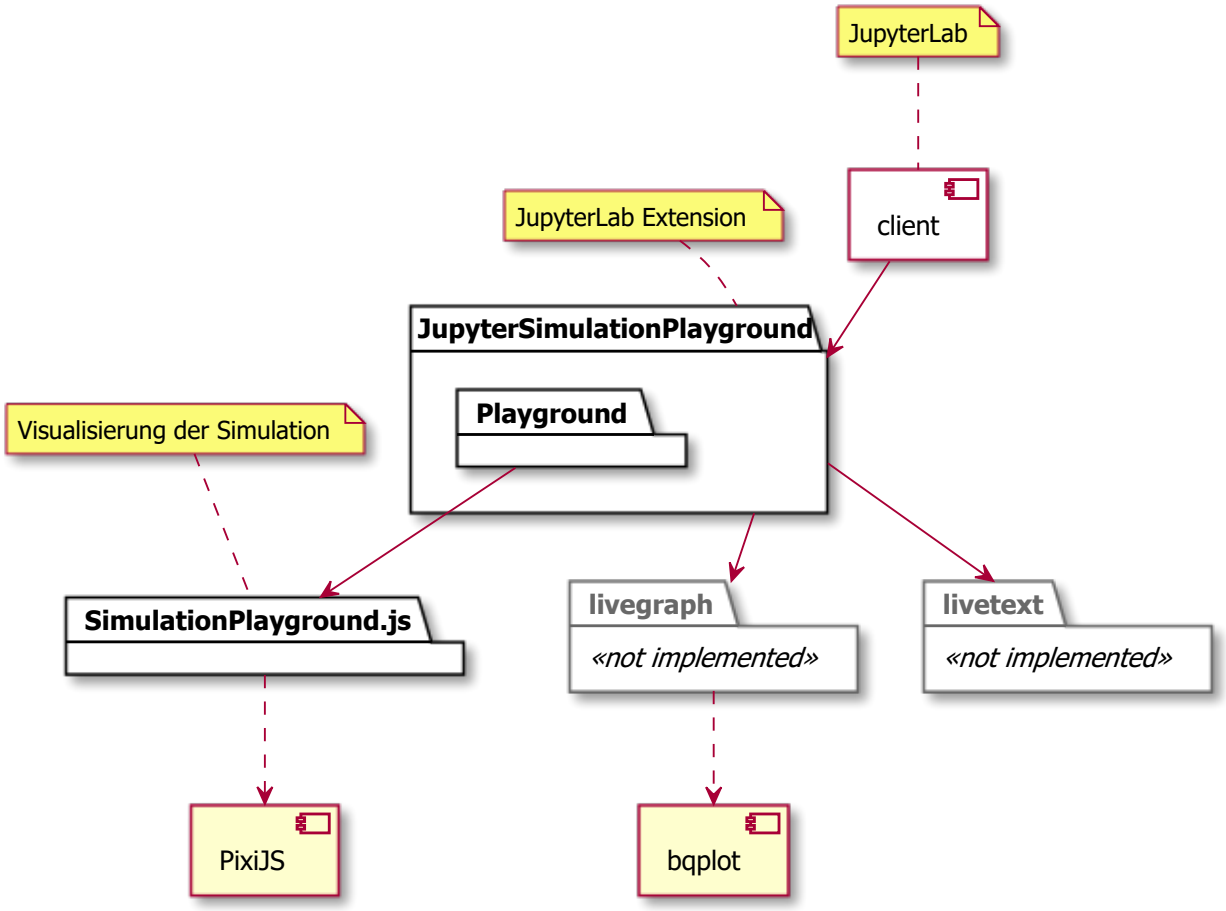

Abbildung 4.1: Architekturübersicht der JupyterLab Extension

`JupyterSimulationPlayground` beinhaltet das Widget `Playground`, welches mit der Verwendung von `SimulationPlayground.js` eine Simulation visualisiert. `SimulationPlayground.js` könnte auch von anderen Simulationsbibliotheken zur Animation verwendet werden, beispielsweise von einer SimPy Alternative die

in JavaScript implementiert ist.

Für die Animation einer Simulation sind folgende Komponenten zu visualisieren:

- **Spielfeld** (`Playground`): Innerhalb des Spielfelds wird die Simulation dargestellt.
- **Area**: Jedes Spielfeld besteht aus verschiedenen Bereichen mit unterschiedlichen Eigenschaften.
- **Agent**: Ein Agent hält sich innerhalb des Spielfelds auf und ändert im Verlauf der Simulation seine Eigenschaften und auch seinen Zustand.
- **Ressourcen**: Ressourcen haben eine begrenzte Kapazität und können von Agenten in Anspruch genommen werden. Falls die Kapazität ausgelastet ist, kann es sein, dass Agenten warten bis die Ressource freigegeben wird.

Diese Strukturen und deren Eigenschaften lassen sich wie folgt zusammenfassen:

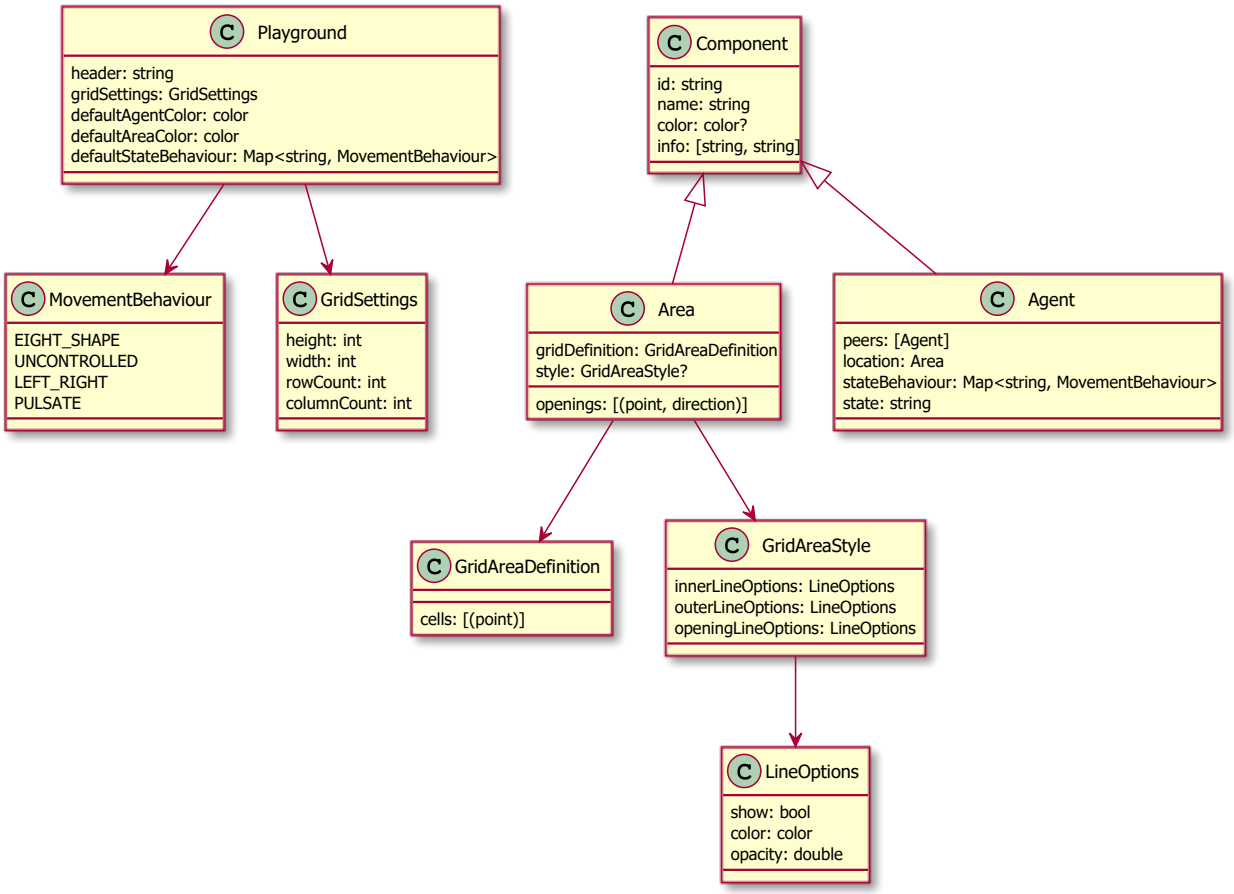

Abbildung 4.2: Benötigte Strukturen zur Visualisierung

Die Strukturen sind die Grundlage für die Animation, welche in folgenden Fällen stattfindet:

- `peers` eines Agenten ändern sich.
  - Kommunikation ist bidirektional, wenn beide sich gegenseitig als Peer referenzieren
  - Kommunikation ist unidirektional wenn nur der Versender den Empfänger als Peer referenziert, aber nicht umgekehrt.
- `location` eines Agenten ändert sich.

- `state` eines Agenten ändert sich.

Einer Änderung von `state` ist ein Spezialfall, es kann für die verschiedenen Werte von `state` eine von der Bibliothek zur Verfügung gestellte Animation definiert werden, welche beim Eintreten des entsprechenden `state` ausgeführt wird. Dazu muss die Eigenschaft `defaultStateBehaviour` des `Playground` definiert werden. Individuelles Verhalten für einen `Agent` kann mit dessen `stateBehaviour` Eigenschaft definiert werden.

4.2.1 JupyterSimulationPlayground & Playground

Dieser Abschnitt beschreibt die Architektur der `JupyterSimulationPlayground` und der `Playground` Komponenten, welche eine bestehende Simulation mit einer Animation integriert.

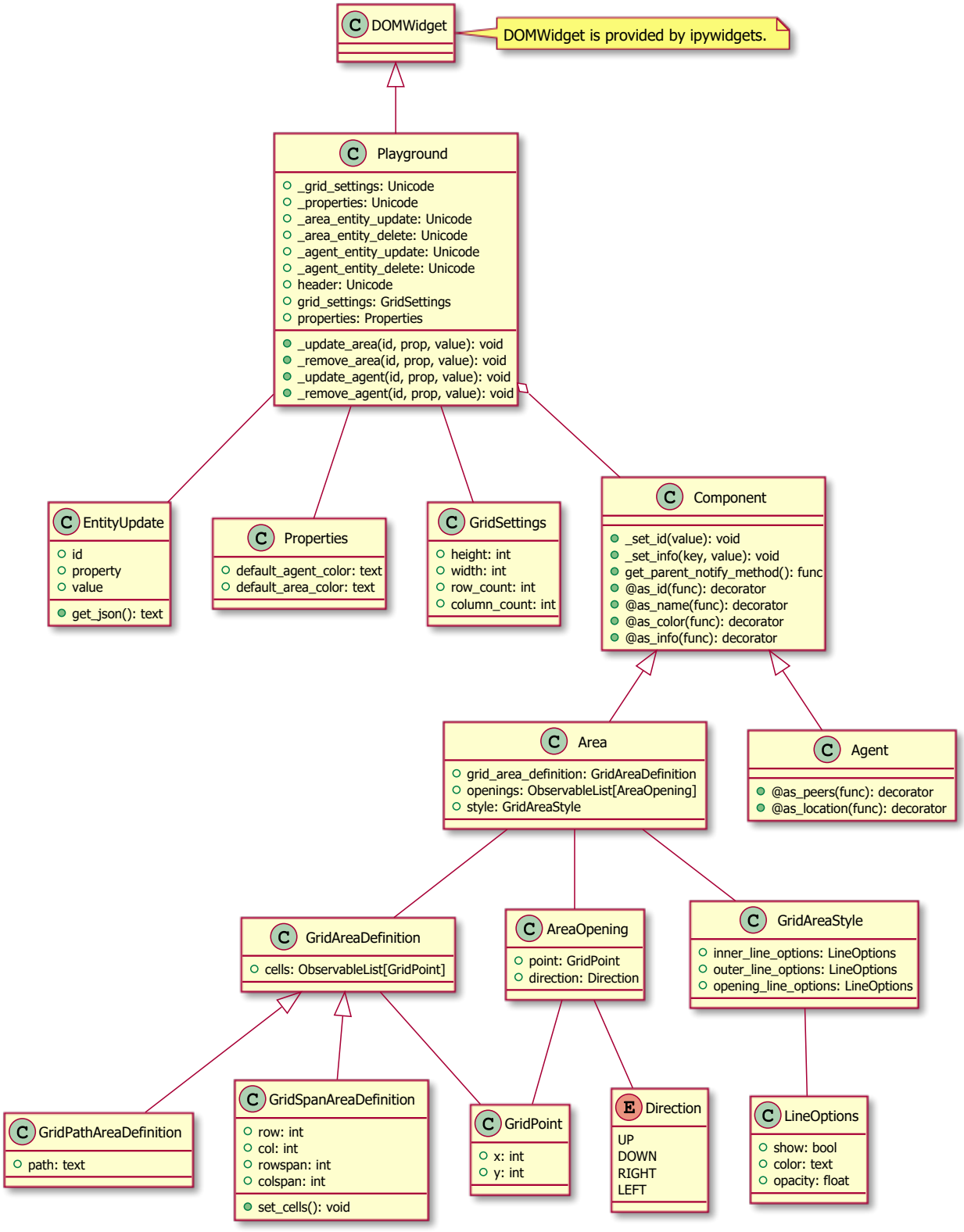

Abbildung 4.3: Übersicht über die Pythonstrukturen

4.2.1.1 Verwendung von Decorators

In der obigen Abbildung haben die Klassen `Component` und `Agent` öffentliche Funktionen welche mit einem `@` beginnen. Dieses `@` markiert, dass diese Funktionen einen `Decorator`[1] darstellen. Als Parameter nehmen Decorators eine Funktion entgegen und dekorieren diese. Folgendes Codesnippet verschafft einen Eindruck für die Verwendung:

```
@decorator
def decorated_function():
  pass
```

Dieses Dekorieren von Funktionen hat zur Folge, dass Aufrufe von `decorated_function` zur `decorator` Funktion umgeleitet werden. Funktionen welche als `decorator` funktionieren geben eine Funktion zurück.

Ein simples Beispiel ist wie folgt:

Ein Decorator `log_func_call` gibt jeweils den Namen einer Funktion aus bevor sie aufgerufen wird und nach dem Aufruf der Funktion gibt der Decorator aus, dass die Funktion aufgerufen wurde.

```
from functools import wraps


def log_func_call(func):
    @wraps(func)
    def func_wrapper(*args, **kwargs):
        print(f"'{func.__name__}' is going to be called")
        func_result = func(*args, **kwargs)
        print(f"'{func.__name__}' has been called")
        return func_result

    return func_wrapper


@log_func_call
def print_number(num):
    print(num)


print_number(42)
```

Output:

```
> 'print_number' is going to be called
> 42
> 'print_number' has been called
```

Der Einsatz dieses Konzept ermöglicht es, dass Klassen, welche für die Simulationslogik relevant sind, sehr leicht um Animationskapazitäten erweitert werden können. Gegeben sei folgende (stark vereinfachte) Klasse für die Abbildung von Menschen in einer Simulation:

```
class Human:
  @property
  def name(self):
    return self._name

  @name.setter
```

[1] https://realpython.com/primer-on-python-decorators/

```
  def name(self, value):
    self._name = value
```

Um diese Klasse `Human` nun fähig für Animationen zu machen, wird sie wie folgt abgeändert.

```
class Human(Agent):
  def __init__(self, parent):
    Agent.__init__(self, parent)

  @property
  def name(self):
    return self._name

  @name.setter
  @Agent.as_name
  def name(self, value):
    self._name = value
```

Die Klasse `Human` erbt von der definierten Basisklasse `Agent`, dass ermöglicht die Verwendung der Decorators welche von `Agent` bereitgestellt werden. Die Eigenschaft `name` wird nun um den Decorator `@Agent.as_name` ergänzt.

Bei der Ausführung des folgenden Codes

```
parent = Playground()

jane = Human(parent)
jane.name = 'Jane Doe'

john = Human(parent)
john.name = 'John Doe'
```

wird die Eigenschaft `name` der jeweiligen Instanzen von `Human` gesetzt. Aufgrund des Decorators `@Agent.as_name` wird zusätzlich weiterer Code der Basisklasse `Agent` ausgeführt, der dafür sorgt, dass der `parent` dieses Simulationsobjekts über diese Änderung informiert wird. Der `parent` ist eine Instanz der Klasse `Playground`. Diese wird nun wiederum ihr eigenes, mit dem Webfrontend synchronisiertes Property `_agent_entity_update` setzen. Somit wird also der Animationslogik im Webfrontend mitgeteilt, dass für die beiden Agenten `jane` und `john` das `name` Property geändert wurde. Anwender dieser Bibliothek müssen sich entsprechend nicht darum kümmern manuell Updates an die Animationslogik zu senden, lediglich muss markiert werden welche Eigenschaften der selbst definierten Klassen zu welchen Eigenschaften eines Animationsobjekts gehören.

4.2.1.2 Playground

Das `Playground` Widget ist eine Erweiterung des `DOMWidget`. Diese Basisklasse wird von `ipywidgets` zur Verfügung gestellt. Die Klasse `Playground` ist die zentrale Komponente, welche die eigentliche Änderungsübermittlung zum Animationsfrontend vornimmt. Properties welche das `sync=true` Tag (bereitgestellt von `Traitlets`) tragen, werden bei jeder Änderung über die Kommunikationskanäle von JupyterLab ans Webfrontend übertragen. Bei der `Playground` Klasse sind das:

- `_grid_settings`
- `_properties`
- `_area_entity_update`
- `_area_entity_delete`
- `_agent_entity_update`
- `_agent_entity_delete`

Einen Überblick über die Übermittlung von Nachrichten aus dem Pythoncode in das Webfrontend ist im Kapitel Performance[2] vorhanden.

4.2.1.3 Component: Area und Agent

Die Klassen `Area` und `Agent` welche von `Component` erben, definieren grundlegende Strukturen welche als Basis für Simulationskomponenten verwendet werden. Die von den Klassen zur Verfügung gestellten Decorators bilden alle Eigenschaften ab, welche nötig sind um ein Element in einer Animation darstellen zu können.

4.2.1.4 Area: weiterführende Erklärungen

Die `Area` Klasse definiert Bereiche innerhalb einer Animation. Bereiche sind Orte, also `location`s an denen `Agent`s sich aufhalten können.

4.2.1.5 Definition der Bereichsgrenzen

Zur Definition der Bereichsgrenzen, oder der durch die Bereiche abgedeckten Fläche stehen 2 verschiedene Varianten zur Verfügung.

`GridSpanAreaDefinition`

Diese Klasse erlaubt die Definition eines Ankerpunkts (`row`, `col`), und die Definition von Spannweiten (`rowspan`, `colspan`). Damit können Rechteckige Bereiche von `Area`s definiert werden.

`GridPathAreaDefinition`

Diese Klasse erlaubt die Definition eines Pfades welcher die Grenzen einer Area abbildet. Damit können komplexere Formen definiert werden.

Folgender Pfad wäre ein Beispiel:

```
1 path = '0,0 10,0 10,15 5,15 5,5 0,5 0,0'
```

4.2.1.5.1 Definition des Wandaussehens

Mit der Klasse `LineOptions` kann das Aussehen der Wände und Türen, und Rasterlinien definiert werden. Dabei sind Konfigurationsoptionen für Farbe, Breite und Opazität verfügbar.

4.2.2 SimulationPlayground.js

Diese Komponente ist zuständig für die Visualisierung der Animation im Browser und dementsprechend die Integration ins Document Object Model des Browsers.

[2] 3.3.1 Performance

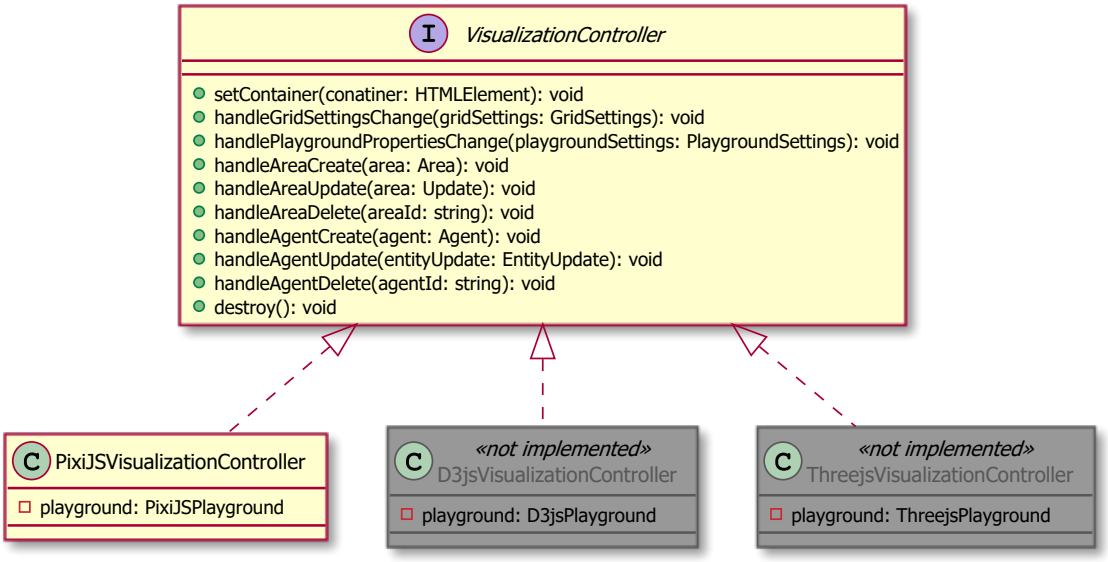

Abbildung 4.4: Architekturübersicht SimulationPlayground.js

Der `VisualizationController` ist das externe Interface dieser Komponente und wird beispielsweise vom `Playground` Widget verwendet. Dadurch ist sichergestellt, dass der `VisualizationController` mit geringem Aufwand ausgetauscht werden kann. Auf der Abbildung wird das verdeutlicht durch die zusätzlichen, noch nicht implementierten Controller.

Eine Implementation von `VisualizationController` verwendet einen `Playground` welcher Funktionen zur Visualisierung und Animation basierend auf der entsprechenden Visualisierungsbibliothek bereitstellt.

Im Abschnitt Erkenntnisse und Fazit[3] wird PixiJS als Visualisierungsbibliothek empfohlen. Im nächsten Abschnitt ist ein entsprechender Architekturvorschlag für `PixiJSPlayground` beschrieben.

4.2.2.1 Architekturvorschlag für PixiJSPlayground

Dieser Abschnitt beschreibt Ideen und Inspiration für eine Implementierung von `PixiJSPlayground` mit PixiJS. Dieser Architekturvorschlag soll dabei die Grundlage für eine iterative Weiterentwicklung der Architektur bilden. Die im Vorschlag enthaltenen Konzepte und Strukturen sind den bei der Implementation des PixiJS Prototypen gesammelten Erfahrungen entsprungen.

Entscheidend ist bei dieser Architektur vor allem, dass eine starke Trennung der Verantwortlichkeiten der einzelnen Komponenten im Kern der Architektur vorhanden ist. So sind die unten beschriebenen Komponenten ausschliesslich dafür zuständig, Eigenschaften und die Änderungen ebendieser in sichtbare UI-Elemente zu verwandeln. Die Definition und Manipulation der Eigenschaften geschieht dabei durch den `PixiJSVisualizationController`.

[3] 3.7 Erkenntnisse und Fazit

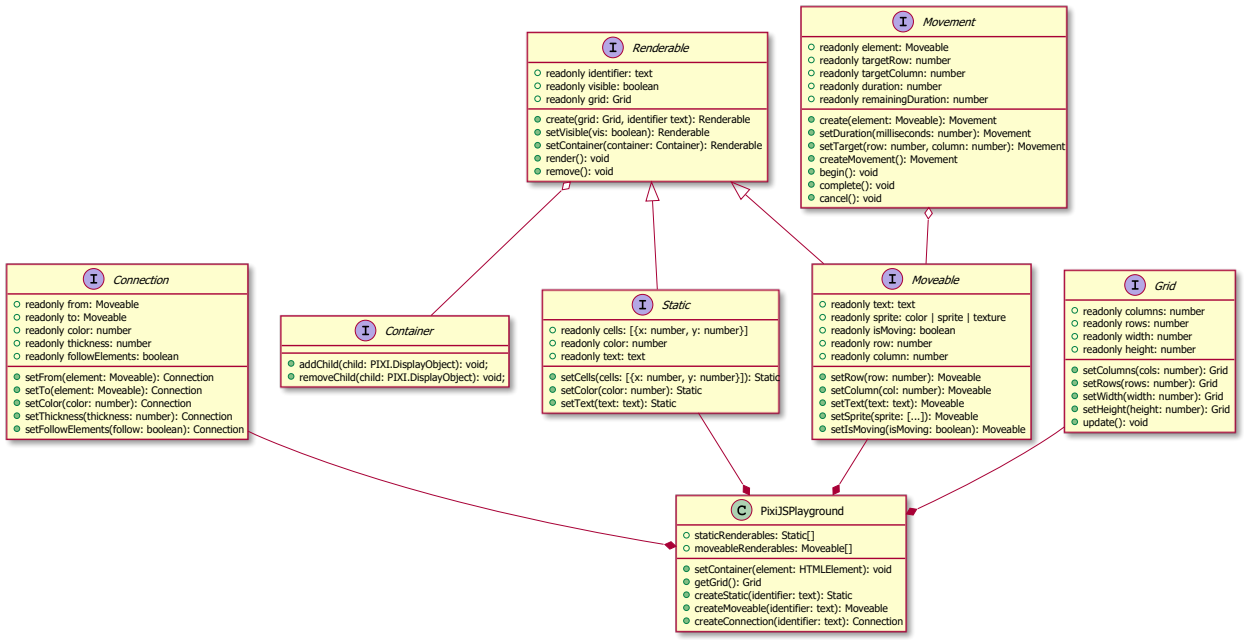

Abbildung 4.5: Architekturvorschlag PixiJS

Anmerkung: Die in der Abbildung beschriebenen Interfaces sind TypeScript Interfaces.

Dieser Architekturvorschlag sieht ein Fluent-API Design für die Visualisierungskomponente mit PixiJS vor. Der vom `PixiJSVisualizationcontroller` kontrollierte `PixiJSPlayground` würde dabei das Erstellen und Zuordnen von verschiedenen `Renderable` Typen verwalten.

Folgender Code zeigt, wie die Komponente `PixiJSPlayground` im `PixiJSVisualizationcontroller` verwendet werden würde:

```
const playground = new PixiJSPlayground();
playground.getGrid()
          .setRows(gridSettings.rowCount)
          .setColumns(gridSettings.columnCount)
          .setWidth(gridSettings.width)
          .setHeight(gridSettings.height)
          .update();
```

Die Manipulation von `Renderables` würde ebenfalls durch den `PixiJSVisualizationcontroller` geschehen. Nach Änderungen an Position oder Aussehen, würden ebendiese Änderungen durch den Aufruf von `render()` abgeschlossen, wobei dann eine spezifisch implementierte Komponente diese Datenänderungen in Änderungen auf dem mit PixiJS kreierten UI umsetzen würde.

Folgender Code zeigt dies Beispielhaft für ein `Static` Objekt:

```
const areaStatic = playground.createStatic(`${area.id}static`);
areaStatic.setVisible(true)
          .setCells(area.gridDefinition.cells)
          .setText(area.name)
          .setColor(area.color)
          .render();
```

`Moveable` erbt von `Renderable` und zeichnet sich dadurch aus, dass es beweglich ist, also dass ein `Movement` Objekt erstellt werden kann, das sich auf das `Moveable` bezieht.

`Connection` `Renderable`s sind jeweils abhängig von 2 `Moveable`s und halten keine Positionsdaten oder Dimensionsdaten. Mit den `Connection` Objekten können beispielsweise Kommunikationsverbindungen zwischen Agenten dargestellt werden.

Diese Strukturen sind ausreichend um die Basiselemente einer Simulationsanimation abzubilden.

Kapitel 5

Implementationsstand des Architekturprototyps

Analog zur Architektur wurde ein Architekturprototyp erstellt. Dieser Prototyp beansprucht nicht vollständig zu sein oder die gesamte Funktionalität abzudecken, aber er dient als Grundlage für die Implementation und zeigt auf, wie das Einhalten vereinzelter Qualitätsziele gewährleistet wird. Der Prototyp basiert auf dem Cookiecutter Template widget-ts-cookiecutter[1] welches von Jupyter Widgets zur Verfügung gestellt wird.

Die einzelnen Komponenten sind anhand der Architektur[2] erstellt worden. Momentan ist die Aufteilung in `JupyterSimulationPlayground` und `SimulationPlayground.js` nur theoretisch umgesetzt, effektiv befindet sich die `SimulationPlayground.js` Komponente in der `Playground` Komponente.

5.1 JupyterSimulationPlayground

`JupyterSimulationPlayground` ist eine JupyterLab Extension die das `Custom Widget Playground` zur Verfügung stellt. Diese Extension stellt die Funktionalität zur Verfügung, um eine Simulation gemäss Architektur mit einer Animation zu verbinden. Die Vorschläge zur Performanceoptimierung[3] sind noch nicht implementiert, die bestehende Lösung kann aber um die gewählte Variante erweitert werden.

Die Komponente zeigt beispielhaft auf wie Unit, Integration und UI Tests implementiert werden. Zusätzlich wird aufgezeigt wie für die gesamte Jupyter Extension Continuous Integration umgesetzt werden kann, durch das automatisierte Durchführen von Tests, Linting und Builds.

Es existiert Dokumentation zur Installation der JupyterLab Extension[4] sowie für die Entwicklung an der Extension. Es existieren keine Benutzerinstruktionen, aber eine Vorlage aus dem Cookiecutter Template, in welcher die Benutzerinstruktionen erstellt werden können.

5.2 Playground

`Playground` ist ein `Custom Widget` basierend auf ipywidget. Das Widget erbt von der bereitgestellten `DOMWidget`[5] Klasse, dadurch kann die Kommunikation zwischen Python und TypeScript wie in der Architektur beschrieben umgesetzt werden. Das `Playground` Widget benutzt die `SimulationPlayground.js` Komponente zum Darstellen eines SVG.

[1] https://github.com/jupyter-widgets/widget-ts-cookiecutter
[2] 4.2 Architektur der JupyterLab Extension
[3] 3.3.1 Performance
[4] https://gitlab.ost.ch/moritz.schiesser/sa_simpy_animationlib/-/blob/master/JupyterSimulationPlayground/README.md
[5] https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Custom.html#DOMWidget-and-Widget

Der TypeScript Teil des Widgets verwendet aktuelle Dependencies, verfügt über eine Funktionalität zum Linten des Codes, demonstriert die Verwendung von Git Hooks und zeigt beispielhaft auf, wie Unit und Integration Tests implementiert werden.

5.3 SimulationPlayground.js

Die `SimulationPlaground.js` Komponente ist der am wenigsten fortgeschrittene Teil dieses Prototyps. Momentan wird für die Visualisierung noch D3 verwendet und es ist nur möglich ein simples SVG zu generieren auf dem noch keine Simulation dargestellt wird. Die in der Architektur definierte Schnittstelle `VisualizationController` wird noch nicht zur Verfügung gestellt.

Es existieren Unit Tests, welche beispielhaft aufzeigen, wie Tests bei der Implementation implementiert werden können.

5.4 Weiters Vorgehen

Der Prototyp bildet eine Grundlage, auf der die Implementation durchgeführt werden kann. Die Komponenten `JupyterSimulationPlayground` und `Playground` bilden die beschriebene Architektur ab und Erweiterungen können vorgenommen werden. Die `SimulationPlayground.js` Komponente basiert noch nicht auf der vorgeschlagenen Architektur. Anhand des bestehenden Architekturvorschlags kann die `SimulationPlayground.js` Komponente zur iterativen Entwicklung der Architektur verwendet werden und in den Architekturprototypen integriert werden.

Kapitel 6

Fazit

Die in dieser Arbeit untersuchten Technologien und untersuchten Lösungsansätze zeigen, dass die Umsetzung einer Animationsbibliothek für SimPy mit einer stabilen und erweiterbaren Architektur gut möglich ist. Angetroffene Herausforderungen wie die Performance der Datenübertragung konnten mit einem iterativen Architekturentwicklungsansatz bewältigt werden. Die in dieser Arbeit demonstrierten Ansätze einer auf PixiJS aufbauenden Lösung zur Animation zeigen, dass eine modern aussehende, dynamische Oberfläche offeriert werden kann, welche sowohl von Simulationsentwicklern, als auch von externen nicht-technisch versierten Partnern verstanden und eingeschätzt werden kann. Aufgrund der Möglichkeiten mit welchen die Architektur erweitert werden kann, ist vorstellbar, dass zusätzlich zu einer Implementation welche mit 2 Dimensionen visualisiert, eine Implementation für 3 Dimensionen erstellt werden kann.

Kapitel 7

Vision und Ausblick

Das Potenzial dieser Animationsbibliothek ist noch lange nicht ausgeschöpft, es gibt noch unzählige Funktionen und Visionen, die das Produkt ergänzen oder erweitern könnten. Vor allem mit Ausblick auf die Bachelorarbeit die als Folgearbeit auf diese Studienarbeit durchgeführt wird, soll die Vision festhalten werden. Dazu wird zwischen Major Version Features und möglichen Features unterschieden. Major Version Features sind nach momentaner Einschätzung ein fixer Bestandteil einer Folgearbeit, mögliche Features hingegen sind Ideen und Vorschläge welche noch besprochen und ausgearbeitet werden sollten.

7.1 Major Version Features

Die Beschreibung der nachfolgenden Funktionalitäten sind das MVP für die erste Version 1.0.0 der Bibliothek.

7.1.1 Prototyp Funktionalität

Für die Prototypen sind Ideen und Vorschläge in der Form von Use Cases im Kapitel Anforderungen für Prototypen[1] beschrieben. Folgende Use Cases sind Teil des MVP für die Version 1.0.0.

- Spielfeld visualisieren[2]
- Spielfeldbereiche visualisieren[3]
- Agenten visualisieren und animieren[4]
- Agenten Kommunikation visualisieren[5]

7.1.2 Visualisieren von 5000 Agenten

Der momentane Prototyp kann schon mit einer hohen Anzahl Agenten umgehen, es fehlen aber noch aussagekräftige Tests und eine Spezifikation der Leistungsanforderungen.

7.1.3 Visualisieren von globalen Variablen

Ein Teil einer Simulation sind globale Variablen, wie die Anzahl Kunden oder die benötigten Ressourcen für möglichst effiziente Abläufe. Globalen Variablen ändern sich während die Simulation läuft, dieser Verlauf wird als Diagramme dargestellt und während dem Durchführen der Simulation animiert. Am Ende der Simulation kann der gesamte Verlauf betrachtet werden. Diese Grafiken können auch exportiert werden.

Typische Grafiken sind:

[1] 3.1 Anforderungen an Prototypen
[2] 3.1.1 Spielfeld visualisieren
[3] 3.1.2 Spielfeldbereiche visualisieren
[4] 3.1.3 Agenten visualisieren und animieren
[5] 3.1.4 Agenten Kommunikation visualisieren

- Liniendiagramme
- Kuchendiagramme

Für die Implementierung der Diagramme und Grafiken kann bqplot[6] verwendet werden oder falls damit die Anforderungen nicht genügend abgedeckt werden können die Diagramme auch mit einer JavaScript Bibliothek wie D3[7] implementiert werden.

7.1.4 Visualisierung auf Anfrage auslösen

Der momentane Prototyp animiert die Simulation in Echtzeit, jede Änderung wird sofort animiert. Bei ressourcenintensiven oder schnell ablaufenden Simulationen besteht die Anforderung Animation nur auf Anfrage auszulösen. Es muss eine Funktionalität zur Verfügung stehen, die als Teil der Simulation aufgerufen werden kann und die Daten für die visuelle Darstellung anpasst.

Diese Funktionalität könnte auch die Grundlage für eine triviale Stepper-Funktionalität bilden, wenn zusätzlich zu jedem Aufruf für die Visualisierung die Simulation angehalten wird und nach dem Verarbeiten einer entsprechenden Benutzerinteraktion wieder fortgesetzt wird.

7.1.5 Ausführlichere Darstellung der Kommunikation

In den Prototypen wird Kommunikation bidirektional dargestellt. Kommunikation kann aber auch nur in eine Richtung erfolgen, mit diesem Feature werden visuelle Komponenten hinzugefügt, mit denen die Richtung der Kommunikation dargestellt wird.

7.1.6 Ressourcen animieren

Die Bereiche sind momentan als Ressourcen implementiert, alle anderen Arten von Ressourcen werden noch nicht explizit behandelt. Ressourcen könnten ebenfalls über einen animierbaren Zustand verfügen, welcher beispielsweise die Auslastung (hoch, tief) abbildet. Diese Auslastungen können zusätzlich wie im Abschnitt oben beschrieben als Diagramm dargestellt werden.

7.1.7 Dokumentation

Alle Funktionalitäten der vorhandenen Packages gemäss Overview sind in einer Form dokumentiert die verständlich und erklärend ist. Es ist klar, wofür und wie die Bibliothek verwendet wird. Dafür wird eine geeignete Form verwendet, wie beispielsweise readthedocs[8].

7.2 Mögliche Features

7.2.1 Agent verfolgen

Ein Betrachter einer animierten Simulation hat die Möglichkeit einen Agenten auszuwählen, der jederzeit das Zentrum des Bildausschnitts ist. Diese Funktion ist im Abschnitt UC5: Agent verfolgen[9] genauer beschrieben.

7.2.2 Animation im Zeitintervall

Eine weitere Möglichkeit die Performance zu optimieren ist, dass nur in einem gewissen Zeitintervall visualisiert wird. Die Idee ist dabei, dass zu Beginn der Simulation ein Zeitintervall angegeben wird und dann in diesem Intervall die Daten visualisiert werden.

Der Nachteil dieser Art ist, dass dadurch ein Teil der Simulation nie visualisiert wird, und die visuelle Darstellung deshalb nicht mehr dem entspricht, was eigentlich in der Simulation ablauft. Wenn beispielsweise

[6] https://github.com/bqplot/bqplot
[7] https://d3js.org/
[8] https://readthedocs.org/
[9] 3.1.5 Agent verfolgen

ein Agent innerhalb eines Zeitintervalls von A nach B über C geht, wird der direkte Weg von A nach B animiert. Die Information, dass der Agent eigentlich via C nach B gelangt, ist je nach Anwendung nicht ersichtlich. Für dieses Problem muss entweder noch eine Lösung gefunden werden oder die Dokumentation muss diesen Nachteil explizit erwähnen.

7.2.3 3D Visualisierung

Der Prototyp visualisiert die Simulation im 2D Bereich. Im Vergleich mit Alternativen wie AnyLogic[10] oder Simio[11] stellt sich schnell heraus, dass Simulationen in 3D sehr gefragt sind.

Die Idee dieses Features ist, dass die Simulation in 3D dargestellt werden kann, unter der Berücksichtigung, dass das Benutzen der Bibliothek komplexer und aufwändiger wird. Eine Visualisierung in 3D muss viele verschiedene Einflüsse beachten, damit sie als korrekt wahrgenommen wird. Damit es für den Anwender möglichst einfach wird, sollte möglichst für wiederkehrenden Anforderungen 3D Modelle zur Verfügung stehen aus welchen Entwickler auswählen können. Zusätzlich muss eine Möglichkeit vorhanden sein eigene 3D Modelle in die Simulation zu integrieren.

7.2.4 GUI Interaktionen

Während die Simulation abläuft, möchte man gegebenenfalls stoppen, verlangsamen oder beschleunigen. Diese Interaktionen können über ein visuelles Element vorgenommen werden. Wie diese Interaktionen in SimPy integriert werden können ist aktuell noch unklar.

7.2.5 Kommunikationszustand

Kommunikation kann auch einen Zustand haben, beispielsweise kann Kommunikation welche eine gewisse Zeit dauert als intensiv eingestuft werden. Die Änderungen des Kommunikationszustands kann man animieren.

7.2.6 Erweiterung für andere Simulationsarten

Zusätzlich zur agentenbasierten Simulation werden noch Funktionalitäten für andere Simulationsarten angeboten. Mehr Informationen zu anderen Simulationsarten werden im Module `System Modeling and Simulation` vermittelt.

7.2.7 Weiter Grafiken

Die Grafikbibliothek könnte mit den folgenden Diagrammen erweitert werden:

- Heatmap
- Blasendiagramme
- Histogramme
- Balkendiagramme

7.2.8 Vorlagen für die gängigsten Anforderungen

Wenn man alternative Produkte betrachtet, bieten diese meistens Lösungen die schon auf gängige Anforderungen zugeschnitten sind, und auf welchen man aufbauen kann. Im Fall dieser Arbeit kann man Vorlagen für gängige Simulationen und Simulationsobjekte die am INS verwendet werden erstellen.

Diese Vorlagen sind Objekte die in der SimPy Simulation verwendet werden können und deren Verwendung an einer Beispielsimulation demonstriert wird. Für diese Objekte ist, falls benötigt, auch eine entsprechende visuelle Darstellung vorhanden, welche von der Animation verwendet werden kann.

[10] https://www.anylogic.de/
[11] https://www.simio.com/

Kapitel 8

Appendix

8.1 d3_danceclub/dancelub_combination.ipynb

Verfügbar auf GitLab[1]

```
from simpy import RealtimeEnvironment, Resource
from random import expovariate, paretovariate, triangular, random
import random
import ipywidgets.widgets as widgets
from ipywidgets.widgets import IntProgress, interactive
from enum import Enum
from functools import wraps
from time import sleep
import jsons
import DanceclubWidget as dc


class Tracker:
    def __init__(self):
        self.wig = dc.Club()
        self.registered_items = {}
        self.tracked_props = {}
        self.wig.value = "Tracker's Danceclub"
        self.wig.height = 600
        self.wig.width = 600
        self.items_kind = { 'CELL': [], 'AGENT': [] }

    def get_widget(self):
        return self.wig

    def print_all(self):
        print(jsons.dumps(self.tracked_props))

    def update_state(self, item, new_val, prop_name):
        self.set_prop(self.registered_items[item](), prop_name, new_val)

    def set_prop(self, item_id, item_prop, prop_value):
```

[1] https://gitlab.ost.ch/moritz.schiesser/sa_simpy_animationlib/-/blob/master/research/d3_danceclub/danceclub_combination.ipynb

```
33          self.tracked_props[item_id][item_prop] = prop_value
34          self.update_widget()
35
36      def update_widget(self):
37          agent_dict = {}
38          cell_dict = {}
39          for agent in self.items_kind['AGENT']:
40                  agent_dict[agent] = self.tracked_props[agent]
41
42          for cell in self.items_kind['CELL']:
43                  cell_dict[cell] = self.tracked_props[cell]
44          data = {}
45          data['CELLS'] = cell_dict
46          data['AGENTS'] = agent_dict
47          self.wig.json = jsons.dumps(data)
48
49      def register_item(self, inst, id_func, kind):
50          self.registered_items[inst] = id_func
51          self.tracked_props[self.registered_items[inst]()] = {}
52          self.items_kind[kind].append(id_func())
53          #print(f'{inst} registered: {id_func()}')
54
55      def track(self, func):
56          @wraps(func)
57          def func_wrapper(*args, **kwargs):
58              func_exec = func(*args, **kwargs)
59              self.update_state(args[0], getattr(args[0], func.__name__), func.__name__)
60              return func_exec
61          return func_wrapper
62
63      def as_kind(self, func):
64          @wraps(func)
65          def func_wrapper(*args, **kwargs):
66              func_exec = func(*args, **kwargs)
67              self.update_state(args[0], getattr(args[0], func.__name__), 'kind')
68              return func_exec
69          return func_wrapper
70
71      def this_changes(self, prop):
72          def inner_changes(func):
73              @wraps(func)
74              def func_wrapper(*args, **kwargs):
75                  func_exec = func(*args, **kwargs)
76                  self.update_state(args[0], getattr(args[0], prop.fget.__name__),
                        prop.fget.__name__)
77                  return func_exec
78              return func_wrapper
79          return inner_changes
80
81
82  tracker = Tracker()
83  wig = tracker.get_widget()
84  def modell_danceclub(until, max_guests):
85
```

```
86      toilet_time = 3
87      bar_time = 3
88      thirst_level = 50
89      sip_size = 15
90      dance_time = 10
91      guest_action_timeout = 2
92
93      class Musicstyle(str, Enum):
94          SCHLAGER = "SCHLAGER"
95          TECHNO = "TECHNO"
96          DISCOSTAMPFER = "DISCOSTAMPFER"
97
98      class Activity(str, Enum):
99          DANCING = "DANCING"
100         TALKING = "TALKING"
101         WAITING = "WAITING"
102         ON_TOILET = "ON_TOILET"
103
104     class Location(str, Enum):
105         DANCEFLOOR = "DANCEFLOOR"
106         ENTRY = "ENTRY"
107         TOILET = "TOILET"
108         BAR = "BAR"
109
110     class Cell(Resource):
111         def __init__(self, env, capacity, name, location_type, pos_x, pos_y, id, size_x,
                size_y):
112             super().__init__(env, capacity)
113             self._id = id
114             self._name = name
115             tracker.register_item(self, self.get_id, 'CELL')
116             self.id = id
117             self.pos_x = pos_x
118             self.pos_y = pos_y
119             self.name = name
120             self.location_type = location_type
121             self.width = size_x
122             self.height = size_y
123
124         @property
125         def location_type(self):
126             return self._location_type
127
128         @location_type.setter
129         #@tracker.track
130         def location_type(self, value):
131             self._location_type = value
132
133         @property
134         def pos_x(self):
135             return self._pos_x
136
137         @pos_x.setter
138         @tracker.track
```

```
        def pos_x(self, value):
            self._pos_x = value

        @property
        def pos_y(self):
            return self._pos_y

        @pos_y.setter
        @tracker.track
        def pos_y(self, value):
            self._pos_y = value

        @property
        def name(self):
            return self._name

        @name.setter
        @tracker.track
        def name(self, value):
            self._name = value

        @property
        def id(self):
            return self._id

        @id.setter
        @tracker.track
        def id(self, value):
            self._id = value


        def get_id(self):
            return self.id

        @property
        def width(self):
            return self._width

        @width.setter
        @tracker.track
        def width(self, value):
            self._width = value

        @property
        def height(self):
            return self._height

        @height.setter
        @tracker.track
        def height(self, value):
            self._height = value

    class Guest:
        def __init__(self, env, number, liked_music_styles, thirst_level):
```

```
193             self.env = env
194             self.number = number
195             self._id = number
196             tracker.register_item(self, self.get_id, 'AGENT')
197             self.id = number
198             self.state = Activity.WAITING
199             self.location = Location.ENTRY
200             self.liked_music_styles = liked_music_styles
201             self.thirst_level = thirst_level
202             self.known_peers = []
203             self.current_peers = []
204             self.drink_level = 0
205             self.bladder_level = 0
206             self.action = env.process(self.run())
207 
208         def run(self):
209             while True:
210                 # todo: improve condition to include peers, currentmusic
211                 if self.bladder_level > 90:
212                     with self.env.toilet.request() as wait:
213                         self.clear_current_peers()
214                         self.location = Location.TOILET
215                         self.state = Activity.WAITING
216                         # wait in line
217                         yield wait
218                         # go to toilet
219                         self.state = Activity.ON_TOILET
220                         yield self.env.timeout(paretovariate(toilet_time))
221                         # blatter is empty again
222                         self.bladder_level = 0
223 
224                 #and self.thirst_level > random.randint(0, 100)
225                 if self.drink_level > 0:
226                     #loc_sip_size = random.randint(0, sip_size)
227                     self.bladder_level += sip_size
228                     self.drink_level = self.drink_level - sip_size
229                     yield self.env.timeout(5)
230                 # check if guest likes music
231                 # todo: include peers
232                 if env.dancefloor.music in self.liked_music_styles and self.state is not
                        Activity.DANCING:
233                     with self.env.dancefloor.request() as wait:
234                         self.clear_current_peers()
235                         self.location = Location.DANCEFLOOR
236                         # wait if dancefloor is full
237                         self.state = Activity.WAITING
238                         yield wait
239                         # dance dance dance :)
240                         self.state = Activity.DANCING
241                         yield self.env.timeout(paretovariate(dance_time))
242                 # if guest is dancing & knows someone else dancing, they'll go dance
                        with them
243                 if self.state is Activity.DANCING:
244                     for friend in self.known_peers:
```

```
                        if friend.location == Location.DANCEFLOOR and friend.state is
                            Activity.DANCING and friend not in self.current_peers:
                            friend.add_current_peer(self)
                            self.add_current_peer(friend)
                if env.dancefloor.music not in self.liked_music_styles and self.state is
                    Activity.DANCING:
                    self.clear_current_peers()
                    # guest doesn't like the current song -> they will go stand by the
                        bar
                    self.location = Location.BAR
                    self.state = Activity.WAITING
                    yield self.env.timeout(paretovariate(dance_time))
                # todo: include peers and dancefloor
                if self.drink_level <= 0:
                    with self.env.bar.request() as wait:
                        self.clear_current_peers()
                        self.location = Location.BAR
                        self.state = Activity.WAITING
                        # wait at bar
                        yield wait
                        # wait for drink
                        yield self.env.timeout(paretovariate(bar_time))
                        # drink is full again
                        self.drink_level = 100
                # if guest is at bar & knows someone there, they'll go talk with them
                if self.location is Location.BAR:
                    for friend in self.known_peers:
                        if friend.location == Location.BAR and friend.state is not
                            Activity.TALKING:
                            friend.add_current_peer(self)
                            self.add_current_peer(friend)
                            self.state = Activity.TALKING
                            friend.state = Activity.TALKING
                yield self.env.timeout(guest_action_timeout)


        @property
        def bladder_level(self):
            return self._bladder_level

        @bladder_level.setter
        #@tracker.track
        def bladder_level(self, value):
            self._bladder_level = value

        @property
        def drink_level(self):
            return self._drink_level

        @drink_level.setter
        #@tracker.track
        def drink_level(self, value):
            self._drink_level = value

```

```
        @property
        def state(self):
            return self._state

        @state.setter
        @tracker.track
        def state(self, value):
            self._state = value

        @property
        def number(self):
            return self._number

        @number.setter
        def number(self, value):
            self._number = value

        @property
        def id(self):
            return self._id

        @id.setter
        @tracker.track
        def id(self, value):
            self._id = value

        def get_id(self):
            return self.id

        @property
        def location(self):
            return self._location

        @location.setter
        @tracker.track
        def location(self, value):
            self._location = value


        @property
        def current_peers(self):
            return self._current_peers

        @current_peers.setter
        def current_peers(self, value):
            self._current_peers = value

        @property
        def current_peers_id(self):
            current_peer_id = []
            for peer in self.current_peers:
                current_peer_id.append(peer.number)
            return current_peer_id

```

```
349
350         def print(self):
351             return
352             #print(f'guest {self.number} is {self.state} at  {self.location}')
353             if self.current_peers is not []:
354                 for friend in self.current_peers:
355                     print(f'\t\t with: guest {friend.number}')
356
357         def run_entry(self):
358             return
359
360         def add_known_peer(self, other):
361             #print(f'guest {self.number} now knows guest {other.number}')
362             self.known_peers.append(other)
363
364         def get_current_peers(self):
365             return self.current_peers
366
367         @tracker.this_changes(current_peers_id)
368         def add_current_peer(self, other):
369             self.current_peers.append(other)
370
371         @tracker.this_changes(current_peers_id)
372         def remove_current_peer(self, other):
373             self.current_peers.remove(other)
374             if len(self.current_peers) == 0 and self.state is not Activity.DANCING:
375                 self.state = Activity.WAITING
376
377         @tracker.this_changes(current_peers_id)
378         def clear_current_peers(self):
379             for friend in self.current_peers:
380                 friend.remove_current_peer(self)
381             self.current_peers.clear()
382
383
384     class DJ:
385         def __init__(self, env, music_changing_timeout):
386             self.env = env
387             self.music_changing_timeout = music_changing_timeout
388             self.action = env.process(self.run())
389
390         def run(self):
391             while True:
392                 self.env.dancefloor.music = random.choice(list(Musicstyle))
393                 yield self.env.timeout(self.music_changing_timeout)
394
395     class Bar(Cell):
396         def __init__(self, env, capacity, pos_x, pos_y, name="BAR"):
397             super().__init__(env, capacity, name, Location.BAR, pos_x, pos_y, 0, 300,
                 300)
398
399     class Toilet(Cell):
400         def __init__(self, env, capacity, pos_x, pos_y, name="TOILET"):
401             super().__init__(env, capacity, name, Location.TOILET, pos_x, pos_y, 1, 300,
```

```
            300)

class Dancefloor(Cell):
    def __init__(self, env, capacity, pos_x, pos_y, name="DANCEFLOOR"):
        super().__init__(env, capacity, name, Location.DANCEFLOOR, pos_x, pos_y, 2,
            300, 300)

    @property
    def music(self):
        return self._music

    @music.setter
    @tracker.track
    def music(self, value):
        self._music = value


class Entry(Cell):
    def __init__(self, env, capacity, pos_x, pos_y, name="ENTRY"):
        super().__init__(env, capacity, name, Location.ENTRY, pos_x, pos_y, 3, 300,
            300)

class Watchman:
    def __init__(self, env):
        self.env = env
        self.action = env.process(self.run())
        tracker.register_item(self, self.get_id)

    def get_id(self):
        return "watchman"

    @property
    def sim_time(self):
        return self._sim_time

    @sim_time.setter
    @tracker.track
    def sim_time(self, value):
        self._sim_time = value

    def run(self):
        while True:
            self.guest_state()
            # self.dancefloor_state()
            self.sim_time = self.env.now
            yield self.env.timeout(1)

    def guest_state(self):
        for g in self.env.guests:
            g.print()

    def dancefloor_state(self):
        print(f'music: {self.env.dancefloor.music}')

```

```
453         def toilet_state(self):
454             return
455 
456     def add_guests(env, amount):
457         env.guests = []
458         for i in range(4, amount + 4):
459             guest = Guest(env, i, [random.choice(list(Musicstyle)),
                    random.choice(list(Musicstyle))], expovariate(5))
460             env.guests.append(guest)
461 
462         def get_friend(own):
463             new_friend = random.choice(env.guests)
464             if own == new_friend:
465                 return get_friend(own)
466             else:
467                 return new_friend
468 
469         def get_two_friends(own):
470             new_friend1 = get_friend(own)
471             new_friend2 = get_friend(own)
472             while new_friend1 == new_friend2:
473                 new_friend2 = get_friend(own)
474             return (new_friend1, new_friend2)
475 
476         for guest in env.guests:
477             (f1, f2) = get_two_friends(guest)
478             guest.add_known_peer(f1)
479             guest.add_known_peer(f2)
480 
481 
482     env = RealtimeEnvironment(factor=1)
483     env.dancefloor = Dancefloor(env, 15, 0, 300)
484     env.toilet = Toilet(env, 1, 300, 300)
485     env.bar = Bar(env, 2, 300, 0)
486     env.entry = Entry(env, 200, 0, 0)
487     env.dj = DJ(env, 5)
488     #env.watchman = Watchman(env)
489     add_guests(env, 10)
490     wig.enable_draw = True
491     env.run(until)
492     return env
493 
494 btn = widgets.Button(description='start')
495 
496 def start_click(btn):
497     modell_danceclub(60, 5)
498 
499 btn.on_click(start_click)
500 widgets.VBox([btn, wig])
```

Literaturverzeichnis

[1] Projekt Jupyter, "JupyterLab Documentation," 20-Dec-2021. [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/

[2] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer, "D3: Data-Driven Documents." 2011 [Online]. Available: http://idl.cs.washington.edu/papers/d3

[3] Vega Project, "Vega and D3," 18-Dec-2021. [Online]. Available: https://vega.github.io/vega/about/vega-and-d3/

[4] SimPy, "SimPy." [Online]. Available: https://simpy.readthedocs.io/en/latest/

[5] Project Jupyter, "Low Level Widget Tutorial," 09-Sep-2020. [Online]. Available: https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Low%20Level.html

[6] Project Jupyter, "Low Level Widget Tutorial," 09-Sep-2020. [Online]. Available: https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Low%20Level.html#Comms

[7] effbot.org, "What kinds of global value mutation are thread-safe?" 08-Nov-2020. [Online]. Available: https://web.archive.org/web/20201108091210/http://effbot.org/pyfaq/what-kinds-of-global-value-mutation-are-thread-safe.htm

[8] python.org, "GlobalInterpreterLock," 22-Dec-2020. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[9] Tommy Krüger, "D3.js - Performance Test," 08-Sep-2013. [Online]. Available: http://tommykrueger.com/projects/d3tests/performance-test.php

[10] Project Jupyter, "Extension Developer Guide," 18-Dec-2021. [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html

[11] Colin Eberhardt, "Rendering One Million Datapoints with D3 and WebGL." 01-May-2020 [Online]. Available: https://blog.scottlogic.com/2020/05/01/rendering-one-million-points-with-d3.html

[12] Donghao Ren, Bongshin Lee, and Tobias Höller, "Stardust Accessible and Transparent GPU Support for Information Visualization Rendering." 2017 [Online]. Available: https://stardustjs.github.io/publications/eurovis2017-stardust.pdf

[13] mpmramos, *Playful Dog*. 2021 [Online]. Available: https://skfb.ly/6YPIV

[14] goodboydigital.com, "pixi.js bunnymark," 13-Dec-2021. [Online]. Available: https://www.goodboydigital.com/pixijs/bunnymark/