OST
Ostschweizer
Fachhochschule

Entwicklung einer Animationsbibliothek für SimPy

Bachelorarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2022

Autoren:	David Kühnhanss, Moritz Schiesser
Betreuer:	Marc Sommerhalder, Prof. Dr. Andreas Rinkel
Experte:	Knut Schmahl
Gegenleser:	Prof. Dr. Markus Stolze

Abstract

Ausgangslage

Simulationen werden verwendet, um reale Situationen unter reproduzierbaren sowie steuerbar variierenden Bedingungen nachzubilden, und damit Verhaltensweisen und Systeme 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

Aufbauend auf eine vorhergehende Studienarbeit soll eine Bibliothek für die Animation von Simulationsmodellen entwickelt werden. Die in der Studienarbeit gewonnenen Erkenntnisse sowie die darin erarbeiteten Konzepte sollen, wo möglich, infrage gestellt, angepasst, erweitert und übernommen werden. Die Bibliothek soll, integriert in eine JupyterLab Extension, als Open-Source Projekt veröffentlicht werden, inklusive der Distribution über die gängigen Paketverwaltungssysteme. Weiter sollen als Beispiele für die Verwendung der Bibliothek die Beispiele SimPy's um eine Animation erweitert werden.

Ergebnisse

Die drei Komponenten **simplay**, **simplay-web** und **simplay-jupyter** sind veröffentlicht und über die gängigen Paketverwaltungssysteme installierbar. Die Komponente **simplay-jupyter** ermöglicht es, unter Verwendung von **simplay** und **simplay-web**, SimPy Animation zu visualisieren und in JuptyerLab anzuzeigen. Es existiert ein Open-Source Projekt, gehostet auf GitHub, welches öffentlich dokumentiert ist. Die Beispiele aus SimPy wurden um eine Animation erweitert und können als Anwendungsbeispiel dienen. User-Tests haben gezeigt, dass die Komponenten einfach zu installieren und zu verwenden sind. Die API des Python Packages **simplay** ist verständlich für Entwicklerinnen und Entwickler welche mit SimPy vertraut sind.

Eigenständigkeitserklärung

Erklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selbst und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum:

Chur, 09.01.2023

Name, Unterschrift:

David Kühnhanss

Eigenständigkeitserklärung

Erklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selbst und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum:

Ennenda, 09.01.2023

Name, Unterschrift:

Moritz Schiesser

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Management Summary

Simulationen werden verwendet, um reale Situationen unter reproduzierbaren sowie steuerbar variierenden Bedingungen nachzubilden, und damit Verhaltensweisen und Systeme zu untersuchen und vorherzusagen. SimPy ist ein schlankes Python Framework für diskrete Ereignissimulation und wird zum Beschreiben von Simulationen verwendet. Es ist ein Open-Source Projekt, welches unter der MIT-Lizenz veröffentlicht ist, wodurch das Verwenden von SimPy für kommerzielle Angebote möglich ist. Mit SimPy werden Simulationen programmiert und nicht in einem Tool von Hand zusammengestellt. In den folgenden zwei Abschnitten werden zwei Vorteile der modernen Softwareentwicklung hervorgehoben, welche mit der Verwendung von SimPy ohne Probleme angewendet werden können und es ermöglichen, professionell und effizient zu arbeiten.

Strukturierung: Sobald die Simulation wächst, kann der Code in unterschiedliche Files und Strukturen aufgeteilt werden. Dadurch wird, trotz vieler Zeilen Code, eine gute Übersicht gewährleistet, wodurch wiederum Anpassungen durch Entwickelnde effizienter und gezielter vorgenommen werden können. Eine durchdachte Strukturierung unterstützt und ermutigt Entwicklerinnen und Entwickler, wiederverwendbare Strukturen bereitzustellen. Dank solchen wiederverwendbaren Strukturen können Erweiterungen in deutlich geringerer Zeit hinzugefügt werden sowie Fehlerquellen reduziert werden.

Versionsverwaltung: Sobald komplexe Problemstellungen in Angriff genommen werden, reicht eine einzelne Person zur Bewältigung nicht aus, weshalb in Teams gearbeitet wird. Jahrelange Erfahrungen und Probleme mit genau dieser Tatsache haben in der Softwareentwicklung dazu geführt, Versionsverwaltungssysteme zu entwickeln und zu verwenden. Mit Versionsverwaltungssystemen wird eine optimale Zusammenarbeit sowie Nachvollziehbarkeit von Änderungen gewährleistet. Probleme durch die Bearbeitung desselben Files, zur selben Zeit sowie durch unterschiedliche Mitarbeitende können einfach gelöst werden, sodass Arbeitende deutlich weniger blockiert sind. Da Änderungen mit Kommentaren versehen werden, erhalten sie Kontext und sind einfacher nachvollziehbar. Zusätzlich unterstützen Tools wie GitHub oder GitLab das Zwei-Augen-Prinzip optimal, womit die Qualität neuer Änderungen sichergestellt ist, und Wissenstransfer im Entwicklungsteam erzeugt wird.

Mit SimPlay werden in SimPy programmierte Simulationen um Animationen erweitert. Für die Strukturen in SimPy können mit SimPlay dynamisch Bilder, Positionen, beschreibende Texte, Farben und Interaktionen definiert werden. Das Erweitern der Simulation mit einer Animation bietet einen erheblichen Vorteil in der Validierung. Die Animation kann beispielsweise zusammen mit Kunden betrachtet werden und gemeinsam können Verhaltensweisen erläutert und nachvollzogen werden. Die Erstellung von Beispielen im Rahmen der öffentlichen Dokumentation, sowie durchgeführte User-Tests zeigen, dass die Integration von SimPlay in bestehende Simulationen schnell und einfach ist. SimPlay kann für beliebige, in SimPy geschriebene Simulationen verwendet werden. Wie die Simulation visualisiert wird, ist den Entwicklerinnen und Entwicklern überlassen. Sie erweitern die Simulation mit zusätzlichem Code für die Animation. Damit Entwicklerinnen und Entwickler sich mit SimPlay zurechtfinden, enthält die Dokumentation umfangreiche Anleitungen und Verwendungsbeispiele. SimPlay ermöglicht eine Integration in JupyterLab. JupyterLab bietet die Möglichkeit, sogenannte Notebooks zu erstellen. Notebooks stellen eine webbasierte interaktive Programmierumgebung zur

Verfügung, in welcher mit SimPlay erweiterte SimPy Simulationen geschrieben und betrachtet werden können. Simulationsbetrachtende können die Visualisierung kontrollieren. Dazu können folgende Funktionen verwendet werden: “Starten / Pausieren”, “ein Schritt vorwärts”, “Zurücksetzen”, “Beschleunigen / Verlangsamen” und “Vorwärtsspulen / Zurückspulen”. SimPlay ist ebenfalls ein Open-Source Projekt, welches unter der MIT-Lizenz und BSD 3-Clause-Lizenz veröffentlicht ist. Damit ist das Verwenden von SimPlay für kommerzielle Angebote möglich und erwünscht.

Kapitel 2

Ausgangslage

2.1 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 eine Bibliothek für die Animation von Simulationsmodellen entwickelt werden. Die Arbeit umfasst eine vorgängige Recherche von Animationsmöglichkeiten sowie die Evaluation von Technologiekandidaten. Jupyter Notebooks werden in der Lehre als Mittel für interaktive Lernumgebungen eingesetzt, und bieten daher eine geeignete Plattform für die Vermittlung von Simulationsmodellen. Die Resultate der vorhergegangenen Studienarbeit sollen infrage gestellt und weiterentwickelt werden, sodass die erarbeiteten Konzepte, sofern sinnvoll, übernommen werden. Mit den gewonnenen Erkenntnissen soll ein Open-Source Projekt aufgesetzt werden, welches es ermöglicht, SimPy Simulationen innerhalb von JupyterLab zu animieren.

2.2 Zielsetzung

Die erarbeiteten Komponenten erlauben es, unter Berücksichtigung der offenen Art von SimPy, ebenfalls offene und flexible Animationen zu erstellen. Die im Rahmen eines Open-Source Projektes erzeugten Pakete lassen sich über die gängigen Paketverwaltungssysteme installieren und sind somit leicht zugänglich. Beispielsimulationen, welche SimPy anbietet, sind um Animation erweitert und dienen als Anwendungsbeispiele. Schnittstellen, Verwendungsweisen und Komponenten sind dokumentiert und können von Anwenderinnen und Anwendern erlernt und genutzt werden.

2.3 Resultate der Studienarbeit

Die Studienarbeit hat gezeigt, dass es möglich ist, Simulationsmodelle mit Jupyter Notebooks zu animieren [1]. Erste Prototypen zeigen, dass das Ökosystem von Jupyter erweiterbar ist und visualisierte Simulationen in Notebooks integriert werden können. In der Studienarbeit lag der Fokus vor allem auf agentenbasierten Simulationen, eine Art von Simulation, welche in SimPy umsetzbar ist, aber nicht explizit unterstützt wird. Um auch andere Arten von Simulationen animieren zu können, konzentriert sich diese Arbeit nicht auf agentenbasierte Simulationen, sondern soll eine allgemeine Lösung für die Animation von Simulationsmodellen in Jupyter Notebooks bieten. Erweiterungen der Bibliothek für spezielle Anwendungsfälle, wie beispielsweise einer Unterstützung von agentenbasierten Simulationen, sind jedoch denkbar, und könnten in einer zukünftigen Arbeit umgesetzt werden. Die in der Studienarbeit ausgearbeitete Architektur sowie dessen Prototypen werden in dieser Arbeit als Ressourcen für Inspiration und als Grundlage für eine Neuauflage des Arbeitsauftrags verwendet. Insbesondere die Wahl von PixiJS [2] als Technologie für die Darstellung der Simulationen wird in dieser Arbeit weiterverfolgt, da die Studienarbeit gezeigt hat, dass es möglich ist, mit PixiJS eine performante und flexible Lösung zu erstellen.

Kapitel 3

Technologische Grundlagen

Dieses Kapitel beschreibt die für das Verständnis der Arbeit notwendigen Grundlagen, Frameworks und Technologien.

3.1 SimPy

SimPy ist ein Python Framework für diskrete Ereignissimulationen. SimPy reduziert dabei sämtliche Elemente der Simulation auf Ereignisse. Selbst Prozesse sind als Ereignisse zu verstehen. Neben Prozessen gibt es in SimPy auch noch Ressourcen, Container und Stores, welche von Prozessen oder anderen Ressourcen genutzt werden können. Simulationen können in Echtzeit oder "so schnell wie möglich" ausgeführt werden, wobei sich diese Arbeit auf letztere Variante konzentriert. SimPy erlaubt es, mit sehr wenig Aufwand simple Simulationen zu betreiben, welche sich durch eine hohe Flexibilität auszeichnen. Die Dokumentation von SimPy [3] gibt als simples Beispiel eine Simulation mit zwei Uhren, bei der eine Uhr schneller läuft als die andere:

```
1  import simpy
2  def clock(env, name, tick):
3      while True:
4          print(name, env.now)
5          yield env.timeout(tick)
6
7  env = simpy.Environment()
8  env.process(clock(env, 'fast', 0.5))
9  env.process(clock(env, 'slow', 1))
10 env.run(until=2)
```

Die Ausgabe der Simulation ist:

```
1 fast 0
2 slow 0
3 fast 0.5
4 slow 1
5 fast 1.0
6 fast 1.5
```

Die Simulation wird durch die Methode `run` gestartet, welche als Parameter die Dauer der Simulation in Simulationszeit annimmt. Wie in dem Beispiel oben zu sehen ist, kann eine sehr einfache Simulation in wenigen Zeilen geschrieben werden.

SimPy beschreibt sich selbst als "Overkill" für Simulationen, welche keine Interkationen zwischen Prozessen oder Ressourcen benötigen [3]. Ein einfaches Beispiel einer Simulation mit Interaktionen zwischen Prozessen

und Ressourcen, könnte wie folgt aussehen:

```python
import random
import simpy

SERVICE_TIME = (3, 5)
NUM_CUSTOMERS = 5
ARRIVAL_TIME = (0, 10)

def customer(env, name, counter: simpy.Resource):
    """A customer arrives, is served and leaves."""
    yield env.timeout(random.randint(*ARRIVAL_TIME))
    print(f'{name} arrives at the supermarket at {env.now:.2f}.')
    with counter.request() as req:
        yield req
        print(f'{name} checks out at the counter at {env.now:.2f}.')
        yield env.timeout(random.randint(*SERVICE_TIME))
        print(f'{name} leaves the supermarket at {env.now:.2f}.')

env = simpy.Environment()
counter = simpy.Resource(env, capacity=1)
for i in range(NUM_CUSTOMERS):
    env.process(customer(env, f'Customer {i}', counter))
env.run()
```

Die Ausgabe der Simulation ist (abhängig von der Zufallszahlengenerierung, es wurde kein Seed [1] gesetzt):

```
Customer 2 arrives at the supermarket at 2.00.
Customer 2 checks out at the counter at 2.00.
Customer 4 arrives at the supermarket at 3.00.
Customer 3 arrives at the supermarket at 5.00.
Customer 2 leaves the supermarket at 7.00.
Customer 4 checks out at the counter at 7.00.
Customer 0 arrives at the supermarket at 8.00.
Customer 1 arrives at the supermarket at 8.00.
Customer 4 leaves the supermarket at 12.00.
Customer 3 checks out at the counter at 12.00.
Customer 3 leaves the supermarket at 15.00.
Customer 0 checks out at the counter at 15.00.
Customer 0 leaves the supermarket at 19.00.
Customer 1 checks out at the counter at 19.00.
Customer 1 leaves the supermarket at 24.00.
```

Der Supermarkt hat dabei nur eine Kasse, welche von den Kundinnen und Kunden benutzt werden kann. Die Kundinnen und Kunden kommen in einem zufälligen Zeitabstand in den Supermarkt und warten, bis sie an der Kasse bedient werden können. An der Kasse werden die Personen zufällig lange bedient, bevor sie den Supermarkt wieder verlassen. Die Simulation läuft, bis alle Personen bedient wurden. Um an der Kasse bedient zu werden, müssen Kundinnen und Kunden die Ressource `counter` anfordern und, falls bereits besetzt, warten bis diese frei ist.

Ressourcen können auch mit Prioritäten angefragt werden, wobei die Ressource dann an den Prozess mit der höchsten Priorität vergeben wird.

Ressourcen haben nach Freigabe wieder die ursprüngliche Kapazität, im Gegensatz zu Containern, bei welchen die durch den Request definierte Menge abgezogen wird. Folgendes Beispiel verdeutlicht die Unterschiede

[1] Eine Zufallszahl, welche als Startwert für einen Zufallszahlengenerator verwendet wird

zwischen Ressourcen und Containern:

```
import random
import simpy

HUNGER_LEVEL = (3, 6)
SERVE_TIME = (1, 2)
COOK_TIME = (2, 7)
COOKED_FOOD = (3, 6)
INITIAL_FOOD = 21
NUM_PEOPLE = 5
ARRIVAL_TIME = (0, 5)

def person(env, name, foodtray: simpy.Container, server: simpy.Resource):
    """A person arrives, is served and leaves."""
    yield env.timeout(random.randint(*ARRIVAL_TIME))
    print(f'{name} arrives at the cafeteria at {env.now:.2f}.')
    with server.request() as req:
        yield req
        hunger = random.randint(*HUNGER_LEVEL)
        print(f'{name} is served at {env.now:.2f}, wants {hunger} food units.')
        yield foodtray.get(hunger)
        yield env.timeout(random.randint(*SERVE_TIME))
        print(f'{name} leaves the cafeteria at {env.now:.2f}.')
        print(f'Cafeteria has {foodtray.level} units of food in the tray.')

def cook(env, foodtray: simpy.Container):
    """A cook cooks food and puts it in the cafeteria."""
    while True:
        yield env.timeout(random.randint(*COOK_TIME))
        food = random.randint(*COOKED_FOOD)
        print(f'Cook puts {food} units of food in the tray at {env.now:.2f}.')
        foodtray.put(food)
        print(f'Cafeteria has {foodtray.level} units of food in the tray.')

env = simpy.Environment()
foodtray = simpy.Container(env, capacity=INITIAL_FOOD, init=INITIAL_FOOD)
server = simpy.Resource(env, capacity=1)
env.process(cook(env, foodtray))
for i in range(NUM_PEOPLE):
    env.process(person(env, f'Person {i}', foodtray, server))
env.run(until=20)
```

Die Ausgabe der Simulation ist (abhängig von der Zufallszahlengenerierung, es wurde kein Seed gesetzt):

```
Person 4 arrives at the cafeteria at 1.00.
Person 4 is served at 1.00 and wants 5 food units.
Person 1 arrives at the cafeteria at 3.00.
Person 4 leaves the cafeteria at 3.00.
Cafeteria has 16 units of food in the tray.
Person 1 is served at 3.00 and wants 6 food units.
Person 1 leaves the cafeteria at 4.00.
Cafeteria has 10 units of food in the tray.
Cook puts 6 units of food in the tray at 5.00.
Cafeteria has 16 units of food in the tray.
```

```
Person 0 arrives at the cafeteria at 5.00.
Person 2 arrives at the cafeteria at 5.00.
Person 3 arrives at the cafeteria at 5.00.
Person 0 is served at 5.00 and wants 4 food units.
Person 0 leaves the cafeteria at 7.00.
Cafeteria has 12 units of food in the tray.
Person 2 is served at 7.00 and wants 6 food units.
Person 2 leaves the cafeteria at 9.00.
Cafeteria has 6 units of food in the tray.
Person 3 is served at 9.00 and wants 3 food units.
Cook puts 4 units of food in the tray at 10.00.
Cafeteria has 7 units of food in the tray.
Person 3 leaves the cafeteria at 10.00.
Cafeteria has 7 units of food in the tray.
Cook puts 6 units of food in the tray at 16.00.
Cafeteria has 13 units of food in the tray.
Cook puts 4 units of food in the tray at 18.00.
Cafeteria has 17 units of food in the tray.
```

Eine Cafeteria hat einen Serviceangestellten und eine Köchin. Die Köchin kocht für eine zufällig lange Zeit neue Mahlzeiten, welche dann in die Cafeteria gebracht werden. Der Serviceangestellte bedient die Kundinnen und Kunden, welche für eine zufällig lange Zeit an der Kasse warten müssen und eine zufällige Menge an Mahlzeiten benötigen. Das `foodtray` ist ein Container, welcher die Mahlzeiten enthält. Die Mahlzeiten werden vom Serviceangestellten aus dem Foodtray entnommen, womit die Mahlzeiten vom Foodtray entfernt werden. Die Mahlzeiten werden dem Foodtray hinzugefügt, wenn die Köchin neue Mahlzeiten kocht.

Die obigen Beispiele decken die grundlegenden Konzepte von SimPy ab. Weiterführende Informationen zu SimPy sind in der SimPy Dokumentation [3] verfügbar.

3.2 Project Jupyter

Project Jupyter ist die Organisation, welche hinter Jupyter Notebook, JupyterLab und JupyterHub steht. JupyterLab ist eine Webanwendung, in welcher Notebooks verwaltet werden. Mit Notebooks können interaktive Dokumente erstellt werden, welche Code, Text und Bilder enthalten können. Die Notebooks können den in Zellen definierten Code ausführen und dessen Ausgabe anzeigen. Die Ausgabe kann Text, Bilder, Tabellen, Videos, Audio oder interaktive Widgets sein. Im Falle dieser Arbeit wird die Visualisierung der Simulationen als interaktives Widget dargestellt.

Jupyter Notebooks werden - unter anderem auch an der OST - in der Lehre eingesetzt. Das folgende Bild zeigt ein Jupyter Notebook, in dem mit Python versucht wird, ein Yazzi zu würfeln:

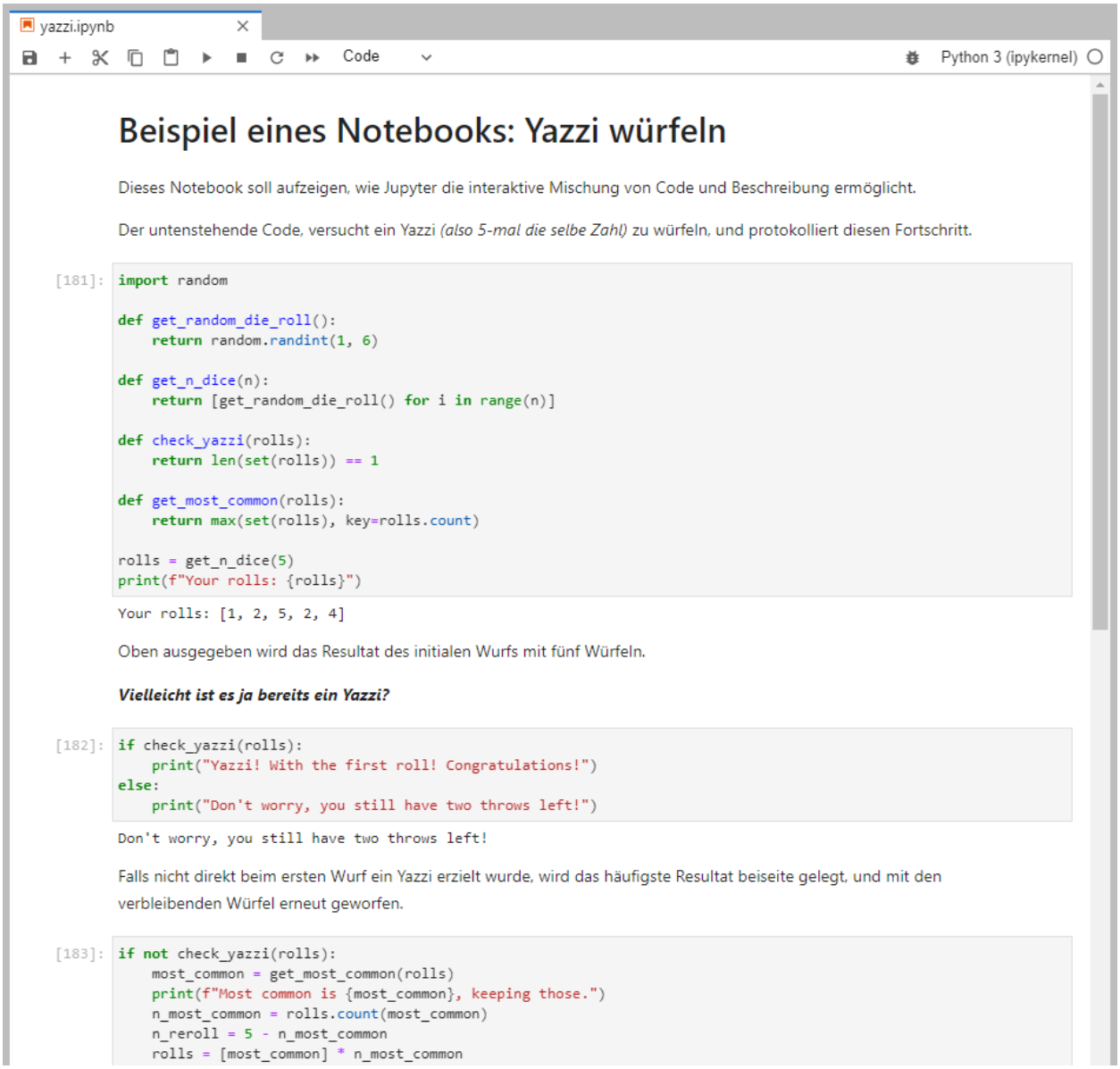


Abbildung 3.1: Jupyter Notebook: Yazzi

Auf dem Bild lässt sich gut erkennen, dass der Code in verschiedene Zellen aufgeteilt ist, welche einzeln ausgeführt werden können. Zwischen den Zellen ist jeweils ein Textfeld definiert, welches Text in Markdown-Syntax enthält. Die Zellen teilen sich einen gemeinsamen Ausführungskontext, das heisst, dass Variablen, welche in einer Zelle definiert werden, auch in anderen Zellen verwendet werden können. Dieses Mischen von Code und Text wird gerne eingesetzt, um den Lesenden die einzelnen Schritte zu erklären.

Jupyter Notebooks erlauben verschiedene Exportmöglichkeiten, unter anderem auch als Markdown. Im Anhang dieser Arbeit ist ein Beispiel für ein Jupyter Notebook als Markdown [2] zu finden.

[2] 13.2 Beispiel eines Jupyter Notebooks: Yazzi würfeln

Kapitel 4

Anforderungen

4.1 Funktionale Anforderungen

Das folgende Diagramm ist eine Übersicht über die Anforderungen. In den anschliessend folgenden Kapiteln sind die jeweiligen Use Cases ausführlich beschrieben. In den Anforderungen wird davon ausgegangen, dass Simulations-Entwicklerinnen und -Entwickler und Betrachtende mit einem Jupyter Notebook arbeiten, die Simulation in JupyterLab ausführen und anschliessend ebenfalls in JupyterLab betrachten.

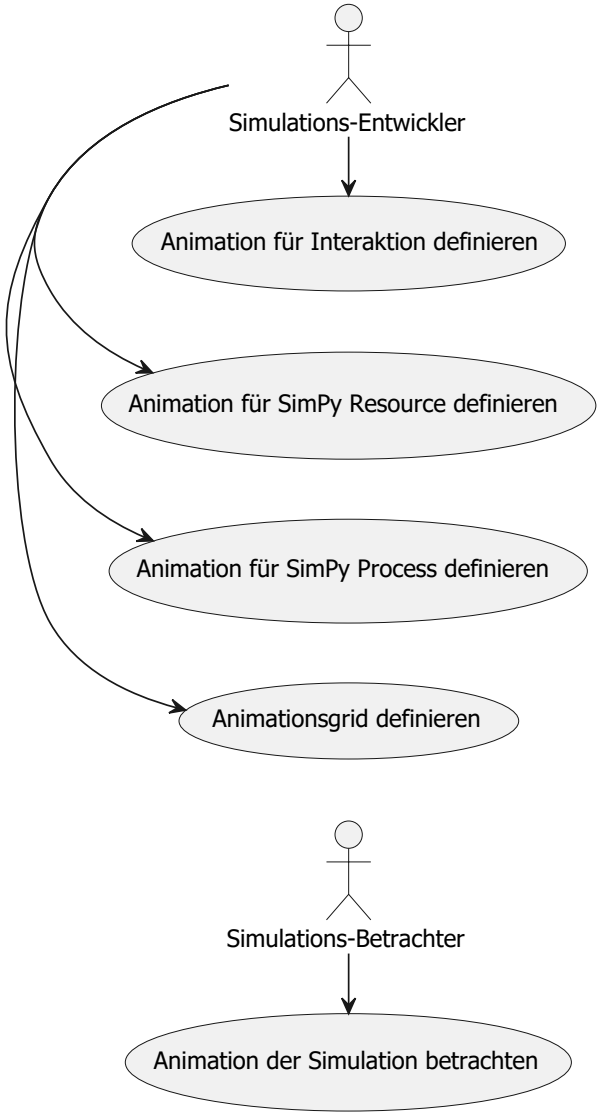


Abbildung 4.1: Funktionale Anforderungen

4.1.1 Animation der Simulation betrachten

Akteur:

Simulations-Betrachter

Voraussetzung:

Ein Jupyter Notebook mit einer funktionierenden SimPy Simulation, welche bereits ausgeführt wurde und eine EventQueue erstellt hat. Die durch diese Arbeit zu erstellenden Komponenten sind gemäss Dokumentation installiert.

Beschreibung:

Ein Simulations-Betrachter möchte die Animation einer Simulation in einem Jupyter Notebook betrachten. Dazu wird eine Liste von Simulationsevents zur Verfügung gestellt. Ein Simulations-Betrachter kann durch eine Interaktion den Start der Animation auslösen. Es stehen weitere Interaktionen zum Pausieren, Zurücksetzen, Vorwärtsspulen, Zurückspulen und Verändern der Abspielgeschwindigkeit zur Verfügung.

4.1.2 Animationsgrid definieren

Akteur:

Simulations-Entwickler

Voraussetzung:

Ein Jupyter Notebook, in welchem die Simulation entwickelt werden kann, sowie die zusätzlichen durch diese Arbeit zu erstellenden Python Komponenten sind gemäss Dokumentation installiert.

Beschreibung:

Entwicklerinnen und Entwickler definieren das Grid, in welchem die Animation einer Simulation dargestellt wird. Entwicklerinnen und Entwickler definieren ausserdem die Bereiche, mit deren Hintergrundfarbe und Grösse sowie die gesamte Grösse des Grids.

4.1.3 Animation für SimPy Process definieren

Akteur:

Simulations-Entwickler

Voraussetzung:

- Ein Jupyter Notebook, in welchem die Simulation entwickelt werden kann, sowie die zusätzlichen durch diese Arbeit zu erstellenden Python Komponenten sind gemäss Dokumentation installiert.
- Das Grid für die Animation ist definiert.

Beschreibung:

Entwicklerinnen und Entwickler möchten einen SimPy Process animieren. Dazu wird die SimPy Simulation mit zusätzlichen Definitionen für die Animation von Prozessen ergänzt. Entwicklerinnen und Entwickler haben die Möglichkeit, die Position eines Process innerhalb des Grid zu definieren, einen Process an eine andere Position zu bewegen, das Aussehen eines Process zu definieren, indem ein Bild zur Verfügung gestellt wird und einen Text zu definieren, welcher in der Nähe des Process angezeigt wird.

4.1.4 Animation für SimPy Resource definieren

Akteur:

Simulations-Entwickler

Voraussetzung:

- Ein Jupyter Notebook, in welchem die Simulation entwickelt werden kann, sowie die zusätzlichen durch diese Arbeit zu erstellenden Python Komponenten sind gemäss Dokumentation installiert.
- Das Grid für die Animation ist definiert.

Beschreibung:

Entwicklerinnen und Entwickler möchten Ressourcen einer Simulation animieren. Dazu wird die SimPy Simulation mit zusätzlichen Definitionen für die Animation von Ressourcen ergänzt. Entwicklerinnen und Entwickler können die unterschiedlichen Arten von Ressourcen, Container mit deren Level, Stores mit deren Anzahl Objekte, PreemptiveResource, PriorityResource und normale Ressourcen animieren. Entwicklerinnen und Entwickler haben die Möglichkeit, die Position einer Resource innerhalb des Grid zu definieren, eine Resource an eine andere Position zu bewegen, das Aussehen einer Resource zu definieren, indem ein Bild zur Verfügung gestellt wird und einen Text zu definieren, welcher in der Nähe der Resource angezeigt wird.

4.1.5 Animation für Interaktion definieren

Akteur:

Simulations-Entwickler

Voraussetzung:

- Ein Jupyter Notebook, in welchem die Simulation entwickelt werden kann, sowie die zusätzlichen durch diese Arbeit zu erstellenden Python Komponenten sind gemäss Dokumentation installiert.
- Das Grid für die Animation ist definiert.
- Die interagierenden Komponenten sind vorhanden.

Beschreibung:

Entwicklerinnen und Entwickler einer Simulation möchten die Interaktion zwischen zwei Komponenten visuell darstellen. Dazu wird die SimPy Simulation mit zusätzlichen Definitionen für die Animation ergänzt. Entwicklerinnen und Entwickler verwenden bei der Definition die eindeutigen Referenzen der Komponenten, damit klar ist, zwischen welchen Komponenten die Interaktion stattfindet.

4.2 Nicht funktionale Anforderungen

4.2.1 Open-Source Setup

Beschreibung:

Die SimPlay Bibliothek steht unter einer Open-Source Lizenz und ist in einem Repository auf einem gängigen, öffentlichen Versionsverwaltungssystem, wie beispielsweise GitHub, verfügbar. Um einen möglichst kurzen und simplen Releasezyklus zu gewährleisten, verfügt das Repository über eine entsprechende Funktionalität, welche gewährleistet, dass mit wenigen Klicks und innerhalb von Minuten ein öffentlich zugängliches Release verfügbar ist, welches mit einem Paketmanager wie `npm` [1] oder `PyPI` [2] installiert werden kann. Weiter sollte die Einstiegshürde für neue Contributors möglichst tief sein, indem eine Dokumentation bereitgestellt wird, welche dabei hilft, das Projekt und dessen Schnittstellen zu verstehen. Damit wird eine möglichst einfache Erweiterbarkeit durch Contributors unterstützt. Zusätzlich ist das Repository so gegliedert, dass sich Benutzende der Bibliothek einfach zurechtfinden.

[1] Node Package Manager, ein Paketmanager für Node.js
[2] Python Package Index, online verfügbare Sammlung von Python-Paketen

Überprüfbarkeit:

Das Setup, und wie ein möglichst schneller, simpler Releasezyklus gewährleistet wird, ist im Kapitel Open-Source Projekt [3] beschrieben. Die Dokumentation ist ebenfalls als Teil des Resultats im Abschnitt Dokumentation [4] beschrieben. Ob Benutzerinnen und Benutzer sich zurechtfinden, wird mit User-Tests [5] überprüft.

4.2.2 Benutzer Flexibilität

Beschreibung:

Die SimPlay Bibliothek ermöglicht eine schnelle und einfache Erweiterung von bestehenden Simulationen. Dazu müssen Entwicklerinnen und Entwickler lediglich die Simulation um die entsprechenden Definitionen für die Animation erweitern und angemessene Bilder für die visuelle Repräsentation der Simulationskomponenten bereitstellen.

Überprüfbarkeit:

Das Erreichen dieser Anforderung wird mit User-Tests [6] überprüft.

4.2.3 Benutzerfreundlichkeit der Dokumentation

Beschreibung:

Die Dokumentation der SimPlay Bibliothek ist so aufgebaut, dass sie für Benutzende verständlich ist. Die Dokumentation ist in mehrere Abschnitte gegliedert, welche die einzelnen Funktionalitäten der Bibliothek beschreiben. Als Benutzerin und Benutzer soll es dabei möglich sein, die relevanten Informationen schnell zu finden.

Überprüfbarkeit:

Das Erreichen dieser Anforderung wird mit User-Tests [7] überprüft.

4.2.4 Zugänglichkeit

Beschreibung:

Die SimPlay Bibliothek ist für Benutzende mit einer bestehenden JupyterLab Installation einfach zu installieren. Der Installationsvorgang soll dabei keine Komplexität mit sich bringen und schnell durchgeführt werden können. Die Anleitung zur Installation ist in der Dokumentation vollständig enthalten.

Überprüfbarkeit:

Das Erreichen dieser Anforderung wird mit User-Tests [8] überprüft.

[3] 9.5. Open-Source Projekt
[4] 9.4 Dokumentation
[5] 13.6 User-Test Handout
[6] 13.6 User-Test Handout
[7] 13.6 User-Test Handout
[8] 13.6 User-Test Handout

Kapitel 5

Vision

Das sehr proprietäre Universum von Simulationsprogrammen und deren Animationsfähigkeiten, ist ein Hindernis für die Verbreitung von Simulationen in der Lehre. Die Vision von SimPlay ist es, eine Plattform zu schaffen, welche es ermöglicht, Simulationen in einem existierenden Open-Source Ökosystem frei zugänglich zu animieren. Mit der Integration des Projekts in JupyterLab sind interaktive Dokumente leicht erstellbar und bieten damit eine Grundlage für die Verbreitung und Vermittlung von animierten Simulationen.

SimPy hat sich als eine offene und flexible Plattform für die Simulation von Systemen positioniert. Dabei ist es wichtig festzuhalten, dass SimPy primär die Aufgabe des Event-Schedulings übernimmt und ansonsten keine Einschränkungen in der Modellierung vornimmt. Dieser Ansatz ist für Entwicklerinnen und Entwickler, welche mit Python vertraut sind, sehr angenehm. Komplexe Abhängigkeiten können modelliert werden, ohne dass die Komplexität eines UI [1] Editors oder einer proprietären Programmiersprache hinzugefügt werden muss. Diese Philosophie ist auch in SimPlay weiterhin zu verfolgen, und ist ein wichtiges Ziel der Arbeit. Im Rahmen von Beispielen und Dokumentation sollen die Fähigkeiten von SimPlay gezeigt werden und Vorschläge für die Integration der Bibliothek in bestehende Projekte gegeben werden.

Weiter soll die Schnittstelle von SimPlay so gestaltet werden, dass die Integration von SimPlay in bestehende Projekte möglichst einfach ist. Bestehende Simulation müssen nicht komplett neu implementiert werden, sondern können mit minimalen Erweiterungen des Codes um eine Animation ergänzt werden.

Mit dem grundlegenden Aufbau der Visualisierungsfläche als Koordinatensystem wird eine intuitive und naheliegende Ausgangslage für die Visualisierung von Simulationen geschaffen. Entwicklerinnen und Entwickler können sich auf die Modellierung konzentrieren und müssen sich nicht darum kümmern, eine komplexe Visualisierung selbst zu implementieren.

Mit der Strukturierung dieser Arbeit als Open-Source Projekt steht es Simulationsforscherinnen und -forschern weltweit offen, die Bibliothek zu erweitern und zu verbessern, sollte dies für ihre Anwendung notwendig sein. Nach Abschluss der Arbeit soll das Projekt in einem Zustand sein, in dem es durch die Community weiterentwickelt werden kann. Insbesondere sollen Strukturen für die Veröffentlichung der Pakete und der Dokumentation bestehen, sodass die Community die Arbeit aufnehmen kann, ohne dass sie sich mit der Konfiguration des Projekts beschäftigen muss.

[1] User Interface, die grafische Oberfläche eines Programms

Kapitel 6

Technologieentscheide

Anhand der Vision [1] und den Anforderungen [2] stellt sich die Frage, wie diese umzusetzen sind. In diesem Kapitel werden verschiedene Technologien vorgestellt, welche zum Implementieren verwendet werden. Wo bereits entsprechende Entscheidungen oder Recherchen in der Studienarbeit getroffen wurden, wird auf ebendiese verwiesen, um Wiederholungen zu vermeiden.

6.1 Bestehende Lösungen

Eine Recherche zu bestehenden Lösungen wurde bereits im Rahmen der vorhergehenden Studienarbeit durchgeführt [1, p. 5]. Die Studienarbeit hat gezeigt, dass keine der bestehenden Lösungen mit dem zu entwickelnden Produkt vergleichbar ist.

6.2 Integration mit SimPy

SimPy wird in der Aufgabenstellung vorausgesetzt. SimPy ist eine Python Bibliothek, weshalb die Integration der Visualisierung und der Simulation entsprechend in Python implementiert wird. Es wird Python 3.9 verwendet. Das Ziel dieser Komponente ist das Erstellen einer Liste von Events. Bei dieser Liste handelt es sich um eine Liste aller Simulationsevents und deren Timestamps im JSON [3] Format. Da JSON Teil der `ECMAScript Language Specification` ist, ist eine optimale Integration mit JavaScript gewährleistet [4, Ch. 25.5]. Die in TypeScript [5] geschriebene Visualisierungskomponente kann die Events entsprechend einfach verwenden.

Für die EventQueue könnten auch alternative Datenformate wie XML [4] oder reiner Text verwendet werden. Allerdings müsste dafür ein eigener Parser oder eine zusätzliche Bibliothek verwendet werden. Mit der Verwendung von JSON ist das Parsen in der Visualisierungskomponente ohne zusätzlichen Aufwand gewährleistet.

Im Python Code wird für das Erstellen von JSON die Bibliothek `jsons` [6] verwendet. Dies, da diese Bibliothek die Übersetzung von in Python üblichem Snake_Case zu in JavaScript üblichem camelCase während der Serialisierung ermöglicht.

Die Integration mit Simpy wird mit `pytest` [7] getestet. `pytest` wurde bereits in der Studienarbeit verwendet und hat sich bewährt. `pytest` ist in der Python Community bekannt, mit über 9'000 Sternen auf GitHub

[1] 5 Vision
[2] 4 Anforderungen
[3] JavaScript Object Notation, textbasiertes, strukturiertes Format zum Speichern von Informationen, häufig in der Webentwicklung verwendet
[4] Extensible Markup Language, ein textbasiertes, strukturiertes Format zum Speichern von Informationen

ist es eines der meist markierten Python Test Frameworks [1]. Der Einsatz eines bekannten Tools hilft, die Einstiegshürden für zukünftige Contributors niedrig zu halten.

6.3 Visualisierung

Verschiedene Technologien zur Visualisierung der Simulation wurden bereits als Teil der Studienarbeit untersucht [1, p. 6]. Wie in der Studienarbeit empfohlen, verwendet diese Arbeit PixiJS [2] für die Visualisierung der Simulation [1, p. 32].

Für die Implementation der Visualisierung wird TypeScript [5] verwendet. TypeScript selbst hat den Grund für die Entstehung von TypeScript ausführlich beschrieben, in diesem Abschnitt werden die wesentlichen Vorteile kurz zusammengefasst [8]. TypeScript ergänzt JavaScript mit Types und ermöglicht dadurch eine Überprüfung von Types während der Transpiletime. Die Komplexität des Hinzufügens von Änderungen kann mit TypeScript deutlich verringert werden. Dadurch unterstützt TypeScript die Entwickelnden darin, weniger Bugs zu produzieren.

Für die Tests der Visualisierungskomponente wird `Floss` [9] eingesetzt. `Floss` ist ein Tool, welches von PixiJS zur Verfügung gestellt wird und auch von PixiJS intern für Tests verwendet wird. `Floss` ist ein Test Runner, welcher die Tests in einer `Electron` Instanz ausführt. Der Vorteil gegenüber herkömmlichen Test Runners ist, dass mit `Floss` PixiJS vorhanden ist und aufgerufen werden kann. Das heisst Aufrufe von PixiJS müssen nicht gemockt werden. Dadurch können die Tests den PixiJS Zustand abfragen und überprüfen, ob die korrekten Objekte erstellt worden sind.

6.4 Integration in JupyterLab

Für die Integration in JupyterLab wurde in der Studienarbeit ein Custom ipywidget [10] empfohlen [1, pp. 34, 44]. In dieser Arbeit wurden entsprechende Möglichkeiten erneut evaluiert, wie unter anderem das Erstellen einer JupyterLab Extension, gemäss Beschreibung im Extension Developer Guide [11] von JupyterLab. JupyterLab bietet drei verschiedene Arten von Extensions: `Application plugins`, `MIME renderer plugins` und `Theme plugins`.

6.4.1 Application Plugins

Im folgenden Abschnitt ist JupyterLab's Beschreibung zu Application Plugins zusammengefasst [11]. Application Plugins sind ein fundamentaler Teil der JupyterLab Funktionalität. Sie können mit JupyterLab interagieren und Services anderer Plugins verwenden. Ein solches Plugin kann auch selbst Services in JupyterLab zur Verfügung stellen. Diese Art von Plugin wird beispielsweise von JupyterLab verwendet, um das Main Menu System, den File Browser und Notebooks inklusive Console und File Editor zur Verfügung zu stellen. Entwickelnde einer Extension für Jupyter haben dieselben Möglichkeiten. Eine Sammlung von Beispielen für Application Plugins ist auf GitHub [12] verfügbar.

6.4.2 MIME renderer Plugins

Im folgenden Abschnitt ist JupyterLab's Beschreibung zu MIME renderer Plugins zusammengefasst [11]. MIME [5] renderer Plugins werden dazu verwendet, "MIME data" in einem Notebook oder File darzustellen. Diese Art von Plugin wird, während es von JupyterLab geladen wird, zu einem Application Plugin konvertiert. JupyterLab verwendet MIME renderer Plugins beispielsweise zur Darstellung von PDF Files [13], von JSON Files oder für die Betrachtung von Vega [6] Files. Entwickelnde einer Extension können für jeden MIME-Type eine Extension erstellen, welche die Daten als Teil eines Notebooks oder als eigenständiges File darstellt.

[5] Multipurpose Internet Mail Extensions, ein Format zur Beschreibung von Dateitypen
[6] Eine Visualisierungsgramatik, welche das Teilen von interaktiven Visualisierungsdesigns ermöglicht

6.4.3 Theme Plugins

Im folgenden Abschnitt ist JupyterLab's Beschreibung zu Theme Plugins zusammengefasst [11]. Theme Plugins ermöglichen es, JupyterLab individuell zu gestalten, indem von Entwickelnden ein Theme erstellt wird. Ein solches Theme definiert eigene Werte für die CSS [7] Variablen von JupyterLab und kann zusätzliche Schriftarten und Bilder beinhalten. JupyterLab verwendet Theme Plugins für die Bereitstellung von unterschiedlichen Themes, wie beispielsweise des Light Theme [14].

6.4.4 Fazit

Zur Verarbeitung der Simulationsdaten eignet sich ein MIME renderer Plugin. Das Theme Plugin verfügt über keine Funktionalität, mit welcher die Simulationsdaten dargestellt werden können. Anstelle des MIME renderer Plugin könnte das Application Plugin verwendet werden. MIME renderer Plugins und Application Plugins ermöglichen die Darstellung von Daten und können Interaktionen verarbeiten, wobei das Application Plugin über eine grosse Anzahl weiterer Funktionalitäten verfügt. Alle Anforderungen an SimPlay können als Teil eines MIME renderer Plugin implementiert werden und es werden keine weiteren Funktionalitäten benötigt, welche durch die Verwendung eines Application Plugin verfügbar wären. Deshalb eignet sich das MIME renderer Plugin am Besten für SimPlay.

Mit einem MIME renderer Plugin kann die EventQueue eigenständig, oder als Zelle eines Notebooks dargestellt werden. Für die Verwendung der von JupyterLab bereitgestellten Extension Points, wird weniger Code und weniger Konfiguration benötigt, als für die Variante mit ipywidget. Weiter kann das Produkt um eine Abhängigkeit verringert werden.

Als Teil des Technologieentscheids für ein MIME renderer Plugin, wurde auch die Anforderung der "Livevisualisierung" infrage gestellt. Wenn nicht live animiert wird, kann die Simulation eigenständig abgespielt werden. Ein eigenständiges Abspielen ermöglicht es, Kontrollstrukturen zum Pausieren, Verlangsamen, Beschleunigen, Zurücksetzen, Vorwärtsspulen oder Zurückspulen in der Animation hinzuzufügen. Die Simulation wird deshalb nicht live animiert.

Für die Implementation der Integration in JupyterLab wird das von JupyterLab bereitgestellte `mimerender-cookiecutter-ts` [15] Template verwendet. Das Template verwendet ebenfalls TypeScript [5]. Für Tests empfiehlt das Template die Verwendung von Jest [16]. Zukünftige Contributors könnten schon Erfahrungen mit JupyterLab Erweiterungen mitbringen oder möchten Erfahrung mit JupyterLab Erweiterungen sammeln. Um die Einstiegshürde möglichst tief zu halten und die in der Jupyter Community etablierten Strukturen und Konzepte zu erhalten, werden die von JupyterLab empfohlenen Technologien verwendet.

Die Komponente zur Integration mit JupyterLab verbindet alle Teile des Projekts miteinander, deshalb eignet sich die Komponente für die Durchführung von UI Tests. Dafür wird JupyterLab mit dem Tool `Galata` [17] im Browser ferngesteuert und getestet. `Galata` ist Teil von JupyterLab und erweitert den Playwright Test Runner [18] um Helferfunktionen für das Testen von JupyterLab UIs. `Galata` wurde explizit für das Testen von JupyterLab UI's entwickelt. In der Studienarbeit hat sich diese Technologie bereits bewährt [1].

6.5 Dokumentation

Für die Dokumentation wird Read the Docs [19] verwendet. Read the Docs ist eine Plattform, mit welcher Dokumentation von Open-Source Projekten gehostet werden kann. Als Alternative würde sich eine Dokumentation direkt in GitHub anbieten, entweder unter Verwendung von GitHub Pages [20] oder dem GitHub Wiki [21]. Read the Docs wird bei Pythonprojekten oft eingesetzt, um Dokumentation für Packages zur Verfügung zu stellen. Die Dokumentation von SimPy wird ebenfalls mit Read the Docs bereitgestellt, Benutzende von SimPy sind mit dem Layout und der Bedienung vertraut. Da eine Integration von SimPlay möglichst einfach sein sollte, wird mit dieser Entscheidung gewährleistet, dass die Struktur der Dokumentation sich an SimPy orientieren kann und der zusätzliche Aufwand für Benutzende möglichst gering ist. Read the Docs stellt ein

[7] Cascading Style Sheets, eine Syntax zur Deklaration visueller Eigenschaften von Websites

Sphinx Theme [22] zur Verfügung, welches in der Dokumentation von SimPy verwendet wird und ebenfalls in der SimPlay Dokumentation eingesetzt wird.

Zusätzlich bietet Read the Docs Unterstützung, mehrere Versionen der Dokumentation gleichzeitig zu hosten. Dadurch wird sichergestellt, dass Benutzenden für unterschiedliche Versionen eine Dokumentation zur Verfügung steht und dass bei einer Weiterentwicklung keine Hindernisse bei der Verfügbarkeit neuer Dokumentationen auftreten. Read the Docs kann so konfiguriert werden, dass bei Änderungen auf dem Repository automatisch neue Versionen der Dokumentation erstellt werden. Das ermöglicht es, dass Contributors sich nicht darum kümmern müssen, ob, wann und wie die Dokumentation gehostet wird.

Ein weiterer Punkt, in welchem die Verwendung von Read the Docs und somit Sphinx einen Vorteil gegenüber den Alternativen hat, ist die `autosummary` [23] Funktionalität, welche es ermöglicht, die Docstring [24] Kommentare in die Dokumentation einzubinden.

Kapitel 7

Prototyp

Um die zu Beginn der Arbeit skizzierte Vision [1] und deren Ziele auf Machbarkeit und ausreichende Funktionalität zu überprüfen, wird ein Prototyp entwickelt, welcher alle wesentlichen funktionalen Anforderungen [2] des angedachten Systems beinhaltet. Das primäre Ziel des Prototyps ist es, zu zeigen, dass die Ideen der Vision implementierbar sind und im Rahmen des Projekts erreicht werden können. Die Vision wird als ausreichend ausgereift angesehen, wenn ein oder mehrere Beispiele von SimPy mit dem Prototypen rudimentär visualisiert werden können. Für den initialen "Durchstich" ist dabei die Architektur dessen sekundär, wobei die aus der Entwicklung des Prototyps gewonnenen Erkenntnisse in die Architektur [3] einfliessen.

Der Prototyp ist nicht direkt mit der Simulationsumgebung in Python verbunden, die generierte Liste an Events wird serialisiert und als File im Visualisierungsteil des Protoypen referenziert. Dieser Ansatz erzeugt zugleich eine natürliche Entkopplung der beiden Projektteile. Grundsätzlich kann die Simulation als Generator von Events verstanden werden, welche dann vom Web-UI abgespult werden. Im Prinzip ist es vorstellbar, dass alternative Generatoren für diese EventQueue entwickelt werden.

7.1 Prototyp in Python

(Der Prototyp respektiert die in Python gängigen Konventionen zu Naming und Casing nicht.)

Der in Python entwickelte Prototyp für die Erzeugung und Serialisierung der Visualisierungsevents umfasst nur jene Komponenten von SimPy, welche in den zu visualisierenden Beispielen benötigt werden. Diese sind:

- `Resource`
- `Container`
- `Process`

wobei Prozesse in SimPy üblicherweise nicht als Klasse implementiert werden. Um die Hilfsstrukturen zur Visualisierung ebenfalls für Prozesse bereitzustellen, müssen Prozesse als Klasse implementiert werden.

Der Prototyp stellt in Python folgende Klassen bereit:

- `AnimatedResource`
- `AnimatedContainer`
- `AnimatedProcess`

welche alle von `AnimatedComponent` erben. In `AnimatedComponent` sind die für die Erzeugung der Events notwendigen Informationen gesammelt, sodass als Folge davon eine möglichst leicht zu verwendende Schnittstelle für die Visualisierung bereitgestellt wird.

[1] 5 Vision
[2] 4.1 Funktionale Anforderungen
[3] 8 Architektur

Die visuellen Zustandsänderungen werden über statische Methoden der dafür zuständigen Klassen deklariert. Diese Methoden erstellen dann jeweils einen Eintrag in der EventQueue, welche vom Web-UI abgearbeitet wird. Für die Sammlung der Events ist der `AnimationManager` zuständig. Ebenfalls werden auf dem `AnimationManager` Rahmenbedingungen, wie visuelle Objekte und das Grid, auf dem die Objekte platziert werden sollen, definiert.

Um den Prototypen in Python für eine Visualisierung verwenden zu können, ist es notwendig, den `AnimationManager` zu instanziieren und in dem SimPy-Environment zu registrieren.

```
env = Environment()
env.animationManager = AnimationManager(env)
```

Damit sowohl `MyResource` als auch `MyProcess` als visuelles Element dargestellt werden können, muss auf ein `visual` verwiesen werden. Ein `visual` ist ein PNG [4]. `Visuals` werden auf dem `AnimationManager` registriert.

```
env.animationManager.registerVisual('RESOURCEVISUAL', 'MyResourceVisual.png')
env.animationManager.registerVisual('PROCESSVISUAL', 'MyProcessVisual.png')
```

Die visuellen Elemente werden dann in der `__init__`-Methode der jeweiligen Klasse deklariert.

```
class MyResource(AnimatedResource):
    def __init__(self, env, capacity, id):
        super().__init__(env, capacity, id, 'RESOURCEVISUAL')

class MyProcess(AnimatedProcess):
    def __init__(self, env, id):
        super().__init__(env, id, 'PROCESSVISUAL')
```

Wenn der Prozess `MyProcess` deklarieren möchte, dass eine Interaktion mit der Ressource `MyResource` stattfindet, wird folgende Zeile benötigt:

```
AnimationUtil.setInteracting(self, my_resource)
```

Der Aufruf dieser Methode muss im Instanzkontext erfolgen, sodass `self` auf den Prozess verweist.

`AnimationUtil` wiederum wird ein Event erzeugen und dem `AnimationManager` übergeben.

```
component.env.animationManager.addEvent(AnimationEvent(
    component.id,
    component.env.now,
    "setInteracting",
    withId=with_component.id
))
```

Events enthalten die Information über den aktuellen Zeitpunkt der Simulation. Damit wird sichergestellt, dass sie später vom Web-UI zum korrekten Zeitpunkt ausgewertet und dargestellt werden. Die Events können nach dem Durchlauf der Simulation aus dem `AnimationManager` als JSON exportiert werden und beispielsweise als JSON-Datei abgespeichert werden.

7.2 Prototyp in TypeScript mit PixiJS

Der Prototyp in TypeScript [5] ist eine Web-UI, welchem der Pfad zu einer JSON-Datei übergeben wird. Diese JSON-Datei enthält die Informationen über die visuellen Elemente und deren Zustandsänderungen abhängig von der Simulationszeit, also die vom `AnimationManager` erstellte EventQueue. Nach Aufbau des Grids und dessen Areas, kann die Visualisierung der Simulation konfiguriert und gestartet werden.

[4] Portable Network Graphics, ein Speicherformat für Bilddaten, erlaubt Transparenz

Das Web-UI stellt dafür die folgenden Interaktions- und Informationselemente bereit:

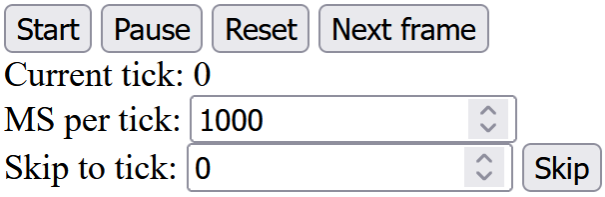


Abbildung 7.1: Interaktionselemente im Web-Frontend des Prototypen

- `Start` startet die Simulation
- `Pause` pausiert die Simulation
- `Reset` setzt die Simulation auf Zeitpunkt 0 zurück
- `Next frame` führt den nächsten Zeitschritt der Simulation aus
- `Current tick` zeigt den aktuellen Zeitschritt der Simulation an
- `MS per tick` definiert die Anzahl Millisekunden, die ein Zeitschritt der Simulation dauern soll
- `Skip to tick` überspringt die Simulation bis zum angegebenen Zeitschritt

7.2.1 Beispiel mit SimPy's Gas Station

Für das Beispiel der Tankstelle, welches von SimPy bereitgestellt wird [25], sieht die Visualisierung wie folgt aus:

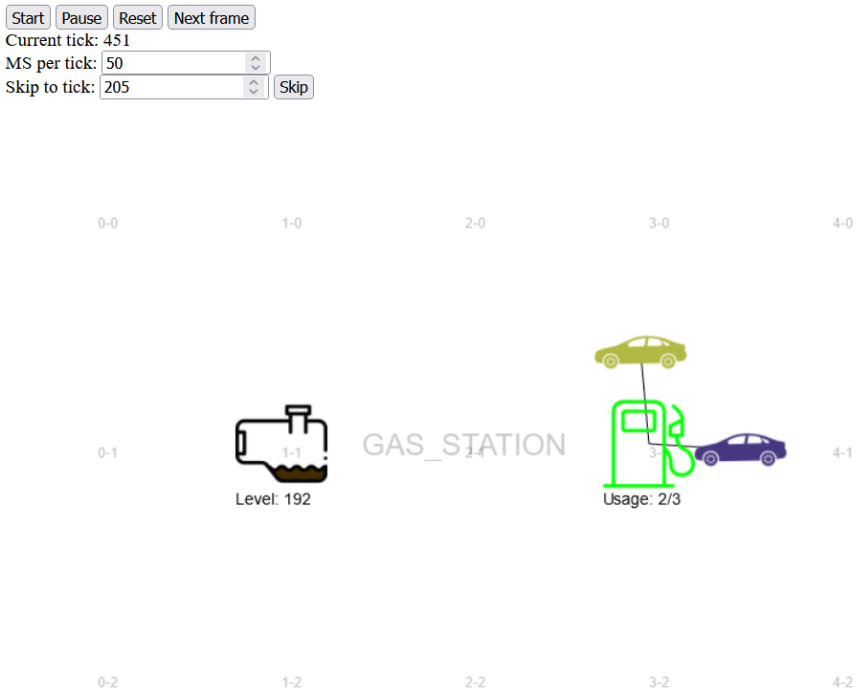


Abbildung 7.2: Tankstellen-Visualisierung

Im Bild verwendete Icons: [26] [27] [28]

Damit das obige Bild zustande kommt, sind unter anderem die folgenden Events nötig:

- `setVisible`, um die Fahrzeuge, die Zapfsäule und den Tank darzustellen
- `moveNear`, um die Fahrzeuge zu den Zapfsäulen zu bewegen
- `setInteracting`, um die Interaktion zwischen den Fahrzeugen und der Zapfsäule zu visualisieren
- `setSpriteFrame`, um den Füllstand des Tanks zu visualisieren

Später in der Simulation werden die aktuell sichtbaren Fahrzeuge mit `setInvisible` ausgeblendet und die Interaktion zwischen den Fahrzeugen und der Zapfsäule mit `setNotInteracting` beendet. Da der Tank stetig geleert wird, wird er irgendwann beinahe leer sein. Sobald der Füllstand unter 10% sinkt, wird ein Tanklastwagen gerufen, welcher den Tank wieder befüllt. 220 Ticks nach der obigen Abbildung sieht die Visualisierung dann, mit einem fast leeren Tank, wie folgt aus:

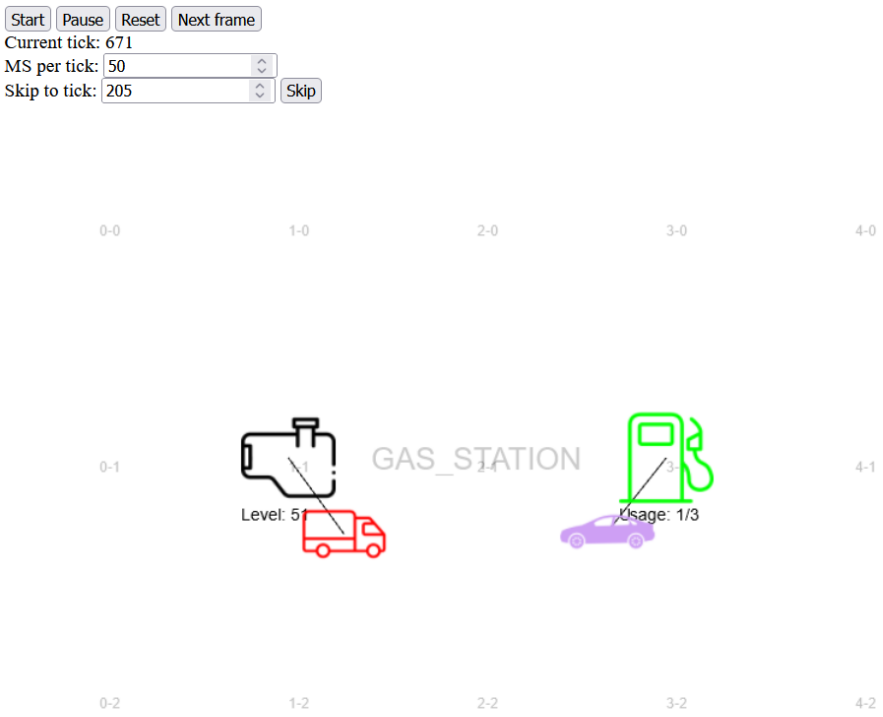


Abbildung 7.3: Tankstellen-Visualisierung mit Lastwagen

Im Bild verwendete Icons: [26] [27] [28] [29]

Wie oben sichtbar wird der Tank wieder befüllt und die Fahrzeuge fahren erneut zur Zapfsäule.

7.2.2 Beispiel mit SimPy's Machine Shop

Für das Beispiel der Maschinenfabrik, welches von SimPy bereitgestellt wird [30], sieht die Visualisierung wie folgt aus:

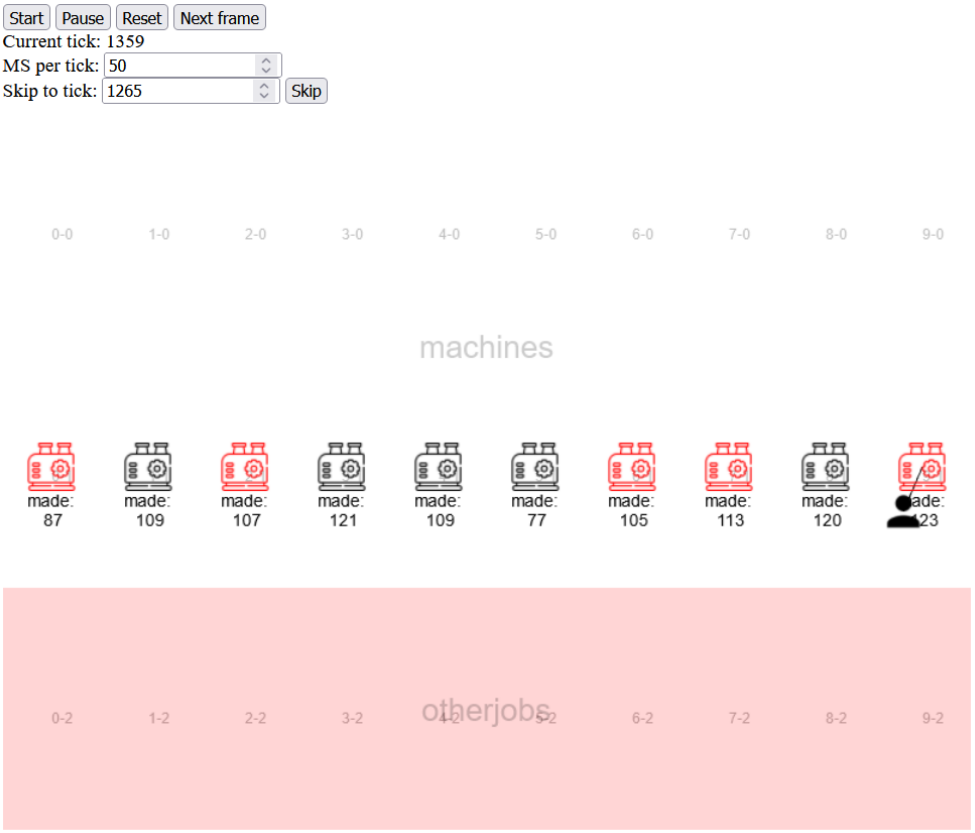


Abbildung 7.4: Maschinenfabrik-Visualisierung

Im Bild verwendete Icons: [31] [32]

Es sind zwei verschiedene Bereiche auf dem Grid definiert: einer für den Arbeitsbereich des Arbeiters und einer für die zehn Maschinen. Im vorhergehenden Beispiel ist es so, dass die Ressourcen jeweils an einer Position fixiert sind und die Prozesse sich um diese Ressourcen bewegen. Im Beispiel dieses Machine Shops ist das umgekehrt, der Arbeiter ist eine Ressource und die Maschinen sind die Prozesse. Die Maschinen, welche rot sind, benötigen Wartungsarbeiten und produzieren daher nichts. Sobald Maschinen Wartungsarbeiten benötigen, fordern sie den Arbeiter an, welcher dann die Wartungsarbeiten durchführt. Während der Arbeiter die Wartungsarbeiten durchführt, interagiert er mit ihnen, wie bei der Maschine ganz rechts im Bild zu sehen ist.

Dieses Beispiel ist insofern interessant, als das die Simulation sehr lange dauert, insgesamt 40'320 Ticks. Der Prototyp ist fähig, mit dieser Grösse umzugehen. Einzig bei der Verwendung des `Skip`-Buttons kann es zu einer kurzen Verzögerung kommen, da die Events der EventQueue von Beginn bis Ende abgearbeitet werden müssen. Hier würde es sich anbieten, periodische Keyframes zu erstellen, welche den aktuellen Zustand der Simulation und Visualisierung zwischenspeichern, sodass diese bei Bedarf geladen werden können, und als Startpunkt für die weitere Berechnung fungieren.

Angenommen eine Simulation hat 90'000 Ticks, und ein Keyframe wird alle 10'000 Ticks erstellt, so würde bei der Anforderung zum Tick 83'478 vorzuspulen, die Simulation nur noch 3'478 Ticks berechnen müssen, also die Differenz von 80'000 und 83'478. Diese Funktionalität ist im Prototypen nicht implementiert.

7.3 Fazit

Der Prototyp ist in der Lage, die gewählten Beispielsimulationen von SimPy zu visualisieren. Der Aufwand, um eine bestehende Simulation um die Visualisierungsfähigkeit zu erweitern, ist gering. Es zeigt sich, dass die Beispiele Gas Station und Machine Shop in jeweils weniger als 15 Minuten visualisiert werden konnten. Mit dem Prototyp wird erfolgreich gezeigt, dass die skizzierte Vision möglich und angemessen ist sowie die beschriebenen Anforderungen umsetzbar sind.

Der Prototyp definiert eine erste Version von Schnittstellen und eine erste Aufteilung von Funktionalitäten und Komponenten. Diese Ideen und das verarbeitete Feedback während des Erstellens des Prototyps beeinflussen die schlussendlichen Designentscheide der Architektur [5] massgebend.

[5] 8 Architektur

Kapitel 8

Architektur

Das Projekt besteht aus drei Komponenten, welche weitgehend unabhängig voneinander entwickelt werden. Das folgende Diagramm gibt eine Übersicht:

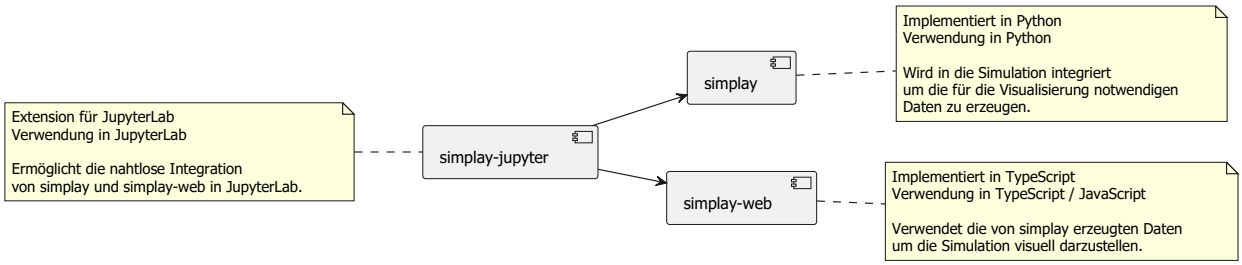


Abbildung 8.1: Übersicht über Komponenten

Die Komponenten `simplay` und `simplay-web` können unabhängig voneinander verwendet werden. Mit `simplay-jupyter` wird diese Verwendung zusammengeführt und vereinfacht. Es findet allerdings eine bewusste Entkoppelung von `simplay` und `simplay-web` statt, sodass bei Bedarf andere Generatoren oder Visualisierungkomponenten für die erzeugten oder zu visualisierenden Daten entwickelt werden könnten, sofern die Struktur der SimulationData [1] eingehalten wird.

8.1 simplay

`simplay` wird zum Erzeugen der EventQueue verwendet und ist in Python entwickelt.

8.1.1 Aufbau

Die Komponente `simplay` besteht aus Klassen, welche als Simulationskomponenten verwendet werden und die Methoden für die Visualisierungsdeklarationen beinhalten, und dem Visualisierungsmanager, welcher ebendiese Deklarationen sammelt und serialisiert.

Damit mit SimPy erstellte Simulationsmodelle leicht um Visualisierungsfähigkeit ergänzt werden können, existieren für die von SimPy beschriebenen Klassen spezielle Klassen, welche Hilfsstrukturen implementieren.

[1] 8.2 SimulationData

Folgendes Diagramm gibt eine Übersicht:

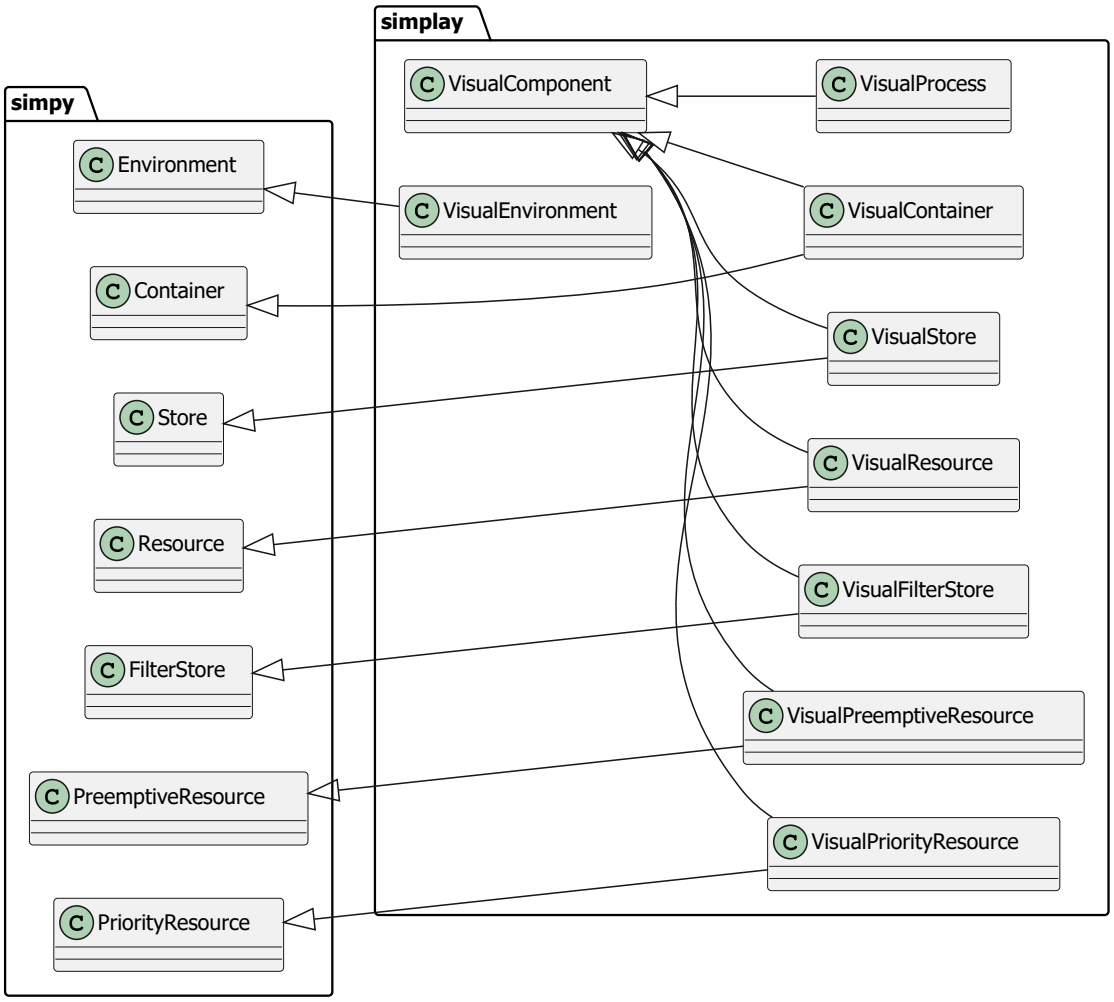


Abbildung 8.2: Übersicht über Grundbausteine

Die Klasse `VisualComponent` ist die Basisklasse für die zu visualisierenden Elemente der Simulation und implementiert die Funktionalität, welche für alle Visualisierungsbausteine gleich ist. Die anderen Klassen erben davon und erweitern sie um spezialisierte Hilfsstrukturen. Jeweilige Spezialisierungen der SimPy Klassen erlauben es, visuelle Indikatoren von Zustandsveränderungen direkt in der API [2] von SimPy zu integrieren. Sollte eine Autorin oder ein Autor einer zu visualisierenden Simulation mehr Kontrolle über die Visualisierungslogik wünschen, ist es möglich, direkt von `VisualComponent` zu erben.

[2] Application Programming Interface, eine Liste von Methoden, welche programmatisch aufgerufen werden können, um Objekte zu manipulieren oder Informationen zu erhalten

8.1.1.1 VisualComponent

Die Klasse `VisualComponent` beinhaltet die für die Deklaration von Visualisierungsevents notwendigen Informationen, wie die ID einer Komponente und die zu visualisierende Grafik. Weiter ist auf der `VisualComponent` ein `type` definiert, welcher angibt, welche Art von Komponente dies ist, also ein Prozess, eine Ressource, ein Container oder ein Store. Das `visual` ist jeweils eine Referenz auf ein zuvor mit dem `VisualizationManager` registriertes PNG.

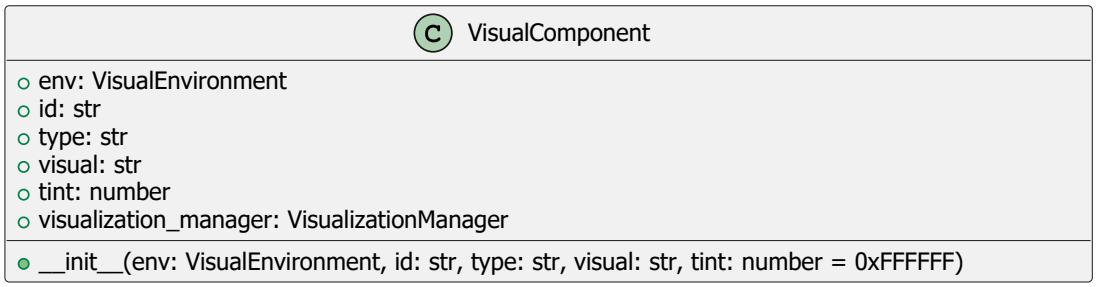


Abbildung 8.3: VisualComponent

Die konkreten Implementationen der Komponenten erben von `VisualComponent` und implementieren die Funktionalität, welche für die jeweilige Komponente notwendig ist. Sobald ein `VisualComponent` instanziiert wird, registriert es sich auf dem `VisualizationManager` mit der Methode `add_entity`. Mit dieser Registrierung ist es in den zu serialisierenden Daten vorhanden. Ebenfalls stellt die Klasse die Methoden für Visualisierungsdeklarationen bereit, auf die später [3] genauer eingegangen wird.

8.1.1.2 VisualizationManager

`VisualizationManager` ist die zentrale Komponente, welche Visualisierungsdeklarationen sammelt und serialisieren kann. Auf dem `VisualizationManager` werden visuelle Ressourcen registriert, welche von den Visualisierungsbausteinen verwendet werden können. Weiter können Informationen zum Grid, auf welchem die Visualisierungen angezeigt werden, definiert werden.

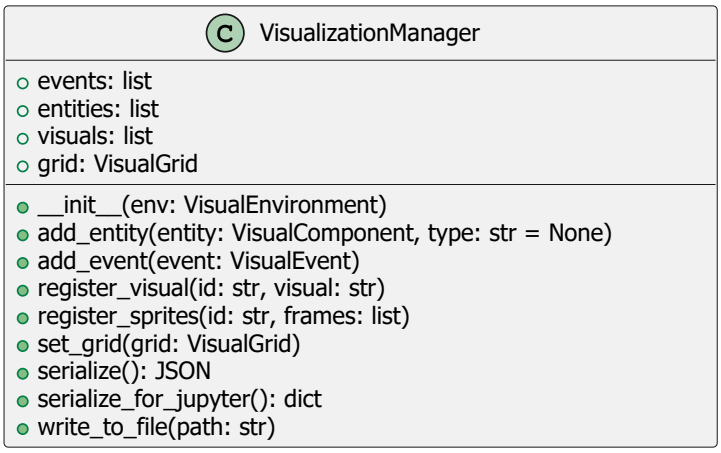


Abbildung 8.4: VisualizationManager

[3] 8.1.1.4 Visualisierungsdeklarationen

Eine genauere Beschreibung zur Struktur des Inhalts von `entities` und `visuals` ist in den Abschnitten Visuals [4] und Entities [5] des Abschnitts SimulationData [6] zu finden.

8.1.1.2.1 VisualGrid

`VisualGrid` enthält Informationen über das Grid, auf welchem die Visualisierungen angezeigt werden sollen. `VisualGrid` erlaubt die Definition von Areas, in welche das Grid aufgeteilt werden kann.

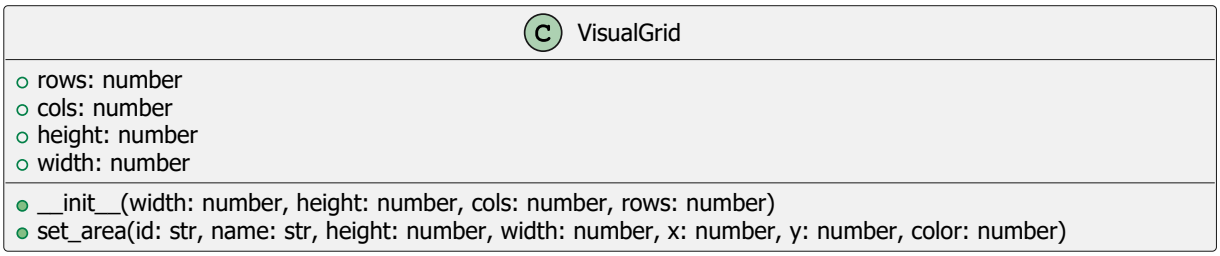


Abbildung 8.5: VisualGrid

8.1.1.2.2 VisualEvent

`VisualEvent` bildet die visuelle Repräsentation einer Zustandsänderung ab. `VisualEvent` enthält Informationen über die Komponente, welche sich verändert hat, die Art der Veränderung und den Zeitpunkt der Veränderung.

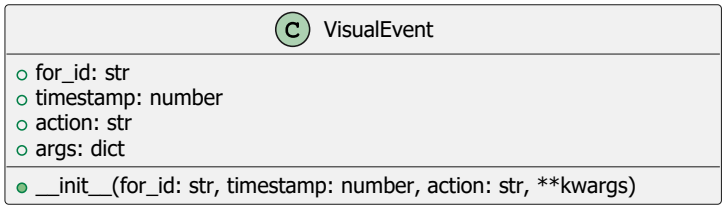


Abbildung 8.6: VisualEvent

Es fungiert als Basisklasse für die spezifischen Events, welche die verschiedenen Zustandsänderungen abbilden.

[4] 8.2.1.3 Visuals
[5] 8.2.1.4 Entities
[6] 8.2 SimulationData

Das folgende Diagramm gibt eine Übersicht über die verschiedenen Events:

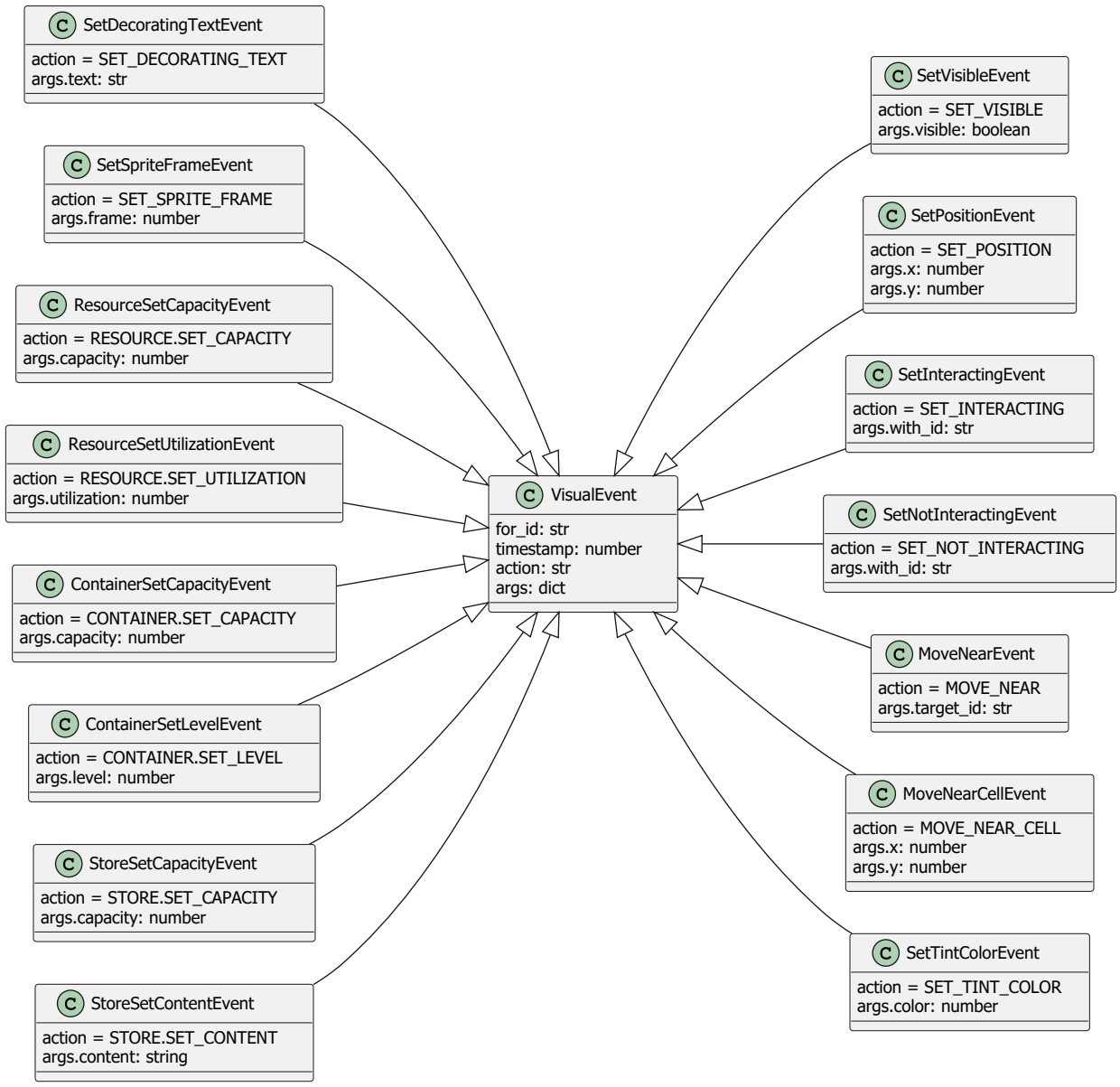


Abbildung 8.7: VisualEvent-Spezialisierungen

Die Vererbungen definieren jeweils einen Konstruktor, der die Spezialisierung der `args` bestimmt, wie in folgenden Beispielen zu sehen ist:

```
class SetPositionEvent(VisualEvent):
    def __init__(self, for_id, timestamp, x, y):
        super().__init__(for_id, time, "SET_POSITION", x=x, y=y)

class SetVisibleEvent(VisualEvent):
    def __init__(self, for_id, timestamp, visible):
        super().__init__(for_id, time, "SET_VISIBLE", visible=visible)
```

Die explizite Definition dieser Klassen erlaubt Benutzenden der Bibliothek, selbst die gewünschten Events zu erzeugen, und diese an den `VisualizationManager` zu übergeben. Damit wird der Philosophie von SimPy, Anwenderinnen und Anwender der Bibliothek so viel Freiheit wie möglich zu geben, Rechnung getragen.

8.1.1.3 VisualEnvironment

Das `VisualEnvironment` ist eine spezielle Umgebung, welches das SimPy Environment um die für die Visualisierung benötigten Strukturen erweitert. Insbesondere wird dafür gesorgt, dass der `VisualizationManager` mit der Umgebung verbunden wird und somit von allen Simulationsobjekten aus verwendet werden kann.

8.1.1.4 Visualisierungsdeklarationen

Die Deklaration der visuellen Änderungen geschieht über Methoden, welche die `VisualEvent` Objekte erzeugen und an den `VisualizationManager` weiterleiten. All diese Methoden sind in der Klasse `VisualComponent` definiert. Die Parameter der Methoden sind abhängig vom Kontext der Deklarationen.

Das folgende Diagramm gibt einen Überblick:

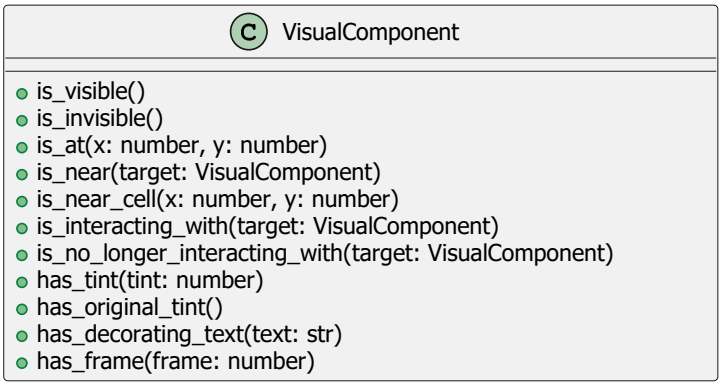


Abbildung 8.8: Visualisierungsdeklarationen

Die Methodennamen folgen einer klaren Struktur. Sie beginnen entweder mit `is_` oder `has_`, und vermitteln so, dass diese Änderungen sofort wirksam werden und den aktuellen Zustand der Simulation abbilden. Weiter haben diese Methodennamen einen Wiedererkennungswert und heben sich von den Methodennamen der SimPy API ab.

Diese Strukturierung der Methodennamen ist ein Resultat der User-Tests [7].

Änderungen an Kapazität, Auslastung, Füllstand oder Inhalt von Ressourcen müssen nicht explizit deklariert werden. SimPlay erkennt diese Änderungen automatisch und deklariert sie entsprechend. So wird sichergestellt, dass die Visualisierung immer den aktuellen Zustand abbildet.

[7] 13.7 User-Test Feedback

8.1.2 Beispielhafte Verwendung

Untenstehend ist ein Beispiel für die Verwendung der oben beschriebenen Komponenten gezeigt. Er ist auf das Wesentliche limitiert und implementiert keine echte Simulation.

```
from simplay import VisualEnvironment, VisualResource, VisualProcess, VisualGrid

class SampleProcess(VisualProcess):
    def __init__(self, env, id, resource):
        super().__init__(env, id, "SAMPLE_PROCESS")
        self.resource = resource

    def run(self):
        self.is_visible()
        self.is_at(2, 1)
        # ...
        with self.resource.request() as req:
            yield req
            self.interacts_with(self.resource)
            self.is_near(self.resource)
            # ...
            self.is_no_longer_interacting_with(self.resource)
            self.is_near_cell(2, 1)

class SampleResource(VisualResource):
    def __init__(self, env, id, capacity):
        super().__init__(env, id, "SAMPLE_RESOURCE", capacity)
        self.capacity = capacity


env = VisualEnvironment()
env.visualization_manager.register_visual("SAMPLE_PROCESS", "process.png")
env.visualization_manager.register_visual("SAMPLE_RESOURCE", "resource.png")
grid = VisualGrid(10, 10, 5, 5)
grid.set_area("AREA_1", "Area 1", 2, 2, 0, 0, 0x00FF00)
grid.set_area("AREA_2", "Area 2", 2, 2, 2, 0, 0xFF0000)
env.visualzation_manager.set_grid(grid)
sample_resource = SampleResource(env, "SAMPLE_RESOURCE_1", 10)
sample_process = SampleProcess(env, "SAMPLE_PROCESS_1", sample_resource)
env.process(sample_process)
env.run(until=100)
env.visualzation_manager.write_to_file("output.simplay")
```

8.2 SimulationData

Der folgende Abschnitt beschreibt die Struktur des `SimulationData` Objekts. In diesem Projekt wird es in Python durch `simplay` erstellt. Die erzeugten Daten werden an `simplay-web` übergeben und dargestellt. Die Übergabe der Daten erfolgt entweder als Datei im JSON Format, falls `simplay` und `simplay-web` manuell verwendet werden, oder durch `simplay-jupyter` im Rahmen eines Jupyter Notebooks.

8.2.1 Struktur

Das `SimulationData` Objekt ist ein JSON-Objekt mit folgenden root-Elementen:

```
{
  "grid": {},
  "events": [],
  "visuals": [],
  "entities": []
}
```

8.2.1.1 Grid

Das `grid` Element beschreibt die Grösse des Grids, in welchem die Animation abgespielt wird und die Areas, welche sich darauf befinden.

```
{
  "width": number,
  "height": number,
  "rows": number,
  "cols": number,
  "areas": []
}
```

`width` und `height` beschreiben die Grösse des Grids in Pixel. `rows` und `cols` beschreiben die Anzahl der Zeilen und Spalten des Grids. `areas` enthält die Areas, welche sich auf dem Grid befinden. Sie sind wie folgt definiert:

```
{
  "id": string,
  "name": string,
  "color": number,
  "gridDefinition": {
    "height": number,
    "width": number,
    "x": number,
    "y": number
  }
}
```

`id` und `name` beschreiben die ID und den Namen der Area. `color` beschreibt die Farbe der Area in Hexadezimal. `gridDefinition` beschreibt die Position und Grösse der Area im Grid, wobei `height` und `width` die Anzahl der Zeilen und Spalten und `x` und `y` die Position der oberen linken Zelle definieren.

8.2.1.2 Events

Das `events` Attribut enthält eine Liste von Events, welche während der Simulation aufgetreten sind. Die Basisstruktur ist wie folgt:

```
{
  "forId": string,
  "timestamp": number,
  "action": string,
  "args": {}
}
```

`forId` beschreibt die ID des Objekts, welches durch das Event betroffen ist. `timestamp` beschreibt den Zeitpunkt, zu welchem das Event aufgetreten ist. `action` beschreibt die Art des Events. `args` beschreibt die zusätzlichen Parameter des Events.

Ein Event zur Definition der Sichtbarkeit eines Objekts kann beispielsweise wie folgt definiert werden:

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_VISIBLE",
  "args": {
    "visible": boolean
  }
}
```

Alle verfügbaren Events sind im Anhang, in der SimulationData Event Definition [8], detailiert beschrieben.

8.2.1.3 Visuals

Dies ist eine Liste der möglichen Visuals, auf welche ein visuelles Element verweisen kann. Die Struktur ist wie folgt:

```
{
    "id": string,
    "frames": string[],
}
```

`id` beschreibt die ID des Visuals. `frames` ist ein Array von zu Base64 [9] serialisierten einzelnen Frames des Visuals. Frames sind im PNG Format unterstützt. Diese Struktur ist bewusst so gewählt, dass die visuelle Repräsentation eines Objekts nicht auf einen einzigen Frame beschränkt ist. Damit können Simulationsobjekte mehrere Zustände visuell abbilden, ohne dass explizit zwischen Objekten mit dieser Anforderung und Objekten ohne diese Anforderung unterschieden werden muss.

8.2.1.4 Entities

Dies ist eine Liste der Entities, welche im Verlauf der Simulation existieren.

```
{
    "id": string,
    "type": "PROCESS" | "RESOURCE" | "CONTAINER" | "STORE" | "CUSTOM",
    "visual": string,
    "tint": number,
}
```

[8] 13.9 SimulationData Event Definition
[9] Ein Verfahren zur Kodierung binärer Daten in ASCII-Zeichen

`id` beschreibt die ID des Entities. `type` beschreibt den Typ des Entities. `visual` beschreibt die ID des Visuals, welches die Entity repräsentiert, und muss in der Liste der Visuals enthalten sein. `tint` beschreibt die Tint-Farbe des Entities. Falls `type` auf `CUSTOM` gesetzt ist, ist das Element kein aktiver Teil der Simulation, sondern dient nur der Darstellung. Der `type` wird auch dafür verwendet, um unterscheiden zu können, welche Simulationsobjekte nebst dem "dekorativen" Text noch einen "informativen" Text anzeigen sollen, was für "RESOURCE", "CONTAINER" und "STORE" der Fall ist.

8.3 simplay-web

Die Komponente `simplay-web`, wird zum Abspielen einer Simulation innerhalb eines Browsers verwendet. Als Rendering Engine wird PixiJS [2] verwendet.

8.3.1 Aufbau

`simplay-web` stellt eine Schnittstelle zur Verfügung, welche folgende Funktionalitäten ermöglicht:

- Eventlist übergeben
- Definition von visuellen Elementen übergeben
- Grid übergeben
- Starten
- Pausieren
- Abspielgeschwindigkeit ändern
- Vorwärtsspulen
- Zurückspulen
- Zurücksetzen

`simplay-web` besteht aus den folgenden Komponenten:

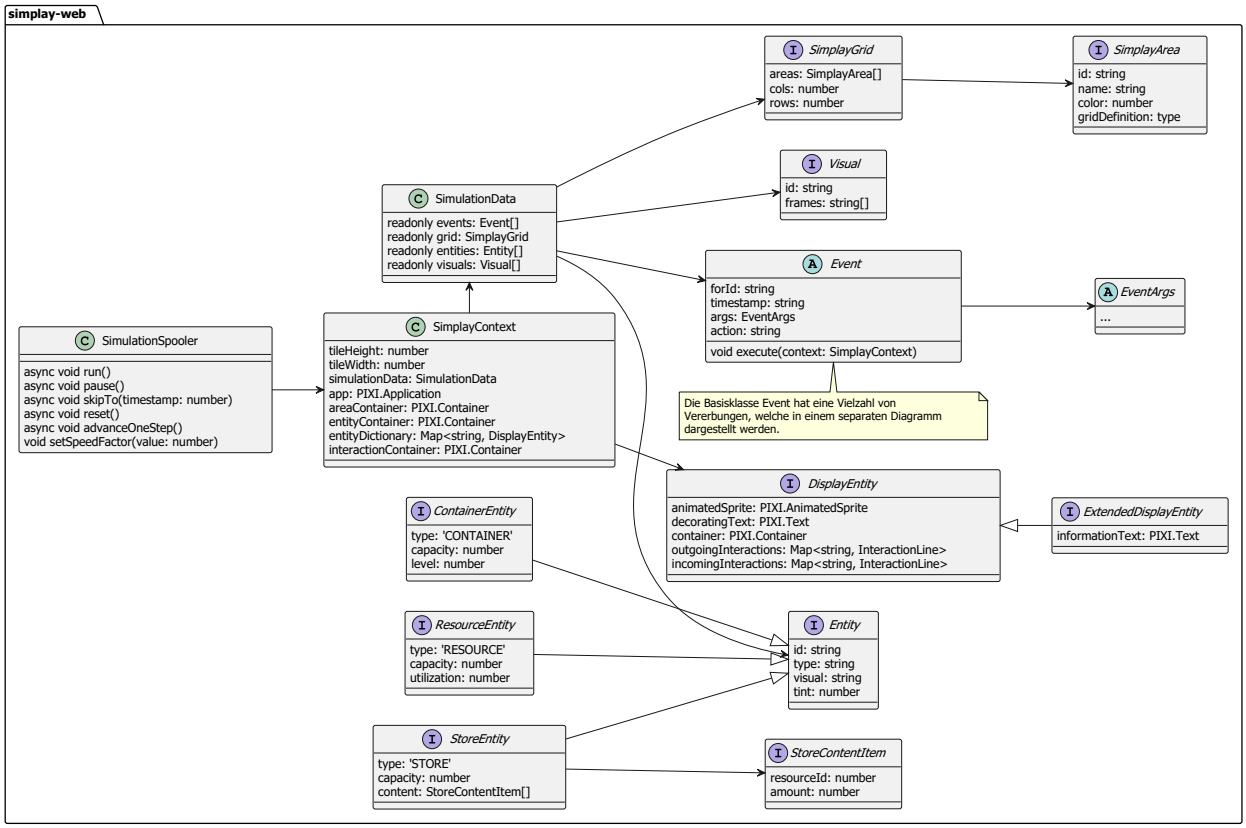


Abbildung 8.9: simplay-web: Komponenten

`SimulationSpooler` ist für die Kontrolle des Abspielens der Simulation verantwortlich. Dafür werden die Events interpretiert, Entities erstellt und mit Verwendung der `Grid` Komponente ein Grid berechnet. Ein Grid eignet sich, um der Benutzerin oder dem Benutzer eine möglichst flexible, aber dennoch strukturierte Möglichkeit zu geben, das Spielfeld zu gestalten [33, pp. 10–13]. Die Entities, welche im `SimplayContext`

gehalten werden, sind jeweils als `PIXI.AnimatedSprite` Objekte implementiert, welche die entsprechenden Visuals verwenden. `SimulationSpooler` ist die externe API von `simplay-web`. Zum Verarbeiten der Events steht für jedes Event ein eigenes Command zur Verfügung, welches die entsprechenden Veränderungen in der PixiJS Application [34] vornimmt, sobald `execute` aufgerufen wird. Diese Struktur folgt dem "Command" Pattern [35, pp. 233–242].

Die Struktur der konkreten Eventklassen, welche von `Event` erben, sowie deren Argumente ist wie folgt:

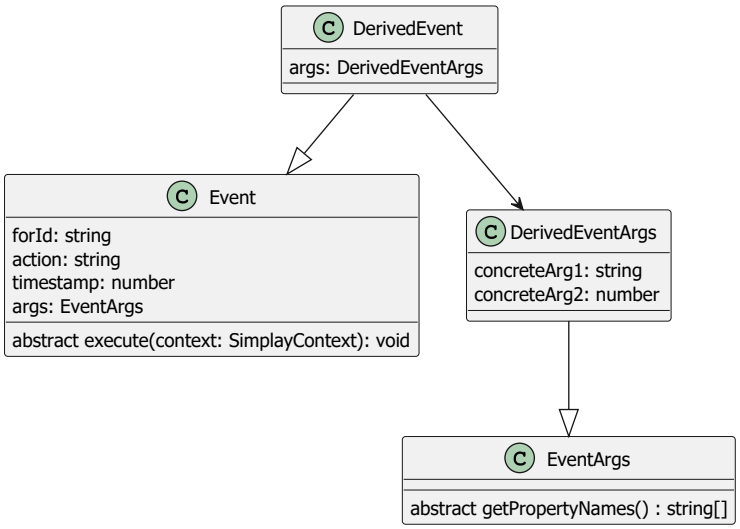


Abbildung 8.10: simplay-web: Vererbungen der Events

Eine Übersicht aller konkreten Eventklassen, welche von `Event` erben, ist im Anhang in simplay-web: Events [10] verfügbar.

Damit die serialisierten Events in Instanzen der jeweiligen Klassen umgewandelt werden können, wird die `eventFactory` verwendet. Diese Methode entscheidet aufgrund des `type` Feldes, welches in jedem Event enthalten ist, welche Klasse instanziiert werden soll. Die erkannten Typen der Events sowie der EventArgs werden dann einer generischen Factorymethod übergeben, welche ebendiese Klassen instanziiert und sie zurückgibt. Damit die Konstruktoren der Klassen aufgerufen werden können, müssen sie den Typen `EventCtor<T extends Event>` und `EventArgsCtor<T extends EventArgs>` entsprechen.

Das iterative Aufrufen der `eventFactory` wird von der `simulationDataFactory` übernommen, welche die serialisierten Events in eine Liste von `Event` Instanzen umwandelt.

[10] 13.10 simplay-web: Events

Das folgende Sequenzdiagramm zeigt den Ablauf aus Sicht der verwendenden Komponente, inklusive des Abspielens und Manipulierens der Simulation:

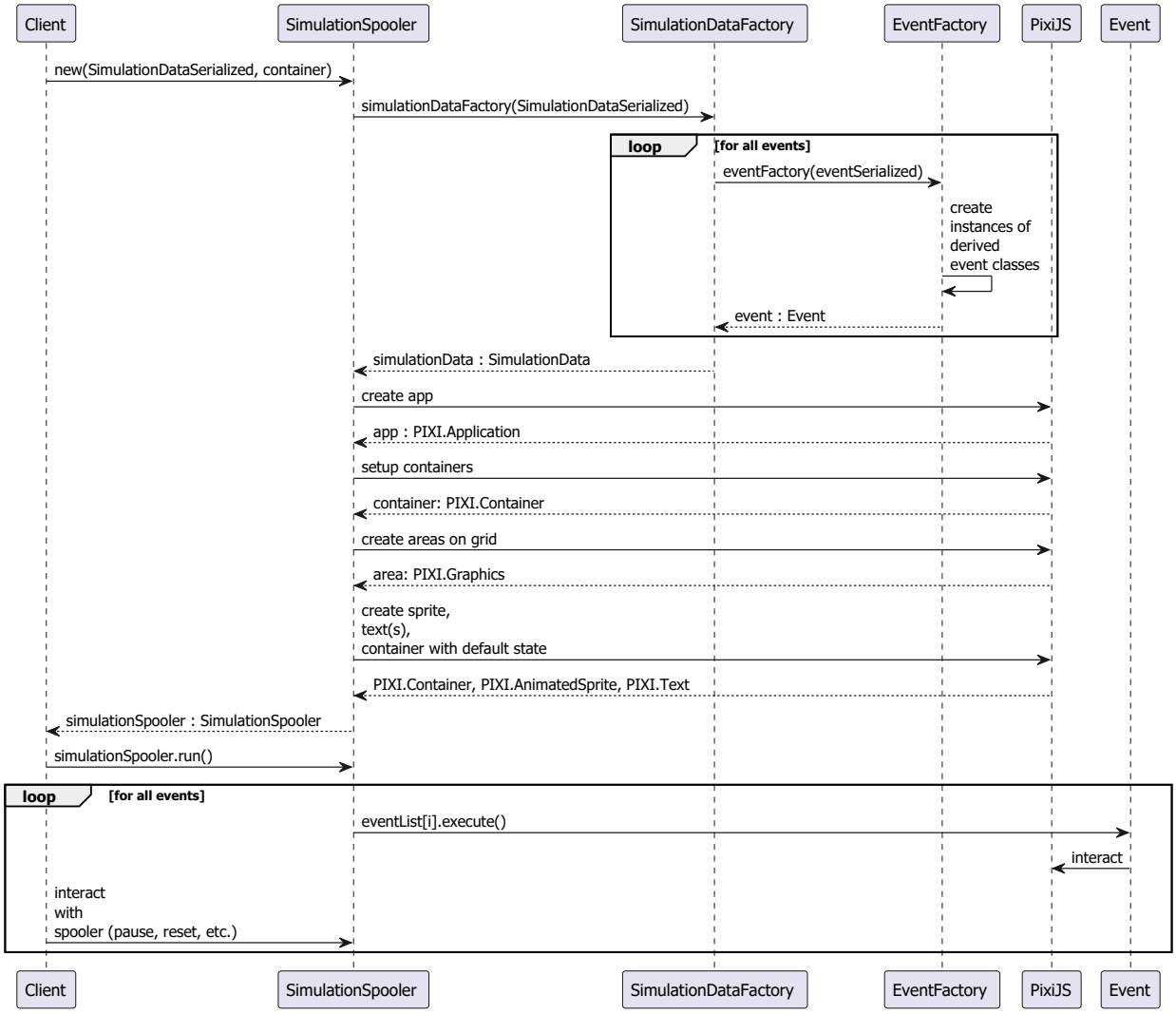


Abbildung 8.11: simplay-web: Aufruf

Die externe Schnittstelle der `simplay-web` Komponente ist der `SimulationSpooler`:

```
class SimulationSpooler {
  constructor(
    simulationData: SimulationDataSerialized,
    container: HTMLElement
  ) {}
  private spoolTimestamp(timestamp: number) {}
  addStepChangedEventListener(listener: (timestamp: number)) {}
  removeStepChangedEventListener(listener: (timestamp: number)) {}
  getTotalSteps() {}
  async run() {}
  async pause() {}
  async advanceOneStep() {}
```

```
13   async skipTo() {}
14   async reset() {}
15   setSpeedFactor(value: number) {}
16 }
```

`SimulationSpooler` wird mit den serialisierten Daten der Simulation und dem Container, in welchem die Simulation dargestellt werden soll, instanziiert.

`spoolTimestamp` ist eine private Methode, welche die Events für den übergebenen Timestamp aus der Liste der Events holt und diese ausführt.

`addStepChangedEventListener` und `removeStepChangedEventListener` ermöglichen das Hinzufügen und Entfernen von Event-Listenern, welche aufgerufen werden, wenn sich der aktuelle Zeitschritt ändert.

`getTotalSteps` gibt die totale Anzahl der Zeitschritte zurück, welche in der Simulation vorkommen.

`run` startet die Simulation, indem die Methode `spoolTimestamp` wiederholt aufgerufen wird. Da jeder Zeitschritt der Simulation gleich lange dauern soll, wird jeweils im Inneren des `while` Loops die verbleibende Zeit abgewartet. Um dies zu ermöglichen, ist die Funktion asynchron.

`pause` pausiert das Abspulen der Events, und stellt sicher, dass die Bearbeitung des aktuellen Schritts abgeschlossen ist.

`advanceOneStep` führt den nächsten Zeitschritt der Simulation, respektive die zum Zeitschritt zugehörigen Events aus.

`skipTo` springt zum übergebenen Timestamp, ebenfalls unter Verwendung der `spoolTimestamp` Methode.

`reset` setzt die Simulation zurück. Dazu werden die erstellten Entities auf ihren Ursprungszustand zurückgesetzt und alle Interaktionen zwischen den Entities zerstört.

Mit der Methode `setSpeedFactor` kann die Geschwindigkeit der Simulation verändert werden.

Folgender Code gibt eine beispielhafte Übersicht, wie die Simulation mit dem `SimulationSpooler` gestartet werden kann:

```
1  const simData = { ... }
2  const simulationSpooler = new SimulationSpooler(simData,
       document.getElementById('containerId'));
3
4  simulationSpooler.run();
5
6  // Die Simulation wird nach einer Sekunde pausiert
7  setTimeout(() => {simulationSpooler.pause()}, 1000);
8  // Die Simulation wird nach einer Sekunde wieder gestartet
9  setTimeout(() => {simulationSpooler.run()}, 1000);
10 // Die Geschwindigkeit der Simulation wird nach einer Sekunde verdoppelt
11 setTimeout(() => {simulationSpooler.speedFactor(2)}, 1000);
12 // Die Simulation wird nach einer Sekunde auf den Zeitschritt 300 gesprungen
13 setTimeout(() => {simulationSpooler.skipTo(300)}, 1000);
```

8.4 simplay-jupyter

`simplay-jupyter` ist eine JupyterLab Erweiterung [11], welche es ermöglicht, in Jupyter Notebooks Simulationen zu visualisieren. Genauer ist `simplay-jupyter` ein MIME renderer plugin [11]. In `simplay-jupyter` fügen sich die zuvor in diesem Kapitel beschriebenen Komponenten und Datenstrukturen zu einem Ganzen zusammen. Ein mit `simplay` erstelltes `SimulationData` Objekt wird von `simplay-jupyter` entgegengenommen und mit `simplay-web` visualisiert.

Damit JupyterLab das MIME renderer plugin `simplay-jupyter` aufrufen kann, muss es installiert sein. Sobald das Plugin installiert ist, kann es entweder durch das Öffnen eines Files mit der Dateiendung `.simplay` oder einem Aufruf von `IPython.display()` gestartet werden. Dieser Aufruf kann in einem Notebook mit der folgenden, von `simplay` bereitgestellten Funktion, ausgelöst werden:

```
1 env = VisualEnvironment()
2 ...
3 from IPython.display import display
4 display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

Wie `simplay-jupyter` und JupyterLab interagieren ist in der folgenden Abbildung aufgezeigt:

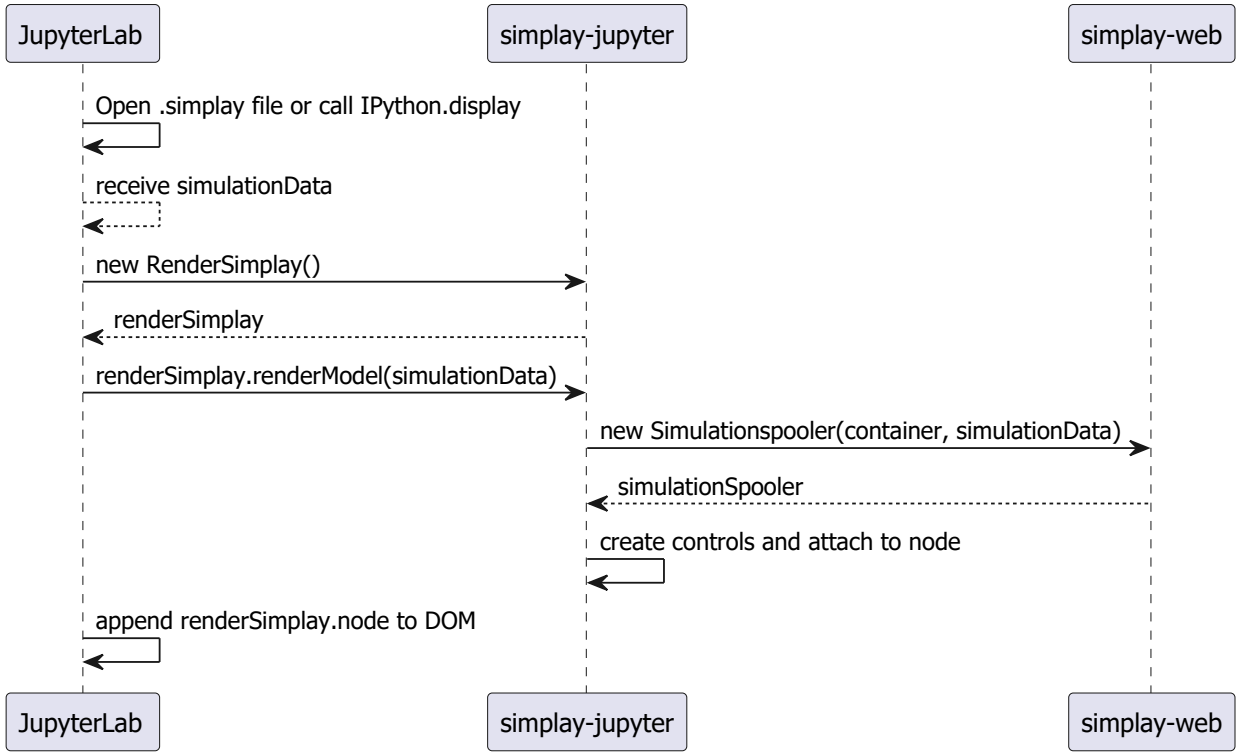


Abbildung 8.12: simplay-jupyter und JupyterLab

8.4.1 Aufbau

Für die Integration mit JupyterLab wird ein Objekt vom Typ IExtension [36] bereitgestellt, welches eine `renderFactory` vom Typ IRendererFactory [37] beinhaltet. Diese Factory erstellt eine neue Instanz von `RenderSimplay`, einer Klasse von `simplay-web`, welche IRenderer [38] implementiert. Die `renderModel` Methode von `RenderSimplay` wird von JupyterLab aufgerufen, sobald die Daten, welche das Plugin erhalten hat, dargestellt werden sollen. In dieser Methode werden die für das `simplay-jupyter` Plugin benötigten DOM [11] Elemente erstellt, und die Interaktion mit `simplay-web` wird definiert.

Je nach Funktionalität und Inhalt der DOM Elemente, können für die Elemente eigene Klassen in separaten Files erstellt werden, welche dann in `renderModel` verwendet werden.

Die Komponente `simplay-jupyter` definiert eine Abhängigkeit zu `simplay`. Im User-Test [12] hat sich gezeigt, dass eine Abhängigkeit zu `simplay` gewünscht ist, da somit für die Verwendung innerhalb von JupyterLab nur eine Installation notwendig ist.

8.4.1.1 Design

`simplay-jupyter` erstellt einen Container, in welchem das Animationsgrid und die Elemente zur Kontrolle der Animation dem DOM hinzugefügt werden. Das Animationsgrid befindet sich über den Elementen zur Kontrolle der Animation.

Die Kontrollstrukturen orientieren sich am YouTube Videoplayer [39]. Da YouTube eine der weltweit meistbesuchten Webseiten ist, sind den Benutzenden die Icons und deren Bedeutung schon bekannt [40]. Zur Kontrolle der Funktionen “Starten / Pausieren”, “ein Schritt vorwärts”, “Zurücksetzen” und “Beschleunigen / Verlangsamen” werden Buttons zur Verfügung gestellt. Ein solcher Button besteht immer aus einem Icon und einem beschreibenden Tooltip. Für die Icons werden Icons des Material Design 3 [41] verwendet. Die Funktion “Vorwärtsspulen / Zurückspulen” wird mit einem Slider ähnlich zu einem YouTube Video dargestellt, die vergangene Zeit wird farblich von der noch abzuspielenden Zeit abgegrenzt. Bei einem Klick auf den Slider wird zum entsprechenden Zeitpunkt gesprungen. Wird mit dem Cursor über den Slider gefahren, erscheint ein Tooltip, welcher angibt zu welchem Zeitpunkt der Simulation ein Klick auf den Slider führt.

Für den Style wie beispielsweise Schriftart, Schriftgrösse oder Schriftfarbe werden die Standards von JupyterLab verwendet. Falls dennoch zusätzliche Styles benötigt werden, werden diese in einem CSS File definiert oder via `HTMLElement.style` [42] gesetzt.

[11] Document Object Model, eine Struktur zur Verwaltung von HTML-Elementen
[12] 13.7 User-Test Feedback

Kapitel 9

Resultat

Das Resultat besteht aus den Packages `simplay`, `simplay-web`, `simplay-jupyter` und der Benutzerdokumentation. Um das Bild zu vervollständigen, ist im letzten Abschnitt `Open-Source Projekt` beschrieben, wie und wo diese Packages existieren.

9.1 simplay

`simplay` steht als eigenständiges Package in PyPI [43] zur Verfügung. `simplay` ist ein Python Package, das mit dem Kommando `pip install simplay` [1] installiert werden kann. Die Dokumentation des Package ist Teil des SimPlay Read the Docs [44], welche im Abschnitt Dokumentation [2] dieses Kapitels beschrieben ist.

`simplay` wird von Simulations-Entwickelnden benötigt, um eine Simulation um Animationsinformationen zu erweitern. Anhand dieser Informationen wird die EventQueue aller Simulationsschritte, in der im Abschnitt SimulationData [3] beschriebenen Struktur, erstellt.

Das Package stellt Funktionalitäten für alle Funktionalen Anforderungen [4] der Rolle `Simulations-Entwickler` zur Verfügung. Zusätzlich hat sich anhand der User-Tests [5] gezeigt, dass die Benutzer Flexibilität [6] gewährleistet ist. Der Tester konnte ohne weitere Fragen eine eigene Simulation erweitern und hat bestätigt, dass das Erweitern ohne intensiven Lernaufwand möglich ist.

Die Verwendung von `simplay` ist im Anhang Verwendung von `simplay` mit JupyterLab: Beispiel 1 [7], den Beispielen der Dokumentation [8] oder im Abschnitt `simplay` [9] ersichtlich.

`simplay` ist eine Abhängigkeit von `simplay-jupyter` und ist automatisch mit der Installation von `simplay-jupyter` vorhanden. Benutzende müssen `simplay` nicht einzeln installieren, sondern werden `simplay` im Standardfall via `simplay-jupyter` erhalten.

`simplay` ist Teil des SimPlay Repository und der Source Code wird auf SimPlay - GitHub [45] gehostet. Wie der Source Code des Package strukturiert ist und wie er verwaltet wird, ist im Kapitel Open-Source Projekt [10] beschrieben.

[1] Python Package Installer, ein Paketmanager für Python
[2] 9.4 Dokumentation
[3] 8.2 SimulationData
[4] 4.1 Funktionale Anforderungen
[5] 13.7 User-Test Feedback
[6] 4.2.2 Benutzer Flexibilität
[7] 13.3 Verwendung von `simplay` mit JupyterLab: Beispiel 1
[8] 13.5.4 Examples
[9] 8.1 simplay
[10] 9.5 Open-Source Projekt

9.2 simplay-web

`simplay-web` steht als eigenständiges Package in npm [46] zur Verfügung. `simplay-web` ist ein JavaScript Package, welches mitsamt TypeScript Types ausgeliefert wird. Das Package kann mit dem Kommando `npm i simplay-web` installiert werden. Dieses Package ist nicht im SimPlay Read the Docs beschrieben, da Benutzende, welche eine SimPy Simulation animieren, dieses Package vermutlich nie direkt verwenden. Für Interessierte am `simplay-web` Package stehen Informationen in dessen README.md [47] zur Verfügung.

`simplay-web` visualisiert eine EventQueue, welche der im Abschnitt SimulationData [11] beschriebenen Struktur folgt. Ein von `simplay-web` bereitgestellte Visualisierung für das Gas Station Beispiel von SimPy sieht folgendermassen aus:

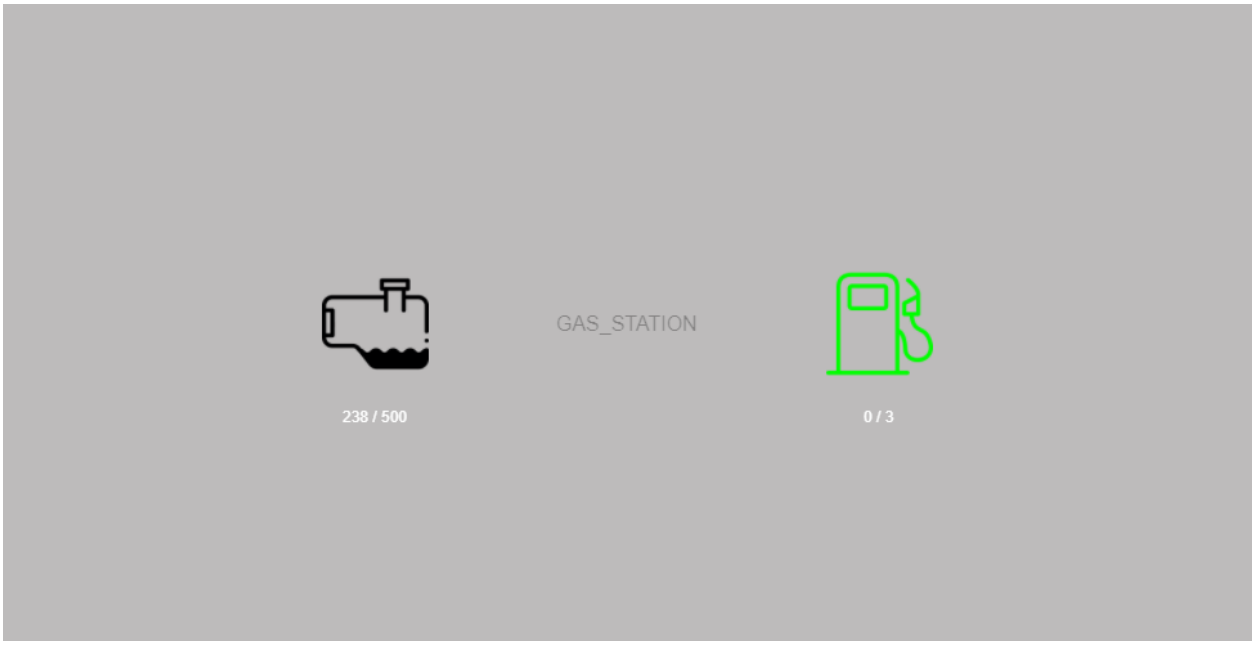


Abbildung 9.1: simplay-web: Gas Station

Im Bild verwendete Icons: [26] [28]

Zusätzlich bietet `simplay-web` Schnittstellen zur Kontrolle der Visualisierung an. Diese Schnittstellen sind im Abschnitt simplay-web [12] im Kapitel Architektur beschrieben. Sie werden beispielsweise von `simplay-jupyter` aufgerufen, um Benutzenden die Kontrolle der Visualisierung zu ermöglichen.

`simplay-web` ist Teil des SimPlay Repository und der Source Code wird auf SimPlay - GitHub gehostet. Wie der Source Code des Package strukturiert ist und wie er verwaltet wird, ist im Kapitel Open-Source Projekt [13] beschrieben.

[11] 8.2 SimulationData
[12] 8.3 simplay-web
[13] 9.5 Open-Source Projekt

9.3 simplay-jupyter

`simplay-jupyter` ist als eigenständiges Package in PyPI [48] sowie npm [49] vorhanden. `simplay-jupyter` ist ein Jupyter Mime Renderer Plugin, welches mit dem Ausführen von `pip install simplay_jupyter` oder via JupyterLab UI von npm installiert wird. Die User-Tests [14] haben bestätigt, dass die Installation von `simplay-jupyter` sehr einfach durchzuführen ist.

Die Dokumentation des Package ist Teil des SimPlay Read the Docs [44], welche im Abschnitt Dokumentation [15] dieses Kapitels beschrieben ist. `simplay-jupyter` ist für Simulations-Entwickelnde das Package, welche sie installieren müssen, um die Visualisierung der Simulation in JupyterLab zu betrachten. Für die Visualisierung verwendet `simplay-jupyter` `simplay-web`, indem es die von `simplay` erstellte EventQueue an `simplay-web` übergibt und im UI Kontrollstrukturen anbietet.

`simplay-jupyter` ermöglicht das Darstellen der Simulation als Teil eines Notebooks, wie auf folgendem Screenshot ersichtlich:

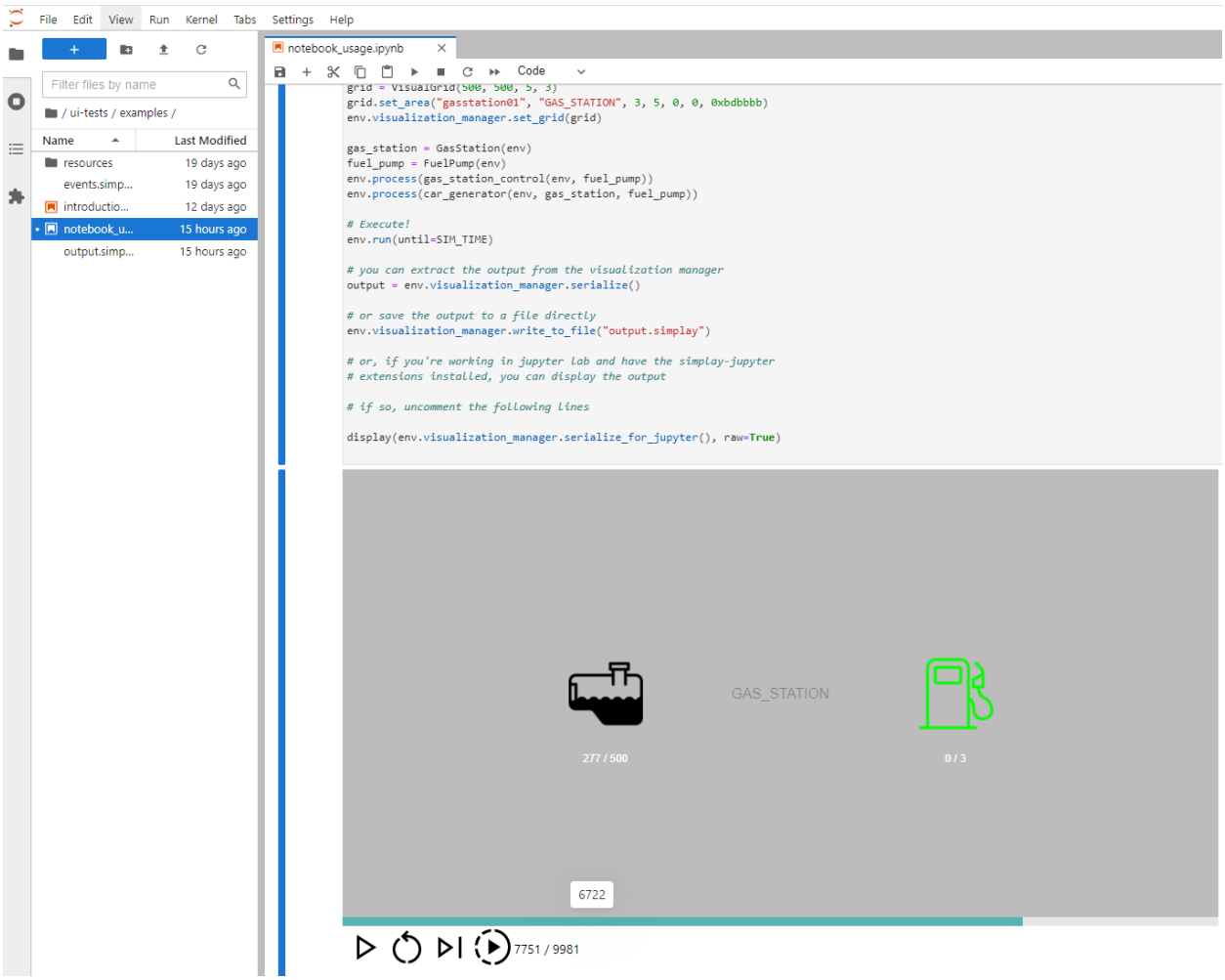


Abbildung 9.2: simplay-jupyter: Gas Station Notebook

Im Bild verwendete Icons: [26] [28]

[14] 13.7 User-Test Feedback
[15] 9.4 Dokumentation

Ausserdem ermöglicht `simplay-jupyter` die Darstellung von Files mit der Dateiendung `.simplay`, wie auf folgendem Screenshot ersichtlich:

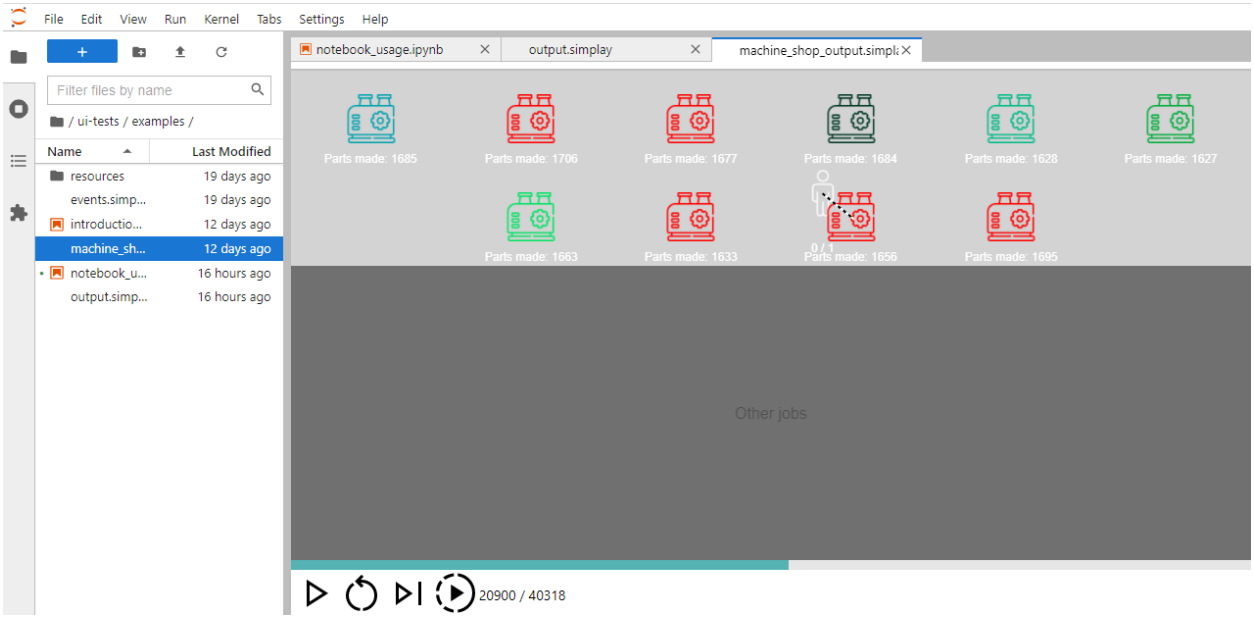


Abbildung 9.3: simplay-jupyter: machine_shop.simplay

Im Bild verwendete Icons: [32] [50]

In beiden Betrachtungsweisen stehen Buttons zur Kontrolle der Animation zur Verfügung. Die Interaktionen, wie in der Anforderung Animation der Simulation betrachten [16] beschrieben, stehen den Simulationsbetrachterinnen und -betrachtern zur Verfügung. Es wird sowohl der momentane Zeitpunkt der Simulation als auch die gesamte Anzahl Schritte angezeigt. Diese Ansicht kann auf Zeit umgestellt werden. Dann wird angenommen, dass es sich bei einem Timestamp um eine Sekunde handelt und daraus entsprechend die Zeit berechnet und angezeigt.

Eine barrierefreie Bedienung wird unterstützt. Die Buttons zum "Starten / Pausieren", "Zurücksetzen", "ein Schritt vorwärts", "Beschleunigen / Verlangsamen" sowie die Anzeige des momentanen und der totalen Zeitschritte, können durch die Navigation mit der Tastatur bedient werden und verfügen über beschreibenden Text. Der Zustand und die Veränderung der Visualisierung werden nicht als Text bereitgestellt. Falls Benutzende nicht imstande sind die Visualisierung zu betrachten, ist empfohlen den Zustand der Simulation mit Aufrufen von `print` auszugeben und diese Ausgabe anstelle der Visualisierung zu konsumieren.

Gemäss dem `Jupyter Accessibility Statement` erfüllt der Inhalt von JupyterLab nicht komplett die Anforderungen des "accessibility standard" [51]. Entwicklerinnen und Entwickler von Extensions werden dazu ermutigt, Barrierefreiheit unterstützende Strukturen zu verwenden. Mit den momentanen Kontrollstrukturen ist die Visualisierung barrierefrei steuerbar und unterstützt JupyterLab im Anbieten eines barrierefreien Benutzererlebnisses.

`simplay-jupyter` ist Teil des SimPlay Repository und der Source Code wird auf SimPlay - GitHub [45] gehostet. Wie der Source Code des Packages strukturiert ist und wie er verwaltet wird, ist im Kapitel Open-Source Projekt [17] beschrieben.

[16] 4.1.1 Animation der Simulation betrachten
[17] 9.5 Open-Source Projekt

9.4 Dokumentation

Die Dokumentation zur Verwendung von SimPlay wird mit Read the Docs [19] bereitgestellt. Die SimPlay Dokumentation ist unter simplay.readthedocs.io [44] abrufbar.

Diese Dokumentation enthält Instruktionen, wie das Package `simplay` zusammen mit dem Package `simplay-jupyter` verwendet wird. Die Dokumentation richtet sich an Benutzende, welche eine bestehende oder zukünftige Simulation um eine Animation erweitern, und das Resultat in JupyterLab betrachten möchten. Benutzende, welche einen eigenen Abspieler für ihre Simulation schreiben wollen und sich für das Package `simplay-web` interessieren, werden auf die entsprechende Dokumentation im README.md [47] von `simplay-web` verwiesen. Diese Trennung wird bewusst gemacht, sodass der grosse Teil von Benutzenden, welche eine Simulation animieren wollen, dieses Ziel möglichst einfach erreichen.

9.4.1 Struktur

Die Struktur und der Inhalt der Kapitel unterstützt Benutzende beim Erweitern einer Simulation um eine Animation. Anhand der User-Tests [18] wurde die Einfachheit und das Verständnis der Dokumentation überprüft und wo nötig entsprechend angepasst.

Die Dokumentation ist in die fünf folgenden Kapitel unterteilt:

9.4.1.1 Overview

Übersicht zu SimPlay, mit Beschreibung und GIF sowie wichtigen Links.

9.4.1.2 Getting Started

Installationseinleitung und Beschreibung zum Animieren von Simulation. Diese Seite richtet sich an alle Benutzenden von SimPlay, ist aber darauf ausgelegt, dass sich vor allem Neueinsteiger zurechtfinden.

9.4.1.3 Examples

Wie in der Vision [19] beschrieben, sind alle Beispiele von SimPy in animierter Form vorhanden. Die folgenden Abschnitte beinhalten Screenshots der Visualisierung aller SimPlay Beispiele.

[18] 13.7 User-Test Feedback
[19] 5 Vision

9.4.1.3.1 Bank Renege

```
Bank renege
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 0.2533 Customer01: Here I am
 3.0310 Customer00: Finished
 3.0310 Customer01: Waited  2.778
 4.1225 Customer01: Finished
11.5450 Customer02: Here I am
11.5450 Customer02: Waited  0.000
11.8475 Customer03: Here I am
14.0644 Customer04: Here I am
14.1473 Customer03: RENEGED after  2.300
16.2430 Customer04: RENEGED after  2.179
16.5547 Customer05: Here I am
19.1663 Customer05: RENEGED after  2.612
19.9920 Customer02: Finished
```

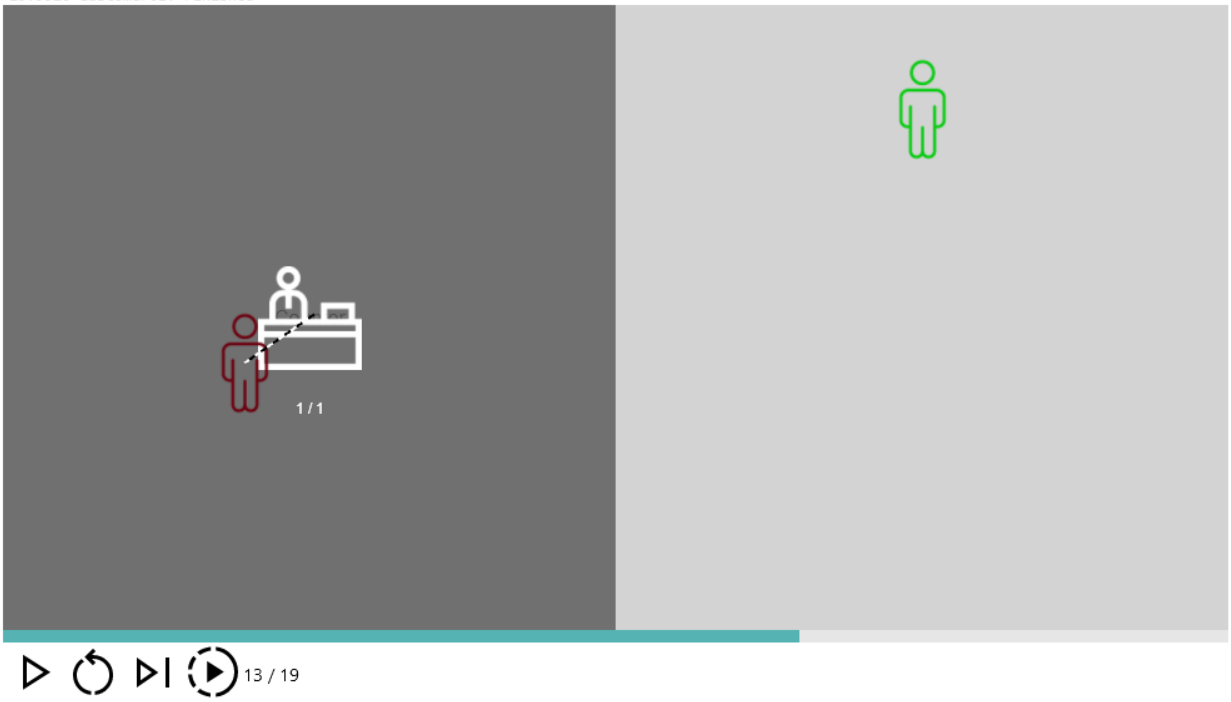


Abbildung 9.4: simplay-jupyter: Bank Renege

Im Bild verwendete Icons: [50] [52]

9.4.1.3.2 Carwash

```
Carwash
Check out http://youtu.be/fXXmeP9TvBg while simulating ... ;-)
Car 0 arrives at the carwash at 0.00.
Car 1 arrives at the carwash at 0.00.
Car 2 arrives at the carwash at 0.00.
Car 3 arrives at the carwash at 0.00.
Car 0 enters the carwash at 0.00.
Car 1 enters the carwash at 0.00.
Car 4 arrives at the carwash at 5.00.
Carwash removed 55% of Car 0's dirt.
Carwash removed 87% of Car 1's dirt.
Car 0 leaves the carwash at 5.00.
Car 1 leaves the carwash at 5.00.
Car 2 enters the carwash at 5.00.
Car 3 enters the carwash at 5.00.
Carwash removed 77% of Car 2's dirt.
Carwash removed 52% of Car 3's dirt.
Car 2 leaves the carwash at 10.00.
Car 3 leaves the carwash at 10.00.
Car 4 enters the carwash at 10.00.
Car 5 arrives at the carwash at 14.00.
Car 5 enters the carwash at 14.00.
Carwash removed 64% of Car 4's dirt.
Car 4 leaves the carwash at 15.00.
Carwash removed 82% of Car 5's dirt.
Car 5 leaves the carwash at 19.00.
```

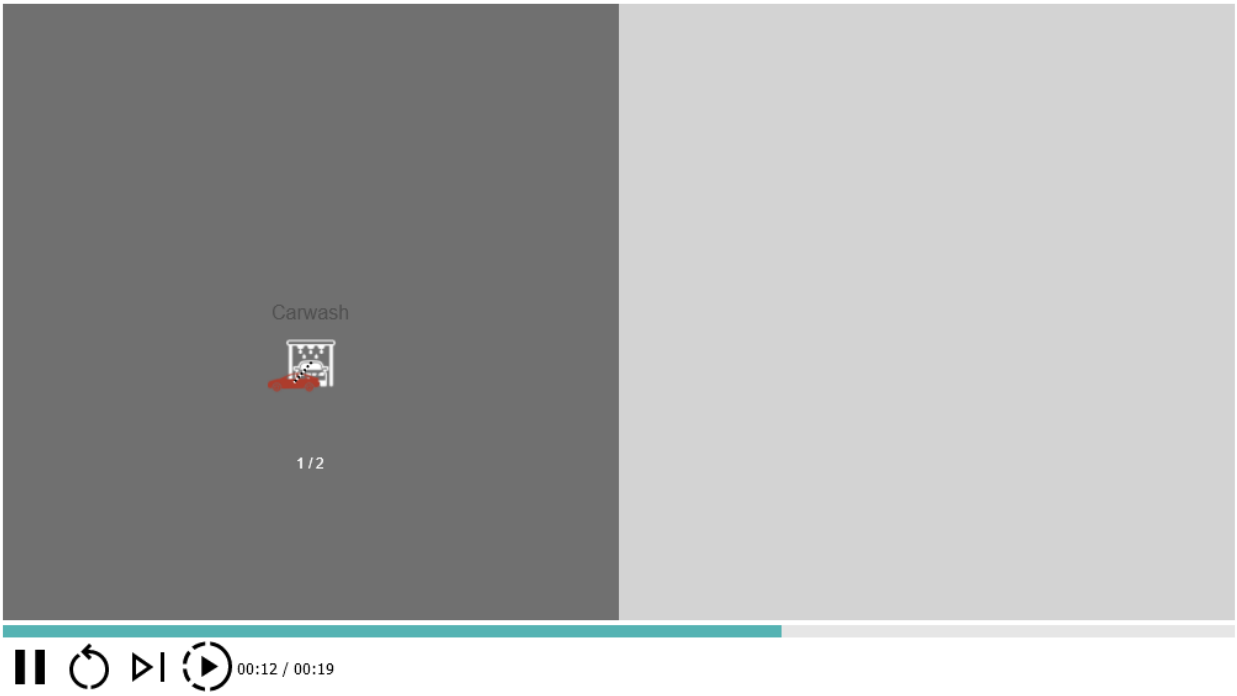


Abbildung 9.5: simplay-jupyter: Carwash

Im Bild verwendete Icons: [53] [27]

9.4.1.3.3 Machine Shop

```
Machine shop
Machine shop results after 4 weeks
Machine 0 made 3307 parts.
Machine 1 made 3285 parts.
Machine 2 made 3345 parts.
Machine 3 made 3377 parts.
Machine 4 made 3285 parts.
Machine 5 made 3278 parts.
Machine 6 made 3222 parts.
Machine 7 made 3292 parts.
Machine 8 made 3338 parts.
Machine 9 made 3338 parts.
```

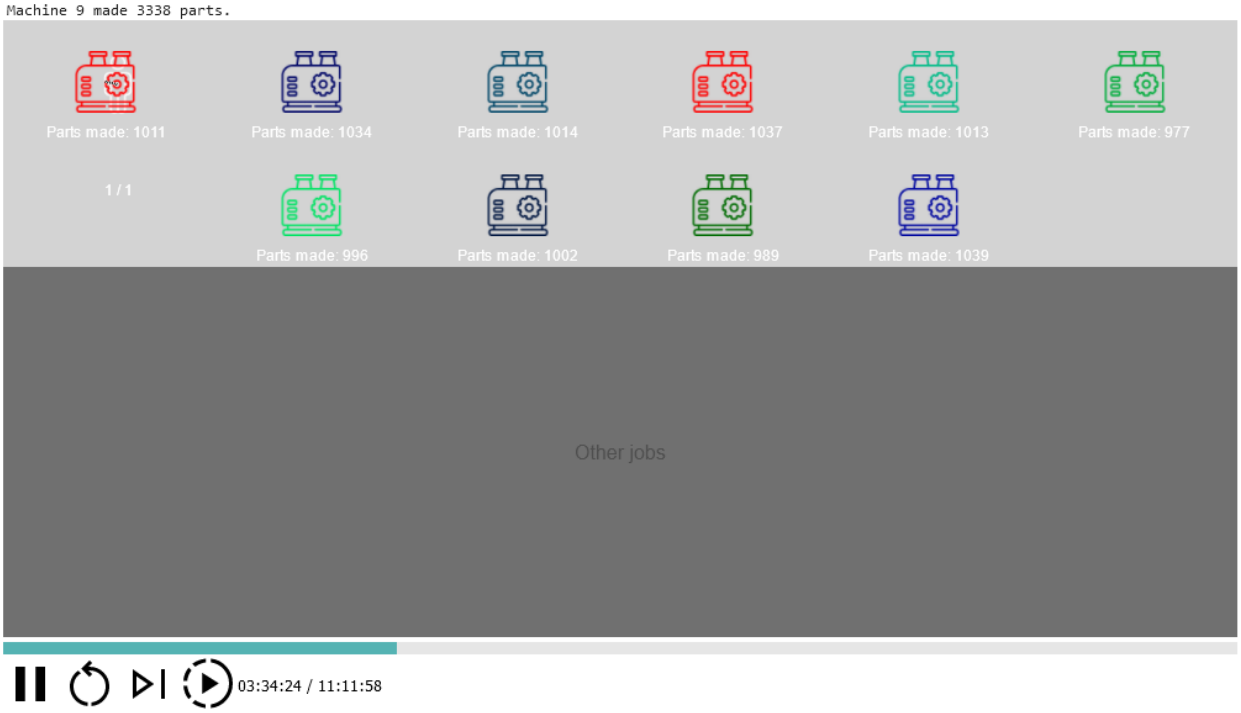


Abbildung 9.6: simplay-jupyter: Machine Shop

Im Bild verwendete Icons: [32] [50]

9.4.1.3.4 Movie Renege

```
Movie renege
Movie "Python Unchained" sold out 30.0 minutes after ticket counter opening.
  Number of people leaving queue when film sold out: 14
Movie "Kill Process" sold out 43.0 minutes after ticket counter opening.
  Number of people leaving queue when film sold out: 11
Movie "Pulp Implementation" sold out 40.5 minutes after ticket counter opening.
  Number of people leaving queue when film sold out: 16
```

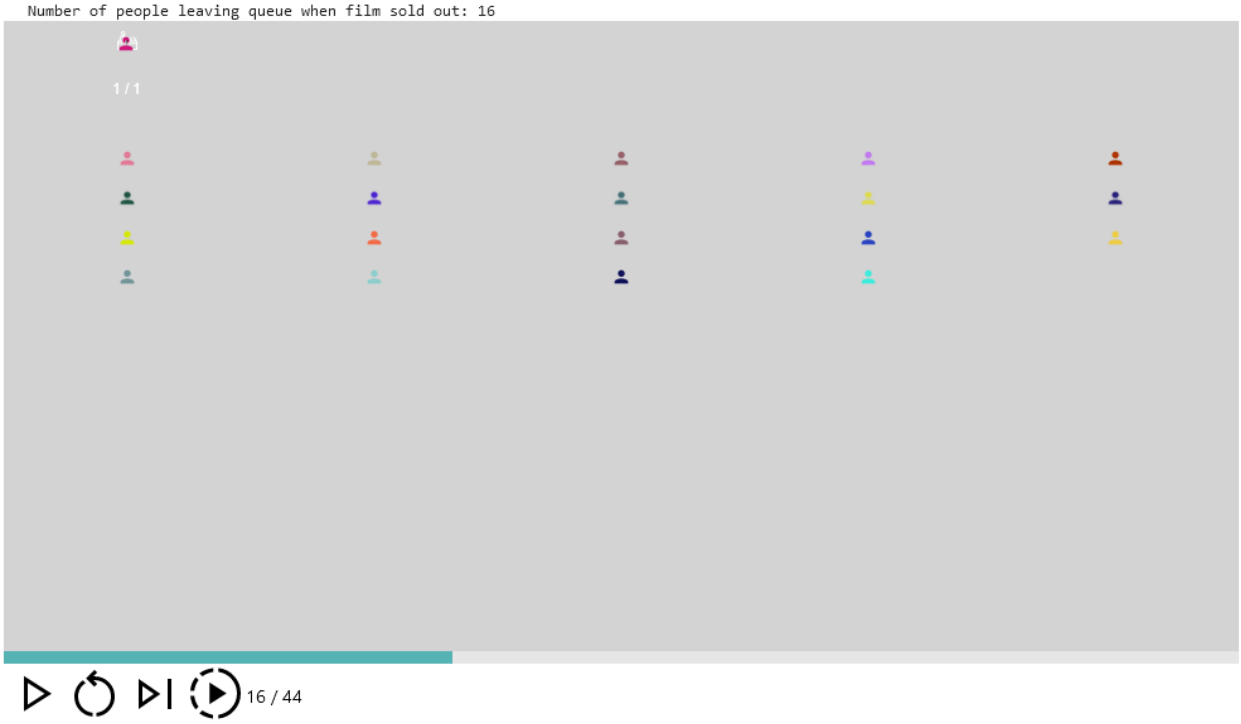


Abbildung 9.7: simplay-jupyter: Movie Renege

Im Bild verwendete Icons: [54] [52]

9.4.1.3.5 Gas Station

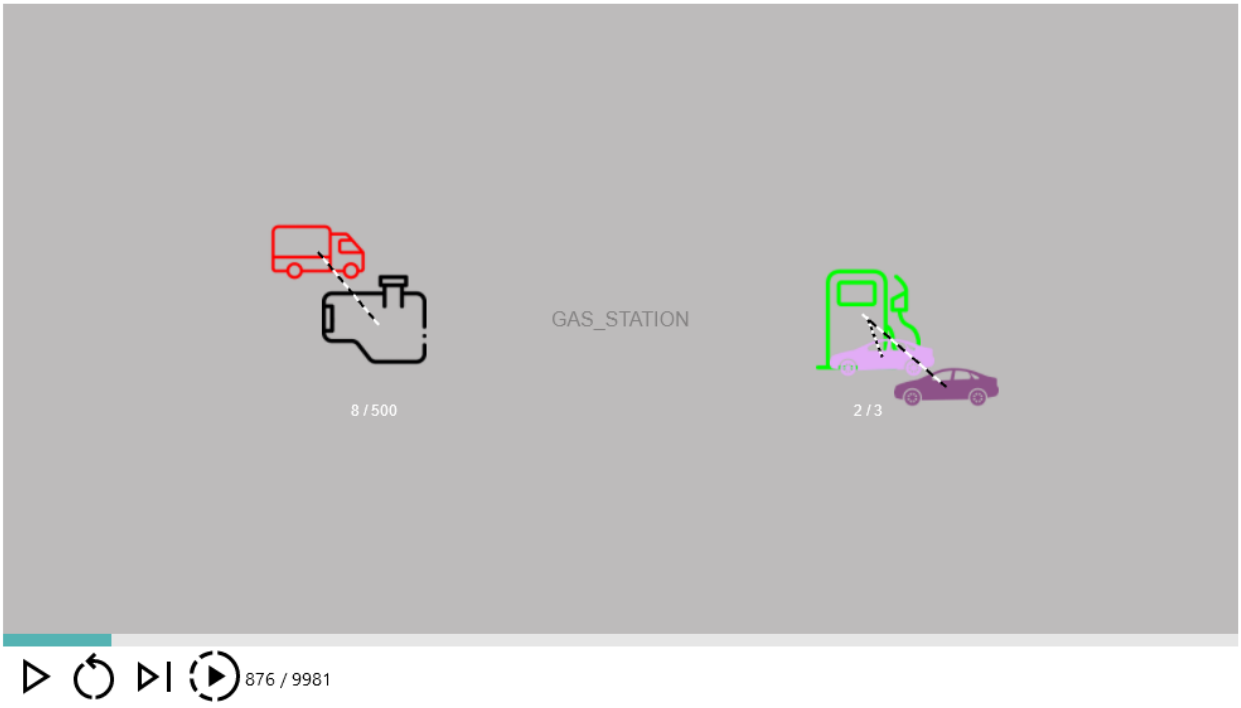


Abbildung 9.8: simplay-jupyter: Gas Station

Im Bild verwendete Icons: [26] [27] [28] [29]

9.4.1.3.6 Process Communication: Pipe

```
Process communication

One-to-one pipe communication

at time 6: Consumer A received message: Generator A says hello at 6.
at time 15: Consumer A received message: Generator A says hello at 15.
at time 23: Consumer A received message: Generator A says hello at 23.
at time 30: Consumer A received message: Generator A says hello at 30.
LATE Getting Message: at time 38: Consumer A received message: Generator A says hello at 37
at time 48: Consumer A received message: Generator A says hello at 48.
at time 55: Consumer A received message: Generator A says hello at 55.
at time 63: Consumer A received message: Generator A says hello at 63.
at time 74: Consumer A received message: Generator A says hello at 74.
LATE Getting Message: at time 82: Consumer A received message: Generator A says hello at 81
at time 92: Consumer A received message: Generator A says hello at 92.
```

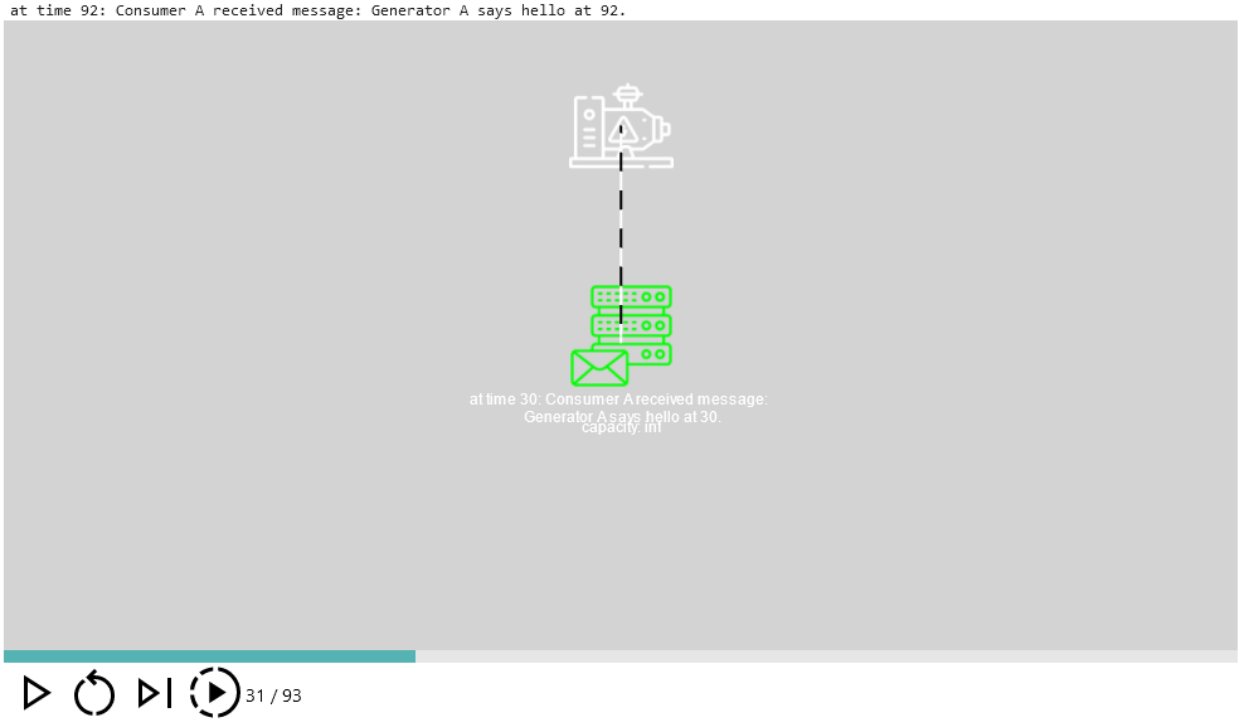


Abbildung 9.9: simplay-jupyter: Process Communication: Pipe

Im Bild verwendete Icons: [55] [56]

9.4.1.3.7 Process Communication: Broadcast Pipe

```
Process communication

One-to-many pipe communication

at time 6: Consumer A received message: Generator A says hello at 6.
at time 6: Consumer B received message: Generator A says hello at 6.
at time 14: Consumer A received message: Generator A says hello at 14.
at time 14: Consumer B received message: Generator A says hello at 14.
at time 21: Consumer A received message: Generator A says hello at 21.
at time 21: Consumer B received message: Generator A says hello at 21.
at time 32: Consumer A received message: Generator A says hello at 32.
at time 32: Consumer B received message: Generator A says hello at 32.
at time 39: Consumer A received message: Generator A says hello at 39.
at time 39: Consumer B received message: Generator A says hello at 39.
at time 47: Consumer A received message: Generator A says hello at 47.
at time 47: Consumer B received message: Generator A says hello at 47.
LATE Getting Message: at time 55: Consumer A received message: Generator A says hello at 54
LATE Getting Message: at time 55: Consumer B received message: Generator A says hello at 54
at time 65: Consumer A received message: Generator A says hello at 65.
at time 65: Consumer B received message: Generator A says hello at 65.
at time 75: Consumer A received message: Generator A says hello at 75.
at time 75: Consumer B received message: Generator A says hello at 75.
at time 82: Consumer B received message: Generator A says hello at 82.
LATE Getting Message: at time 83: Consumer A received message: Generator A says hello at 82
at time 92: Consumer A received message: Generator A says hello at 92.
at time 92: Consumer B received message: Generator A says hello at 92.
```

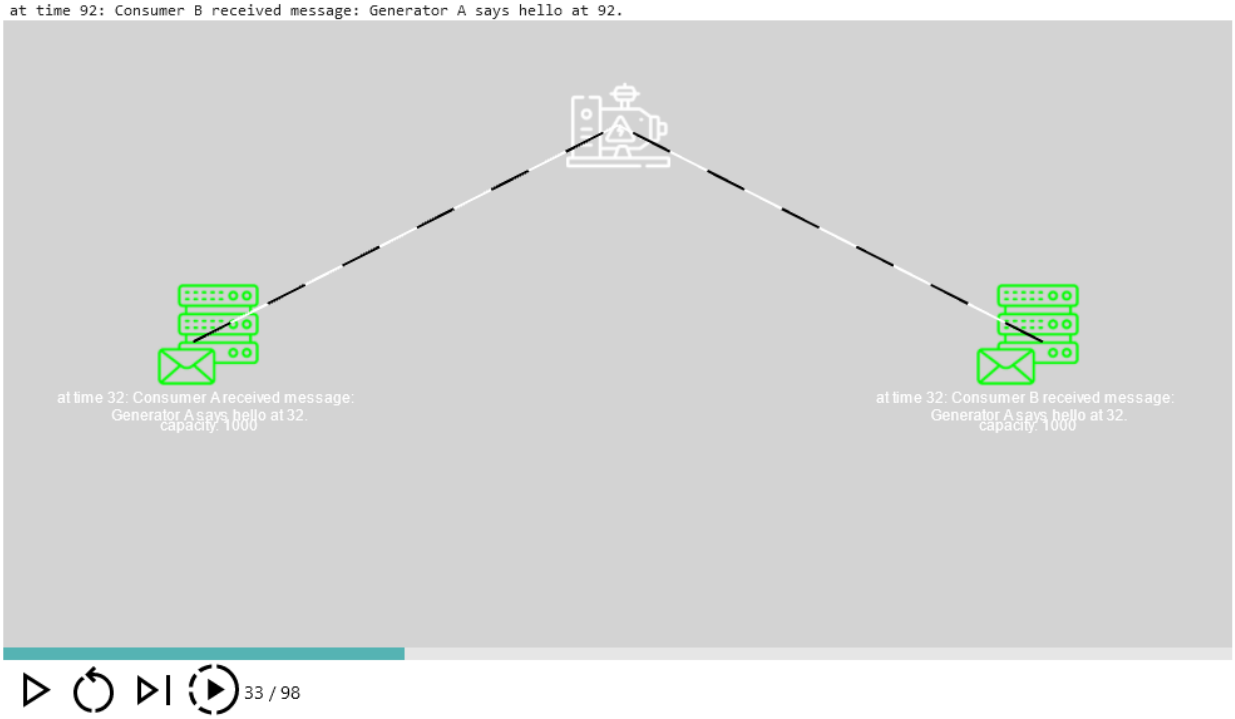


Abbildung 9.10: simplay-jupyter: Process Communication: Broadcast Pipe

Im Bild verwendete Icons: [55] [56]

9.4.1.3.8 Event Latency

```
Event Latency
Received this at 15 while Sender sent this at 5
Received this at 20 while Sender sent this at 10
Received this at 25 while Sender sent this at 15
Received this at 30 while Sender sent this at 20
Received this at 35 while Sender sent this at 25
Received this at 40 while Sender sent this at 30
Received this at 45 while Sender sent this at 35
Received this at 50 while Sender sent this at 40
Received this at 55 while Sender sent this at 45
Received this at 60 while Sender sent this at 50
Received this at 65 while Sender sent this at 55
Received this at 70 while Sender sent this at 60
Received this at 75 while Sender sent this at 65
Received this at 80 while Sender sent this at 70
Received this at 85 while Sender sent this at 75
Received this at 90 while Sender sent this at 80
Received this at 95 while Sender sent this at 85
```

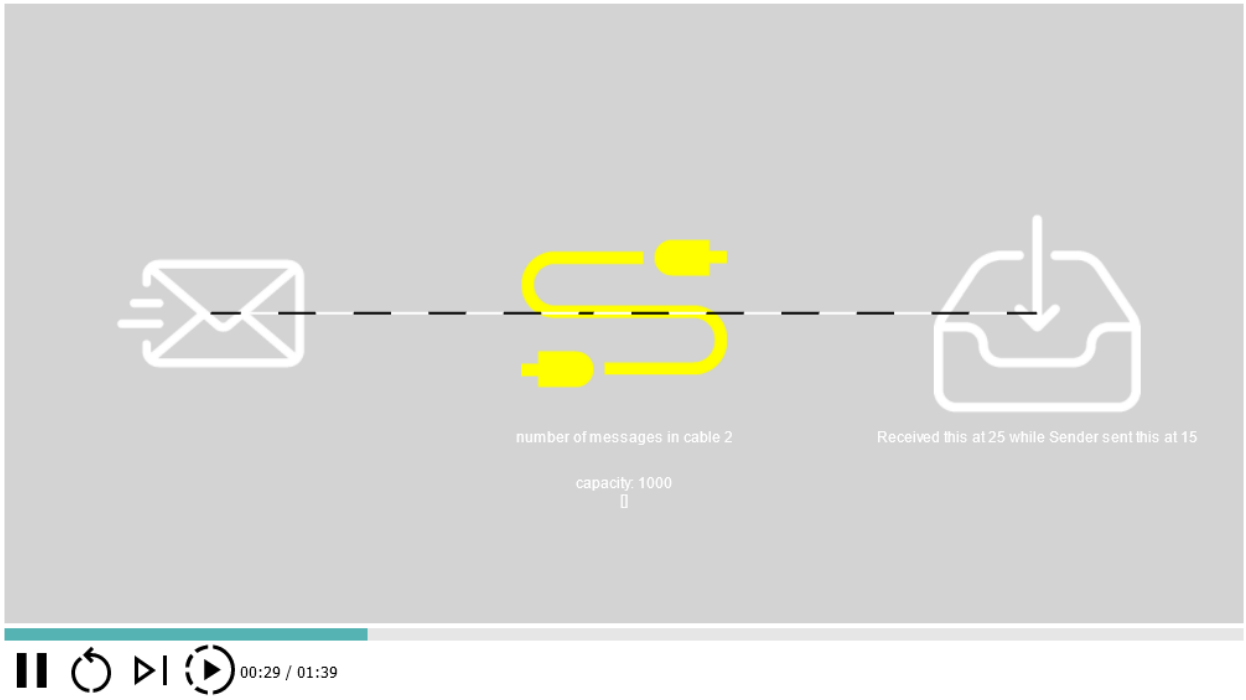


Abbildung 9.11: simplay-jupyter: Event Latency

Im Bild verwendete Icons: [57] [58] [59]

9.4.1.4 SimPlay in depth

Genauere Beschreibung zu den Themen `Container`, `Store` und `Resource`.

9.4.1.5 API Reference

Die Dokumentation des `simplay` Package, automatisch generiert mit Sphinx autosummary [23].

9.4.2 Installationsanleitung

Da die öffentlichen Anleitungen in Englisch verfasst sind, befindet sich im Anhang eine kurze deutsche Übersetzung der Installationsschritte [20] sowie die vollständige öffentliche Dokumentation [21].

9.5 Open-Source Projekt

Wie in der Zielsetzung beschrieben, soll dieses Projekt den gängigen Open-Source Standards entsprechen, unter anderem mit der Hoffnung eine durch die Community gestützte Weiterentwicklung zu ermöglichen. Eine gängige Plattform zum Hosten von Open-Source Projekte ist GitHub, weshalb dieses Projekt auf GitHub gehostet wird: SimPlay - GitHub [45].

Das Projekt umfasst insgesamt drei Packages welche in einem Repository gehalten werden. Um eine einfache Integration und Adaption zu ermöglichen, existiert eine öffentliche Dokumentation auf simplay.readthedocs.io [44].

9.5.1 Struktur

Der Repository-Root sieht wie folgt aus:

```
.
|-- .github
|
|-- docs
|
|-- src
    |-- simplay
    |-- simplay-web
     -- simplay-jupyter
```

9.5.1.1 .github

Im Ordner `.github` befinden sich die Konfigurationsdateien für GitHub Actions. Diese Actions sorgen dafür, dass neue Versionen der Pakete automatisch auf Package-Repositories wie PyPI [60] und npm [61] veröffentlicht werden und dass für Änderungen Überprüfungen durchgeführt werden.

9.5.1.2 docs

Im Ordner `docs` befindet sich die Dokumentation des Projekts. Diese Dokumentation wird automatisch bei jedem Push auf dem `main`-Branch aktualisiert und ist unter simplay.readthedocs.io [44] verfügbar. Weiter setzt das Projekt auf `autosummary`, um die Dokumentation, welche in den Python Modulen vorhanden ist, automatisch zu generieren und mit Read the Docs darzustellen. Read the Docs greift dabei standardmässig immer auf den aktuellen Zustand des `main`-Branches zu.

[20] 13.1 Installationsanleitung
[21] 13.5 Öffentliche Dokumentation

9.5.1.3 src

Im Ordner `src` befinden sich die drei Packages des Projekts. Diese Packages sind in drei Unterordner aufgeteilt:

- `simplay`
- `simplay-web`
- `simplay-jupyter`

Alle Packages in einem Repository zu halten, hat den Vorteil, dass die Packages untereinander leichter integriert werden können. Sollten zukünftig Probleme mit diesem Ansatz auftreten, ist es problemlos möglich, die Packages in einzelne Repositories auszulagern.

9.5.1.4 Testing

Das Projekt verwendet verschiedene Methoden, um die drei Packages abhängig von ihrem jeweiligen Ökosystem zu testen. Die Tests werden im Rahmen von Continuous Integration (CI) / Continuous Deployment (CD) automatisch ausgeführt.

9.5.1.4.1 simplay

Das Package `simplay` wurde mit dem PyPI-Standard aufgesetzt [62] und verwendet `pytest` für die Tests. Die Tests sind so strukturiert, dass pro Modul ein Test-Modul erstellt wird. Die Testdateien sind unterteilt in Testklassen, welche die jeweiligen Klassen testen und Testfunktionen, welche die jeweiligen Funktionen testen. Sowohl die Testdateien als auch die Testklassen und Testfunktionen sind mit einem `test_`-Präfix versehen. Die Unterteilung der einzelnen Testfunktionen in die zugehörigen Klassen dient in diesem Fall der Übersichtlichkeit und der besseren Strukturierung der Tests und nicht der Funktionalität. Diese Art wurde bewusst gewählt, sodass alle Testfunktionen selbst dafür verantwortlich sind, ihre Abhängigkeiten aufzubauen, ohne dass diese von anderen Testfunktionen beeinflusst werden. Dies hat zwar zur Folge, dass bei Änderungen an Konstruktorsignaturen ein erhöhter Wartungsaufwand besteht, gleichzeitig wird damit jedoch gewährleistet, dass die Tests lesbar, und resilient gegenüber Änderungen sind.

9.5.1.4.2 simplay-web

Für die Ausführung der Tests wird der von PixiJS bereitgestellte Test Runner `Floss` [9] verwendet. Die Strukturierung der Tests repliziert dabei die Strukturierung der Code-Dateien. Die Tests folgen derselben Struktur wie die Code-Dateien.

9.5.1.4.3 simplay-jupyter

`simplay-jupyter` ist die Integration mit JupyterLab und definiert Abhängigkeiten zu `simplay` und `simplay-web`. Deshalb beinhaltet diese Komponente Unit Tests und UI Tests. Mit den UI Tests wird das Zusammenspiel aller drei Komponenten innerhalb von JupyterLab getestet. Die UI Tests werden verwendet, um zu überprüfen, ob die Integration mit JupyterLab funktioniert. Die Unit Tests folgen derselben Struktur wie die Code-Dateien.

9.5.1.5 Continuous Integration (CI) / Continuous Deployment (CD)

Das Projekt verwendet GitHub Actions [63] um die CI und CD zu unterstützen.

9.5.1.5.1 Continuous Integration (CI)

Zur Gewährleistung einer möglichst problemlosen Zusammenarbeit mehrerer Contributors, empfiehlt es sich, CI anzuwenden. Ob CI angewendet wird, hängt letztlich aber von den Entwickelnden ab. Sie sind dafür verantwortlich, ihren Code regelmässig zu commiten, keinen kaputten Code ins Repository hochzuladen, kaputte Builds zu reparieren, automatisierte Tests zu schreiben, Tests und Inspections durchzuführen und zu verhindern, dass sie auf fehlerhaftem Code weiterarbeiten [64, p. 23].

GitHub bietet die Möglichkeit, automatisiert Kommandos durchzuführen und das Resultat den Entwickelnden bereitzustellen. Um CI bestmöglich zu unterstützen, setzt das Simplay Projekt mehrere GitHub Actions ein. Für jedes der drei Packages stehen entsprechende Jobs zur Verfügung. Sobald ein Push Request mit Änderungen im entsprechenden Package erstellt wird, werden bei der initialen sowie jeder späteren Änderung, die entsprechenden GitHub Actions ausgelöst.

Im Buch `Continuous integration : improving software quality and reducing risk` sind im Kapitel `Features of CI` die Prozesse für optimales Continuous integration beschrieben [64, Ch. 1]. Mit den GitHub Actions für die drei Packages werden die Prozesse wie folgt unterstützt:

Source Code Compilation

Alle Abhängigkeiten werden installiert und das Build Kommando ausgeführt. Damit wird überprüft, ob der Code kompiliert. Zusätzlich wird bei diesem Schritt ein Artefakt erstellt, welches auf dem entsprechenden Package-Repository veröffentlicht werden kann.

Inspection

Es wird eine statische Analyse des Codes durchgeführt.

Testing

Es werden die automatischen Unit Test und für das Package `simplay-jupyter` die UI Tests durchgeführt.

Deployment

Ist im nächsten Kapitel beschrieben.

Database Integration

Wurde bewusst verzichtet, da das Projekt über keine Datenbank verfügt.

Documentation

Bei jeder Änderung auf dem `main` Branch wird die Dokumentation aktualisiert.

Feedback

Durch die Verwendung von GitHub Actions ist es gewährleistet, dass eine Entwicklerin oder ein Entwickler Feedback zu den Prozessen erhält.

9.5.1.5.2 Continuous Deployment

Da es sich bei dem Produkt um öffentliche verfügbare Packages handelt, wird darauf verzichtet, für jede Änderung ein Artefakt in den entsprechenden Packagemanagern zu veröffentlichen. Die Artefakte werden automatisch publiziert, sobald via GitHub UI ein Release erstellt wird. Da sich die drei Packages im selben Repository befinden, wird für alle dieselbe Version verwendet. Die Versionsnummern folgen dem `Semantic Versioning` Pattern [65].

9.5.2 Metriken

Um eine Übersicht über den Inhalt des Projekts zu geben, sind in diesem Kapitel Metriken der Projekte aufgelistet.

9.5.2.1 Lines of Code

Die "Lines of Code" wurden mit dem Tool sloc [66] ermittelt.

Package	LOC (physical)	LOC (logical)	LOC (comment)	LOC (blank)
simplay	1'582	884	523 [22]	223
simplay (tests)	1080	919	0	161
simplay-web	1'550	1'374	45	133
simplay-web (tests)	3'174	2809	106	260
simplay-jupyter	950	820	16	114
simplay-jupyter (tests)	711	624	7	82
Total	**9'047**	**7'430**	**697**	**973**

9.5.2.2 Number of Merged Pull Requests

68 [67]

9.5.2.3 Code Coverage

Der Anhang enthält im Abschnitt Code Coverage [23] eine komplette Ausgabe ebendieser.

Package	Statements	Branches	Functions	Lines
simplay	97%			
simplay-web	96.63%	93.68%	99.13	96.95%
simplay-jupyter	97.34%	93.33%	93.8%	97.31%

9.6 Einfluss auf die Laufzeit von Simulationen

Die Leistungsfähigkeit, obwohl nicht explizit als nichtfunktionale Anforderung erwähnt, ist ein wichtiger Aspekt des Produkts. Aufgrund des teilweise iterativen Ansatzes von Simulationsexperimenten ist es wichtig, dass der Einsatz von `simplay` nicht zu exponentiellen Veränderungen in der Laufzeit führt. Die visuellen Zustandsänderungsdeklarationen, welche mit `simplay` vorgenommen werden, führen im Regelfall zu einer Erhöhung der Simulationslaufzeit. Dies allerdings nur, wenn die reine `simpy` Simulation bereinigt wird, also I/O vermieden wird. Die Beispiele welche von `simpy` zur Verfügung gestellt werden, haben als einzige Möglichkeit zur Rapportierung der Resultate die Konsolenausgabe. Da Konsolenausgabe gezwungenermassen immer I/O verursacht, gibt es Situationen in den Simulationen, in denen die angepassten Beispiele mit Einsatz von `simplay`, aber ohne Konsolenausgabe, schneller sind. Da `simplay` ein Reporting ohne Konsolenausgabe ermöglicht, ist es teilweise sogar möglich, die Simulationslaufzeit zu verbessern. In der Realität ist es so, dass die Veränderung der Laufzeit stark von der gewählten Simulation abhängt.

[22] Diese grosse Anzahl an Kommentaren stammt von den docstrings
[23] 13.4 Code Coverage

Die untenstehenden Messungen sind für das Beispiel Carwash:

SIMTIME	Carwash mit simplay	Carwash ohne simplay	Carwash ohne simplay, und ohne `print`
1	0.000397	0.000441	0.000055
10	0.000490	0.000825	0.000092
100	0.001186	0.004063	0.000428
1000	0.007569	0.043669	0.003284
10'000	0.074498	0.411762	0.031244

(Resultate in Sekunden)

Was auffällt, ist dass die Simulation mit Einsatz von `simplay` bei einer `SIMTIME` von 10'000 im Vergleich zum originalen Beispiel drastisch schneller ist. Dies liegt daran, dass die Carwash Simulation sehr viele, schnell aufeinanderfolgende `print` Ausgaben hat, wodurch die Laufzeit leidet.

Zum Vergleich die untenstehende Tabelle mit dem Beispiel Machine Shop:

SIMTIME	Machine Shop mit simplay	Machine Shop ohne simplay	Machine Shop ohne simplay, und ohne `print`
1	0.000501	0.000200	0.000119
10	0.000594	0.000245	0.000172
100	0.001199	0.000649	0.000547
1000	0.007221	0.003726	0.003607
10'000	0.066416	0.035249	0.034306

(Resultate in Sekunden)

Das Beispiel der "Machine Shop" Simulation von `simpy` enhält während der Simulationsdurchführung keine `print` Aufrufe. Erst nach Abschluss der Simulation werden die Resultate der jeweiligen Maschinen ausgegeben. Entsprechend erhöht sich die Laufzeit unter Einsatz von `simplay`.

Der Anhang enthält im Abschnitt Messungen zur Laufzeitveränderung [24] weitere Messresultate

9.6.1 Fazit

Die Laufzeitveränderung durch `simplay` ist konstant. Dies bedeutet, dass die Laufzeitveränderung nicht exponentiell wächst, sondern linear. Wenn die Ausführungsdauer der Simulation ein Aspekt der Entwicklung ist, muss davon ausgegangen werden, dass eine Simulation rund doppelt so lange dauert, wenn `simplay` verwendet wird. Diese Aussage lässt sich nicht verallgemeinern, da die effektive Veränderung abhängig von der Frequenz von Zustandsdeklarationen ist. Die Messung der Beispiele zeigt aber, dass dies eine realistische Annahme ist.

[24] 13.11 Messungen zur Laufzeitveränderung

Kapitel 10

Fazit

Mit SimPlay ist eine Möglichkeit geschaffen, Simulationen auf Basis von SimPy einfach und intuitiv innerhalb von JupyterLab zu erweitern, auszuführen und anzusehen. Die verschiedenen Elemente der Simulation können mit wenigen Zeilen Code animiert werden. Mit der Wahl von PixiJS als Rendering Engine ist eine flexible und performante Basis geschaffen, welche moderne Möglichkeiten der Webentwicklung nutzt. Die Einbettung in JupyterLab ermöglicht es, SimPlay in der Lehre als Vermittlungs- und Lernmittel zu verwenden. User-Tests zeigen, dass die Strukturierung der API verständlich ist und der Einsatz von SimPlay keine langwierige Einarbeitung erfordert. Die vorhandenen Kontrollstrukturen zur Beschleunigung und Verlangsamung der Simulation erlauben es, kritische Zeitpunkte genau zu untersuchen. Mit der geschaffenen Möglichkeit zur Animation können Entwicklerinnen und Entwickler sowie Kundinnen und Kunden die Simulationen interaktiv erleben und verstehen. Durch die Strukturierung als Open-Source Projekt können Entwicklerinnen und Entwickler die Entwicklung von SimPlay mitgestalten.

SimPlay ist nach wie vor in einem frühen Entwicklungsstadium. Grundlegende Features sind vorhanden, allerdings gibt es diverse Ideen und Möglichkeiten, das Projekt weiterzuentwickeln.

Kapitel 11

Ausblick

Das Produkt ist Open-Source, weshalb die Entwicklung stetig weiter geht. Im Rahmen dieser Bachelorarbeit wurde die erste Major-Version erstellt und veröffentlicht. Damit die Weiterentwicklung von SimPlay optimal unterstützt werden kann, sind alle offenen Tasks als Issues im öffentlich verfügbaren Simplay GitHub Repository [68] erfasst.

In diesem Kapitel werden die bedeutendsten zukünftig möglichen Änderungen beschrieben.

11.1 Animierte Bewegungen

GitHub-Issue 75 [69]

Momentan werden Positionsänderungen über verschiedene Methoden, wie beispielsweise `self.is_at(1, 1)` oder `self.is_near(resource)`, ausgelöst. Diese Positionierungsänderungen haben alle gemeinsam, dass die Änderung sofort ausgeführt wird. Das heisst, dass für die Bewegung von `x` nach `y` keine Zeit benötigt wird. Beim Implementieren der SimPy Beispiele hat sich herausgestellt, dass die Bewegung selbst selten eine Rolle in der Simulation einnimmt, sondern das vor allem wichtig ist, dass die Position geändert werden kann. Flüssige Bewegungen sind auch deshalb nicht Teil der Anforderungen an Version 1.0.0.

Eine möglichst prägnante Visualisierung sollte über flüssige Bewegungen verfügen. Wichtig ist, dass in diesem Fall den Entwickelnden der Simulation klar ist, dass für die Bewegung entsprechend Zeit zur Verfügung gestellt wird.

Die Funktionalität könnte über Code, wie er unten vorliegt, ausgelöst werden.

```
1  ...
2  moving_time = 1
3  x = 3
4  y = 3
5  self.is_moving_to(x, y, moving_time)
6  self.env.timeout(moving_time)
7  path = [(1, 1), (2, 1), (2, 2)]
8  self.is_moving_to(x, y, moving_time, path)
9  self.env.timeout(moving_time)
10 ...
```

Der bisherige Ansatz erlaubt es, eine Bewegung eines Objekts von einer Position zu einer anderen Position zu visualisieren. Es würde sich auch anbieten, dass während dieser Bewegung das Aussehen des Objekts verändert werden kann, sodass beispielsweise für eine Person eine fliessende gehende Bewegung dargestellt wird. PixiJS unterstützt eine solche Funktionalität mit PIXI.AnimatedSprite [70], welches bereits von SimPlay verwendet

wird. SimPlay müsste um eine Schnittstelle erweitert werden, welche es erlaubt, die entsprechenden Frames für die Bewegung zu definieren, in welcher Reihenfolge die Frames angezeigt werden und wie lange.

11.2 Fenster für print Output

GitHub-Issue 76 [71]

GitHub-Issue 77 [72]

Personen, welche mit SimPy arbeiten, sind sich gewohnt, Ausgaben mit `print` Aufrufen zu erzeugen. Während dem Betrachten der Animation der Simulation werden diese Ausgaben nicht angezeigt. Die Idee dieser Anforderung ist es, eine zusätzliche API für die Ausgabe von Nachrichten hinzuzufügen. Diese Nachrichten verfügen dann ebenfalls über einen Timestamp und werden als Teil der Animation angezeigt. Die Nachrichten könnten beispielsweise in einem transparenten Fenster über dem Grid angezeigt werden. Falls Benutzende die Nachrichten sehen möchten, kann dieses Fenster aufgeklappt werden und sonst entsprechend versteckt werden.

Die Funktionalität könnte über Code, wie er unten vorliegt ausgelöst werden.

```
self.logs("Hello World")
```

Die Struktur der Ausgabe könnte folgende Struktur haben:

```
[Timestamp] [OriginatorID] [Message]
```

Weiter könnte es interessant sein, Nachrichten dieser Art direkt in der Visualisierung darzustellen. Komponenten in der Visualisierung könnten dabei in Form von Sprechblasen Nachrichten anzeigen. Die API welche die Nachrichten anzeigt könnte dabei folgendermassen aussehen.

```
self.is_speaking(text, duration_in_steps)
```

Mit dem Parameter `duration_in_steps` würde dabei die Anzeigedauer der Sprechblase definiert werden.

11.3 Default-Icons

GitHub-Issue 72 [73]

Aus dem User-Test ist hervorgegangen, dass für kleine, simple Simulationen der Aufwand für die Suche von Icons ein beträchtlicher Teil der Zeit für die Erweiterung der Animation einnimmt. Daher könnte `simplay` "Default-Icons" mitliefern, welche die Suche nach passenden Bildern (zumindest teilweise) überflüssig machen. Für diese Erweiterung existieren zwei verschiedene Ansätze.

11.3.1 Idee

Der `VisualizationManager` von `simplay` wird um eine Methode erweitert, welche `base64` kodierte Bilder als Parameter entgegennimmt. Der `base64` String kann dabei aus beliebiger Quelle stammen.

11.3.2 Icons in `simplay`

Die Icons werden als `base64` Strings in der `simplay` Bibliothek hinterlegt. Diese können dem `VisualizationManager` übergeben werden.

Folgender Code zeigt die Idee:

```
from simplay import SimplayIcon

env.visualization_manager.register_base64('MONKEY', SimplayIcon.MONKEY)
env.visualization_manager.register_base64('BANANA', SimplayIcon.BANANA)
```

Dieser Ansatz bietet den Vorteil, dass es nicht notwendig ist, eine zusätzliche Installation durchzuführen, um die Icons zu nutzen. Weiter ist die Maintainability für das gesamte Open-Source Projekt geringer.

11.3.3 Icons in eigenem Package

Die Icons werden als `base64` Strings in einem separaten Package hinterlegt. Dieses Package muss zusätzlich zu `simplay` installiert werden. Die Icons können dem `VisualizationManager` übergeben werden.

Folgender Code zeigt die Idee:

```
1 from simplay_icons import SimplayIcon
2
3 env.visualization_manager.register_base64('MONKEY', SimplayIcon.MONKEY)
4 env.visualization_manager.register_base64('BANANA', SimplayIcon.BANANA)
```

Bei einer direkten Mitlieferung von Icons im `simplay` Paket, wird das Paket sehr viel grösser. Es ist nicht attraktiv, dass die Icons in der `simplay` Bibliothek liegen, da diese nicht für die Benutzung der Bibliothek notwendig sind, sondern nur für die Darstellung der Simulation, und kein funktionaler Bestandteil der Bibliothek sind.

11.3.4 Fazit

Die Bereitstellung von Default-Icons ist für Benutzende attraktiv und senkt die Hürde für die Erstellung von Animationen. Die Frage ist, ob die Icons in der `simplay` Bibliothek oder in einem separaten Package hinterlegt werden sollen. Weiter muss festgelegt werden, welche Icons bereitgestellt werden sollen. Die Spannweite kann von simplen geometrischen Formen wie Kreisen, Dreiecken oder Quadraten, bis hin zu Bildern, wie Affen, Bananen oder Flugzeugen reichen. Für das Erstellen von Bildern wäre das Projekt auf die Mithilfe der Community angewiesen.

11.4 Automode für Gridgrösse

GitHub-Issue 78 [74]

Die Grösse des Grids wird momentan in Pixel angegeben. Dafür wird folgender Aufruf verwendet (für ein 1000 Pixel breiten und 500 Pixel hohen Grid): `grid = VisualGrid(1000, 500, 10, 5)`.

Dieser Aufruf bietet den Benutzenden von SimPy die ganze Kontrolle über die Grösse des Grids und ermöglicht es auch, dementsprechend absolute Positionen zu definieren. Der Nachteil dieses Ansatzes ist, dass bei unterschiedlichen Bildschirmgrössen die Visualisierung grösser als der verfügbare Platz sein kann.

Die Idee dieses Features ist es, einen "Auto Mode" anzubieten, welcher entsprechend die maximale verfügbare Breite und Höhe verwendet. Damit wird eine saubere Darstellung der Visualisierung auf unterschiedlichen Bildschirmen gewährleistet sowie ein "Full Screen Mode".
Ein entsprechender Aufruf könnte folgendermassen aussehen: `grid = VisualGrid('auto', 'auto', 10, 5)`. Für die Definition der automatischen Höhe müsste ermittelt werden, wie hoch der Bereich eines Notebook innerhalb von JupyterLab ist, ohne das eine Scrollbar verwendet wird. Dieser Ansatz könnte schwierig umzusetzen sein. Da ein Notebook aus einer beliebigen Anzahl Zellen besteht, ist die Höhe nicht begrenzt und kann deshalb auch nicht mit `height: 100%` definiert werden. Um dieses Problem zu umgehen, kann eine absolute Höhe angegeben werden, da ein Notebook immer über einen Scrollbereich verfügt spielt das keine Rolle für die Darstellung auf unterschiedlichen Bildschirmen. Als Alternative kann ein Default Wert definiert werden oder ein Verhältnis von Breite und Höhe, mit welchem die Höhe berechnet werden kann.

Der Nachteil des "Auto" Mode ist, dass die Positionierung auf unterschiedlichen Bildschirmen unterschiedlich ausfällt. Je nach Simulation ist eine absolute, sich nicht ändernde Positionierung aber essenziell. Deshalb ist der "Auto Mode" eine ergänzende Funktionalität, aber kein Ersatz für das Definieren von absoluter Breite und Höhe.

11.5 Bild Grösse definieren

GitHub-Issue 79 [75]

Die Grösse eines Bildes ist davon abhängig, wie gross die Zelle ist, in welcher sich das Bild befindet. Je mehr Zellen existieren, desto kleiner ist das Bild. Je nach Simulation werden viele Zellen benötigt, um geringe Bewegungen oder genaue Positionierungen zu ermöglichen. Falls eine solche Simulation aber nur aus wenigen Akteuren besteht, ist es für eine ansprechendere Visualisierung oft hilfreich, die Bilder grösser darzustellen.

Für das Definieren der Grösse bieten sich zwei Varianten an. Die Grössen kann entweder in Pixel oder in Zellen angegeben werden. Für das Definieren der Grösse in Pixel sind folgende Aufrufe denkbar:

```
1 height_in_px = 100
2 width_in_px = 100
3 self.has_visual_heigth(height_in_px)
4 self.has_visual_width(width_in_px)
5 self.has_visual_size(height_in_px, width_in_px)
```

Die Aufrufe, welche nur Breite oder Höhe setzen, verwenden für das Definieren des nicht gesetzten Werts das bisherige Verhältnis.

Für das Definieren der Grösse in Zellen ist folgender Aufrufe denkbar:

```
1 cols = 3
2 rows = 3
3 self.has_visual_size(cols, rows)
```

Falls beide Varianten angeboten werden, müssen die Namen entsprechend angepasst werden. Der Vorteil der Variante mit der Angabe in Zellen ist, dass durch dieses Abstraktionslevel gewährleistet ist, dass Änderungen der absoluten Grösse des Animationsgrids und entsprechend eine Änderung der Zellengrösse berücksichtigt werden.

11.6 Performanceoptimierung

GitHub-Issue 80 [76]

GitHub-Issue 81 [77]

Bei der Animation ist es für eine möglichst ansprechende Darstellung wichtig, dass Änderungen mit möglichst wenig Verzögerung dargestellt werden. Um das zu gewährleisten, ist es sinnvoll, “teuere” Funktionalitäten zu kennen, und wenn möglich anzupassen. Eine Funktionalität mit entsprechendem Potenzial ist jene des Zurückspulens. Falls zurückgespult wird, wird die Animation zurückgesetzt und alle Events bis zum ausgewählten Zeitpunkt werden erneut durchlaufen. Da diese Events aber zuvor bereits verarbeitet wurden, sind Massnahmen zur Optimierung denkbar. Um diesen teuren Aufwand zu verhindern, könnten Keyframes erstellt werden. In diesem Kontext ist ein Keyframe der Zustand der gesamten Visualisierung zu einem spezifischen Zeitpunkt. Ein solches Keyframe könnte beispielsweise alle 5000 Schritte erstellt werden. Der zuvor teure Aufwand zur erneuten Berechnung aller Schritte würde in diesem Fall um einiges günstiger und nie teurer als das Berechnen von 4999 Schritten werden.

11.7 Fehlermeldungen

GitHub-Issue 82 [78]

Fehler werden momentan in der Console der Development Tools angezeigt. Development Tools werden von den gängigsten Browsern zur Verfügung gestellt, wie beispielsweise von Chrome [79]. Für Webentwicklerinnen und -entwickler ist das ein bekannter Ort für Fehlermeldungen. Betrachterinnen und Betrachter der Simulation werden die Fehlermeldungen nicht sehen, es sei denn, sie kennen und öffnen die Development Tools. Die Visualisierung sollte so erweitert werden, dass Fehlermeldungen, wenn beispielsweise das Zurücksetzen oder Pausieren nicht funktioniert hat, angezeigt werden können.

Die Fehlermeldungen sollten aussagekräftig und Anweisungen für das weitere Vorgehen enthalten, wie beispielsweise, dass die Seite neu geladen werden sollte.

11.8 Levels mit Frames verbinden

GitHub-Issue 83 [80]

Momentan sind Level und Nutzungsänderungen von Ressourcen und deren Darstellung unabhängig voneinander. Das heisst, eine Änderung des Levels wird nur visuell repräsentiert, wenn danach ein entsprechender Aufruf an SimPlay stattfindet. Beim Implementieren der SimPy Beispiele hat sich gezeigt, dass solche Events oft aufeinanderfolgen und Helferfunktionen dafür nützlich sind.

SimPlay kann zum Unterstützen dieser Funktionalität für `Container` um folgende API erweitert werden:

```
change_direction = ChangeDirection.DOWN
threshold = 50
frame_nr = 3
mycontainer.add_level_frame_rule(change_direction, threshold, frame_nr)

change_direction = ChangeDirection.UP
threshold = 200
frame_nr = 2
mycontainer.add_level_frame_rule(change_direction, threshold, frame_nr)
```

und für `Resource`:

```
change_direction = ChangeDirection.DOWN
threshold = 2
frame_nr = 1
myresource.add_utilization_frame_rule(change_direction, 2, 1)
```

Damit wird gewährleistet, dass für eine Änderung des Levels oder der Nutzung in eine spezifische Richtung automatisch das Frame angepasst wird, ohne dass dieser Aufruf für jede Änderung einzeln definiert werden muss.

11.9 Export

GitHub-Issue 74 [81]

Für flexiblere Verwendung der Animation, ist es wünschenswert, dass die Animation als Video oder GIF exportiert werden kann. Das Resultat des Exports kann leicht und ohne Installation von JupyterLab betrachtet werden, und es ist möglich, die Animation in anderen Kontexten zu verwenden. Obschon die Interaktivität verloren geht, wird dafür Portabilität gewonnen.

Im Rahmen dieser Arbeit werden keine konkreten technischen Lösungen dafür vorgeschlagen. Es ist vorstellbar, dass `simplay-web` im Rahmen der `SimulationSpooler` API um entsprechende Funktionen erweitert wird. Die Methodensignatur könnte wie folgt aussehen:

```
exportAnimation(exportOptions: ExportOptions): Promise<blob>;

interface ExportOptions {
    format: "gif" | "mp4";
    fps: number;
    width: number;
    height: number;
    animationSpeed: number;
}
```

Ein naheliegender Ansatz für die Umsetzung dieser Funktionalität ist es, die einzelnen Frames der Animation zu speichern, und diese dann zu einem Video oder GIF zu encodieren.

11.10 Binder

GitHub-Issue 73 [82]

Die Plattform Binder [83] erlaubt es, Jupyter Notebooks öffentlich zur Verfügung zu stellen. Die Dokumentation verfügt bereits über mehrere Beispiele, welche alle mit Binder öffentlich zur Verfügung gestellt werden könnten. Das ermöglicht es Benutzenden und Interessierten, ein interaktives laufendes Jupyter Notebook der SimPlay Beispiele zu betrachten.

Kapitel 12

Glossar

Begriff	Definition
API	Application Programming Interface, eine Liste von Methoden, welche programmatisch aufgerufen werden können, um Objekte zu manipulieren oder Informationen zu erhalten
Base64	Ein Verfahren zur Kodierung binärer Daten in ASCII-Zeichen
CSS	Cascading Style Sheets, eine Syntax zur Deklaration visueller Eigenschaften von Websites
DOM	Document Object Model, eine Struktur zur Verwaltung von HTML-Elementen
FPS	Frames per Second, Anzahl Aktualisierungen welche eine Rendering Engine pro Sekunde durchführt
JSON	JavaScript Object Notation, textbasiertes, strukturiertes Format zum Speichern von Informationen, häufig in der Webentwicklung verwendet
MIME	Multipurpose Internet Mail Extensions, ein Format zur Beschreibung von Dateitypen
Node.js	Runtime Environment für JavaScript, häufig in der Webentwicklung verwendet
npm	Node Package Manager, ein Paketmanager für Node.js
pip	Python Package Installer, ein Paketmanager für Python
PNG	Portable Network Graphics, ein Speicherformat für Bilddaten, erlaubt Transparenz
PyPI	Python Package Index, online verfügbare Sammlung von Python-Paketen
Seed	Eine Zufallszahl, welche als Startwert für einen Zufallszahlengenerator verwendet wird
UI	User Interface, die grafische Oberfläche eines Programms
Vega	Eine Visualisierungsgrammatik, welche das Teilen von interaktiven Visualisierungsdesigns ermöglicht
XML	Extensible Markup Language, ein textbasiertes, strukturiertes Format zum Speichern von Informationen

Kapitel 13

Anhang

13.1 Installationsanleitung

Da die öffentlichen Anleitungen in Englisch verfasst sind, wird hier eine deutsche Übersetzung der Installationsschritte gegeben.

13.1.1 Installation

Als Voraussetzung sind folgende Punkte zu beachten

- Python 3.9 oder höher
- JupyterLab 3.0 oder höher

Für die Verwendung wird davon ausgegangen, dass Benutzende mit Python, JupyterLab und SimPy vertraut sind.

Da die Packages auf den jeweiligen Repositories veröffentlicht sind, können diese mit dem Paketmanager von Python installiert werden.

```
pip install simplay-jupyter
```

`simplay-jupyter` ist das Package, welches die Integration in JupyterLab ermöglicht. Es enthält die notwendigen Erweiterungen, um die Simulationen in JupyterLab ausführen zu können. Da es `simplay` als Abhängigkeit hat, wird dieses automatisch mit installiert.

Nach der Installation muss JuptyerLab neu gestartet werden, damit die Erweiterungen geladen werden.

Ein Beispiel eines Notebooks welches das Projekt einsetzt kann aus der öffentlichen Dokumentation [84], oder dem im Anhang verfügbaren Abschnitt Verwendung von `simplay` mit JupyterLab: Beispiel 1 [1] entnommen werden.

13.2 Beispiel eines Jupyter Notebooks: Yazzi würfeln

Dieses Notebook soll aufzeigen, wie Jupyter die interaktive Mischung von Code und Beschreibung ermöglicht.

Der untenstehende Code, versucht ein Yazzi *(also 5-mal die selbe Zahl)* zu würfeln, und protokolliert diesen Fortschritt.

```
import random

def get_random_die_roll():
```

[1] 13.3 Verwendung von `simplay` mit JupyterLab: Beispiel 1

```
    return random.randint(1, 6)

def get_n_dice(n):
    return [get_random_die_roll() for i in range(n)]

def check_yazzi(rolls):
    return len(set(rolls)) == 1

def get_most_common(rolls):
    return max(set(rolls), key=rolls.count)

rolls = get_n_dice(5)
print(f"Your rolls: {rolls}")
```

```
Your rolls: [1, 2, 5, 2, 4]
```

Oben ausgegeben wird das Resultat des initialen Wurfs mit fünf Würfeln.

Vielleicht ist es ja bereits ein Yazzi?

```
if check_yazzi(rolls):
    print("Yazzi! With the first roll! Congratulations!")
else:
    print("Don't worry, you still have two throws left!")
```

```
Don't worry, you still have two throws left!
```

Falls nicht direkt beim ersten Wurf ein Yazzi erzielt wurde, wird das häufigste Resultat beiseite gelegt, und mit den verbleibenden Würfel erneut geworfen.

```
if not check_yazzi(rolls):
    most_common = get_most_common(rolls)
    print(f"Most common is {most_common}, keeping those.")
    n_most_common = rolls.count(most_common)
    n_reroll = 5 - n_most_common
    rolls = [most_common] * n_most_common
    new_rolls = get_n_dice(n_reroll)
    rolls += new_rolls
    print(f"Your current rolls: {rolls}")
```

```
Most common is 2, keeping those.
Your current rolls: [2, 2, 3, 3, 1]
```

Der aktuelle Stand der Würfel wird erneut ausgegeben.

Ist es ein Yazzi?

```
if check_yazzi(rolls):
    print("Yazzi! Congratulations!")
else:
    print("Fear not, you still have one more chance to get it :)")
```

```
Fear not, you still have one more chance to get it :)
```

Erneut wird aus allen Würfeln das häufigste Resultat beiseite gelegt und mit den verbleibenden Würfeln erneut geworfen.

```python
if not check_yazzi(rolls):
    most_common = get_most_common(rolls)
    print(f"Most common: {most_common}")
    n_most_common = rolls.count(most_common)
    print(f"Number of most common: {n_most_common}")
    n_reroll = 5 - n_most_common
    print(f"Number of dice to reroll: {n_reroll}")
    rolls = [most_common] * n_most_common
    new_rolls = get_n_dice(n_reroll)
    rolls += new_rolls
    print(f"Final rolls: {rolls}")
```

```
Most common: 2
Number of most common: 2
Number of dice to reroll: 3
Final rolls: [2, 2, 4, 1, 3]
```

Das Resultatset wird ein letztes Mal überprüft.

Ist es gelungen, ein Yazzi zu würfeln?

```python
if check_yazzi(rolls):
    print("Yazzi! Congratulations!")
if not check_yazzi(rolls):
    print("No Yazzi. Better luck next time!")
```

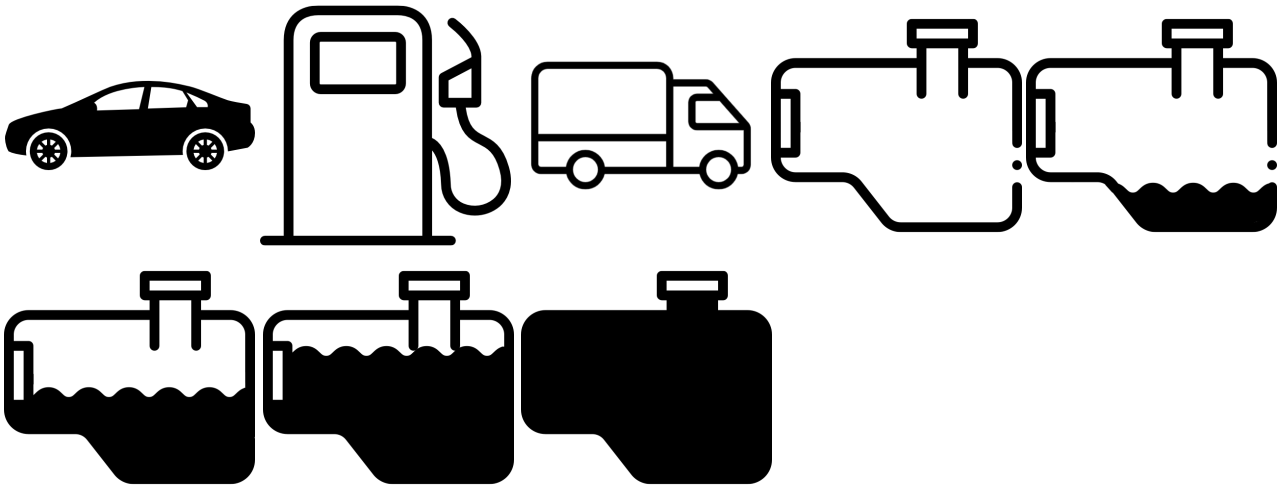
```
No Yazzi. Better luck next time!
```

13.3 Verwendung von `simplay` mit JupyterLab: Beispiel 1

Das untenstehende Beispiel basiert auf dem Beispiel von SimPy, welches hier [25] zu finden ist.

Im untenstehenden Beispiel sind folgende Bilder verwendet:

Damit der ***`tint`*** *den gewünschten Effekt erzielt, sind die effektiv verwendeten Bilder PNGs mit transparentem Hintergrund, und weissem Vordergrund. Um die Bilder hier zeigen zu können, wurden sie mit schwarzer Farbe überlagert.*



```
"""
Gas Station Refueling example
Covers:
- Resources: Resource
- Resources: Container
- Waiting for other processes
Scenario:
  A gas station has a limited number of gas pumps that share a common
  fuel reservoir. Cars randomly arrive at the gas station, request one
  of the fuel pumps and start refueling from that reservoir.
  A gas station control process observes the gas station's fuel level
  and calls a tank truck for refueling if the station's level drops
  below a threshold.
"""
from IPython.display import display
import itertools
import random

from simplay import (
    VisualEnvironment, VisualContainer, VisualProcess,
    VisualResource, VisualGrid
)

RANDOM_SEED = 42
GAS_STATION_SIZE = 500  # liters
THRESHOLD = 20  # Threshold for calling the tank truck (in %)
FUEL_TANK_SIZE = 50  # liters
FUEL_TANK_LEVEL = [5, 25]  # Min/max levels of fuel tanks (in liters)
REFUELING_SPEED = 2  # liters / second
TANK_TRUCK_TIME = 300  # Seconds it takes the tank truck to arrive
T_INTER = [10, 100]  # Create a car every [min, max] seconds
SIM_TIME = 10000  # Simulation time in seconds


class Car(VisualProcess):
    def __init__(self, name, env, gas_station, fuel_pump):
        color = int(random.random() * 0xFFFFFF)
        super().__init__(env, name, visual="CAR", tint=color)
        self.gas_station = gas_station
        self.fuel_pump = fuel_pump

    def run(self):
        self.is_visible()
        self.is_near_cell(2, 1)
        fuel_tank_level = random.randint(*FUEL_TANK_LEVEL)

        with self.gas_station.request() as req:
            start = self.env.now
            # Request one of the gas pumps
            yield req
            self.is_near(self.gas_station)
            self.is_interacting_with(self.gas_station)

            # Get the required amount of fuel
```

```
            liters_required = FUEL_TANK_SIZE - fuel_tank_level
            yield self.fuel_pump.get(liters_required)

            # The "actual" refueling process takes some time
            yield self.env.timeout(liters_required / REFUELING_SPEED)
            self.is_no_longer_interacting_with(self.gas_station)

        self.is_invisible()


def gas_station_control(env, fuel_pump):
    """Periodically check the level of the *fuel_pump* and call the tank
    truck if the level falls below a threshold."""
    truck = TankTruck(env, fuel_pump)
    while True:
        if fuel_pump.level / fuel_pump.capacity * 100 < THRESHOLD:
            # We need to call the tank truck now!
            # Wait for the tank truck to arrive and refuel the station
            yield env.process(truck.run())

        yield env.timeout(10)  # Check every 10 seconds


class TankTruck(VisualProcess):
    def __init__(self, env, fuel_pump):
        super().__init__(env, "Tank Truck", visual="TANK_TRUCK", tint=0xFF0000)
        self.fuel_pump = fuel_pump

    def run(self):
        self.is_visible()
        self.is_at(0, 0)
        self.is_near(self.fuel_pump)
        self.is_interacting_with(self.fuel_pump)
        yield self.env.timeout(TANK_TRUCK_TIME)
        ammount = self.fuel_pump.capacity - self.fuel_pump.level
        yield self.fuel_pump.put(ammount)
        self.is_no_longer_interacting_with(self.fuel_pump)
        self.is_invisible()


def car_generator(env, gas_station, fuel_pump):
    """Generate new cars that arrive at the gas station."""
    for i in itertools.count():
        yield env.timeout(random.randint(*T_INTER))
        car = Car("Car %d" % i, env, gas_station, fuel_pump)
        env.process(car.run())


class GasStation(VisualResource):
    def __init__(self, env):
        super().__init__(env, "Gas Station", 3, visual="GAS_STATION",
                         tint=0x00FF00)
        self.is_at(3, 1)
        self.is_visible()
```

```
109
110
111 class FuelPump(VisualContainer):
112     def __init__(self, env):
113         super().__init__(
114             env,
115             "Fuel Pump",
116             capacity=GAS_STATION_SIZE,
117             init=GAS_STATION_SIZE,
118             visual="FUEL_PUMP",
119             tint=0x000001,
120         )
121         self.is_at(1, 1)
122         self.is_visible()
123         self.has_frame(4)
124
125     def update_sprite(self):
126         fillPercentage = self.level / self.capacity
127         if fillPercentage < 0.25:
128             self.has_frame(0)
129         elif fillPercentage < 0.5:
130             self.has_frame(1)
131         elif fillPercentage < 0.75:
132             self.has_frame(2)
133         elif fillPercentage < 1:
134             self.has_frame(3)
135         else:
136             self.has_frame(4)
137
138     def get(self, amount):
139         cget = super().get(amount)
140         self.update_sprite()
141         return cget
142
143     def put(self, amount):
144         cput = super().put(amount)
145         self.update_sprite()
146         return cput
147
148
149 # Setup and start the simulation
150 random.seed(RANDOM_SEED)
151
152 # Create environment and start processes
153 env = VisualEnvironment()
154 env.visualization_manager.register_visual("CAR", "./resources/car.png")
155 env.visualization_manager.register_visual(
156     "TANK_TRUCK", "./resources/truck.png")
157 env.visualization_manager.register_visual(
158     "GAS_STATION", "./resources/gaspump.png")
159 env.visualization_manager.register_sprites(
160     "FUEL_PUMP",
161     [
162         "./resources/pump_000.png",
```

```
        "./resources/pump_025.png",
        "./resources/pump_050.png",
        "./resources/pump_075.png",
        "./resources/pump_100.png",
    ],
)

grid = VisualGrid(500, 500, 5, 3)
grid.set_area("gasstation01", "GAS_STATION", 3, 5, 0, 0, 0xbdbbbb)
env.visualization_manager.set_grid(grid)

gas_station = GasStation(env)
fuel_pump = FuelPump(env)
env.process(gas_station_control(env, fuel_pump))
env.process(car_generator(env, gas_station, fuel_pump))

# Execute!
env.run(until=SIM_TIME)

# you can extract the output from the visualization manager
output = env.visualization_manager.serialize()

# or save the output to a file directly
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in JupyterLab and have the simplay-jupyter
# extensions installed, you can display the output

display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.4 Code Coverage

13.4.1 simplay

```
Name                            Stmts   Miss  Cover
-------------------------------------------------
src\simplay\__init__.py            23      0   100%
src\simplay\components.py         197     12    94%
src\simplay\core.py               117      6    95%
src\simplay\events.py              95      0   100%
src\simplay\primitives.py          68      0   100%
src\simplay\visualization.py       56      0   100%
-------------------------------------------------
TOTAL                             556     18    97%
```

13.4.2 simplay-web

```
-----------------------------------|---------|----------|---------|---------|-------------------
File                               | % Stmts | % Branch | % Funcs | % Lines |
    Uncovered Line #s
-----------------------------------|---------|----------|---------|---------|-------------------
All files                          |   96.63 |    93.68 |   99.13 |   96.95 |
 src                               |   92.64 |     92.3 |   97.29 |    93.5 |
```

```
 Entity.ts                          |   98.61 |    95.23 |     100 |   98.61 | 67
 Grid.ts                            |   61.11 |      100 |   66.66 |    64.7 | 15-38
 SimplayContext.ts                  |     100 |      100 |     100 |     100 |
 SimulationData.ts                  |     100 |      100 |     100 |     100 |
 SimulationSpooler.ts               |     100 |    85.71 |     100 |     100 | 149
src/event                           |   98.89 |    94.64 |     100 |   98.88 |
 ContainerSetCapacityEvent.ts       |     100 |      100 |     100 |     100 |
 ContainerSetCapacityEventArgs.ts   |     100 |      100 |     100 |     100 |
 ContainerSetLevelEvent.ts          |     100 |      100 |     100 |     100 |
 ContainerSetLevelEventArgs.ts      |     100 |      100 |     100 |     100 |
 Event.ts                           |     100 |      100 |     100 |     100 |
 EventArgs.ts                       |     100 |      100 |     100 |     100 |
 EventFactory.ts                    |     100 |      100 |     100 |     100 |
 InteractionLine.ts                 |     100 |      100 |     100 |     100 |
 MoveNearCellEvent.ts               |     100 |      100 |     100 |     100 |
 MoveNearCellEventArgs.ts           |     100 |      100 |     100 |     100 |
 MoveNearEvent.ts                   |     100 |      100 |     100 |     100 |
 MoveNearEventArgs.ts               |     100 |      100 |     100 |     100 |
 ResourceSetCapacityEvent.ts        |     100 |      100 |     100 |     100 |
 ResourceSetCapacityEventArgs.ts    |     100 |      100 |     100 |     100 |
 ResourceSetUtilizationEvent.ts     |     100 |      100 |     100 |     100 |
 ResourceSetUtilizationEventArgs.ts |     100 |      100 |     100 |     100 |
 SetDecoratingTextEvent.ts          |     100 |      100 |     100 |     100 |
 SetDecoratingTextEventArgs.ts      |     100 |      100 |     100 |     100 |
 SetInteractingEvent.ts             |    92.3 |       50 |     100 |    92.3 | 20
 SetInteractingEventArgs.ts         |     100 |      100 |     100 |     100 |
 SetNotInteractingEvent.ts          |   76.92 |       50 |     100 |   76.92 | 22-24
 SetNotInteractingEventArgs.ts      |     100 |      100 |     100 |     100 |
 SetPositionEvent.ts                |     100 |      100 |     100 |     100 |
 SetPositionEventArgs.ts            |     100 |      100 |     100 |     100 |
 SetSpriteFrameEvent.ts             |     100 |      100 |     100 |     100 |
 SetSpriteFrameEventArgs.ts         |     100 |      100 |     100 |     100 |
 SetTintColorEvent.ts               |     100 |      100 |     100 |     100 |
 SetTintColorEventArgs.ts           |     100 |      100 |     100 |     100 |
 SetVisibleEvent.ts                 |     100 |      100 |     100 |     100 |
 SetVisibleEventArgs.ts             |     100 |      100 |     100 |     100 |
 StoreSetCapacityEvent.ts           |     100 |       75 |     100 |     100 | 29
 StoreSetCapacityEventArgs.ts       |     100 |      100 |     100 |     100 |
 StoreSetContentEvent.ts            |     100 |      100 |     100 |     100 |
 StoreSetContentEventArgs.ts        |     100 |      100 |     100 |     100 |
------------------------------------|---------|----------|---------|---------|-------------------
```

13.4.3 simplay-jupyter

```
---------------------|---------|----------|---------|---------|---------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|---------------------
All files            |   97.34 |    93.33 |    93.8 |   97.31 |
 accurateSlider.ts   |   94.82 |    88.23 |   94.73 |   94.73 | 32-34,62-64
 index.ts            |   96.42 |      100 |   86.48 |    96.4 | 176,187,267,303,335
 speedSelector.ts    |     100 |      100 |     100 |     100 |
 startPauseButton.ts |     100 |      100 |     100 |     100 |
 stepInfo.ts         |     100 |      100 |     100 |     100 |
 utils.ts            |     100 |      100 |     100 |     100 |
```

```
11 --------------------|---------|----------|---------|---------|--------------------
```

13.5 Öffentliche Dokumentation

Die Dokumentation ist unter simplay.readthedocs.io [44] verfügbar. Die hier enthaltene Dokumentation, entspricht der `simplay` Version `1.0.1`: https://simplay.readthedocs.io/en/1.0.1/. Sie wurde automatisch generiert und strukturell überarbeitet.

13.5.1 Documentation for SimPlay

- Overview
  - Structure of the project
- Getting started
  - Install SimPlay
  - Adding visual information to your simulation
  - Playing the Visualization
- Examples
  - Bank Renege
  - Carwash
  - Machine Shop
  - Movie Renege
  - Gas Station Refueling
  - Process Communication
  - Event Latency
- SimPlay in depth
  - Using Resources
  - Using Containers
  - Using Stores
- API Reference
  - `simplay`
  - `simplay.core` – Core components of Simplay
  - `simplay.visualization` — Visualization
  - `simplay.components` — Components
  - `simplay.events` — Visual Event Types
  - `simplay.primitives` — Primitives

13.5.2 Overview

SimPlay is a project that aims to bring simple animation capabilities to the SimPy project.

SimPlay is integrated into JupyterLab and is capable of displaying an animation of a SimPy simulation in a notebook:

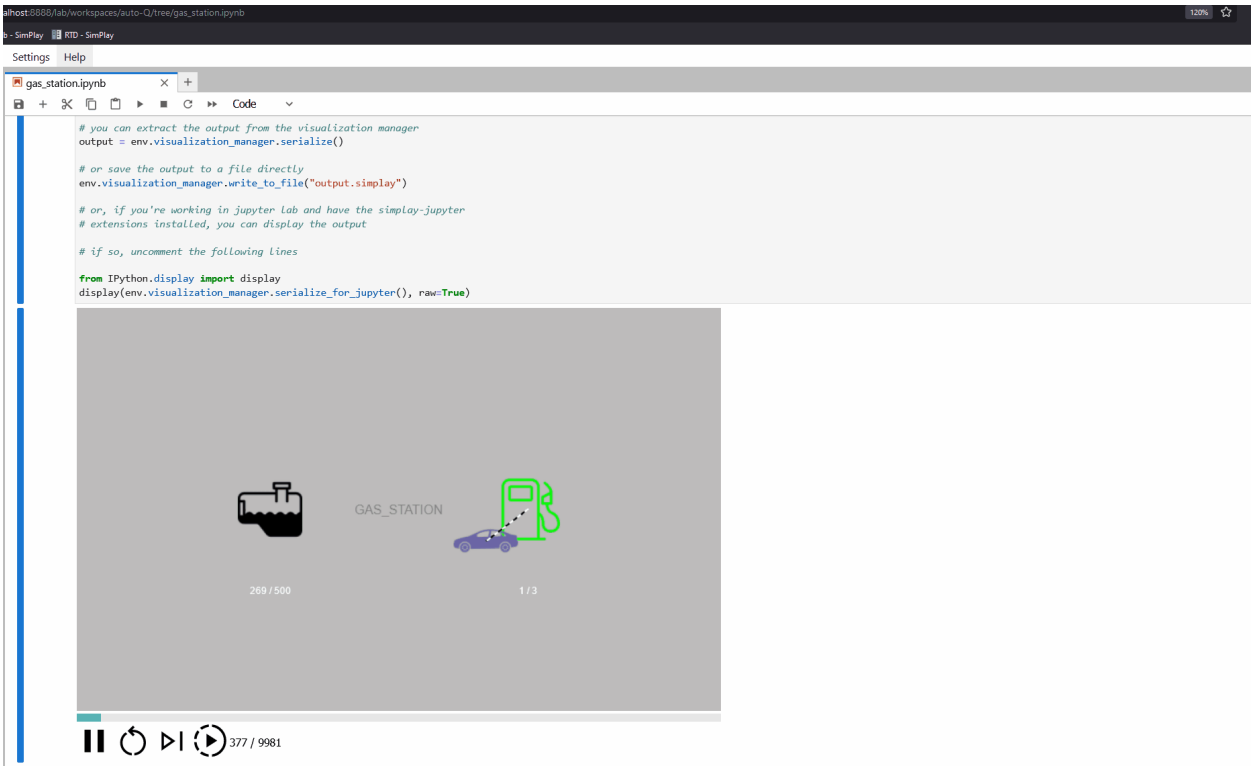


Abbildung 13.1: `simplay` in JupyterLab

Check out the Getting started section for further information, including how to Install SimPlay.

The API documentation is available at API Reference.

13.5.2.1 Structure of the project

SimPlay is split into three parts:

- The **SimPlay** package, which is a Python package that can be integrated into SimPy simulations. It is used to generate the animation data and to provide a simple API to do so. It is available on PyPI at https://pypi.python.org/pypi/simplay.
- The **SimPlay-Web** package, which is a TypeScript package that is used to display the generated animation data. It is available on npm at https://www.npmjs.com/package/simplay-web.
- The **SimPlay-Jupyter** package, which depends on both the packages mentioned above. It is used to display the animation data in Jupyter Lab. It is available on PyPI at https://pypi.python.org/pypi/simplay-jupyter.

13.5.3 Getting started

13.5.3.1 Install SimPlay

To use Simplay, first decide how you want to use it. Three components exist:

- SimPlay-Jupyter (the python package allowing you to view and enhance any SimPy Simulation within a jupyter notebook)
- SimPlay (the python library allowing you to generate the SimulationData)

- SimPlay-Web (the npm packages allowing you to render the SimulationData)

To install the jupyter extension, execute the following command:

> **NOTE**: This guide assumes you have a working JupyterLab installation. The minimum version required is 3.0.

```
pip install simplay-jupyter
```

NOTE: After installing simplay-jupyter, JupyterLab needs to be restarted.

The following installation steps are only necessary if you don't want to run within JupyterLab. Otherwise continue with Adding visual information to your simulation.

To install SimPlay:

```
pip install simplay
```

To install SimPlay-Web:

```
npm install simplay-web
```

13.5.3.2 Adding visual information to your simulation

SimPlay is designed to integrate with SimPy as easily as possible. The usual structures and components of SimPy are recreated as objects with capabilities to animate the simulation.

The components known from SimPy are prefixed with `Visual`. For example, `Environment` becomes `VisualEnvironment`. The `VisualEnvironment` is needed by other components, such as `VisualProcess`, to gain information about the visualization.

To get started, import the `VisualEnvironment`:

```
from simplay import VisualEnvironment
```

Then create a instance of `VisualEnvironment` and start adding components:

```
env = VisualEnvironment()
# add components here
env.run()
```

The components provided by SimPy, such as simple processes and resources, must be classes in order to be able to be integrated into the visualization.

A simple process can be created as follows:

```
from simplay import VisualProcess

class MyProcess(VisualProcess):
    def __init__(self, env, id):
        super().__init__(env, id, visual="SOMEPNG", tint=0x00FF00)

    def run(self):
        while True:
            print(f'{self.id} is running')
            yield self.env.timeout(1)
```

Then, create an instance of the process and add it to the environment:

```
env = VisualEnvironment()
p = MyProcess(env, 'MyProcess')
env.process(p.run())
env.run()
```

Keep in mind, that the `id` parameter should be unique across the whole environment.

Reading the `__super__` call of the `MyProcess` constructor carefully, notice that it takes a `visual` and a `tint` parameter. In the example, the value of `visual` is `SOMEPNG`. In order for the visualization to work, the `VisualizationManager` - which exists on `VisualEnvironment` - needs to know where to find the visual. Register visuals with the following call:

```
env.visualization_manager.register_visual('SOMEPNG', 'path/to/your/visual.png')
```

> **NOTE**: We recommend using PNG files with a transparent background, and a white foreground. This way, you can most effectively use the `tint` parameter.

The `tint` parameter multiplies the color of the visual with the given color. If no tint is to be applied, set it to 0xFFFFFF, which is the default value, so all pixel values are kept the same. The tint parameter must be a whole integer.

After having successfully created a process, it is time to learn how SimPlay is able to log visual changes of the simulation.

The visual components provide various methods to declare visual changes. These methods always start with `is_` or `has_` and create the corresponding event. Find a complete list of events in the simplay.events — Visual Event Types section. The following section only provides a few examples for these methods, but should make the use of them clear.

The following example shows how to set the position of a component:

```
from simplay import VisualEnvironment, VisualProcess, VisualComponent

class MyProcess(VisualProcess):
    def __init__(self, env, id):
        super().__init__(env, id, visual="SOMEPNG", tint=0x00FF00)
        self.is_at(self, 5, 5)
        self.is_visible()

    def run(self):
        while True:
            print(f'{self.id} is running')
            yield self.env.timeout(1)
```

The code above now sets the position of the component to (5, 5), at the time of the simulation when the process is created, and declares the component visible.

The parameters of the `is_at` function refer to row and column values of a grid.

This is where the `VisualGrid` comes into (Sim)play.

The `VisualGrid` is a component that is used to map the simulation space to the screen space.

The following example shows how to create a `VisualGrid` and add it to the environment:

```
from simplay import VisualEnvironment, VisualGrid
env = VisualEnvironment()
# create a grid
grid = VisualGrid(1000, 500, 10, 5)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xbdbbbb)
```

```
# add the grid to the environment
env.visualization_manager.set_grid(grid)
```

The code above creates a grid with a width of 1000 pixels and a height of 500 pixels, split into 10x5 cells.

> **NOTE:** Be aware that different machines have different screen resolutions and if the visualization is viewed on a different machine it may appear different. Also be aware that when looking at a file directly in JupyterLab, there is no scrolling enabled and some parts may get cut off.

The grid must be registered with the `VisualizationManager` of the environment. Additionally, the code above adds an area to the grid. The area is a rectangle that is drawn on the grid, and can be used to visually separate different parts of the simulation. The area is defined by the id, the text that is displayed in the area, the height (in cells) and the width (in cells), and the top-left position (in cells) of the area. The following is a visual representation of this, where 'X' marks the cells where this area is drawn, and ' ' marks the cells where it is not:

```
+---+---+---+---+---+---+---+---+---+---+
| X | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
| X | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
| X | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
| X | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
| X | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
```

> **NOTE**: It is recommended to not have areas with a white background. This is because the decorating and informational texts are also drawn in white, and thus would not be visible. Further, in order to correctly have tints applied to the components, it is recommended to have all-white transparent PNGs, and if no tint is applied, then the visual is invisible.

This guide covers the basics of SimPlay. Learn more about Containers, Resources and Stores in SimPlay in depth or view some Examples to see how SimPlay can be used in practice. For detailed information about the API have a look at API Reference.

13.5.3.3 Playing the Visualization

Follow the instructions under Install SimPlay to install the simplay extension for jupyter. Once the installation is complete, start a new notebook and import the `simplay` module:

```
from simplay import VisualEnvironment, VisualGrid

env = VisualEnvironment()
# create a grid
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
# add the grid to the environment
env.visualization_manager.set_grid(grid)

class MyProcess(VisualProcess):
    def __init__(self, env, id):
        super().__init__(env, id, visual="SOMEPNG", tint=0x00FF00)
        self.is_at(5, 5)

    def run(self):
```

```
        while True:
            print(f'{self.id} is running')
            yield self.env.timeout(1)

env.process(MyProcess(env, 1))
env.run(until=10)
```

The code above is the same as the one in the previous section, but now it is executed in a jupyter notebook. To display the visualization, use the `display` function provided by `IPython.display`:

```
from IPython.display import display
output = env.visualization_manager.serialize_for_jupyter()
display(output, raw=True)
```

The extension will now automatically display the visualization in the notebook.

Since `simplay` creates JSON output, save the output to a file if desired:

```
env.visualization_manager.write_to_file("output.simplay")
```

Then, open the `.simplay` file in JupyterLab and the visualization will be displayed.

How to use resources, containers and stores is explained in SimPlay in depth.

13.5.4 Examples

The following examples are based on SimPy's examples.

13.5.4.1 Bank Renege

NOTE: The following images are referenced in the example below.

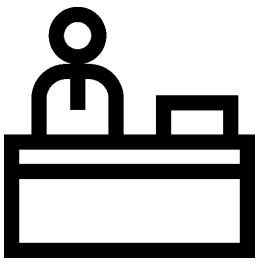


Abbildung 13.2: counter

Reception icons created by Freepik - Flaticon

Abbildung 13.3: user

Person icons created by DinosoftLabs - Flaticon

```
"""
Bank renege example

Covers:

- Resources: Resource
- Condition events

Scenario:
A counter with a random service time and customers who renege. Based on the
program bank08.py from TheBank tutorial of SimPy 2. (KGM)

"""
from IPython.display import display
import random
from simplay import VisualEnvironment, VisualProcess, VisualResource, VisualGrid

RANDOM_SEED = 42
NEW_CUSTOMERS = 6  # Total number of customers
INTERVAL_CUSTOMERS = 10.0  # Generate new customers roughly every x seconds
MIN_PATIENCE = 1  # Min. customer patience
MAX_PATIENCE = 3  # Max. customer patience

free_positions = [[1, 0], [1, 1], [1, 2]]


def source(env, number, interval, counter):
    """Source generates customers randomly"""
    for i in range(number):
        c = Customer(env, 'Customer%02d' % i, counter, time_in_bank=12.0)
        env.process(c.run())
        t = random.expovariate(1.0 / interval)
        yield env.timeout(t)


class Customer(VisualProcess):
    def __init__(self, env, id, counter, time_in_bank):
        color = int(random.random() * 0xFFFFFF)
        super().__init__(env, id, visual="CUSTOMER", tint=color)
        self.counter = counter
        self.time_in_bank = time_in_bank
        self.position = free_positions.pop(0)

    def run(self):
        """Customer arrives, is served and leaves."""
        self.is_visible()
        self.is_at(self.position[0], self.position[1])
        arrive = self.env.now
        print('%7.4f %s: Here I am' % (arrive, self.id))

        with self.counter.request() as req:
            patience = random.uniform(MIN_PATIENCE, MAX_PATIENCE)
            # Wait for the counter or abort at the end of our tether
            results = yield req | env.timeout(patience)
```

```
55
56              wait = self.env.now - arrive
57
58              if req in results:
59                  # We got to the counter
60                  self.is_near(self.counter)
61                  self.is_interacting_with(self.counter)
62                  print('%7.4f %s: Waited %6.3f' % (self.env.now, self.id, wait))
63
64                  tib = random.expovariate(1.0 / self.time_in_bank)
65                  yield self.env.timeout(tib)
66                  self.is_invisible()
67                  self.is_no_longer_interacting_with(self.counter)
68                  free_positions.append(self.position)
69                  print('%7.4f %s: Finished' % (self.env.now, self.id))
70
71              else:
72                  # We reneged
73                  print('%7.4f %s: RENEGED after %6.3f' %
74                        (self.env.now, self.id, wait))
75                  self.is_invisible()
76                  self.is_no_longer_interacting_with(self.counter)
77                  free_positions.append(self.position)
78
79
80  # Setup and start the simulation
81  print('Bank renege')
82  random.seed(RANDOM_SEED)
83  env = VisualEnvironment()
84  grid = VisualGrid(1000, 1000, 2, 3)
85  grid.set_area("counter", "Counter", 3, 1, 0, 0, 0x707070)
86  env.visualization_manager.set_grid(grid)
87  env.visualization_manager.register_visual(
88      "CUSTOMER", "./bank_assets/user.png")
89  env.visualization_manager.register_visual(
90      "COUNTER", "./bank_assets/counter.png")
91
92  # Start processes and run
93
94
95  class Counter(VisualResource):
96      def __init__(self, env):
97          super().__init__(env, "Counter", 1, visual="COUNTER")
98          self.is_at(0, 1)
99          self.is_visible()
100
101
102 counter = Counter(env)
103 env.process(source(env, NEW_CUSTOMERS, INTERVAL_CUSTOMERS, counter))
104 env.run()
105
106 # you can extract the output from the visualization manager
107 output = env.visualization_manager.serialize()
108
```

```
# or save the output to a file directly
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in jupyter lab and have the simplay-jupyter
# extensions installed, you can display the output

# if so, uncomment the following lines
# from IPython.display import display
# display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.4.2 Carwash

NOTE: The following images are referenced in the example below.

Abbildung 13.4: car

Car icons created by Freepik - Flaticon

Abbildung 13.5: carwash

Carwash icons created by smashingstocks - Flaticon

```
"""
Carwash example.

Covers:

- Waiting for other processes
- Resources: Resource

Scenario:
  A carwash has a limited number of washing machines and defines
  a washing processes that takes some (random) time.

  Car processes arrive at the carwash at a random time. If one washing
```

```
  machine is available, they start the washing process and wait for it
  to finish. If not, they wait until they an use one.

"""
import random

from simplay import VisualEnvironment, VisualProcess, VisualResource, VisualGrid


RANDOM_SEED = 42
NUM_MACHINES = 2  # Number of machines in the carwash
WASHTIME = 5      # Minutes it takes to clean a car
T_INTER = 7       # Create a car every ~7 minutes
SIM_TIME = 20     # Simulation time in minutes

class Carwash(VisualResource):
    """A carwash has a limited number of machines (``NUM_MACHINES``) to
    clean cars in parallel.

    Cars have to request one of the machines. When they got one, they
    can start the washing processes and wait for it to finish (which
    takes ``washtime`` minutes).

    """

    def __init__(self, env, num_machines, washtime):
        super().__init__(env, "Carwash", num_machines, visual="CARWASH")
        self.washtime = washtime
        self.is_at(1, 3)
        self.is_visible()

    def wash(self, car):
        """The washing processes. It takes a ``car`` processes and tries
        to clean it."""
        yield self.env.timeout(WASHTIME)
        print("Carwash removed %d%% of %s's dirt." %
              (random.randint(50, 99), car))


class Car(VisualProcess):
    def __init__(self, env, id, cw):
        color = int(random.random() * 0xFFFFFF)
        super().__init__(env, id, visual="CAR", tint=color)
        self.cw = cw

    def run(self):
        """The car process (each car has a ``name``) arrives at the carwash
        (``cw``) and requests a cleaning machine.

        It then starts the washing process, waits for it to finish and
        leaves to never come back ...

        """
        print('%s arrives at the carwash at %.2f.' % (self.id, env.now))
```

```
        self.is_visible()
        self.is_near_cell(4, 2)
        with self.cw.request() as request:
            yield request
            self.is_near(self.cw)
            self.is_interacting_with(self.cw)
            print('%s enters the carwash at %.2f.' % (self.id, env.now))
            yield env.process(self.cw.wash(self.id))

            print('%s leaves the carwash at %.2f.' % (self.id, env.now))
            self.is_invisible()
            self.is_no_longer_interacting_with(self.cw)


def setup(env, num_machines, washtime, t_inter):
    """Create a carwash, a number of initial cars and keep creating cars
    approx. every ``t_inter`` minutes."""
    # Create the carwash
    carwash = Carwash(env, num_machines, washtime)

    # Create 4 initial cars
    for i in range(4):
        env.process(Car(env, 'Car %d' % i, carwash).run())

    # Create more cars while the simulation is running
    while True:
        yield env.timeout(random.randint(t_inter - 2, t_inter + 2))
        i += 1
        env.process(Car(env, 'Car %d' % i, carwash).run())


# Setup and start the simulation
print('Carwash')
print('Check out http://youtu.be/fXXmeP9TvBg while simulating ... ;-)')
random.seed(RANDOM_SEED)  ### This helps reproducing the results

# Create an environment and start the setup process
env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 6, 6)
grid.set_area("Carwash", "Carwash", 6, 3, 0, 0, 0x707070)
env.visualization_manager.set_grid(grid)
env.visualization_manager.register_visual(
    "CAR", "./carwash_assets/car.png")
env.visualization_manager.register_visual(
    "CARWASH", "./carwash_assets/carwash.png")
env.process(setup(env, NUM_MACHINES, WASHTIME, T_INTER))


# Execute!
env.run(until=SIM_TIME)
# you can extract the output from the visualization manager
output = env.visualization_manager.serialize()

# or save the output to a file directly
```

```
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in jupyter lab and have the simplay-jupyter
# extensions installed, you can display the output

# if so, uncomment the following lines
#from IPython.display import display
#display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.4.3 Machine Shop

NOTE: The following images are referenced in the example below.

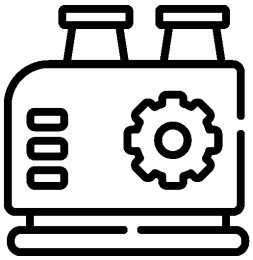


Abbildung 13.6: machine

Machine icons created by Freepik - Flaticon

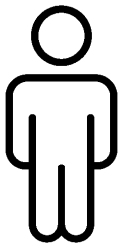


Abbildung 13.7: repairman

Person icons created by DinosoftLabs - Flaticon

NOTE: This simulation takes a while to run.

```
"""
Machine shop example

Covers:

- Interrupts
- Resources: PreemptiveResource

Scenario:
  A workshop has *n* identical machines. A stream of jobs (enough to
  keep the machines busy) arrives. Each machine breaks down
  periodically. Repairs are carried out by one repairman. The repairman
  has other, less important tasks to perform, too. Broken machines
```

```
  preempt theses tasks. The repairman continues them when he is done
  with the machine repair. The workshop works continuously.

"""
import random

from simplay import VisualEnvironment, VisualProcess, VisualPreemptiveResource,
    VisualGrid
import simpy


RANDOM_SEED = 42
PT_MEAN = 10.0         # Avg. processing time in minutes
PT_SIGMA = 2.0         # Sigma of processing time
MTTF = 300.0           # Mean time to failure in minutes
BREAK_MEAN = 1 / MTTF  # Param. for expovariate distribution
REPAIR_TIME = 30.0     # Time it takes to repair a machine in minutes
JOB_DURATION = 30.0    # Duration of other jobs in minutes
NUM_MACHINES = 10      # Number of machines in the machine shop
WEEKS = 4              # Simulation time in weeks
SIM_TIME = WEEKS * 7 * 24 * 60  ### Simulation time in minutes


def time_per_part():
    """Return actual processing time for a concrete part."""
    return random.normalvariate(PT_MEAN, PT_SIGMA)


def time_to_failure():
    """Return time until next failure for a machine."""
    return random.expovariate(BREAK_MEAN)


machine_positions = [[0, 0], [1, 0], [2, 0], [3, 0],
                     [4, 0], [5, 0], [1, 1], [2, 1], [3, 1], [4, 1]]


class Machine(VisualProcess):
    """A machine produces parts and my get broken every now and then.

    If it breaks, it requests a *repairman* and continues the production
    after the it is repaired.

    A machine has a *name* and a numberof *parts_made* thus far.

    """

    def __init__(self, env, name, repairman: VisualPreemptiveResource):
        color = int(random.random() * 0x00FFFF)
        super().__init__(env, name, "MACHINE", color)
        self.parts_made = 0
        self.broken = False
        self.repairman = repairman
        self.position = machine_positions.pop(0)
```

```
        self.is_at(self.position[0], self.position[1])
        self.is_visible()

        # Start "working" and "break_machine" processes for this machine.
        self.process = env.process(self.working())
        env.process(self.break_machine())

    def working(self):
        """Produce parts as long as the simulation runs.

        While making a part, the machine may break multiple times.
        Request a repairman when this happens.

        """
        while True:
            # Start making a new part
            done_in = time_per_part()
            while done_in:
                try:
                    # Working on the part
                    start = self.env.now
                    yield self.env.timeout(done_in)
                    done_in = 0  # Set to 0 to exit while loop.

                except simpy.Interrupt:
                    self.broken = True
                    done_in -= self.env.now - start  # How much time left?

                    # Request a repairman. This will preempt its "other_job".
                    with self.repairman.request(priority=1) as req:
                        yield req
                        self.repairman.is_near(self)
                        self.repairman.is_interacting_with(self)
                        yield self.env.timeout(REPAIR_TIME)
                        self.repairman.is_no_longer_interacting_with(self)
                        self.repairman.is_at(
                            self.repairman.home[0], self.repairman.home[1])
                    self.broken = False
                    self.has_original_tint()

            # Part is done.
            self.parts_made += 1
            self.has_decorating_text("Parts made: " + str(self.parts_made))

    def break_machine(self):
        """Break the machine every now and then."""
        while True:
            yield self.env.timeout(time_to_failure())
            if not self.broken:
                # Only break the machine if it is currently working.
                self.process.interrupt()
                self.has_tint(0xFF0000)


```

```
def other_jobs(env, repairman):
    """The repairman's other (unimportant) job."""
    while True:
        # Start a new job
        done_in = JOB_DURATION
        while done_in:
            # Retry the job until it is done.
            # It's priority is lower than that of machine repairs.
            with repairman.request(priority=2) as req:
                yield req
                try:
                    start = env.now
                    yield env.timeout(done_in)
                    done_in = 0
                except simpy.Interrupt:
                    done_in -= env.now - start


# Setup and start the simulation
print('Machine shop')
random.seed(RANDOM_SEED)  # This helps reproducing the results

# Create an environment and start the setup process
env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 6, 5)
grid.set_area("OtherJobs", "Other jobs", 3, 6, 0, 2, 0x707070)
env.visualization_manager.set_grid(grid)
env.visualization_manager.register_visual(
    "REPAIRMAN", "./machine_shop_assets/repairman.png")
env.visualization_manager.register_visual(
    "MACHINE", "./machine_shop_assets/machine.png")


class Repairman(VisualPreemptiveResource):
    def __init__(self, env, capacity):
        super().__init__(env, "Repairman", capacity, visual="REPAIRMAN")
        self.home = [2, 3]
        self.is_at(2, 3)
        self.is_visible()


repairman = Repairman(env, capacity=1)
machines = [Machine(env, 'Machine %d' % i, repairman)
            for i in range(NUM_MACHINES)]
env.process(other_jobs(env, repairman))

# Execute!
env.run(until=SIM_TIME)

# Analyis/results
print('Machine shop results after %s weeks' % WEEKS)
for machine in machines:
    print('%s made %d parts.' % (machine.id, machine.parts_made))

```

```
# you can extract the output from the visualization manager
output = env.visualization_manager.serialize()

# or save the output to a file directly
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in jupyter lab and have the simplay-jupyter
# extensions installed, you can display the output

# if so, uncomment the following lines
# from IPython.display import display
# display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.4.4 Movie Renege

NOTE: The following images are referenced in the example below.

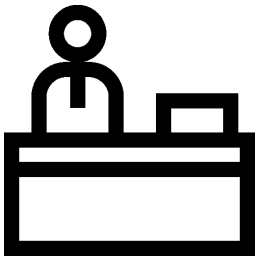


Abbildung 13.8: counter

Reception icons created by Freepik - Flaticon

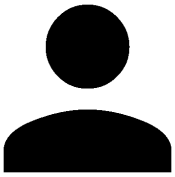


Abbildung 13.9: customer

Person icons created by inkubators - Flaticon

```
"""
Movie renege example

Covers:

- Resources: Resource
- Condition events
- Shared events

Scenario:
```

```
  A movie theatre has one ticket counter selling tickets for three
  movies (next show only). When a movie is sold out, all people waiting
  to buy tickets for that movie renege (leave queue).

"""
import collections
import random

from simplay import VisualEnvironment, VisualGrid, VisualResource, VisualProcess


RANDOM_SEED = 42
TICKETS = 50  # Number of tickets per movie
SIM_TIME = 120  # Simulate until

row_current_x = 1
row_current_y = 0

COLS = 5
ROWS = 16

def define_next_moviegoer_position():
    global row_current_x
    global row_current_y
    if row_current_x == COLS:
        row_current_y += 1
        row_current_x = 0
    else:
        row_current_x += 1


class Moviegoer(VisualProcess):
    def __init__(self, env, id, movie, num_tickets, theater):
        color = int(random.random() * 0xFFFFFF)
        super().__init__(env, id, "CUSTOMER", color)
        self.movie = movie
        self.num_tickets = num_tickets
        self.theater = theater
        self.is_at(row_current_x, row_current_y)
        define_next_moviegoer_position()
        self.is_visible()

    def run(self):
        """A moviegoer tries to by a number of tickets (*num_tickets*) for
        a certain *movie* in a *theater*.

        If the movie becomes sold out, she leaves the theater. If she gets
        to the counter, she tries to buy a number of tickets. If not enough
        tickets are left, she argues with the teller and leaves.

        If at most one ticket is left after the moviegoer bought her
        tickets, the *sold out* event for this movie is triggered causing
        all remaining moviegoers to leave.

```

```
        """
        with self.theater.counter.request() as my_turn:
            # Wait until its our turn or until the movie is sold out
            result = yield my_turn | self.theater.sold_out[self.movie]
            self.is_near(self.theater.counter)
            self.is_interacting_with(self.theater.counter)
            # Check if it's our turn or if movie is sold out
            if my_turn not in result:
                self.theater.num_renegers[self.movie] += 1
                self.is_invisible()
                self.is_no_longer_interacting_with(self.theater.counter)
                return

            # Check if enough tickets left.
            if self.theater.available[self.movie] < self.num_tickets:
                # Moviegoer leaves after some discussion
                yield self.env.timeout(0.5)
                self.is_invisible()
                self.is_no_longer_interacting_with(self.theater.counter)
                return

            # Buy tickets
            self.theater.available[self.movie] -= self.num_tickets
            if self.theater.available[self.movie] < 2:
                # Trigger the "sold out" event for the movie
                self.theater.sold_out[self.movie].succeed()
                self.theater.when_sold_out[self.movie] = env.now
                self.theater.available[self.movie] = 0
            yield self.env.timeout(1)
            self.is_invisible()
            self.is_no_longer_interacting_with(self.theater.counter)


def customer_arrivals(env, theater):
    """Create new *moviegoers* until the sim time reaches 120."""
    while True:
        yield env.timeout(random.expovariate(1 / 0.5))

        movie = random.choice(theater.movies)
        num_tickets = random.randint(1, 6)
        if theater.available[movie]:
            env.process(Moviegoer(
                env, "Movigoer" + str(random.random()), movie, num_tickets,
                    theater).run())


Theater = collections.namedtuple('Theater', 'counter, movies, available, '
                                            'sold_out, when_sold_out, '
                                            'num_renegers')


# Setup and start the simulation
print('Movie renege')
random.seed(RANDOM_SEED)
```

```
env = VisualEnvironment()
env.visualization_manager.register_visual('COUNTER', './movie_renege_assets/counter.png')
env.visualization_manager.register_visual('CUSTOMER',
    './movie_renege_assets/customer.png')
grid = VisualGrid(1000, 1000, COLS, ROWS)
env.visualization_manager.set_grid(grid)

# Create movie theater
counter = VisualResource(env, "Counter", 1, "COUNTER")
counter.is_at(0, 0)
counter.is_visible()
movies = ['Python Unchained', 'Kill Process', 'Pulp Implementation']
available = {movie: TICKETS for movie in movies}
sold_out = {movie: env.event() for movie in movies}
when_sold_out = {movie: None for movie in movies}
num_renegers = {movie: 0 for movie in movies}
theater = Theater(counter, movies, available, sold_out, when_sold_out,
                  num_renegers)

# Start process and run
env.process(customer_arrivals(env, theater))
env.run(until=SIM_TIME)

# Analysis/results
for movie in movies:
    if theater.sold_out[movie]:
        print('Movie "%s" sold out %.1f minutes after ticket counter '
              'opening.' % (movie, theater.when_sold_out[movie]))
        print('  Number of people leaving queue when film sold out: %s' %
              theater.num_renegers[movie])

# you can extract the output from the visualization manager
output = env.visualization_manager.serialize()

# or save the output to a file directly
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in jupyter lab and have the simplay-jupyter
# extensions installed, you can display the output

# if so, uncomment the following lines
# from IPython.display import display
# display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.4.5 Gas Station Refueling

NOTE: The following images are referenced in the example below.

Abbildung 13.10: car

Car icons created by Freepik - Flaticon

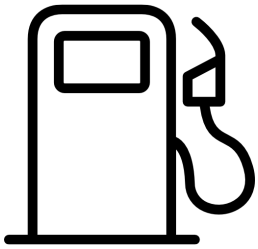


Abbildung 13.11: gaspump

Fuel icons created by Good Ware - Flaticon

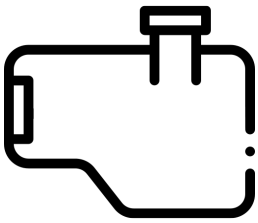


Abbildung 13.12: pump_000

Adapted from Gas Tank icon created by Freepik - Flaticon

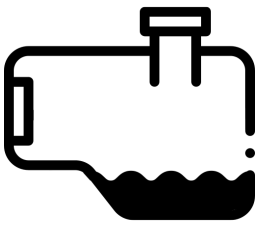


Abbildung 13.13: pump_025

Adapted from Gas Tank icon created by Freepik - Flaticon

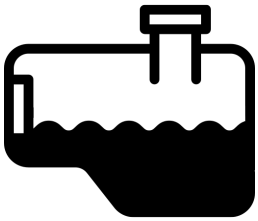


Abbildung 13.14: pump_050

Adapted from Gas Tank icon created by Freepik - Flaticon

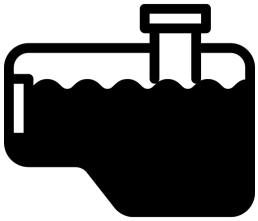


Abbildung 13.15: pump_075

Adapted from Gas Tank icon created by Freepik - Flaticon

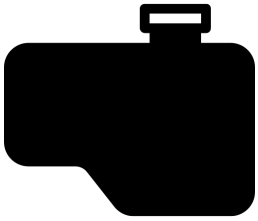


Abbildung 13.16: pump_100

Adapted from Gas Tank icon created by Freepik - Flaticon

Abbildung 13.17: truck

Truck icon created by iconpacks.net

```
"""
Gas Station Refueling example
Covers:
- Resources: Resource
- Resources: Container
- Waiting for other processes
Scenario:
  A gas station has a limited number of gas pumps that share a common
  fuel reservoir. Cars randomly arrive at the gas station, request one
  of the fuel pumps and start refueling from that reservoir.
  A gas station control process observes the gas station's fuel level
  and calls a tank truck for refueling if the station's level drops
  below a threshold.
"""
import itertools
import random

from simplay import (
    VisualEnvironment, VisualContainer, VisualProcess,
    VisualResource, VisualGrid
)

RANDOM_SEED = 42
GAS_STATION_SIZE = 500  # liters
THRESHOLD = 20  # Threshold for calling the tank truck (in %)
FUEL_TANK_SIZE = 50  # liters
FUEL_TANK_LEVEL = [5, 25]  # Min/max levels of fuel tanks (in liters)
REFUELING_SPEED = 2  # liters / second
TANK_TRUCK_TIME = 300  # Seconds it takes the tank truck to arrive
T_INTER = [10, 100]  # Create a car every [min, max] seconds
SIM_TIME = 10000  # Simulation time in seconds


class Car(VisualProcess):
    def __init__(self, name, env, gas_station, fuel_pump):
        color = int(random.random() * 0xFFFFFF)
        super().__init__(env, name, visual="CAR", tint=color)
        self.gas_station = gas_station
        self.fuel_pump = fuel_pump

    def run(self):
        self.is_visible()
        self.is_near_cell(2, 1)
        fuel_tank_level = random.randint(*FUEL_TANK_LEVEL)

        with self.gas_station.request() as req:
            start = self.env.now
            # Request one of the gas pumps
            yield req
            self.is_near(self.gas_station)
            self.is_interacting_with(self.gas_station)

            # Get the required amount of fuel
            liters_required = FUEL_TANK_SIZE - fuel_tank_level
```

```
            yield self.fuel_pump.get(liters_required)

            # The "actual" refueling process takes some time
            yield self.env.timeout(liters_required / REFUELING_SPEED)
            self.is_no_longer_interacting_with(self.gas_station)

        self.is_invisible()


def gas_station_control(env, fuel_pump):
    """Periodically check the level of the *fuel_pump* and call the tank
    truck if the level falls below a threshold."""
    truck = TankTruck(env, fuel_pump)
    while True:
        if fuel_pump.level / fuel_pump.capacity * 100 < THRESHOLD:
            # We need to call the tank truck now!
            # Wait for the tank truck to arrive and refuel the station
            yield env.process(truck.run())

        yield env.timeout(10)  # Check every 10 seconds


class TankTruck(VisualProcess):
    def __init__(self, env, fuel_pump):
        super().__init__(env, "Tank Truck", visual="TANK_TRUCK", tint=0xFF0000)
        self.fuel_pump = fuel_pump

    def run(self):
        self.is_visible()
        self.is_at(0, 0)
        self.is_near(self.fuel_pump)
        self.is_interacting_with(self.fuel_pump)
        yield self.env.timeout(TANK_TRUCK_TIME)
        ammount = self.fuel_pump.capacity - self.fuel_pump.level
        yield self.fuel_pump.put(ammount)
        self.is_no_longer_interacting_with(self.fuel_pump)
        self.is_invisible()


def car_generator(env, gas_station, fuel_pump):
    """Generate new cars that arrive at the gas station."""
    for i in itertools.count():
        yield env.timeout(random.randint(*T_INTER))
        car = Car("Car %d" % i, env, gas_station, fuel_pump)
        env.process(car.run())


class GasStation(VisualResource):
    def __init__(self, env):
        super().__init__(env, "Gas Station", 3, visual="GAS_STATION",
                         tint=0x00FF00)
        self.is_at(3, 1)
        self.is_visible()

```

```
109
110 class FuelPump(VisualContainer):
111     def __init__(self, env):
112         super().__init__(
113             env,
114             "Fuel Pump",
115             capacity=GAS_STATION_SIZE,
116             init=GAS_STATION_SIZE,
117             visual="FUEL_PUMP",
118             tint=0x000001,
119         )
120         self.is_at(1, 1)
121         self.is_visible()
122         self.has_frame(4)
123
124     def update_sprite(self):
125         fillPercentage = self.level / self.capacity
126         if fillPercentage < 0.25:
127             self.has_frame(0)
128         elif fillPercentage < 0.5:
129             self.has_frame(1)
130         elif fillPercentage < 0.75:
131             self.has_frame(2)
132         elif fillPercentage < 1:
133             self.has_frame(3)
134         else:
135             self.has_frame(4)
136
137     def get(self, amount):
138         cget = super().get(amount)
139         self.update_sprite()
140         return cget
141
142     def put(self, amount):
143         cput = super().put(amount)
144         self.update_sprite()
145         return cput
146
147
148 # Setup and start the simulation
149 random.seed(RANDOM_SEED)
150
151 # Create environment and start processes
152 env = VisualEnvironment()
153 env.visualization_manager.register_visual("CAR", "./gas_station_assets/car.png")
154 env.visualization_manager.register_visual(
155     "TANK_TRUCK", "./gas_station_assets/truck.png")
156 env.visualization_manager.register_visual(
157     "GAS_STATION", "./gas_station_assets/gaspump.png")
158 env.visualization_manager.register_sprites(
159     "FUEL_PUMP",
160     [
161         "./gas_station_assets/pump_000.png",
162         "./gas_station_assets/pump_025.png",
```

```
        "./gas_station_assets/pump_050.png",
        "./gas_station_assets/pump_075.png",
        "./gas_station_assets/pump_100.png",
    ],
)

grid = VisualGrid(500, 500, 5, 3)
grid.set_area("gasstation01", "GAS_STATION", 3, 5, 0, 0, 0xbdbbbb)
env.visualization_manager.set_grid(grid)

gas_station = GasStation(env)
fuel_pump = FuelPump(env)
env.process(gas_station_control(env, fuel_pump))
env.process(car_generator(env, gas_station, fuel_pump))

# Execute!
env.run(until=SIM_TIME)

# you can extract the output from the visualization manager
output = env.visualization_manager.serialize()

# or save the output to a file directly
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in jupyter lab and have the simplay-jupyter
# extensions installed, you can display the output

# if so, uncomment the following lines

#from IPython.display import display
#display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.4.6 Process Communication

NOTE: The following images are referenced in the example below.

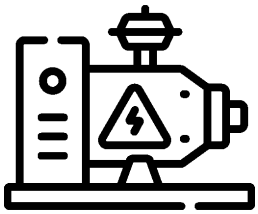


Abbildung 13.18: generator

Generator icons created by Freepik - Flaticon

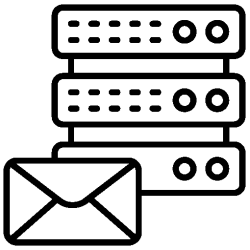


Abbildung 13.19: message_consumer

Seo and web icons created by BizzBox - Flaticon

```
"""
Process communication example

Covers:

- Resources: Store

Scenario:
  This example shows how to interconnect simulation model elements
  together using :class:`~simpy.resources.store.Store` for one-to-one,
  and many-to-one asynchronous processes. For one-to-many a simple
  BroadCastPipe class is constructed from Store.

When Useful:
  When a consumer process does not always wait on a generating process
  and these processes run asynchronously. This example shows how to
  create a buffer and also tell is the consumer process was late
  yielding to the event from a generating process.

  This is also useful when some information needs to be broadcast to
  many receiving processes

  Finally, using pipes can simplify how processes are interconnected to
  each other in a simulation model.

Example By:
  Keith Smith

"""
import random

from simplay import VisualEnvironment, VisualStore, VisualGrid, VisualProcess

import simpy


RANDOM_SEED = 42
SIM_TIME = 100

output_conn_positions = [[0, 1], [2, 1]]
generator_positions = [[1, 0], [2, 0]]
```

```
42
43
44 class BroadcastPipe(object):
45     """A Broadcast pipe that allows one process to send messages to many.
46
47     This construct is useful when message consumers are running at
48     different rates than message generators and provides an event
49     buffering to the consuming processes.
50
51     The parameters are used to create a new
52     :class:`~simpy.resources.store.Store` instance each time
53     :meth:`get_output_conn()` is called.
54
55     """
56
57     def __init__(self, env, capacity=1000):
58         self.env = env
59         self.capacity = capacity
60         self.pipes = []
61
62     def put(self, value):
63         """Broadcast a *value* to all receivers."""
64         if not self.pipes:
65             raise RuntimeError('There are no output pipes.')
66         events = [store.put(value) for store in self.pipes]
67         return self.env.all_of(events)  # Condition event for all "events"
68
69     def get_output_conn(self):
70         """Get a new output connection for this broadcast pipe.
71
72         The return value is a :class:`~simpy.resources.store.Store`.
73
74         """
75         position = output_conn_positions.pop(0)
76         pipe = VisualStore(
77             self.env, "pipe" + str(position[0]) + str(position[1]), "PIPE",
                 capacity=self.capacity)
78         pipe.is_at(position[0], position[1])
79         pipe.is_visible()
80         self.pipes.append(pipe)
81         return pipe
82
83     def has_tint(self, color):
84         [store.has_tint(color) for store in self.pipes]
85
86
87 class Message_generator(VisualProcess):
88
89     def __init__(self, env: VisualEnvironment, id: str, out_pipe: VisualStore):
90         super().__init__(env, id, "GENERATOR", 0xFFFFFF)
91         self.out_pipe = out_pipe
92         position = generator_positions.pop(0)
93         self.is_at(position[0], position[1])
94         self.is_visible()
```

```
95
96      def run(self):
97          """A process which randomly generates messages."""
98          while True:
99              # wait for next transmission
100             yield self.env.timeout(random.randint(6, 10))
101
102             # messages are time stamped to later check if the consumer was
103             # late getting them.  Note, using event.triggered to do this may
104             # result in failure due to FIFO nature of simulation yields.
105             # (i.e. if at the same env.now, message_generator puts a message
106             # in the pipe first and then message_consumer gets from pipe,
107             # the event.triggered will be True in the other order it will be
108             # False
109             msg = (self.env.now, '%s says hello at %d' %
110                    (self.id, self.env.now))
111             if hasattr(self.out_pipe, "pipes"):
112                 [self.is_interacting_with(store)
113                  for store in self.out_pipe.pipes]
114             else:
115                 self.is_interacting_with(self.out_pipe)
116             self.out_pipe.put(msg)
117             # If the out_pipe has a delay in consuming the pipe will be marked yellow
118             self.out_pipe.has_tint(0xFFFF00)
119             yield self.env.timeout(1)
120             if hasattr(self.out_pipe, "pipes"):
121                 [self.is_no_longer_interacting_with(
122                     store) for store in self.out_pipe.pipes]
123             else:
124                 self.is_no_longer_interacting_with(self.out_pipe)
125
126
127 def message_consumer(name, env: VisualEnvironment, in_pipe: VisualStore):
128     """A process which consumes messages."""
129     while True:
130         # Get event for message pipe
131         msg = yield in_pipe.get()
132
133         if msg[0] < env.now:
134             # if message was already put into pipe, then
135             # message_consumer was late getting to it. Depending on what
136             # is being modeled this, may, or may not have some
137             # significance
138             print('LATE Getting Message: at time %d: %s received message: %s' %
139                   (env.now, name, msg[1]))
140             in_pipe.has_decorating_text('LATE Getting Message: at time %d: %s received
                message: \n %s' %
141                                         (env.now, name, msg[1]))
142             in_pipe.has_tint(0xFF0000)
143
144         else:
145             # message_consumer is synchronized with message_generator
146             print('at time %d: %s received message: %s.' %
147                   (env.now, name, msg[1]))
```

```
            in_pipe.has_decorating_text(str('at time %d: %s received message: \n %s.' %
                                            (env.now, name, msg[1])))
            in_pipe.has_tint(0x00FF00)

        # Process does some other work, which may result in missing messages
        yield env.timeout(random.randint(4, 8))


# Setup and start the simulation
print('Process communication')
random.seed(RANDOM_SEED)
env = VisualEnvironment()
env.visualization_manager.register_visual(
    "PIPE", "./process_communication_assets/message_consumer.png")
env.visualization_manager.register_visual(
    "GENERATOR", "./process_communication_assets/generator.png")
grid = VisualGrid(1000, 1000, 3, 3)
env.visualization_manager.set_grid(grid)

# For one-to-one or many-to-one type pipes, use Store
pipe = VisualStore(env, "pipe", "PIPE", capacity=1000)
pipe.is_at(1, 1)
pipe.is_visible()
env.process(Message_generator(env, 'Generator A', pipe).run())
env.process(message_consumer('Consumer A', env, pipe))
print('\nOne-to-one pipe communication\n')

# For one-to many use BroadcastPipe comment out the lines 176 to 181 and comment the
    lines 168 to 173
# (Note: could also be used for one-to-one,many-to-one or many-to-many)
# bc_pipe = BroadcastPipe(env)
# env.process(Message_generator(env, 'Generator A', bc_pipe).run())
# env.process(message_consumer('Consumer A', env, bc_pipe.get_output_conn()))
# env.process(message_consumer('Consumer B', env, bc_pipe.get_output_conn()))
# print('\nOne-to-many pipe communication\n')

env.run(until=SIM_TIME)

# you can extract the output from the visualization manager
output = env.visualization_manager.serialize()

# or save the output to a file directly
env.visualization_manager.write_to_file("output.simplay")

# or, if you're working in jupyter lab and have the simplay-jupyter
# extensions installed, you can display the output

# if so, uncomment the following lines
# from IPython.display import display
# display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.4.7 Event Latency

NOTE: The following images are referenced in the example below.

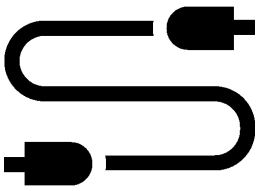


Abbildung 13.20: cable

Cable icons created by Freepik - Flaticon

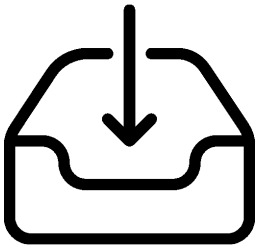


Abbildung 13.21: receiver

Inbox icons created by Pixel perfect - Flaticon

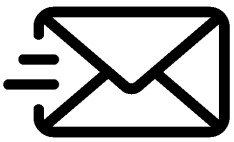


Abbildung 13.22: sender

Email Icons created by IconMark - Flaticon

```
"""
Event Latency example
Covers:
- Resources: Store
Scenario:
  This example shows how to separate the time delay of events between
  processes from the processes themselves.
When Useful:
  When modeling physical things such as cables, RF propagation, etc.  it
  better encapsulation to keep this propagation mechanism outside of the
  sending and receiving processes.
  Can also be used to interconnect processes sending messages
Example by:
  Keith Smith
```

```
"""
import simpy
from simplay import VisualEnvironment, VisualGrid, VisualProcess, VisualStore


SIM_DURATION = 100


class Cable(VisualStore):
    """This class represents the propagation through a cable."""

    def __init__(self, env, delay):
        super().__init__(env, "Cable", "CABLE", capacity=1000)
        self.delay = delay
        self.is_at(1, 0)
        self.number_of_messages_in_cable = 0
        self.is_visible()

    def latency(self, value):
        yield self.env.timeout(self.delay)
        super().put(value)

    def put(self, value):
        self.number_of_messages_in_cable += 1
        self.has_decorating_text(
            str('number of messages in cable %d' % self.number_of_messages_in_cable))
        self.has_tint(0xFFFF00)
        self.env.process(self.latency(value))

    def get(self):
        if self.number_of_messages_in_cable > 0:
            self.number_of_messages_in_cable -= 1
        self.has_decorating_text(
            str('number of messages in cable %d' % self.number_of_messages_in_cable))
        return super().get()


class Sender(VisualProcess):
    def __init__(self, env: VisualEnvironment, cable: Cable):
        super().__init__(env, "Sender", "SENDER")
        self.cable = cable
        self.is_at(0, 0)
        self.is_visible()

    def run(self):
        """A process which randomly generates messages."""
        while True:
            # wait for next transmission
            yield self.env.timeout(4)
            self.is_interacting_with(self.cable)
            yield self.env.timeout(1)
            self.cable.put('Sender sent this at %d' % self.env.now)
            self.is_no_longer_interacting_with(self.cable)

```

```
69
70 class Receiver(VisualProcess):
71     def __init__(self, env: VisualEnvironment, cable: Cable):
72         super().__init__(env, "Receiver", "RECEIVER")
73         self.cable = cable
74         self.is_at(2, 0)
75         self.is_visible()
76
77     def run(self):
78         """A process which consumes messages."""
79         while True:
80             # Get event for message pipe
81             msg = yield self.cable.get()
82             self.is_interacting_with(self.cable)
83             yield self.env.timeout(1)
84             print('Received this at %d while %s' % (self.env.now, msg))
85             self.has_decorating_text(
86                 str('Received this at %d while %s' % (self.env.now, msg)))
87             self.is_no_longer_interacting_with(self.cable)
88
89
90 # Setup and start the simulation
91 print('Event Latency')
92 env = VisualEnvironment()
93 grid = VisualGrid(1000, 1000, 3, 1)
94 env.visualization_manager.set_grid(grid)
95 env.visualization_manager.register_visual(
96     "SENDER", "./event_latency_assets/sender.png")
97 env.visualization_manager.register_visual(
98     "RECEIVER", "./event_latency_assets/receiver.png")
99 env.visualization_manager.register_visual(
100     "CABLE", "./event_latency_assets/cable.png")
101
102 cable = Cable(env, 9)
103 env.process(Sender(env, cable).run())
104 env.process(Receiver(env, cable).run())
105
106 env.run(until=SIM_DURATION)
107
108 # you can extract the output from the visualization manager
109 output = env.visualization_manager.serialize()
110
111 # or save the output to a file directly
112 env.visualization_manager.write_to_file("output.simplay")
113
114 # or, if you're working in jupyter lab and have the simplay-jupyter
115 # extensions installed, you can display the output
116
117 # if so, uncomment the following lines
118 # from IPython.display import display
119 # display(env.visualization_manager.serialize_for_jupyter(), raw=True)
```

13.5.5 SimPlay in depth

How a simulation is set up, and how to work with processes is explained in Getting started.

13.5.5.1 Using Resources

The following example shows how to use the `VisualResource` class:

```
from simplay import VisualEnvironment, VisualResource

class MyResource(VisualResource):
    def __init__(self, env):
        super().__init__(env, "MyResource", 3, visual="SOMEPNG", tint=0x00FF00)
        self.is_at(5, 5)
        self.is_visible()

env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
env.visualization_manager.set_grid(grid)

resource = MyResource(env)
env.run()
```

The `VisualResource` class inherits from the `Resource` class from the `simpy` package. The API is the same, except that the `request` and `release` methods are overridden to reflect for changes in the utilization and capacity of the resource, and will automatically create the corresponding events. Specialized classes like `PreemptiveResource` and `PriorityResource` are also supported, and are inherited by the `VisualPreemptiveResource` and `VisualPriorityResource` respectively.

The code example above creates a custom class for the resource, and by doing so declares the visibility and position of the resource in the constructor. Alternatively, you can create the resource without a custom class.

```
from simplay import VisualEnvironment, VisualResource

env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
env.visualization_manager.set_grid(grid)

resource = VisualResource(env, "MyResource", 3, visual="SOMEPNG", tint=0x00FF00)
resource.is_at(5, 5)
resource.is_visible()

env.run()
```

13.5.5.2 Using Containers

The following example shows how to use the `VisualContainer` class:

```
from simplay import VisualEnvironment, VisualContainer

class MyContainer(VisualContainer):
    def __init__(self, env):
        super().__init__(env, "MyContainer", 3, visual="SOMEPNG", tint=0x00FF00)
        self.is_at(5, 5)
        self.is_visible()
```

```

env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
env.visualization_manager.set_grid(grid)

container = MyContainer(env)
env.run()
```

The `VisualContainer` class inherits from the `Container` class from the `simpy` package. The API is the same, except that the `put` and `get` methods are overridden to reflect for changes in the level and capacity of the container, and will automatically create the corresponding events.

The code example above creates a custom class for the container, and by doing so declares the visibility and position of the container. Alternatively, you can create the container without a custom class.

```
from simplay import VisualEnvironment, VisualContainer

env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
env.visualization_manager.set_grid(grid)

container = VisualContainer(env, "MyContainer", 3, visual="SOMEPNG", tint=0x00FF00)
container.is_at(5, 5)
container.is_visible()

env.run()
```

13.5.5.3 Using Stores

The following example shows how to use the `VisualStore` class:

```
from simplay import VisualEnvironment, VisualStore

class MyStore(VisualStore):
    def __init__(self, env):
        super().__init__(env, "MyStore", 3, visual="SOMEPNG", tint=0x00FF00)
        self.is_at(5, 5)
        self.is_visible()

env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
env.visualization_manager.set_grid(grid)

store = MyStore(env)
env.run()
```

The `VisualStore` class inherits from the `Store` class from the `simpy` package. The API is the same, except that the `put` and `get` methods are overridden to reflect for changes in the contents and capacity of the store, and will automatically create the corresponding events.

The specialized [`FilterStore`] is also supported, and is inherited by the `VisualFilterStore` class.

The code example above creates a custom class for your store, and by doing so declares the visibility and position of the store. Alternatively, you can create the store without a custom class.

```
from simplay import VisualEnvironment, VisualStore

env = VisualEnvironment()
grid = VisualGrid(1000, 1000, 10, 10)
grid.set_area("area51", "ALIENS!", 5, 2, 0, 0, 0xFF0000)
env.visualization_manager.set_grid(grid)

store = VisualStore(env, "MyStore", 3, visual="SOMEPNG", tint=0x00FF00)
store.is_at(5, 5)
store.is_visible()

env.run()
```

More details about the functions and classes of SimPlay are available in API Reference.

13.5.6 API Reference

The API reference provides detailed descriptions of SimPlay's classes and functions.

- simplay
- simplay.core – Core components of Simplay
- simplay.visualization — Visualization
- simplay.components — Components
- simplay.events — Visual Event Types
- simplay.primitives — Primitives

13.5.7 simplay

The simplay module aggregates SimPlay's classes and functions. It is the only module that is imported by default when importing SimPlay.

The following tables list all of the available components in this module.

13.5.7.1 Core

VisualEnvironment([initial_time])	Extends the Environment class with visualization.
VisualComponent(env, id, type, visual[, tint])	Base class for all visual components.
VisualizationManager()	This class acts as a central point for all entities, visuals and events.

13.5.7.2 Components

VisualProcess(env, id, visual[, tint])	Shorthand for creating an entity of type PROCESS.
VisualResource(env, id, capacity, visual[, tint])	Extends the Resource class with
VisualContainer(env, id, visual[, tint, ...])	Extends the Container class with
VisualPreemptiveResource(env, id, capacity, ...)	Extends the PreemptiveResource class
VisualPriorityResource(env, id, capacity, visual)	Extends the PriorityResource class with
VisualStore(env, id, visual[, tint, capacity])	Extends the Store class with visualization.
VisualFilterStore(env, id, capacity, visual)	Extends the FilterStore class with

13.5.7.3 Visualization

VisualGrid(width, height, cols, rows)	This class represents a grid for the visualization.

13.5.7.4 Events

VisualEvent(for_id, timestamp, action, **kwargs)	Base class for all visual events.
SetVisible(for_id, timestamp, visible)	Event to set the visibility of a component.
SetPosition(for_id, timestamp, x, y)	Event to set the position of a component.
SetInteracting(for_id, timestamp, with_id)	Event to set the interaction of a component.
SetNotInteracting(for_id, timestamp, with_id)	Event to set the interaction of a component.
MoveNear(for_id, timestamp, target_id)	Event to move a component near another component.
MoveNearCell(for_id, timestamp, x, y)	Event to move a component near a cell in the grid.
SetTintColor(for_id, timestamp, color)	Event to set the tint color of a component.
SetSpriteFrame(for_id, timestamp, frame)	Event to set the sprite frame of a component.
SetDecoratingText(for_id, timestamp, text)	Event to set the decorating text of a component.
ResourceSetCapacity(for_id, timestamp, capacity)	Event to set the capacity of a resource.
ResourceSetUtilization(for_id, timestamp, ...)	Event to set the utilization of a resource.
ContainerSetCapacity(for_id, timestamp, capacity)	Event to set the capacity of a container.
ContainerSetLevel(for_id, timestamp, level)	Event to set the level of a container.
StoreSetCapacity(for_id, timestamp, capacity)	Event to set the capacity of a store.
StoreSetContent(for_id, timestamp, content)	Event to set the content of a store.

13.5.7.5 Primitives

ComponentType(value)	Enum for component types.
EventAction(value)	Enum for event actions.

13.5.8 `simplay.core` – Core components of Simplay

13.5.8.0.1 *class* simplay.core.VisualEnvironment(initial_time: Union[int, float] = 0)

Bases: `Environment`

Extends the `Environment` class with visualization.

13.5.8.0.2 *class* simplay.core.VisualComponent(env: VisualEnvironment, id: str, type: ComponentType, visual: str, tint: int = 16777215)

Base class for all visual components.

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **type** – The type of the component, one of `ComponentType`..
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.
- **Raises**
  - **TypeError** – If the type is invalid.
  - **TypeError** – If the id is not a string.
  - **TypeError** – If the environment is not a `VisualEnvironment`.
  - **TypeError** – If the visual is not a string.
  - **TypeError** – If the tint is not an integer.

13.5.8.1 *static* create_custom_component(env: VisualEnvironment, id: str, visual: str, pos_x: int = 0, pos_y: int = 0, visible: bool = True, tint: int = 16777215)

Creates a non-simulation component, i.e. a component that is not part of the simulation, but can be used to visualize other components.

The positional parameters are used to directly set the position of the component. If you want to move the component later, use the `is_at()` method.

- **Parameters**
  - **env** – The environment to create the component in.
  - **id** – The id of the component.
  - **visual** – The visual to use for the component.
  - **pos_x** – The x coordinate of the component.
  - **pos_y** – The y coordinate of the component.
  - **visible** – Whether the component should be visible.
  - **tint** – The tint color of the component.
- **Returns**

  The created component.

13.5.8.2 has_decorating_text(text: str)

Adds an `SetDecoratingText` event for the given component to the EventQueue.

- **Parameters**

  text – The text to display.

13.5.8.3 has_frame(frame: int)

Adds an `SetSpriteFrame` event for the given component to the EventQueue.

- **Parameters**

  frame – The index of the frame to display.

13.5.8.4 has_original_tint()

Adds an `SetTintColor` event for the given component to the EventQueue, resetting the tint color to its initial value.

13.5.8.5 has_tint(color: int)

Adds an `SetTintColor` event for the given component to the EventQueue.

- **Parameters**

  color – The color to tint the component with, as an integer. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue.

13.5.8.6 is_at(x: int, y: int)

Adds an `SetPosition` event for the given component to the EventQueue.

- **Parameters**
  - **x** – The x coordinate of the component.
  - **y** – The y coordinate of the component.

13.5.8.7 is_interacting_with(target: VisualComponent)

Adds an `SetInteracting` event for the given component to the EventQueue.

- **Parameters**

  target – The component the first component is interacting with.

- **Raises**

  TypeError – If the target is not a `VisualComponent`.

13.5.8.8 is_invisible()

Adds an `SetVisible` event for the given component to the EventQueue, making it invisible.

13.5.8.9 is_near(target: VisualComponent)

Adds an `MoveNear` event for the given component to the EventQueue.

- **Parameters**

  target – The target component.

- **Raises**

  TypeError – If the target is not a `VisualComponent`.

13.5.8.10 is_near_cell(x: int, y: int)

Adds an `MoveNearCell` event for the given component to the EventQueue.

- **Parameters**
  - **x** – The x coordinate of the target cell.
  - **y** – The y coordinate of the target cell.

13.5.8.11 is_no_longer_interacting_with(target: VisualComponent)

Adds an `SetNotInteracting` event for the given component to the EventQueue.

- **Parameters**

  target – The component the first component is not interacting with anymore.
- **Raises**

  TypeError – If the target is not a `VisualComponent`.

13.5.8.12 is_visible()

Adds an `SetVisible` event for the given component to the EventQueue, making it visible.

13.5.8.12.1 *class* simplay.core.VisualizationManager()

This class acts as a central point for all entities, visuals and events.

13.5.8.13 add_entity(entity: VisualComponent, type: ComponentType)

Add an entity to the visualization.

- **Parameters**
  - **entity** – The entity to add.
  - **type** – The type of the entity, one of `ComponentType`.
- **Raises**
  - **TypeError** – If the type is invalid.
  - **TypeError** – If the entity is not a `VisualComponent`.

13.5.8.14 add_event(event: VisualEvent)

Add an event to the visualization.

- **Parameters**

  event – The event to add.

13.5.8.15 entities()

The entities of the simulation.

13.5.8.16 events()

The events that happened in the simulation.

13.5.8.17 grid()

The grid that is used for the visualization.

13.5.8.18 register_sprites(id: str, frames: List[str])

Register sprites with the manager.

- **Parameters**
  - **id** – The id of the visual, it must be unique and can be used to reference the visual in components.
  - **frames** – A list of paths to the frames of the sprite.
- **Raises**
  - **TypeError** – If the id is not a string.
  - **TypeError** – If the frames are not a list.
  - **ValueError** – If the id is not unique.
  - **ValueError** – If the frames are empty.

13.5.8.19 register_visual(id: str, path: str)

Register a visual with the manager. The visual is a sprite with a list of frames whose length is equal to one.

- **Parameters**
  - **id** – The id of the visual, it must be unique and can be used to reference the visual in components.
  - **path** – The path to the visual.
- **Raises**
  - **TypeError** – If the id is not a string.
  - **TypeError** – If the path is not a string.
  - **ValueError** – If the id is not unique.
  - **ValueError** – If the path is empty.

13.5.8.20 serialize()

Serialize the visualization to a JSON string.

13.5.8.21 serialize_for_jupyter()

Serialize the visualization for use with Jupyter.

13.5.8.22 set_grid(grid: VisualGrid)

Set the grid for the visualization.

- **Parameters**

  grid – The grid to use.
- **Raises**
  - **TypeError** – If the grid is not a `VisualGrid`.
  - **ValueError** – If the grid is None.

13.5.8.23 visuals()

The visuals that have been registered with the manager.

13.5.8.24 write_to_file(filename: str)

Write the visualization to a file.

- **Parameters**

  filename – The name of the file to write to.

13.5.9 `simplay.visualization` — Visualization

13.5.9.0.1 *class* simplay.visualization.VisualGrid(width: int, height: int, cols: int, rows: int)

This class represents a grid for the visualization.

- **Parameters**
  - **width** – The width of the grid in pixels.
  - **height** – The height of the grid in pixels.
  - **cols** – The number of columns in the grid.
  - **rows** – The number of rows in the grid.

13.5.9.1 set_area(id: str, name: str, height: int, width: int, x: int, y: int, color: int)

Set an area in the grid.

- **Parameters**
  - **id** – The id of the area.
  - **name** – The name of the area.
  - **height** – The height of the area in rows.
  - **width** – The width of the area in columns.
  - **x** – The x coordinate (column) of the top left corner of the area.
  - **y** – The y coordinate (row) of the top left corner of the area.
  - **color** – The color of the area.
- **Raises**
  - **TypeError** – If the id is not a string.
  - **TypeError** – If the name is not a string.
  - **TypeError** – If the height is not a positive integer.
  - **TypeError** – If the width is not a positive integer.
  - **TypeError** – If the x is not an integer.
  - **TypeError** – If the y is not an integer.
  - **TypeError** – If the color is not an integer.
  - **ValueError** – If the id is not unique.
  - **ValueError** – If the area overlaps with another area.
  - **ValueError** – If x is negative or greater than the number of columns.
  - **ValueError** – If y is negative or greater than the number of rows.
  - **ValueError** – If x + width is greater than the number of columns.

- **ValueError** – If y + height is greater than the number of rows.

13.5.10 `simplay.components` — Components

13.5.10.0.1 *class* simplay.components.VisualProcess(env: VisualEnvironment, id: str, visual: str, tint: int = 16777215)

Bases: `VisualComponent`

Shorthand for creating an entity of type `PROCESS`.

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.

13.5.10.0.2 *class* simplay.components.VisualResource(env: VisualEnvironment, id: str, capacity: int, visual: str, tint: int = 16777215)

Bases: `VisualComponent`, `Resource`

Extends the `Resource` class with

```
1 visualization.
```

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **capacity** – The capacity of the resource.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.
- **Raises**
  - **ValueError** – If the capacity is not a positive integer.
  - **TypeError** – If the capacity is not a integer.

13.5.10.1 *property* capacity(*: Union[int, float*)

Maximum capacity of the resource.

13.5.10.2 release(request: Request)

Releases the usage of *resource* granted by *request*. This event is triggered immediately. Subclass of `simpy.resources.base.Get`.

13.5.10.3 request()

Request usage of the *resource*. The event is triggered once access is granted. Subclass of `simpy.resources.base.Put`.

If the maximum capacity of users has not yet been reached, the request is triggered immediately. If the maximum capacity has been reached, the request is triggered once an earlier usage request on the resource is released.

The request is automatically released when the request was created within a `with` statement.

13.5.10.3.1 *class* simplay.components.VisualPreemptiveResource(env: VisualEnvironment, id: str, capacity: int, visual: str, tint: int = 16777215)

Bases: `VisualComponent`, `PreemptiveResource`

Extends the `PreemptiveResource` class

```
1 with visualization.
```

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **capacity** – The capacity of the resource.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.
- **Raises**
  - **TypeError** – If the capacity is not a integer.
  - **ValueError** – If the capacity is not a positive integer.

13.5.10.4 *property* capacity(*: Union[int, float*)

Maximum capacity of the resource.

13.5.10.5 release(request: PriorityRequest)

Releases the usage of *resource* granted by *request*. This event is triggered immediately. Subclass of `simpy.resources.base.Get`.

13.5.10.6 request(priority: int = 0, preempt: bool = True)

Request the usage of *resource* with a given *priority*. If the *resource* supports preemption and *preempt* is `True` other usage requests of the *resource* may be preempted (see `PreemptiveResource` for details).

This event type inherits `Request` and adds some additional attributes needed by `PriorityResource` and `PreemptiveResource`

13.5.10.7 users(*: List[PriorityRequest*)

List of `Request` events for the processes that are currently using the resource.

13.5.10.7.1 *class* simplay.components.VisualPriorityResource(env: VisualEnvironment, id: str, capacity: int, visual: str, tint: int = 16777215)

Bases: `VisualComponent`, `PriorityResource`

Extends the `PriorityResource` class with

```
visualization.
```

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **capacity** – The capacity of the resource.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.
- **Raises**
  - **TypeError** – If the capacity is not a integer.
  - **ValueError** – If the capacity is not a positive integer.

13.5.10.8 *property* capacity(*: Union[int, float*)

Maximum capacity of the resource.

13.5.10.9 release(request: PriorityRequest)

Releases the usage of *resource* granted by *request*. This event is triggered immediately. Subclass of `simpy.resources.base.Get`.

13.5.10.10 request(priority: int = 0, preempt: bool = True)

Request the usage of *resource* with a given *priority*. If the *resource* supports preemption and *preempt* is `True` other usage requests of the *resource* may be preempted (see `PreemptiveResource` for details).

This event type inherits `Request` and adds some additional attributes needed by `PriorityResource` and `PreemptiveResource`

13.5.10.10.1 *class* simplay.components.VisualStore(env: VisualEnvironment, id: str, visual: str, tint: int = 16777215, capacity: Union[float, int] = inf)

Bases: `VisualComponent`, `Store`

Extends the `Store` class with visualization.

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **capacity** – The capacity of the store.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.

- **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.

- **Raises**
  - **TypeError** – If the capacity is not a integer or float.
  - **ValueError** – If the capacity is not a positive integer or float.

13.5.10.11 *property* capacity(*: Union[float, int*)

Maximum capacity of the resource.

13.5.10.12 get()

Request to get an *item* from the *store*. The request is triggered once there is an item available in the store.

13.5.10.13 put(item)

Request to put *item* into the *store*. The request is triggered once there is space for the item in the store.

13.5.10.13.1 *class* simplay.components.VisualFilterStore(env: VisualEnvironment, id: str, capacity: int, visual: str, tint: int = 16777215)

Bases: `VisualComponent`, `FilterStore`

Extends the `FilterStore` class with

```
visualization.
```

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **capacity** – The capacity of the store.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.

- **Raises**
  - **TypeError** – If the capacity is not a integer or float.
  - **ValueError** – If the capacity is not a positive integer or float.

13.5.10.14 *property* capacity(_: int

Maximum capacity of the resource.

13.5.10.15 get(filter: ~typing.Callable[[~typing.Any], bool] = <function VisualFilterStore.>)

Request to get an *item* from the *store* matching the *filter*. The request is triggered once there is such an item available in the store.

filter is a function receiving one item. It should return `True` for items matching the filter criterion. The default function returns `True` for all items, which makes the request to behave exactly like `StoreGet`.

13.5.10.16 put(item)

Request to put *item* into the *store*. The request is triggered once there is space for the item in the store.

13.5.10.16.1 *class* simplay.components.VisualContainer(env: VisualEnvironment

Bases: `VisualComponent`, `Container`

Extends the `Container` class with

```
1 visualization.
```

- **Parameters**
  - **env** – The environment instance.
  - **id** – The id of the component.
  - **capacity** – The capacity of the container.
  - **init** – The initial amount of the container.
  - **visual** – The visualization of the component, must be registered in the `VisualizationManager`.
  - **tint** – The tint of the component. The tint is multiplied with the pixel value of each pixel. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue. If the whole image is white, tinting it will change the color of the image. If the image is black, tinting it will have no effect. If no tint should be applied, set it to 0xFFFFFF, which is the default value.
- **Raises**
  - **TypeError** – If the capacity is not a integer or float.
  - **ValueError** – If the capacity is not a positive integer or float.

13.5.10.17 *property* capacity(*: Union[int, float*)

Maximum capacity of the resource.

13.5.10.18 get(amount: Union[int, float])

Request to get *amount* of matter from the *container*. The request will be triggered once there is enough matter available in the *container*.

Raise a `ValueError` if `amount <= 0`.

13.5.10.19 put(amount: Union[int, float])

Request to put *amount* of matter into the *container*. The request will be triggered once there is enough space in the *container* available.

Raise a `ValueError` if `amount <= 0`.

13.5.11 `simplay.events` — Visual Event Types

13.5.11.0.1 *class* simplay.events.VisualEvent(for_id: str, timestamp: Union[int, float], action: EventAction, **kwargs: dict)

Base class for all visual events.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **action** – The action to be performed.
  - **kwargs** – The arguments for the action.
- **Raises**
  - **TypeError** – If component is not a string.
  - **TypeError** – If timestamp is not a number.
  - **TypeError** – If action is not a string.

13.5.11.0.2 *class* simplay.events.SetVisible(for_id: str, timestamp: Union[int, float], visible: bool)

Event to set the visibility of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **visible** – Whether the component should be visible or not.
- **Raises**

  TypeError – If visible is not a boolean.

13.5.11.0.3 *class* simplay.events.SetPosition(for_id: str, timestamp: Union[int, float], x: int, y: int)

Event to set the position of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **x** – The x coordinate of the component.
  - **y** – The y coordinate of the component.
- **Raises**
  - **TypeError** – If x is not an integer.
  - **TypeError** – If y is not an integer.

13.5.11.0.4 *class* simplay.events.SetInteracting(for_id: str, timestamp: Union[int, float], with_id: str)

Event to set the interaction of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **with_id** – The id of the component the first component is interacting with.
- **Raises**

  TypeError – If with_id is not a string.

13.5.11.0.5 *class* simplay.events.SetNotInteracting(for_id: str, timestamp: Union[int, float], with_id: str)

Event to set the interaction of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **with_id** – The id of the component the first component is interacting with.
- **Raises**

  TypeError – If with_id is not a string.

13.5.11.0.6 *class* simplay.events.MoveNear(for_id: str, timestamp: Union[int, float], target_id: str)

Event to move a component near another component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **target_id** – The id of the component the first component should move near.
- **Raises**

  TypeError – If target_id is not a string.

13.5.11.0.7 *class* simplay.events.MoveNearCell(for_id: str, timestamp: Union[int, float], x: int, y: int)

Event to move a component near a cell in the grid.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **x** – The x coordinate (column) of the cell.
  - **y** – The y coordinate (row) of the cell.
- **Raises**

- **TypeError** – If x is not an integer.
- **TypeError** – If y is not an integer.

13.5.11.0.8 *class* simplay.events.SetTintColor(for_id: str, timestamp: Union[int, float], color: int)

Event to set the tint color of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **color** – The tint of the component. This only works with visuals and sprites that have transparent pixels. The tint is applied to the pixelsthat are not transparent. To use HEX values, write them as 0xRRGGBB. For example: 0xFF0000 is red, 0x00FF00 is green, 0x0000FF is blue.
- **Raises**

  TypeError – If color is not a string.

13.5.11.0.9 *class* simplay.events.SetSpriteFrame(for_id: str, timestamp: Union[int, float], frame: int)

Event to set the sprite frame of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **frame** – The frame to set the sprite to.
- **Raises**

  TypeError – If frame is not an integer.

13.5.11.0.10 *class* simplay.events.SetDecoratingText(for_id: str, timestamp: Union[int, float], text: str)

Event to set the decorating text of a component.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **text** – The text to set as decorating text.
- **Raises**

  TypeError – If text is not a string.

13.5.11.0.11 *class* simplay.events.ResourceSetCapacity(for_id: str, timestamp: Union[int, float], capacity: int)

Event to set the capacity of a resource.

- **Parameters**
  - **for_id** – The if of the component this event is for.

- **timestamp** – The timestamp of the event.
- **capacity** – The capacity to set the resource to.

- **Raises**

  TypeError – If capacity is not a positive integer.

13.5.11.0.12 *class* simplay.events.ResourceSetUtilization(for_id: str, timestamp: Union[int, float], utilization: int)

Event to set the utilization of a resource.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **utilization** – The utilization to set the resource to.
- **Raises**

  TypeError – If utilization is not a integer.

13.5.11.0.13 *class* simplay.events.ContainerSetCapacity(for_id: str, timestamp: Union[int, float], capacity: Union[int, float])

Event to set the capacity of a container.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **capacity** – The capacity to set the container to.
- **Raises**

  TypeError – If capacity is not a positive integer.

13.5.11.0.14 *class* simplay.events.ContainerSetLevel(for_id: str, timestamp: Union[int, float], level: Union[int, float])

Event to set the level of a container.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **level** – The level to set the container to.
- **Raises**

  TypeError – If level is not a positive integer or float.

13.5.11.0.15 *class* simplay.events.StoreSetCapacity(for_id: str, timestamp: Union[int, float], capacity: Union[float, int])

Event to set the capacity of a store.

- **Parameters**
  - **for_id** – The if of the component this event is for.

- **timestamp** – The timestamp of the event.
- **capacity** – The capacity to set the store to.

- **Raises**

  TypeError – If capacity is not a positive integer or float.

13.5.11.0.16 *class* simplay.events.StoreSetContent(for_id: str, timestamp: Union[int, float], content)

Event to set the content of a store.

- **Parameters**
  - **for_id** – The if of the component this event is for.
  - **timestamp** – The timestamp of the event.
  - **content** – The content to set the store to.

13.5.12 `simplay.primitives` — Primitives

The classes in this module most likely don't concern you. They are constants used by the other modules.

13.5.12.0.1 *class* simplay.primitives.ComponentType(value)

Enum for component types.

13.5.12.1 CONTAINER(_ = 'CONTAINER_)

13.5.12.2 CUSTOM(_ = 'CUSTOM_)

13.5.12.3 PROCESS(_ = 'PROCESS_)

13.5.12.4 RESOURCE(_ = 'RESOURCE_)

13.5.12.5 STORE(_ = 'STORE_)

13.5.12.5.1 *class* simplay.primitives.EventAction(value)

Enum for event actions.

13.5.12.6 CONTAINER*SET_CAPACITY(* = 'CONTAINER.SET*CAPACITY*)

13.5.12.7 CONTAINER*SET_LEVEL(* = 'CONTAINER.SET*LEVEL*)

13.5.12.8 MOVE*NEAR(* = 'MOVE*NEAR*)

13.5.12.9 MOVE*NEAR_CELL(* = 'MOVE*NEAR_CELL*)

13.5.12.10 RESOURCE*SET_CAPACITY(* = 'RESOURCE.SET*CAPACITY*)

13.5.12.11 RESOURCE*SET_UTILIZATION(* = 'RESOURCE.SET*UTILIZATION*)

13.5.12.12 SET*DECORATING_TEXT(* = 'SET*DECORATING_TEXT*)

13.5.12.13 SET*INTERACTING(* = 'SET*INTERACTING*)

13.5.12.14 SET*NOT_INTERACTING(* = 'SET*NOT_INTERACTING*)

13.5.12.15 SET*POSITION(* = 'SET*POSITION*)

13.5.12.16 SET*SPRITE_FRAME(* = 'SET*SPRITE_FRAME*)

13.5.12.17 SET*TINT_COLOR(* = 'SET*TINT_COLOR*)

13.5.12.18 SET*VISIBLE(* = 'SET*VISIBLE*)

13.5.12.19 STORE*SET_CAPACITY(* = 'STORE.SET*CAPACITY*)

13.5.12.20 STORE*SET_CONTENT(* = 'STORE.SET*CONTENT*)

13.6 User-Test Handout

13.6.1 User-Test 1: SimPlay in Jupyter

Dieses Dokument bietet die Hilfestellung zur Durchführung des User-Tests. Die Resultate des User-Tests sollen dafür verwendet werden können, die im Projekt aufgestellten Ziele zu überprüfen und zu optimieren.

13.6.1.1 Vorbereitung

Damit dieser User-Test durchgeführt werden kann, sind folgende Punkte vorausgesetzt:

- Funktionierende JupyterLab-Umgebung, Mindestversion 3.0
- Funktionierende Python-Umgebung, Mindestversion 3.9
- Vorhandene SimPy Simulation welche um Animation erweitert werden soll (hier kann ggf. eines der Beispiele von SimPy verwendet werden)
- Basiswissen über SimPy und JupyterLab
- Englischkenntnisse, um die Dokumentation von SimPlay zu lesen

13.6.1.2 Ausgansglage

Ein mögliches Szenario für die Anwendung von SimPlay ist, dass eine Benutzer:in von SimPlay erfährt, und sich entscheidet, eine bestehende SimPy Simulation um eine Animation zu erweitern. Die Simulation sowie die Animation soll dabei in JupyterLab ausgeführt werden.

Als Einstiegspunkt ist dabei die öffentliche Dokumentation von SimPlay zu verwenden. Diese Dokumentation ist unter https://simplay.readthedocs.io/en/latest/ zu finden.

Dies ist die einzige Hilfestellung, welche der Benutzer:in zur Verfügung steht.

13.6.1.3 Durchführung

Der User-Test soll in folgenden Schritten durchgeführt werden:

1. Die Benutzer:in soll die Dokumentation von SimPlay durchlesen und sich mit dem Projekt vertraut machen.
2. Die Benutzer:in soll die notwendigen Schritte durchführen, um SimPlay in JupyterLab zu installieren.
3. Die Benutzer:in soll eine SimPy Simulation auswählen, welche um eine Animation erweitert werden soll, und diese Simulation um eine Animation erweitern.
4. Die Benutzer:in soll die Simulation mit der Animation ausführen und die Animation betrachten.

13.6.1.4 Bewertungskriterien

Die Benutzer:in soll die folgenden Fragen beantworten:

- Wie lange hat die Benutzer:in für die Installation von SimPlay benötigt?
- Auf einer Skala von 1-5, wie einfach war es für die Benutzer:in, SimPlay in JupyterLab zu installieren?
- Auf einer Skala von 1-5, wie verständlich und hilfreich war die Dokumentation von SimPlay?
- Wie lange hat die Benutzer:in für die Erweiterung der Simulation benötigt?
- Auf einer Skala von 1-5, wie einfach war es für die Benutzer:in, die Simulation um eine Animation zu erweitern?
- Auf einer Skala von 1-5, wie gut funktioniert die Animation?
- Auf einer Skala von 1-5, wie wird die gesamte Benutzererfahrung von SimPlay bewertet?
- Gibt es Verbesserungsvorschläge für SimPlay, insbesondere für die Dokumentation?

13.6.2 User-Test 2: SimPlay in Jupyter

Dieses Dokument bietet die Hilfestellung zur Durchführung des User-Tests. Der zweite User-Test soll überprüfen, ob die Änderungen welche aufgrund des ersten User-Tests vorgenommen wurden, die gewünschten Ergebnisse erzielen.

13.6.2.1 Vorbereitung

Damit dieser User-Test durchgeführt werden kann, sind folgende Punkte vorausgesetzt:

- Funktionierende JupyterLab-Umgebung, Mindestversion 3.0
- Funktionierende Python-Umgebung, Mindestversion 3.9
- Vorhandene SimPy Simulation welche um Animation erweitert werden soll (hier kann ggf. eines der Beispiele von SimPy verwendet werden)
- Basiswissen über SimPy und JupyterLab
- Englischkenntnisse, um die Dokumentation von SimPlay zu lesen

13.6.2.2 Ausgangslage

Ein mögliches Szenario für die Anwendung von SimPlay ist, dass eine Benutzer:in von SimPlay erfährt, und sich entscheidet, eine bestehende SimPy Simulation um eine Animation zu erweitern. Die Simulation, sowie die Animation soll dabei in JupyterLab ausgeführt werden. Als Einstiegspunkt ist dabei die öffentliche Dokumentation von SimPlay zu verwenden. Diese Dokumentation ist unter https://simplay.readthedocs.io/en/latest/ zu finden. Dies ist die einzige Hilfestellung, welche der Benutzer:in zur Verfügung steht.

13.6.2.3 Durchführung

Der User-Test soll in folgenden Schritten durchgeführt werden:

1. Die Benutzer:in soll die notwendigen Schritte durchführen, die neuste SimPlay Version zu installieren.

2. Die Benutzer:in soll eine SimPy Simulation auswählen, welche um eine Animation erweitert werden soll, und diese Simulation um eine Animation erweitern.
3. Die Benutzer:in soll die Simulation mit der Animation ausführen und die Animation betrachten.

13.6.2.4 Bewertungskriterien

Die Benutzer:in soll die folgenden Fragen beantworten:

- Wie lange hat die Benutzer:in für die Installation von SimPlay benötigt? 1
- Hat die Restrukturierung der Dokumentation die Installation von SimPlay erleichtert?
- Hat die Restrukturierung der API von simplay die Verwendung von SimPlay erleichtert?
- Auf einer Skala von 1 (sehr schlecht) bis 5 (sehr gut), wie gut funktioniert die Animation?
- Auf einer Skala von 1 (sehr schlecht) bis 5 (sehr gut), wie wird die gesamte Benutzererfahrung von SimPlay bewertet?
- Gibt es weitere Anmerkungen und Verbesserungsvorschläge?

13.7 User-Test Feedback

13.7.1 User-Test 1 Feedback

Wie lange hat die Benutzer:in für die Installation von SimPlay benötigt?

- ca. 15 Minuten; mit Verbesserungsvorschlägen (A), (B) und (C) <1 Minute

Auf einer Skala von 1-5, wie einfach war es für die Benutzer:in, SimPlay in JupyterLab zu installieren?

- *Skala: 1 (sehr schwierig) bis 5 (sehr einfach)*
- 5 (falls C)

Auf einer Skala von 1-5, wie verständlich und hilfreich war die Dokumentation von SimPlay?

- *Skala 1 (sehr unverständlich und nicht hilfreich) bis 5 (sehr verständlich und hilfreicht)*
- 5
- Ich habe das aber (so wie wahrscheinlich alles) mehr quer gelesen und selektiv die benötigten Punkte durchgelesen.

Wie lange hat die Benutzer:in für die Erweiterung der Simulation benötigt?

- Weniger als eine halbe Stunde für die essentielle Animation. Für meinen Anwendungsfall reicht mir die beinahe die automatisch angezeigte Auslastung von Ressourcen. Code-mässig ist das trivial (Positionierung, Sichtbarkeit, evt. Interaktion), da ist die Suche nach Icons schwieriger.

Auf einer Skala von 1-5, wie einfach war es für die Benutzer:in, die Simulation um eine Animation zu erweitern?

- *Skala 1 (sehr schwer) bis 5 (sehr einfach)*
- 5

Auf einer Skala von 1-5, wie gut funktioniert die Animation?

- *Skala 1 (sehr schlecht) bis 5 (sehr gut)*
- 3; falls (G) 5
- Die Anzeige in der Zelle hat nicht geklappt, über die Datei aber problemlos

Auf einer Skala von 1-5, wie wird die gesamte Benutzererfahrung von SimPlay bewertet?

- *Skalara 1 (sehr schlecht) bsi 5 (sehr gut)*
- 5
- Wichtig ist mir ja, schnell Licht ins Dunkel meiner Simulation zu bringen, was sehr einfach und schnell gelungen ist!

Gibt es Verbesserungsvorschläge für SimPlay, insbesondere für die Dokumentation?

- (A) `simplay-jupyter` installiert nicht automatisch `simplay` (was man wahrscheinlich erwartet), man läuft daher evt. gleich in einen ImportError - wie wäre es, das als Dependency zu definieren?
- (B) Die Struktur der Dokumentation ist etwas verwirrend rund um die Installation und den ersten Schritten: Installation ist unter Usage und die Animation mit Jupyter kommt erst recht spät. Vielleicht wäre das mehr im Sinne eines Quickstarts verständlicher:
  - Overview *(so lassen)*
  - ~~Usage~~
  - Getting Started
    - Installation *(von Usage)*
    - ~~with simplay~~ Adding Visual Information to your Simulation
    - ~~simplay in depth~~ (nach unten)
    - ~~with simplay-jupyter~~ Playing the Visualization
      - Juypter *(direkt das Beispiel aufgreifen und nur den Code für Jupyter hinzufügen)*
      - ~~with simplay-web~~ JavaScript *(?)*
  - Simplay in Depth
- (C) Nach der Installation muss Jupyter Lab neu gestartet werden, ein Kernel-Restart reicht nicht (also z.B. direkt in `%pip install jupyter-jupyter`). Hier wäre ein Hinweis in der Doku gut, da sonst die Animation ohne erkennbaren Grund nicht funktioniert.
- (D) Syntactic Sugar
  - `tint=0xFFFFFF` als default Parameter definieren, damit man den weglassen kann
  - `self.set_position(2, 2)` statt `BasicVisualUtil.set_position(self.icon, 2, 2)` (es wir ja sowieso bereits von simplay Klassen abgeleitet!)
  - für das interface wäre es schöner ohne Util, z.B. self.set_position
  - Nicht-simpy Komponente (`VisualComponent(env, 'audience', type=ComponentType.CUSTOM, visual='audience', tint=0xFFFFFF)`)
- (E) Dokumentation
  - Fehler: visualization_manager, nicht visual_manager
  - Grid Beispiel nicht quadratisch, sondern z.B. 4x8 (damit x und y unterscheidbar sind)
  - README auf github ergänzen, das wird eher der Einstiegspunkt als readthedocs
- (F) Animation
  - Wieviele Total Steps? z.B. als Text (oder Progress Bar), was man aber eigentlich erwarten würde, wäre ein Slider.
- (G) Animation in der Zelle gibt einen Fehler:

```
Uncaught TypeError: t.events is undefined
simulationData <folder>/376.e1286b0e5a082f19944e.js?v=e1286b0e5a082f19944e:2
fo <folder>/376.e1286b0e5a082f19944e.js?v=e1286b0e5a082f19944e:2
renderModel <folder>/896.d59a5539f68feea93d75.js?v=d59a5539f68feea93d75:1
```

13.7.2 User-Test 2 Feedback

Wie lange hat die Benutzer:in für die Installation von SimPlay benötigt?

- < 5 min

Hat die Restrukturierung der Dokumentation die Installation von SimPlay erleichtert?

- Ja

Hat die Restrukturierung der API von simplay die Verwendung von SimPlay erleichtert?

- Ja

Auf einer Skala von 1 (sehr schlecht) bis 5 (sehr gut), wie gut funktioniert die Animation?

- 5

Auf einer Skala von 1 (sehr schlecht) bis 5 (sehr gut), wie wird die gesamte Benutzererfahrung von SimPlay bewertet?

- 5

Gibt es weitere Anmerkungen und Verbesserungsvorschläge?

- In der Doku kommt noch BasicVisualUtil vor (gibt ImportError)
- Suche in der Doku funktioniert nicht, z.B. https://simplay.readthedocs.io/en/latest/search.html?q=BasicVisualUtil&check (sieht nach fehlendem jQuery aus?)

13.8 Browser Tests

13.8.1 Ziel

Das Ziel dieser Tests ist es, die Visualisierung und Benutzerinteraktion mit SimPlay in der neusten Version von Google Chrome und Mozilla Firefox zu testen. Damit wird sichergestellt, dass die visuelle Darstellung sowie die Interaktionen in beiden Browsern wie erwartet funktioniert.

13.8.2 Voraussetzung

- Zur Durchführung des Tests wird jeweils eines der bestehenden SimPlay Beispiele verwendet und in JupyterLab mit Version höher als 3.0, sowie Python 3.9 oder höher gestartet.
- In JupyterLab ist keine SimPlay Extension installiert, diese wird zu Beginn des Tests installiert.
  - Kann mit Aufruf von `pip uninstall ...` entfernt werden
  - Cache kann mit Aufruf von `pip cache remove *` gelöscht werden
- In Python ist keine SimPlay Komponente installiert, diese wird zu Beginn des Tests installiert.
- JupyterLab ist in einem Ordner gestartet, welcher nichts mit Simplay zu tun hat. Die benötigten Simulationen werden direkt aus der Dokumentation in ein neues Python 3 (ipykernel) Notebook kopiert und dort gestartet.
- Die von der Simulation benötigten Visuals sind im Ordner, in welchem JuptyerLab gestartet ist vorhanden.

13.8.3 Testfälle

13.8.3.1 [A] Play / Pause

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation, wird der Play Button geklickt. Nachdem mehr als 10 Zeiteinheiten vergangen sind, wird der Button erneut geklickt. Danach wird der Play Button ein weiters Mal geklickt.

Betrachtungsweise

- In Notebook
- In `.simplay` File

Simulation

Machine Shop

Erwartetes Verhalten

Die Simulation läuft für mindestens 10 Zeiteinheiten. Bei erneutem Klicken des Buttons wird die Visualisierung pausiert. Bei dritten Klicken des Buttons wird die Visualisierung an derselben Stelle an der sie vorhin angehalten wurde, weiter abgespielt.

13.8.3.2 [B] Reset

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation wird der Play Button geklickt. Nachdem mehr als 10 Zeiteinheiten vergangen sind, wird der Reset Button geklickt. Ein erneutes Klicken startet die Visualisierung wieder von Beginn. Nach 10 Zeiteinheiten wird erneut der Pause Button geklickt und anschliessend der Reset Button.

Betrachtungsweise

- In Notebook
- In `.simplay` File

Simulation

Bank Renege

Erwartetes Verhalten

Die Simulation läuft für mindestens 10 Zeiteinheiten. Beim Klicken des Reset Buttons werden das Grid, sowie die Buttons und der Slider zur Kontrolle der Simulation zurückgesetzt. Dasselbe Verhalten ist auch erwartet, wenn der Reset Button geklickt wird, wenn sich die Visualisierung in einem pausierten Zustand befindet.

13.8.3.3 [C] Ein Schritt vorwärts

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation wird der Play Button geklickt. Der "ein Schritt vorwärts" Button wird zweimal geklickt, einmal wenn die Visualisierung ausgeführt wird, und einmal wenn die Visualisierung im pausierten Zustand ist.

Betrachtungsweise

- In Notebook
- In `.simplay` File

Simulation

Carwash

Erwartetes Verhalten

Die Simulation läuft für mindestens 10 Zeiteinheiten. Beim Klicken des "ein Schritt vorwärts" Buttons wird die Anzeige des momentanten Zeitpunkt um einen Schritt erhöht und bei weiterem Klicken werden die jeweils nächsten Simulationsschritte ausgeführt.

13.8.3.4 [D] Beschleunigen / Verlangsamen

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation wird der Play Button geklickt. Während die Visualisierung läuft, wird der Cursor auf den Speed Button gefahren. Im anschliessend erscheinenden Tooltip wird für jede Geschwindigkeit die Visualisierung betrachtet, und auf dem Slider wird Vorwärts und Zurück gesprungen.

Betrachtungsweise

- In Notebook
- In `.simplay` File

Simulation

Gas Station

Erwartetes Verhalten

Bei Veränderung der Geschwindigkeit wird die Visualisierung entsprechend langsamer oder schneller.

13.8.3.5 [E] Zurückspulen / Vorwärtsspulen

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation wird der Play Button geklickt. Während die Visualisierung läuft, wird auf dem Slider Vorwärts und Zurück gesprungen. Zusätzlich muss mit diesem Test vom Notebook oder File weg navigiert werden und wieder zurück.

Betrachtungsweise

- In Notebook
- In `.simplay` File

Simulation

Movie Renege

Erwartetes Verhalten

Das Vorwärts und Zurückspulen funktioniert, die Visualisierung springt zum ausgewählten Zeitpunkt und ist dort im Play Zustand. Die Visualisierung verhält sich durch das weg und hin Navigieren nicht anders.

13.8.3.6 [F] Zeit anzeigen

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation wird der Play Button geklickt. Während die Visualisierung läuft, wird auf die Zeitanzeige geklickt.

Betrachtungsweise

- In Notebook
- In `.simplay` File

Simulation

Event Latency

Erwartetes Verhalten

Die Simulation läuft. Durch Klicken der Zeitanzeige wird diese verändert und im Zeitformat angezeigt, ebenfalls wird das Tooltip des Sliders im Zeitformat angezeigt. Bei erneutem Drücken wird alles wieder in Schritten angezeigt.

13.8.3.7 [G] Accessibility

Beschreibung

Die Simulation wird in einer Zelle des Notebooks gestartet. In der anschliessend erscheinenden Visualisierung der Simulation wird der Play Button geklickt. Während die Visualisierung läuft, wird mit Tab durch die Buttons zur Kontrolle der Visualisierung navigiert.

Betrachtungsweise

- In Notebook

- In `.simplay` File

Simulation

Process Communication

Erwartetes Verhalten

Es ist möglich mit Tastatur die Buttons zu bedienen.

13.8.4 Testprotokoll

13.8.4.1 06.0.1.2023

Durchführungsdatum: 06.01.2023

Verwendete Versionen:

- Google-Chrome: 108.0.5359.125
- Mozilla Firefox: 108.0.2
- simplay: 0.3.5
- simplay-web: 0.3.5
- simplay-jupyter: 0.3.5

Testfall	Resultat	Massnahmen
A	Wie erwartet	keine
B	Wie erwartet, ausser das der Slider nicht zurückgesetzt wird.	Pull-Request 59
C	Wie erwartet, ausser: Slider springt nicht zum nächsten Zeitpunkt, Momentaner Zeitpunkt wird grösser als totale Anzahl Zeitschritte	Pull-Request 59 Pull-Request 60
D	Wie erwartet	keine
E	Wie erwartet	keine
F	Wie erwaret	keine
G	Wie erwartet	keine

13.9 SimulationData Event Definition

13.9.1 `SET_VISIBLE`

Das `SET_VISIBLE` Event wird ausgelöst, wenn die Sichtbarkeit eines Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_VISIBLE",
  "args": {
    "visible": boolean
  }
}
```

`visible` beschreibt, ob das Objekt sichtbar ist.

13.9.2 `SET_POSITION`

Das `SET_POSITION` Event wird ausgelöst, wenn ein Objekt seine Position ändert.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_POSITION",
  "args": {
    "x": number,
    "y": number
  }
}
```

`x` und `y` beschreiben die neue Position des Objekts. Die Position ist in Zellen des Grids angegeben.

13.9.3 SET_INTERACTING

Das `SET_INTERACTING` Event wird ausgelöst, wenn ein Objekt mit einem anderen Objekt interagiert.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_INTERACTING",
  "args": {
    "withId": string
  }
}
```

`withId` beschreibt die ID des Objekts, mit dem das Objekt interagiert.

13.9.4 SET_NOT_INTERACTING

Das `SET_NOT_INTERACTING` Event wird ausgelöst, wenn ein Objekt mit einem anderen Objekt nicht mehr interagiert.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_NOT_INTERACTING",
  "args": {
    "withId": string
  }
}
```

`withId` beschreibt die ID des Objekts, mit dem das Objekt nicht mehr interagiert.

13.9.5 MOVE_NEAR

Das `MOVE_NEAR` Event wird ausgelöst, wenn ein Objekt sich in der Nähe eines anderen Objekts befindet.

```
{
  "forId": string,
  "timestamp": number,
  "action": "MOVE_NEAR",
  "args": {
    "targetId": string
  }
}
```

`targetId` ist die ID des Objekts, in dessen Nähe das Objekt positioniert wird.

13.9.6 MOVE_NEAR_CELL

Das MOVE_NEAR_CELL Event wird ausgelöst, wenn ein Objekt sich in der Nähe einer Zelle befindet.

```
{
  "forId": string,
  "timestamp": number,
  "action": "MOVE_NEAR_CELL",
  "args": {
    "x": number,
    "y": number
  }
}
```

x und y beschreiben die Position der Zelle, in dessen Nähe das Objekt positioniert wird. Die Position ist in Zellen des Grids angegeben.

13.9.7 SET_TINT_COLOR

Das SET_TINT_COLOR Event wird ausgelöst, wenn die Tint-Farbe eines Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_TINT_COLOR",
  "args": {
    "color": number
  }
}
```

color beschreibt die neue Tint-Farbe des Objekts.

13.9.8 SET_DECORATING_TEXT

Das SET_DECORATING_TEXT Event wird ausgelöst, wenn der Text eines Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_DECORATING_TEXT",
  "args": {
    "text": string
  }
}
```

text beschreibt den Text des Objekts.

13.9.9 SET_SPRITE_FRAME

Das SET_SPRITE_FRAME Event wird ausgelöst, wenn der Frame eines Sprite-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "SET_SPRITE_FRAME",
  "args": {
    "frame": number
```

```
  }
}
```

`frame` beschreibt den neuen Frame des Sprite-Objekts.

13.9.10 RESOURCE.SET_CAPACITY

Das `RESOURCE.SET_CAPACITY` Event wird ausgelöst, wenn die Kapazität eines Ressourcen-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "RESOURCE.SET_CAPACITY",
  "args": {
    "capacity": number
  }
}
```

`capacity` beschreibt die neue Kapazität des Ressourcen-Objekts.

13.9.11 RESOURCE.SET_UTILIZATION

Das `RESOURCE.SET_UTILIZATION` Event wird ausgelöst, wenn die Auslastung eines Ressourcen-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "RESOURCE.SET_UTILIZATION",
  "args": {
    "utilization": number
  }
}
```

`utilization` beschreibt die neue Nutzung des Ressourcen-Objekts.

13.9.12 CONTAINER.SET_CAPACITY

Das `CONTAINER.SET_CAPACITY` Event wird ausgelöst, wenn die Kapazität eines Container-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "CONTAINER.SET_CAPACITY",
  "args": {
    "capacity": number
  }
}
```

`capacity` beschreibt die neue Kapazität des Container-Objekts.

13.9.13 CONTAINER.SET_LEVEL

Das `CONTAINER.SET_LEVEL` Event wird ausgelöst, wenn der Füllstand eines Behälter-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "CONTAINER.SET_LEVEL",
  "args": {
    "level": number
  }
}
```

`level` beschreibt das neue Level des Container-Objekts.

13.9.14 STORE.SET_CAPACITY

Das `STORE.SET_CAPACITY` Event wird ausgelöst, wenn die Kapazität eines Lager-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "STORE.SET_CAPACITY",
  "args": {
    "capacity": number
  }
}
```

`capacity` beschreibt die neue Kapazität des Store-Objekts.

13.9.15 STORE.SET_CONTENT

Das `STORE.SET_CONTENT` Event wird ausgelöst, wenn der Inhalt eines Lager-Objekts geändert wird.

```
{
  "forId": string,
  "timestamp": number,
  "action": "STORE.SET_CONTENT",
  "args": {
    "content": string
  }
}
```

`content` beschreibt den neuen Inhalt des Store-Objekts. Der Aufbau des Inhalts ist dabei nicht vorgegeben und wird als String dargestellt. Benutzende sind selbst dafür verantwortlich, dass der Inhalt korrekt serialisiert werden kann.

13.10 simplay-web: Events

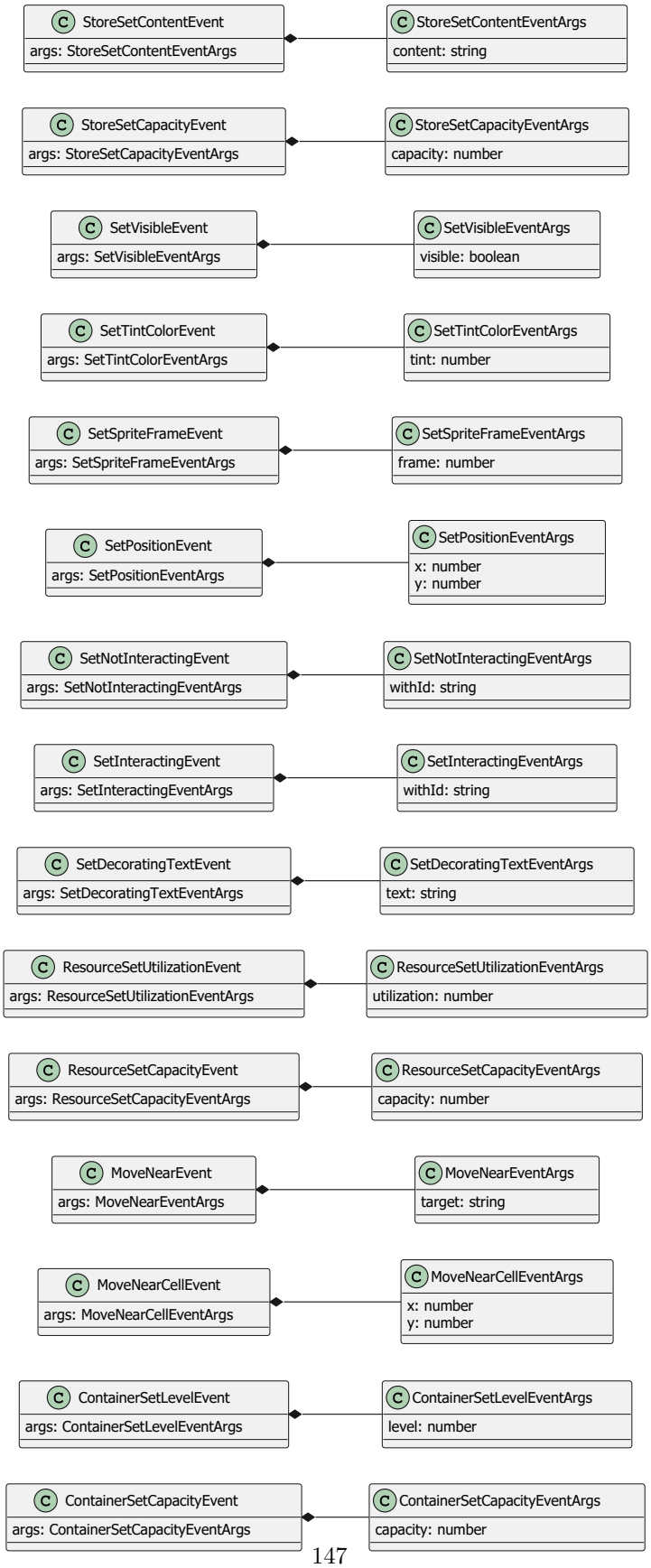


Abbildung 13.23: simplay-web: Events

13.11 Messungen zur Laufzeitveränderung

13.11.1 Machine Shop

SIMTIME	Machine Shop mit simplay	Machine Shop ohne simplay	Machine Shop ohne simplay, und ohne `print`
1	0.000501	0.000200	0.000119
10	0.000594	0.000245	0.000172
100	0.001199	0.000649	0.000547
1000	0.007221	0.003726	0.003607
10'000	0.066416	0.035249	0.034306

(Resultate in Sekunden)

13.11.2 Gas Station

SIMTIME	Gas Station mit simplay	Gas Station ohne simplay	Gas Station ohne simplay, und ohne `print`
1	0.000860	0.000084	0.000026
10	0.000856	0.000084	0.000025
100	0.000949	0.000179	0.000060
1000	0.002287	0.001535	0.000361
10'000	0.014543	0.008529	0.002570

(Resultate in Sekunden)

13.11.3 Event Latency

SIMTIME	Event Latency mit simplay	Event Latency ohne simplay	Event Latency ohne simplay, und ohne `print`
1	0.000429	0.000080	0.000020
10	0.000454	0.000088	0.000025
100	0.002633	0.001363	0.000296
1000	0.024961	0.018555	0.003131
10'000	0.251156	0.189240	0.032171

(Resultate in Sekunden)

13.11.4 Carwash

SIMTIME	Carwash mit simplay	Carwash ohne simplay	Carwash ohne simplay, und ohne `print`
1	0.000397	0.000441	0.000055
10	0.000490	0.000825	0.000092
100	0.001186	0.004063	0.000428
1000	0.007569	0.043669	0.003284
10'000	0.074498	0.411762	0.031244

(Resultate in Sekunden)

Literaturverzeichnis

[1] David Kühnhanss, Moritz Schiesser, "Entwicklung einer Animationsbibliothek für SimPy."

[2] "PixiJS." [Online]. Available: https://www.pixijs.com/

[3] Team SimPy, "Overview of SimPy." [Online]. Available: https://simpy.readthedocs.io/en/latest/

[4] Shu-yu Guo, Michael Ficarra, Kevin Gibbons, "ECMA-262, 13th edition, June 2022 ECMAScript® 2022 Language Specification." [Online]. Available: https://262.ecma-international.org/13.0/

[5] "TypeScript: JavaScript With Syntax For Types." [Online]. Available: https://www.typescriptlang.org/

[6] "For serializing Python objects to JSON (dicts) and back." [Online]. Available: https://github.com/ramonhagenaars/jsons

[7] "pytest." [Online]. Available: https://github.com/pytest-dev/pytest

[8] "TypeScript: Why does TypeScript exist?" [Online]. Available: https://www.typescriptlang.org/why-create-typescript

[9] "Floss." [Online]. Available: https://github.com/pixijs/floss

[10] "Building a Custom Widget - Email widget." [Online]. Available: https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Custom.html

[11] Project Jupyter, "Extension Developer Guide." [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html

[12] "JupyterLab Extensions by Examples." [Online]. Available: https://github.com/jupyterlab/extension-examples

[13] "jupyterlab/pdf-extension." [Online]. Available: https://github.com/jupyterlab/jupyterlab/tree/3.6.x/packages/pdf-extension

[14] "jupyterlab/theme-light-extension." [Online]. Available: https://github.com/jupyterlab/jupyterlab/tree/3.6.x/packages/theme-light-extension

[15] "JupyterLab mimerender-cookiecutter-ts." [Online]. Available: https://github.com/jupyterlab/mimerender-cookiecutter-ts

[16] "Jest · Delightful JavaScript Testing." [Online]. Available: https://jestjs.io/

[17] "galata." [Online]. Available: https://github.com/jupyterlab/jupyterlab/tree/master/galata

[18] "Fast and reliable end-to-end testing for modern web apps | Playwright." [Online]. Available: https://playwright.dev/

[19] Read the Docs, Inc & Mitwirkende, "Start | Read the Docs." [Online]. Available: https://readthedocs.org/

[20] "Websites for you and your projects." [Online]. Available: https://pages.github.com/

[21] "About wikis." [Online]. Available: https://docs.github.com/en/communities/documenting-your-project-with-wikis/about-wikis

[22] "Read the Docs Sphinx Theme." [Online]. Available: https://sphinx-rtd-theme.readthedocs.io/en/stable/

[23] "sphinx.ext.autosummary - Generate autodoc summaries." [Online]. Available: https://www.sphinx-doc.org/en/master/usage/extensions/autosummary.html#module-sphinx.ext.autosummary

[24] "PEP 257 - Docstring Conventions." [Online]. Available: https://peps.python.org/pep-0257/

[25] "Gas Station Refueling." [Online]. Available: https://simpy.readthedocs.io/en/latest/examples/gas_station_refuel.html

[26] "Gas tank icon by Freepik." [Online]. Available: https://www.flaticon.com/free-icons/gas-tank_4230926

[27] "Car icon by Freepik." [Online]. Available: https://www.flaticon.com/de/kostenloses-icon/limousine-automodell_55283

[28] "Pump icon by Good Ware." [Online]. Available: https://www.flaticon.com/de/kostenloses-icon/zapfsaule_784884

[29] "Truck icon by Icon Packs." [Online]. Available: https://www.iconpacks.net/free-icon/truck-1058.html

[30] "Machine Shop." [Online]. Available: https://simpy.readthedocs.io/en/latest/examples/machine_shop.html

[31] "Person Icon by Pixel perfect." [Online]. Available: https://www.flaticon.com/de/kostenloses-icon/einzel_5509366

[32] "Machine icon by Freepik." [Online]. Available: https://www.flaticon.com/de/kostenloses-icon/fabrik-maschine_1670381

[33] Josef Müller-Brockmann, *Rastersystem für die visuelle Gestaltung.*

[34] "PIXI.Application." [Online]. Available: https://pixijs.download/dev/docs/PIXI.Application.html

[35] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.*

[36] "Interface IExtension." [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/api/interfaces/rendermime_interfaces.irendermime.iextension.html

[37] "Interface IRendererFactory." [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/api/interfaces/rendermime_interfaces.irendermime.irendererfactory.html

[38] "Interface IRenderer." [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/api/interfaces/rendermime_interfaces.irendermime.irenderer.html

[39] "YouTube." [Online]. Available: https://www.youtube.com

[40] DataReportal, Hootsuite, We Are Social, "Most Popular Websites Worldwide as of November 2021, by Total Visits (in Billions)." [Online]. Available: https://www.statista.com/statistics/1201880/most-visited-websites-worldwide/

[41] "Icons." [Online]. Available: https://m3.material.io/styles/icons/overview

[42] "HTMLElement.style." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style

[43] "simplay - A library to generate an EventQueue to visualize with simplay-web." [Online]. Available: https://pypi.org/project/simplay/

[44] "simplay - overview." [Online]. Available: https://simplay.readthedocs.io/

[45] "SimPlay." [Online]. Available: https://github.com/team-simplay/SimPlay

[46] "simplay-web." [Online]. Available: https://www.npmjs.com/package/simplay-web

[47] "simplay-web." [Online]. Available: https://github.com/team-simplay/SimPlay/blob/main/src/simplay-web/README.md

[48] "simplay-jupyter - A JupyterLab extension to animate SimPy simulations enhanced with simplay." [Online]. Available: https://pypi.org/project/simplay-jupyter/

[49] "simplay_jupyter." [Online]. Available: https://www.npmjs.com/package/simplay_jupyter

[50] "Person Icons by DinosoftLabs." [Online]. Available: https://www.flaticon.com/free-icon/man_3439380?term=person&page=1&position=56&origin=tag&related_id=3439380

[51] Jupyter Accesibility Team, "Jupyter Accessibility Statement." [Online]. Available: https://jupyter-accessibility.readthedocs.io/en/latest/resources/JupyterLab-a11y-statement.html

[52] "Counter Icon by Freepik." [Online]. Available: https://www.flaticon.com/free-icon/counter_3410112

[53] "Carwas Icon by smashingstocks." [Online]. Available: https://www.flaticon.com/free-icon/carwash_4640519?term=carwash&page=1&position=6&origin=tag&related_id=4640519

[54] "Person Icon by ikubators." [Online]. Available: https://www.flaticon.com/free-icon/profile_3106921

[55] "Generator Icon by Freepik." [Online]. Available: https://www.flaticon.com/free-icon/generator_4815450?term=generator&page=1&position=27&origin=tag&related_id=4815450

[56] "Mail Icon by BizzBox." [Online]. Available: https://www.flaticon.com/free-icon/mail_8474573?term=mail+server&page=1&position=5&origin=search&related_id=8474573

[57] "Email Icon by IconMark." [Online]. Available: https://www.flaticon.com/free-icon/email_2989993?term=email&page=1&position=29&origin=search&related_id=2989993

[58] "Cable Icon by Freepik." [Online]. Available: https://www.flaticon.com/free-icon/audio-jack_2585005?term=cable&page=1&position=7&origin=tag&related_id=2585005

[59] "Inbox Icon by Pixel perfect." [Online]. Available: https://www.flaticon.com/free-icon/inbox_624955

[60] "Finde, installiere und veröffentliche Python-Pakete mit dem Python Package Index." [Online]. Available: https://pypi.org/

[61] "npm." [Online]. Available: https://www.npmjs.com/

[62] PyPA, "Packaging Python Projects." [Online]. Available: https://packaging.python.org/en/latest/tutorials/packaging-projects/

[63] "GitHub Actions - Automate your workflow from idea to production." [Online]. Available: https://github.com/features/actions

[64] Paul M. Duvall, with Steve Matyas and Andrew Glover, *Continuous integration : improving software quality and reducing risk*.

[65] Tom Preston-Werner, "Semantic Versioning 2.0.0 | Semantic Versioning." [Online]. Available: https://semver.org/

[66] "sloc." [Online]. Available: https://github.com/flosse/sloc

[67] "SimPlay merged pull requests." [Online]. Available: https://github.com/team-simplay/SimPlay/pulls?q=is%3Apr+is%3Amerged

[68] "Issues · team-simplay/SimPlay." [Online]. Available: https://github.com/team-simplay/SimPlay/issues

[69] "simplay / simplay-web smooth movement of entities." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/75

[70] "PIXI.AnimatedSprite." [Online]. Available: https://pixijs.download/dev/docs/PIXI.AnimatedSprite.html

[71] "simplay / simplay-web UI component for print(...) calls." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/76

[72] "simplay / simplay-web Speechbubbles for entities." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/77

[73] "simplay provide predefined icons and shapes." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/72

[74] "simplay / simplay-web / simplay-jupyter automatic size of grid." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/78

[75] "simplay / simplay-web define entity size." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/79

[76] "simplay-web organize event list to timestamp dictionary." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/80

[77] "simplay-web keyframes within the animation." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/81

[78] "simplay-web output for errors." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/82

[79] "Console overview." [Online]. Available: https://developer.chrome.com/docs/devtools/console/

[80] "simplay link levels / capacity / utilization to frames." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/83

[81] "simplay Export." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/74

[82] "documentation use Binder." [Online]. Available: https://github.com/team-simplay/SimPlay/issues/73

[83] "binder." [Online]. Available: https://mybinder.org/

[84] "simplay - Examples." [Online]. Available: https://simplay.readthedocs.io/en/latest/examples.html