# Spinal Tap Concert Simulation

**Lodinu Kalugalage**
21442515@student.curtin.edu.au
Perth, Australia

# Contents

# 1 Overview

*Spinal Tap Sim* is is a program created to visualise and simulate the concert of the Mythical Band, *Spinal Tap*. This program is written in python3 (3.10.0) and uses a few dependencies such as matplotlib to simulate and visualise the band's concert based on a `.json` scene description.

This program has a few features, based off the task specification:

- Overhead view of the stage
- Audience view of the stage
- Rendering of the band's props
- Lights with adjustable properties
- Lightgroups for brush-like control over lights
- Gradient lights
- Smoke machines with a few properties
- Eulerian smoke simulation (through phiflow, which does use Moore neighbourhoods within the library)
- Props loaded from arbitrary `.png` files
- Props with scale and position
- Backdrop selection between a file or solid colour
- Choreography file specified in human readable `.json`
    - Adjustable stage size
    - All properties of objects can be adjusted from this file
- Animation system with time intervals which repeats

# 2 User Guide

## 2.1 Running the program

To first run the program, run

```
python3 src/spinal-tap.py
```

in a bash compatible terminal. This will open the window up with the simulation and a progress bar in the terminal. The progress bar will show the progress of the simulation and is used because the simulation takes a while to run due to the smoke.

By default, the program will use the `assets/choreo/one.json` file as the scene description. This can be changed by passing the path to the scene file as an argument to the program. For example, to run the program with the scene `test.json` (test.json would be located at the root of the project structure as a sibling of README.md), run

```
python3 src/spinal-tap.py test.json
```

Additionally, to specify the number of iterations the program simulates other than the default 100, pass the number of iterations as the second argument. For example, to run the program with the scene `test.json` for 100 iterations, run

```
python3 src/spinal-tap.py test.json 100
```

But, if you wanted to use the default scene file, you can replace the scene parameter with `_`, like this.

```
python3 src/spinal-tap.py _ 100
```

# 3 Traceability Matrix

| Feature | Code Reference | Test Reference | Completion |
|---------|----------------|----------------|------------|
| Light Objects | src/light.py | Tested with various different combinations of lights and their positions. | 13/05/2023 |
| Light Groups | src/light.py | Tested by checking if lights changed properly when placed in a light group. | 13/05/2023 |
| Gradient & Solid Colour Lights | src/light.py | Tested by using a gradient light as well as solid light in scene descriptor. | 13/05/2023 |
| Light angling and transparency | src/light.py | Tested roughly by using it in the program, and used when a user changes the scene descriptor. | 14/05/2023 |
| Smoke simulation with moore neighbourhoods | src/smoke.py, line 44 assets/choreo/one.json | Just by changing the positions of the smoke machine(s) in the scene descriptor, the smoke simulation changes. | 20/05/2023 |
| Buoyant smoke | src/smoke.py | Smoke generally up every frame of the simulation, like it would in real life. | 20/05/2023 |
| Prop system | src/prop.py | Tested with usage of the program, as it can load arbitrary .png files, scale them and move them around. | 20/05/2023 |
| Multiple backdrops | src/stage.py, line 32 | Backdrop can be expressed as a solid (matplotlib) colour, a .png file, or `None` for no backdrop to be | 20/05/2023 |

| | | drawn. The backdrop is guaranteed to be drawn at the back. | |
|---|---|---|---|
| Choreography system | src/director.py | Tested with variations to a json file, loading different props and objects in. The system supports the animation of props in the scene as well as time controls. | 20/05/2023 |
| Arbitrary loading of choreography files | src/spinal-tap.py, line 14 | Tested in the command line, with no arguments, _ argument, and an underscore argument with a number. | 20/05/2023 |

# 4 Discussion

I will discuss the features of the program in the order specified in the traceability matrix.

## 4.1 Light Objects

The `Light` class was created with the specification of the task sheet. I started off with creating a class with the properties, position, colour, direction and intensity. Position, colour and intensity were relatively easy to implement. Position is a scalar as the lights only need to move across the x-axis rail. Colour and intensity are shown through the drawn polygon and circle's properties (alpha and colour, or in the case of a gradient, clipping). Another class was created to express gradient colours, called `Colour`.

## 4.2 Light Groups

The `LightGroup` was also relatively simple. It is an aggregation of one or many `Light`s or `LightGroup`s and can manage them; drawing them or being able to be called to set the properties of all the lights in the group. This was used to create the brush-like control over the lights.
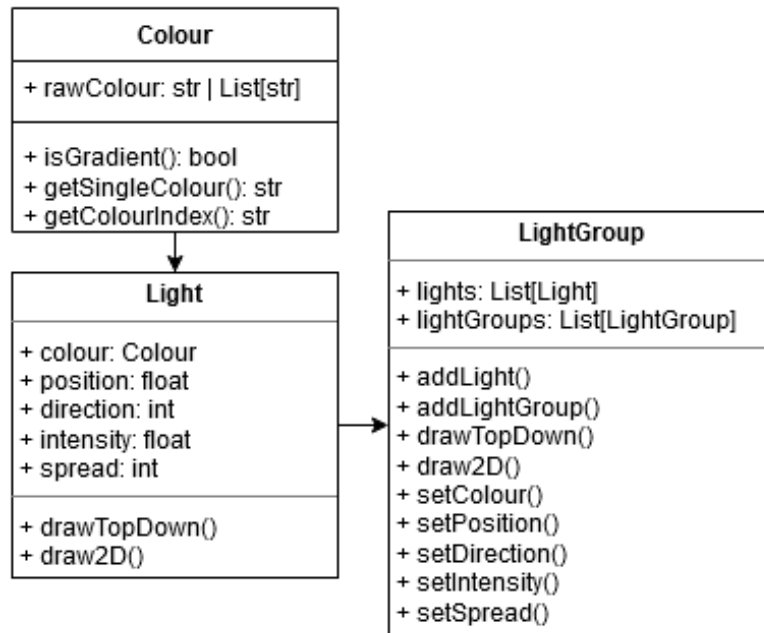


Figure 1: Colour, Light, LightGroup, UML diagrams

## 4.3 Gradient and Solid Colour Lights

This feature was implemented after a lot of bug-fixing, as the z-order (the order in which things are drawn) was off. I used matplotlib's pyplot to create a mask of the light polygons, and then put a gradient which filled the colour in. This was done by using the `alpha` property of the polygon to clip the gradient. This was also used to create the solid colour lights, by clipping the gradient with a solid colour.

## 4.4 Light Angling and Transparency

This feature was also implmented after the lights system was started. I drew out how it would look on a piece of paper and used trigonometry to get the distances and angles of the lights. I then used the `alpha` property of the polygon to clip the gradient, and used the `direction` property of the light to rotate the gradient. The transparency was added when I added a helper function to create and retrieve gradients in `colour.py`.

## 4.5 Smoke Simulation with Moore Neighbourhoods

Smoke simulation was probably the hardest part of the assignment. I read an article (Stam 2003) for the algorithms for advection, diffusion, and projection. But before I started implmenting it, I created stage.py which houses the definitions for the `StageDescriptor` (a class which stores properties about the stage) and `StageDraw` (a class which manages the matplotlib axes for the stage and audience views). Then, I tried to implement it in python, but my naive implmentation based off the paper was too slow, so I looked at using the library `phiflow`. It handles the mathematics and makes the code more concise.
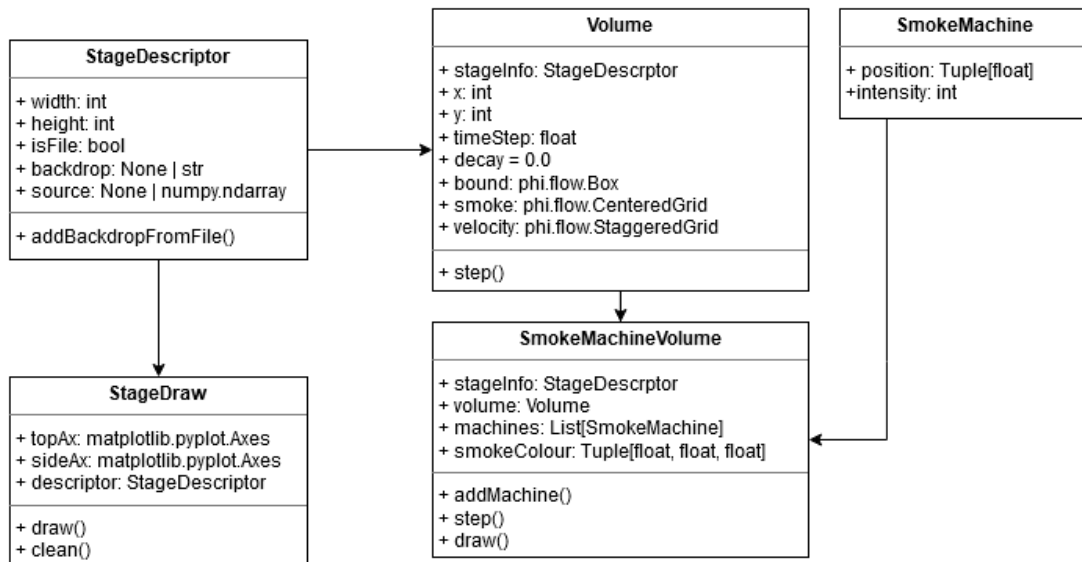


Figure 2: StageDescriptor, StageDraw, Volume, SmokeMachine, SmokeMachineVolume, UML diagrams

## 4.6 Buoyant Smoke

I read through the jupyter notebook (PhiFlowAuthors 2022) describing how to implement buoyancy in smoke, and then implemented it in the code. I used the `density` and `velocity` fields to calculate the buoyancy force, and then added it to the velocity field.

## 4.7 Prop System

The prop system was also simple to create. Using scikit-image and matplotlib, I created a class which can load a .png file, scale it, and move it around. I also added a `draw` method to the class, which draws the prop on the stage.
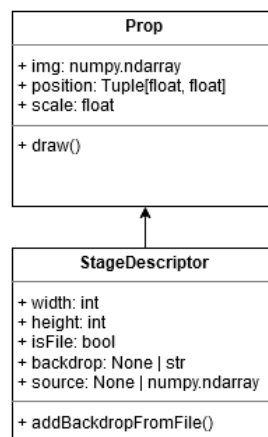


Figure 3: StageDescriptor, Prop, UML diagrams

## 4.8 Multiple backdrops

The first step in creating backdrops was to allow the `StageDraw` class to load a backdrop. I added a `backdrop` property to the class, which can be a `Colour`, `None`, or a `str` (which is the path to a .png file). Then in the draw method of the class, I draw the backdrop based on if it has loaded an image or not.

## 4.9 Choreography System

With my experience of animation, I decided to create a system with allows the user to edit how the simulation would run. I was going to create a markup language in the vein of yaml but more programatic, but realised what I wanted could be done in json relatively easily using the inbuilt python `json` module. I created a class called `Choreography` in src/director.py, which aggregates the other classes created before. It can load a json file, and then run the simulation based on the properties in the file.

## 4.10 Arbitrary loading of choreography files

This code was also relatively easy to implement, as it is just a ternary checking the length of the command line arguments and choosing between a default file, a file specified by the user, or a file specified by the user with a number of simulation cycles appended to it.
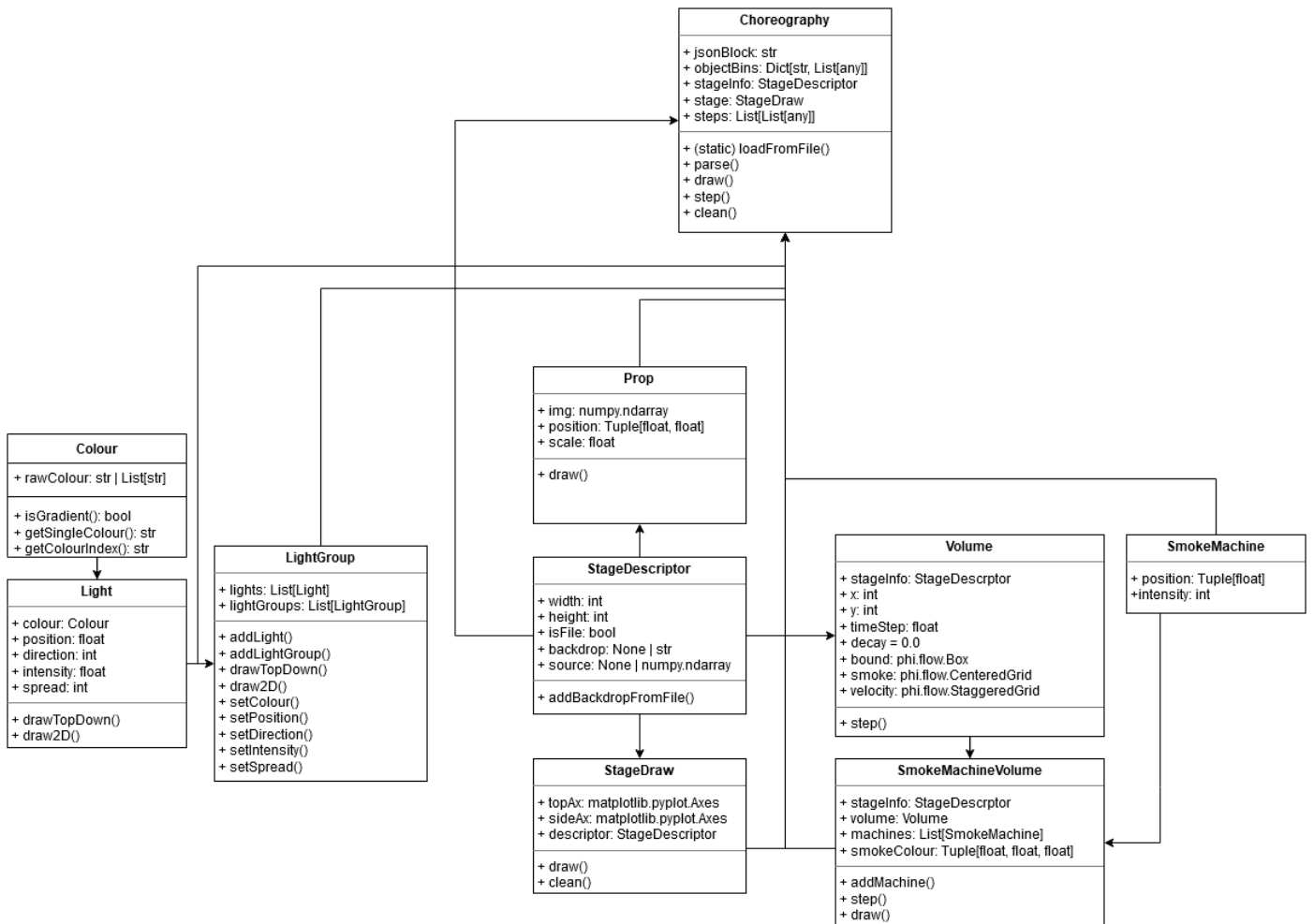
## 4.11 Full UML Diagram



Figure 4: Entire UML Diagram

# 5 Showcase

## 5.1 One.json

This is the first default simulation used by the program. To run it, run the following in the root directory of the project:

```
python3 src/spinal-tap.py
```

or alternatively and more verbosely:

```
python3 src/spinal-tap.py ../assets/choreo/one.json
```

The code should output something similar to the following:



Figure 5: TQDM progress bar

This progress bar is created in `spinal-tap.py` to track the progress of the frames, and is created using the python `tqdm` library. After the progress bar is finished, a folder called __simcache will be filled with the frames of the simulation. The following is a sample of that:
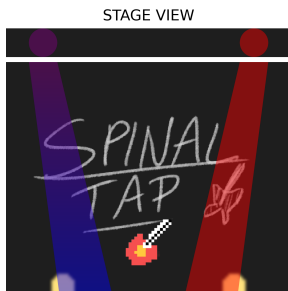


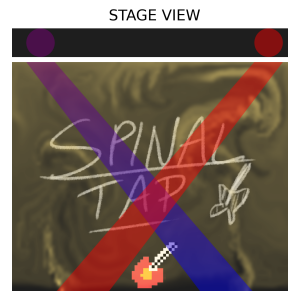Figure 6: First frame of the one.json choreography



Figure 7: Last frame of the one.json choreography

As is visible, there is smoke covering the stage and the lights are moving around. The lights are moving back and forth as specified in the json file, and the smoke is moving upwards with buoyancy. The guitar prop is also moving around, as the position is displaced from the first frame to the last frame.

## 5.2 Two.json

Similarly to the one.json choreography, the two.json choreography can be run with the following command:

```
python3 src/spinal-tap.py ../assets/choreo/two.json
```
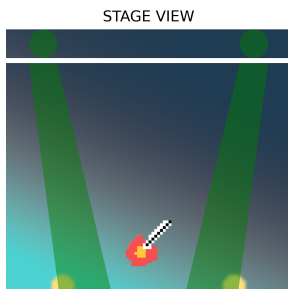


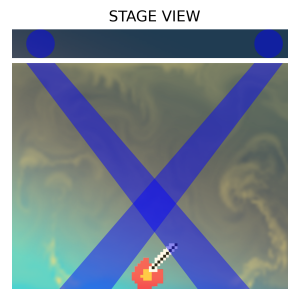Figure 8: First frame of the two.json choreography



Figure 9: Last frame of the two.json choreography

Two.json is a modified version of one.json; It uses a different background which is visible in the stage view. It also has a different light pattern, using a different command to change the colours. The colour changing here demonstrates the `LightGroup`'s ability to manage multiple lights at once. The timing for all of objects is also different; there is a 3 frame gap inserted at the end instead of 1 frame.

### 5.3 Three.json

Similar situation with three.json, run with the following command:

```
python3 src/spinal-tap.py ../assets/choreo/three.json
```
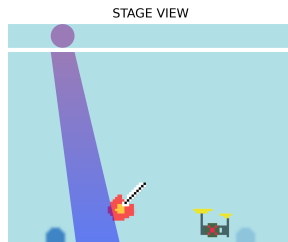


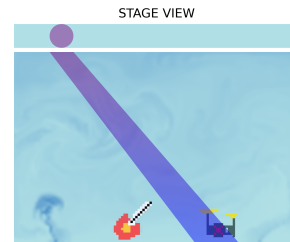Figure 10: First frame of the three.json choreography



Figure 11: Last frame of the three.json choreography

Three.json specifies a different sized stage, and a solid background colour. It also incorporates another prop, the drumkit. The drumkit is moving around the stage, as well as the guitar. The smoke is also put with different intensities (as you can see in Figure 10), so it swirls around differently than if it were all the same intensity.

## 6 Conclusion

In regards to the task specification, I have implemented basically all of the features specified. However, I could not implement direction for smoke machines as the paper I used as a reference did not have an easy way to add a positional velocity as the simulation is with a grid and not particles. But other than that, overall my work is of a high quality (reasonably and not excessively documented, code separation in files, etc.) and I am happy with the result.

## 7 Future Work

Some future work that could take place might be

- Adding a GUI (Graphical User Interface) to the program
- Adding a way to specify the direction of the smoke machines
- Using a faster simulation method or alternative simulation methods for smoke
- Adding a way to specify the colour of the smoke per smoke machine
- Using a faster framework than matplotlib for visualising the simulation (maybe pygame or raw OpenGL, Vulkan, etc.)
- Using a faster language (C/C++, Rust, Zig, etc.) to run the simulation

## Bibliography

PhiFlowAuthors. (2022). *Differentiable fluid simulations with φflow.* (https://github.com/tum-pbs/PhiFlow/blob/master/docs/Fluids_Tutorial.ipynb)

Stam, J. (2003). Real-time fluid dynamics for games. *Gdc*, 1, 2–13.