HACK2HIRE CHALLENGE: ATTRACTOR

Antony Smith Loose

July 23, 2025

The code is hosted on GitHub. I have also deployed a live demo.

1 Methodology

I used the following methodology to complete this task:

1. Understand the question - research topics, think about what is being asked.
2. Plan the user experience - what is our application trying to be, what is its purpose?
3. Plan the code - how do we achieve the planned UX?
4. Test - is it working as expected, are edge cases covered?
5. Deploy - deploy a version on GitHub pages, and package a compressed folder for submission.

2 Understanding the Question

Prior to this project, I had never worked with dynamical systems or attractors; therefore, my first read through of the question provided little insight into the task - I required more context. Identifying the gaps in my knowledge, I decided to spend a day researching dynamics. The majority of that time was spent understanding phase space, technical terms for attractors, and the different types of attractors.

After my research, I was confident I understood the problem space enough to begin answering the question. However, when I re-read the question, I was left slightly confused - what does ”generating” an attractor mean? From my understanding, attractors are a consequence of a dynamical system, so to generate an attractor, you would first need to generate a dynamical system, this understanding leads to two interpretations of the question:

1. Generating an attractor means simulating a provided dynamical system, and visualising the states.
2. Generating an attractor means creating random dynamical systems, then simulating them to find if any attractors arise.

Given the context of the task, the category being graphics and UI, I assumed the first interpretation to be correct.

3 User Experience

My original plan for the user experience is as follows:

1. Input equations for a dynamical system or choose from a predefined set.
2. Set a duration for the trail generated as the system evolves.
3. Play/pause the simulation.
4. Record some portion of the simulation.
5. Scrub the recorded portion of the simulation.

Due to time limitations, I opted to simplify the flow, only allowing users to select from predefined dynamical systems, and using a constant trail duration. However, I developed the code to allow for these features to be added easily:

```
function update(delta, state) {
    /*
    equation can be changed, allowing for new equations to be added dynamically.
    max_age_millis can be changed, allowing for the duration of the trail to
    be changed dynamically.
    */
    const { point, trail, equation, max_age_millis } = state;
    const { x, y, z } = equation(point.x, point.y, point.z, delta / 1000);
    ...
}
```

4 Implementation

The core of this application is the `update` function, every tick it does the following:

- Update the elapsed time - recording the state at this elapsed time if `state.recording` is set to `true`.
- Calculate new x, y, and z positions using the previous state.
- Set the position of the initial point to the newly calculated coordinates.
- Add a new point to the trail.
- Remove old points from the trail.
- Render the new trail.

There are also utility functions for updating state and rendering points/lines.

A vanilla ES6 object is used to store previous states. An ES6 map can be utilised to increase performance. Using the elapsed time as a key allows `O(1)` access to the state at a given time. The limitation of this approach is that it cannot provide states for times that have not previously been calculated.

A combination of recording and dynamically calculating states would be the optimal solution. For example, if `time` does not exist in `states`, we can iterate over the keys of `states` to find the closest time that has an entry, set the current state, then call `update` until `time = elapsed`.

A headless simulation would further optimise the calculation of `states`, this simulation could generate a JSON file containing the `states` for a dynamical system given some initial conditions. Using JSON means that our headless simulation can be written in any language, I would recommend `c` for increased performance. The JSON containing `states` can be imported into our web simulation, allowing the user to instantly view the system for much larger `times`.

5 Challenges

Recording states proved to be the most challenging aspect of this question. I noticed significant latency when recording, this latency followed a specific pattern: every 0.5 seconds, the simulation would freeze briefly; the duration of the freeze increased with the duration of the simulation. It was only noticeable when the simulation had been running for more than 30 seconds. The pattern of latency allowed me to quickly deduce the poorly optimised code:

```
function update_elapsed(state) {
    const remainder = (state.elapsed) % 500;
    if (remainder === 0 && state.recording) {
        state.states[state.elapsed] = {
            point: JSON.parse(JSON.stringify(state.point)),
            trail: JSON.parse(JSON.stringify(state.trail))
        }
    }

    state.elapsed += state.t_step;
}
```

Recording state was causing a bottleneck, I had two hypotheses as to why:

1. The size of the map was consuming too much memory.
2. The keys of the map were colliding frequently, causing open addressing / map resizes.

I quickly discerned hypothesis `1` was likely not the cause - I have worked with larger objects in JavaScript and had no performance issues. To test hypothesis `2`, I refactored to use an ES6 Map, and made the keys `UUIDs` (to minimise collisions); neither resulted in major performance gains. Additionally, from my research it seemed like performance overhead from poor key selection is unlikely in JavaScript.

Confused, I returned to the code. I thought - what if the trail is not being updated correctly? If the trail grows indefinitely, the time to execute `JSON.parse(JSON.stringify(state.trail))` would indefinitely grow. I logged `trail` every tick to verify it was being updated correctly - it was not. The following line of code was broken:

```
trail = trail.filter(point => point.age < max_age_millis);
```

The updated trail was being assigned to a function parameter, meaning old points in `state.trail` were not being removed. The fix was simple:

```
state.trail = trail.filter(point => point.age < max_age_millis);
```