Cornell University

CS 3110

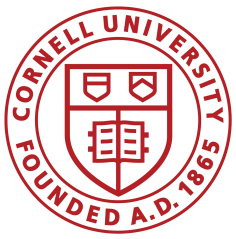
Data Structures and Functional Programming

O My Powder Shell (O My PSH)
Final Design Document

Authors:
Tennyson T Bardwell (ttb33)
Quinn Halpin (qmh4)
Sitar Harel (sh927)

Professor:
Michael Clarkson

December 4, 2016



1 System description

Summary: Implement an ASCII version of the beloved powder game that runs in the terminal. See `http://dan-ball.jp/en/javagame/dust/` for an example.

1.1 Key Features

1. A full, working, implementation of the powder game inside of a terminal.
2. Elements include: sand, water, ice, stone, glass, lava, steam, plant, fire, torch, black hole, water spout, oil, acid, bomb, and stem cell.
3. Accepts mouse interaction and keyboard shortcuts as forms of user interaction to add/remove element and select element type to add.
4. Implement a json config file to define element's properties.
5. Include a saving/loading to/from file.

1.2 Description

The powder game first appeared on dan-ball.jp at `http://dan-ball.jp/en/javagame/dust/` and has since been copied, ported, and expanded many times over. However, to this day, there exists no terminal based, ASCII-only, playable version of this beloved game. We intend to end this atrocity.

Our implementation will be quick enough to be played on low end machines, flexible enough to adapt to different screen sizes, and intuitive enough so that even a new player can play effectively.

We intend for the first implementation of the ASCII powder game to involve minimal physics, no velocity vectors. We want to save the state of the pixel location as a mutable 2D array. The engine outputs the next state of the game based off the current state and on a user's selected element and mouse clicks. All the materials have their own Json file with a list of traits. This makes adding a new element simple. For example, sand can be defined by traits such as color = yellow and movement = slightly viscous. Some elements will also be able to respond to other elements for example, ice will turn all water pixels into ice. A game can be saved and stored to a file.

2 Architecture

All modules will communicate with the main module only (with the exception of their peripherals such as Lamda Term or the file system). This allows only one module, the main module, to be solely concerned with the coordination of the roles of different modules. See figure 1 for the communication and interfaces between modules. Since our architecture makes it difficult to see the flow of logic, data, and state during runtime we have also provided figure 2 which shows the flow of information throughout our program at runtime.

Figure 1: Communications between modules

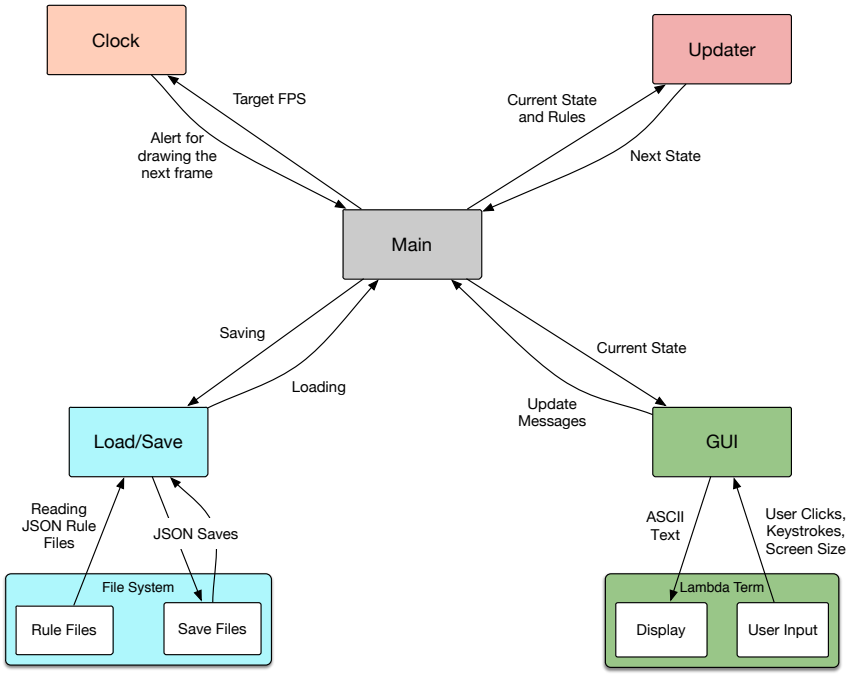
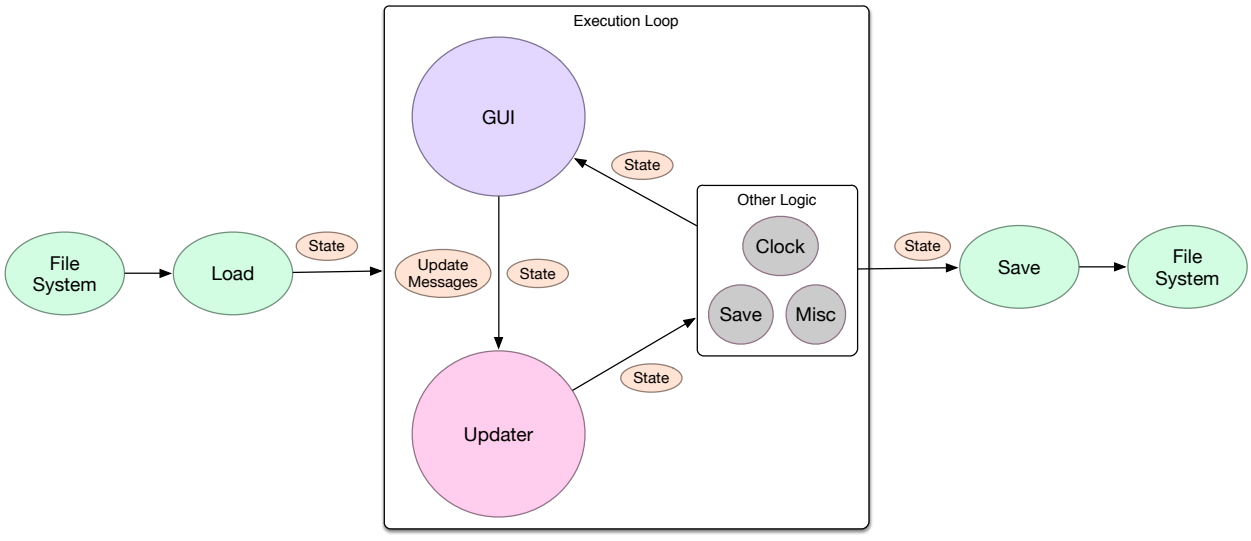


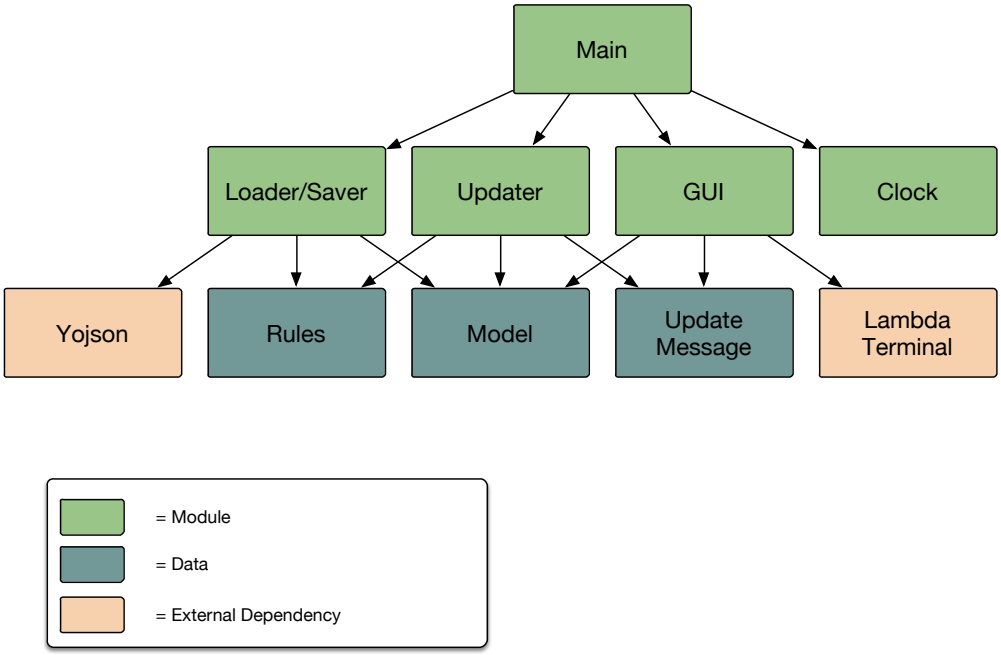
Figure 2: Runtime Flow of Information



3 System design

Each module is completely separate except for 1) main, which manages the coordination of the whole program and 2) the sharing of data objects between modules. This can be seen below in figure 3.

Figure 3: Module Dependency Diagram



3.1 Modules

main: coordinates all other modules, passing them the required state/message updates/rules for them to function and ordering the actions of other modules

load/save: loads rules and states from file to memory representations and reverses the process to save states (never saves rules)

gui: displays the current state to the user using ASCII text through the terminal, also records user interactions and packages them as model updates

updater: manages all changes to the state from updating frame to frame and by processing model updates

clock: prevents the game from running pass a set frame rate

3.2 Updater Logic

The updater can take a single, instantanous state and a set of rules and produces the state at the next time step. How this could work is demonstrated below for the elements sand and water:

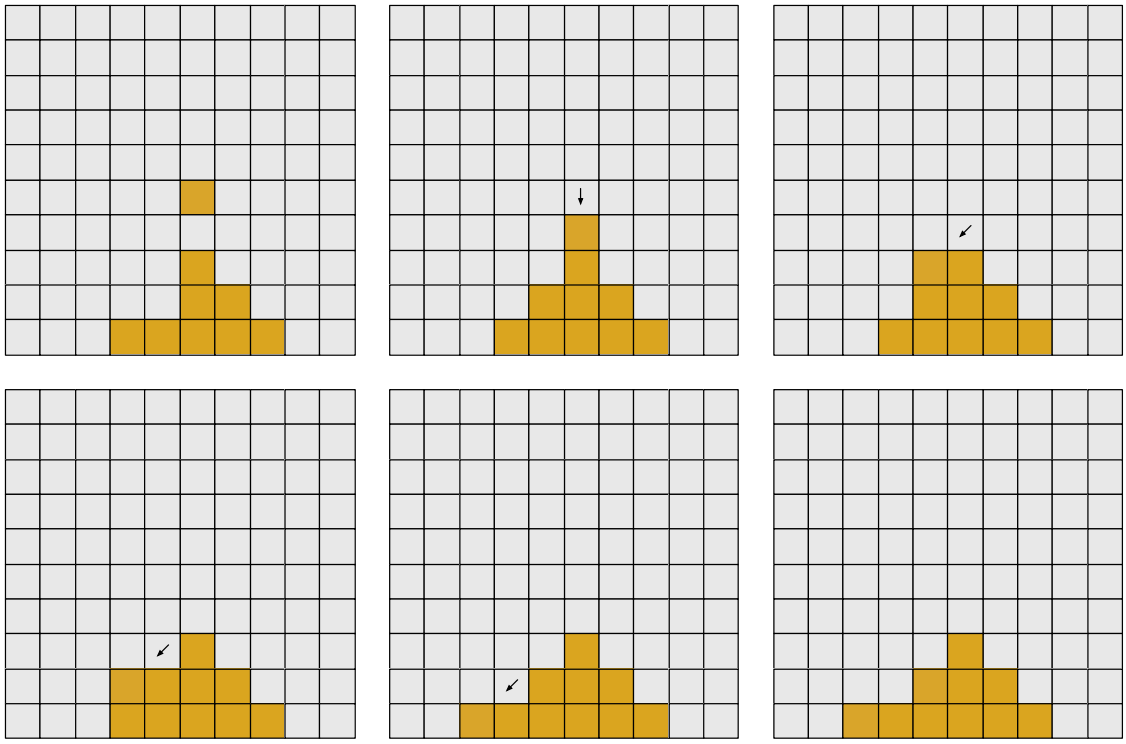
Sand falls to the ground through simple rules:

1. If the space immediately below it is empty then it will move to that space.

2. If a space below it and immediately to the right or left is empty than it moves to one of those spaces (possibly picked randomly).

See an example below in figure 4.

Figure 4: The falling of sand

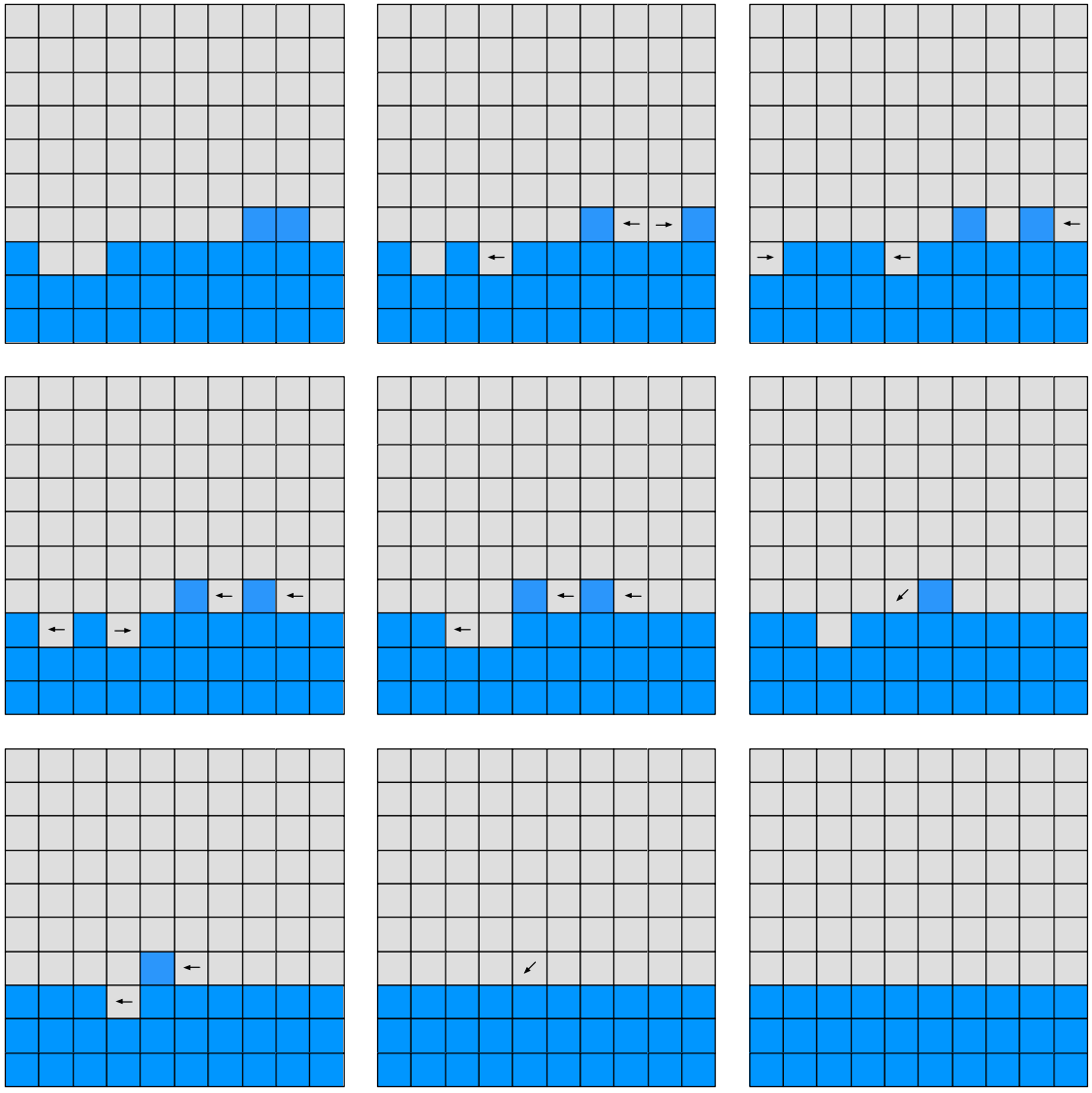


Water merely introduces another rule:

3 If the space immediately next to it on either side is empty then it has a chance of moving to that space (randomly picking if both are empty).

See an example below in figure 5.

Figure 5: The falling of sand



4 Data

The system will need to maintain what particle if any is currently residing at each location on the screen, what are the characteristics of that particle, and what are the rules that the game must follow. We will store the grid representing all the possible locations for characters on the screen in a hash table. We plan to store the type of a particle as a record.

5 External Dependencies

We will be using Lambda-Term (`https://github.com/diml/lambda-term`) and open source library that will allow us to interface with the terminal, not just through keyboard input, but also with the mouse. We will also be using Yojson for parsing JSON data from our rules file and from save files.

6 Testing Plan

We plan on testing our system in three ways. First, we shall have black box testing based on our interface files to test each function. Second, we shall have glass box testing based on implementation of each function in each module. Third, we shall test by playing the game. This method will be important for easily testing the display components.

To hold our team accountable we shall be checking everyone meets their test plan at our bi-weekly meeting and scan over the test suite. Of course, all the code and test suites for each module will be available on the GitHub.