Cornell University

CS 3110

Data Structures and Functional Programming

# Powder Shell
# Final Design Document

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

*Professor:*
Michael Clarkson

December 5, 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—until now. We have corrected this atrocity.

Our implementation is 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.

This implementation of the ASCII powder game involves minimal physics and no velocity vectors. We 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 are defined in a JSON file with a list of rules. This makes adding a new element simple. For example, sand can be defined by traits such as color = yellow. Some elements are also able to respond to other elements for example, ice turns all water pixels into ice. A game can be saved and stored to a file.

# 2 Architecture

All modules 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

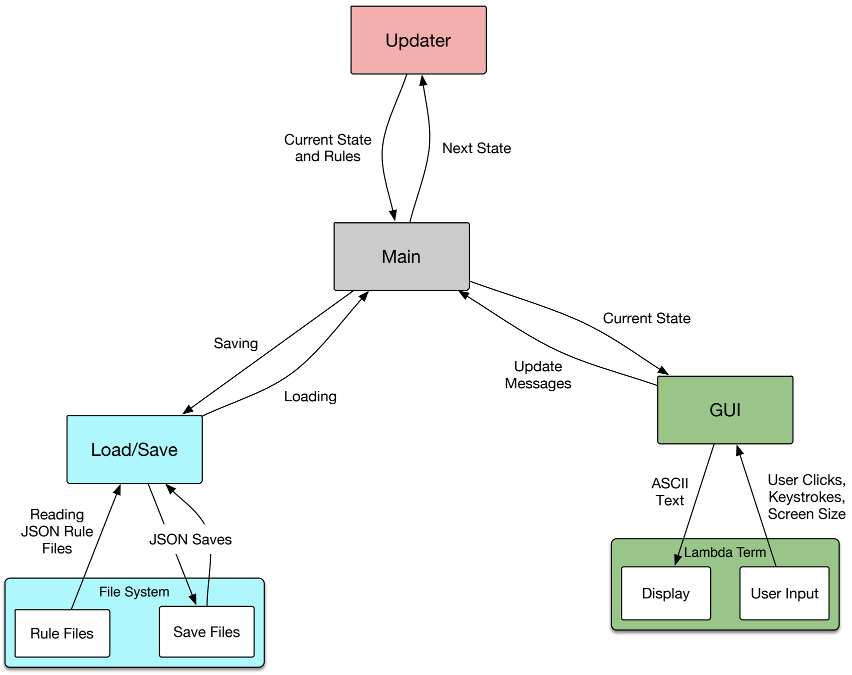
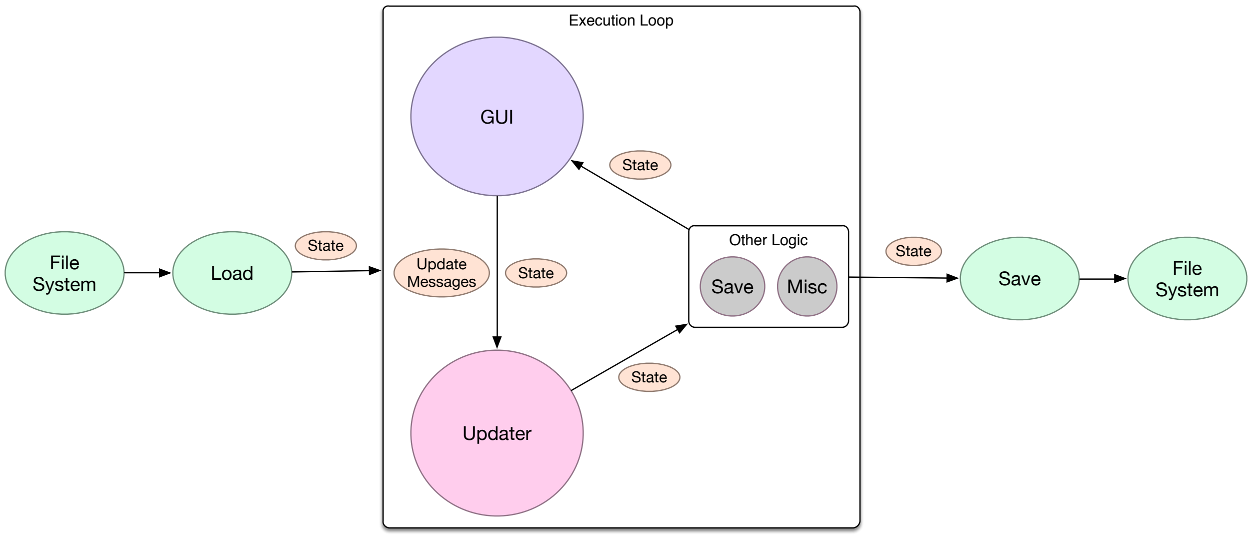


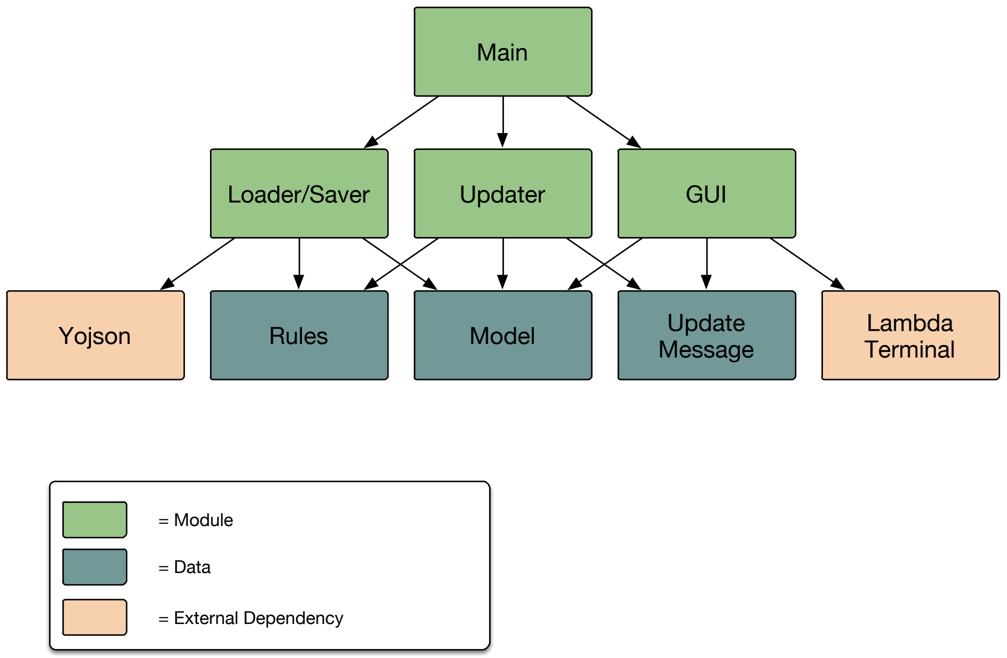
Figure 2: Runtime Flow of Information



# 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

## 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:

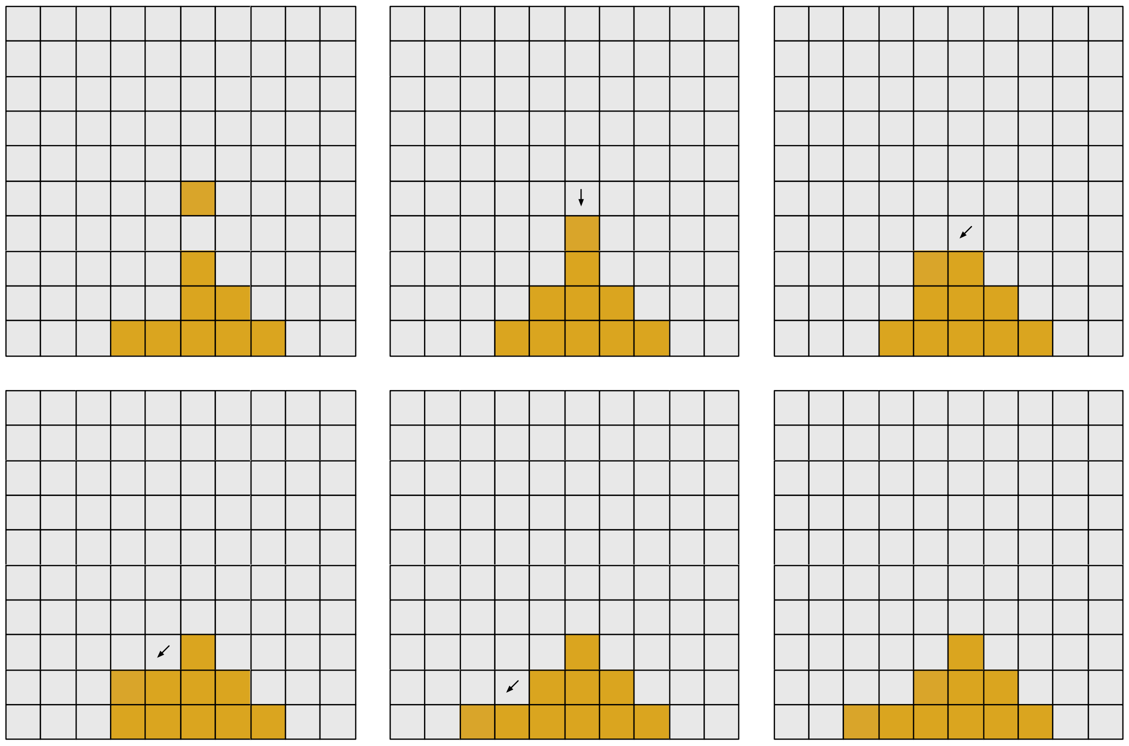
Sand falls to the ground through simple rules:

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

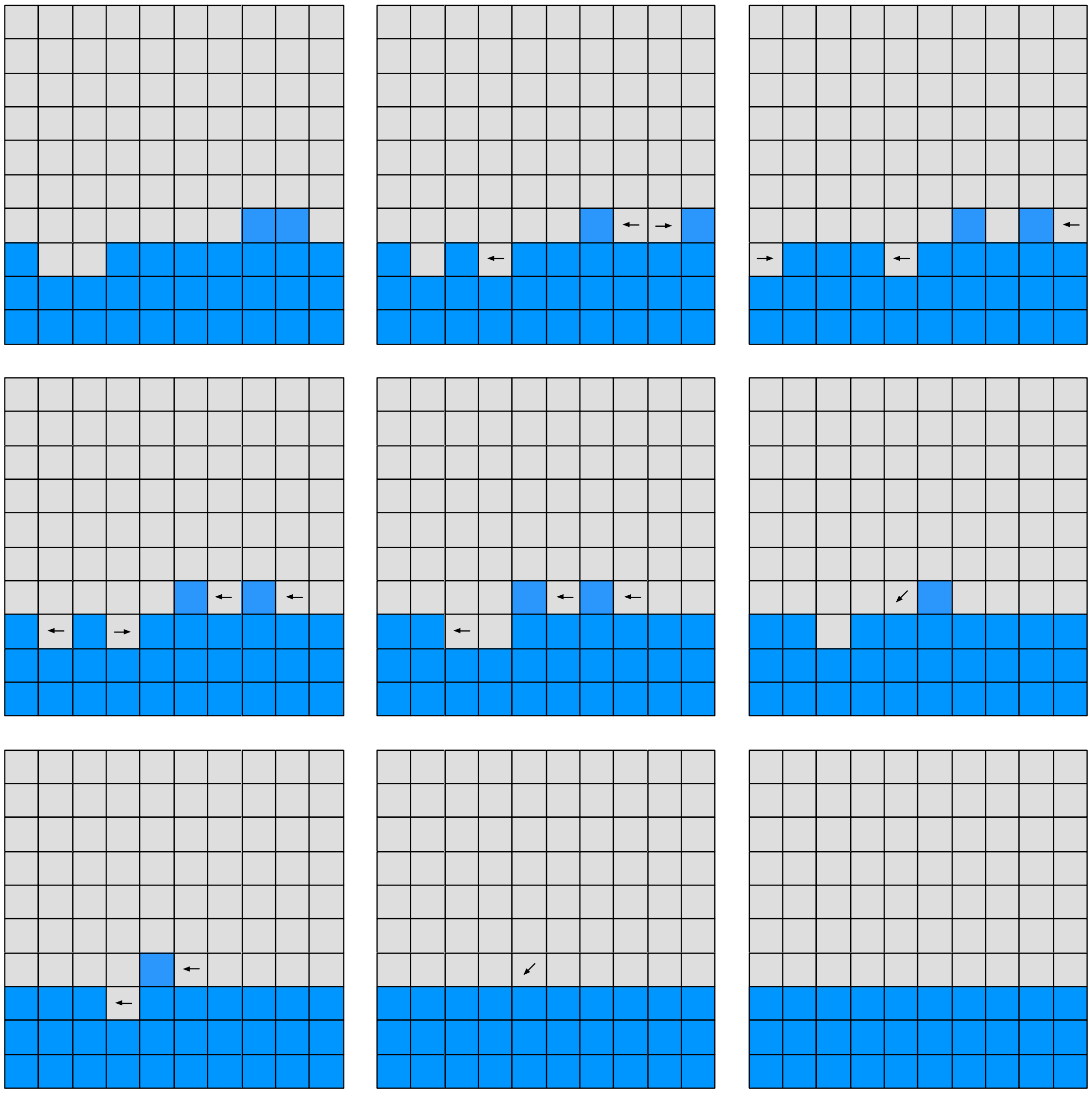


Water merely introduces another rule:

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 grid maintains the location of particles and the name of the particle at that location. The rules are held separately in its own json file and maintains characteristics such as density, interactions, color, change, destroy, and grow. The grid is represented as a (location_t * particle_t) array array, where location_t is type int*int and particle_t is of type name = string.

Rules is an association list with the name of an elem associated with an elem_rules. Elem_rules record consists of a bunch of tuples, records, and lists, that are associated with an element. Something like color of an element, density of an element, and transforming that element to a different element with a certain probability would all belong under Elem_rules.

## 5 External Dependencies

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

## 6 Testing

We have tested our system in three ways. First, we have done black box and glass box testing based on our interface files to test each function in Model interface file and File Manager interface file. Second, we have personally spent hours playing the game to test the GUI and have given the game to other people to test.

We have been holding our team accountable 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.

## 7 Division Of Labor

**Tennyson T Bardwell (ttb33):** Implemented the majority of the updater's logic which increments from one step to the next and applied user input to the grid. Also, defined the initial rules for sand and water, implementing the JSON structure for the rules of these elements, the in memory representation, and the parsing of the JSON. Wrote many updater test cases. Estimated hours of work: 40.

**Quinn Halpin (qmh4):** Quinn implemented the interface for model, writing the grid to a json file, reading a json file to a grid, and wrote test cases for model and filemanager. She spent some time making json parsing to a grid more efficient so that only the locations that are not empty show up in the json file. She did 50 hours of work.

**Sitar Harel (sh927):** Sitar implemented GUI which involved drawing particles to the terminal and managing the entire user interface. He spent over 70 hours deciphering the quirks of lambda-term and figuring out how to get the best user experience out of the terminal. Sitar also implemented the asynchronous logic of main.ml and designed many of the elements in the rules JSON.