Falling Particles

Introduction

"Falling Sand" programs model a virtual world as a grid of particles, each of which behaves according to some set of rules that depends on the material that it is made of. For instance, sand might fall down into empty spaces or water might flow into surrounding areas. These programs allow the user to paint new particles into the virtual world, and they simulate the surprisingly complex interactions that occur given your initial set of rules.

The most straightforward way to write a falling sand program would be to keep a two-dimensional array of integers representing the particles at each location, and to define the rules of the simulation as operations on those integer spaces. This works great for simple simulations, but writing a complex physics system using this technique will quickly get tedious and buggy.

Luckily, you have a powerful tool to make your program more flexible and organized: object-oriented programming! As you've learned in AP Computer Science, applying the principles of object-oriented design to a problem allows you to easily break it into small, structured, and reusable components that work together to form organized and extensible systems.

We'll be creating a Falling Sand-like program that uses object-oriented design to build new functionality at increasing levels of abstraction. Most importantly, you'll have the tools you need to introduce any new mechanics you can think of, allowing you to truly make this project your own.

Program Functionality

The final program will have the following minimum functionality:

- The canvas will be automatically generated and filled with the correct types of particles
- A variety of materials can be painted on a canvas by the user
  - Stationary particles (such as stone)
  - Gravity particles (such as sand)
  - Liquid particles (such as water)
- Materials will interact and behave as expected, and different particle types will inherit their appropriate behaviors

You'll also implement the following creative features

- A material of your choice that extends one of the existing abstract base classes
- A new abstract particle type, implemented by creating a new abstract base class and adding materials that extend it

Project Structure

First, download the starter code and take a look at the provided classes.

- **ParticleDisplay.java** — This class implements the interface and underlying renderer for your particle simulation. You won't need to edit it directly, but you will need to use some of its methods when writing your simulation. Here are the methods you might need:
  - `ParticleDisplay(String title, int height, int width, ArrayList<String> classes)` — This is the main constructor that you'll call to create a ParticleDisplay. In it, you'll provide a `title` for the window, the `height` and `width` of the simulation canvas, and a list of particle `classes` to be included in the interface. The string names you provide in the `classes` list must match the names of your actual particle classes *exactly*.

- `void setColor(int x, int y, Color color)` — Use this method to set the color of any pixel in the display by passing the `x` and `y` coordinates of the pixel. The `color` is provided as a `java.awt.Color` object.
- **Simulator.java** — This class contains the main driver code for your program. It manages all of the particles present in the current simulation and controls the flow of time. You'll be writing most of the methods here, but the `run` method is provided for you.
  - `void locationClicked()` — This method is filled in already, and is functional as is. It handles the creation of new particles based on the selected tool. If you want to add in other tool-based behavior as part of the creative portion, you can edit this method, though this isn't required.
  - `void run()` — Call this method to start the simulation. You shouldn't need to edit this directly.
- **Grid.java** — This class represents a grid of particles, and is used by the simulator. You'll need to write this class, but a skeleton is already provided.
  - Instance Variables
    - `Particle[][] world` — The internal grid of particles
    - `int width, height` — The width and height of the world
  - `Grid(int width, int height)` — The main constructor, which creates a new grid with the provided width and height. This constructor also handles terrain generation.
  - `Particle get(int x, int y)` — Returns the particle at the given coordinates
  - `Particle set(int x, int y, Particle value)` — Inserts the given particle at the given coordinates.
  - `void swap(int x1, int y1, int x2, int y2)` — A utility method that swaps the particles at (x1, y1) and (x2, y2).

Part 1: Building the Simulator

The following exercises will take place in `Simulator.java`. Method stubs are provided for all of the methods you'll need to write, and a few lines of code are already written.

- First, fill in the constructor for this class. The constructor will take in the width and height that the simulator should have, and it should set the corresponding instance variables accordingly. Next, it should identify all of the particle classes that should be added to the palette. You'll see a few of these added already, but you'll need to come back here later when you write your own particle classes. The constructor should then initialize the instance variable `display` with a new `ParticleDisplay` using the constructor described above. Finally, initialize the instance variable `grid` with a new `Grid`, again using the constructor above. We'll write the grid class later, but for now you can assume it has the methods previously described.
- Now that you have a constructor, fill in the `main` method at the top of the file. This method should create a new `Simulator` object with your chosen dimensions (100 by 100 works well), and then call the `run` method on that object.
- The next method to write is `updateDisplay`. This method should iterate through all coordinates within the simulator's width and height, and set the color at the corresponding location in `display` to the result of calling the `color` method on the particle at the coordinates. Your code inside the loop should look something like this: `display.setColor(x, y, grid.get(x,y).color());`
- The `step` method is called repeatedly based on the speed of the simulation, and each time it's called, it should ask one random particle to `step`. To do this, generate a random coordinate pair, and then ask it to step by calling `grid.get(x,y).step();`

Part 2: Representing the Particle Grid

We now have a complete simulator, but it needs to be able to represent and store the actual grid of particles in order to function. Two-dimensional arrays are the perfect tool for this task. We'll write a class `Grid` that represents the internal 2D array of particles and supports useful operations on this array.

- A skeleton for this class can be found in `Grid.java`, with instance variables and method stubs provided. The 2D array of particles is stored in the instance variable `world`, and we also keep track of the dimensions of the grid using the variables `width` and `height`.
- Our first task is to fill in the beginning of the constructor. As you'll see in the file, this constructor takes integer `width` and `height` parameters. First, you'll need to set the corresponding instance variables accordingly. Next, initialize `world` as an empty 2D array of particles with the dimensions `width` and `height`. The constructor will later be completed by adding the initial particles at every point in the grid, but we'll do this after writing a few more methods.
- Since the `world` array is a private variable, we'll need to write methods to enable outside access and mutation of the particles at specific locations. To do this, we'll write a `get` method and a `set` method, each of which will take x and y coordinates as parameters.
  - Fill in the `get` method to return the particle at the requested coordinates, and the `set` methods to replace the particle at the requested coordinates with `value`
- In our simulation, every point in the grid will be occupied by a particle at all times. To meet this requirement, two particle classes will be used at the start of the program: `Air` and `Barrier`. The outside edges of the grid will be occupied by `Barrier` particles, and all other points will be occupied by `Air`. We'll fill the grid with these particles at the end of the constructor.
  - At the end of the constructor, iterate through all of the points within `world` and use the `set` method to initialize each location with the correct particle type. If the point is on any edge of the world, insert a new `Barrier` particle by calling `new Barrier()`. Otherwise, insert a new `Air` particle by calling `new Air()`.
  - Tip: Regardless of the particle type, you'll need to initialize it with its starting coordinates. The constructor takes the surrounding grid as well as the starting coordinates as parameters. For example, `new Barrier(this, x, y)` is how you would instantiate a Barrier object.
- Oftentimes, particles will need to swap locations with other particles during our simulation. For instance, if a `Sand` particle falls onto an `Air` particle below it, the two particles should swap places to simulate the effects of gravity. Fill in the `swap` method to swap the particles at `(x1, y1)` and `(x2, y2)`. At the end of the method, prewritten code passes the new coordinates to the swapped particles—you shouldn't need to modify this part.

Part 3: Abstract Particle Classes

The next step is to write the abstract particle classes that will serve as base classes for the different materials we define.

- `Particle.java`
  - The `Particle` class will be the abstract base class for all other particles. We'll implement it to provide both the methods necessary for polymorphic particle simulation in the `Simulator` class and some utility methods that will be used by the concrete particle classes you write later on.
  - At the top of the file are the instance variable that all particles will have: the `grid` that they're a member of, and their `x` and `y` coordinates.
  - First, write two abstract methods that child classes will implement: `step` and `color`. The first of these methods, `step`, will be called whenever the

particle is asked to do something by the `Simulator`. The second method, `color`, will be called during rendering to get the color of the particle. These should both be public methods. The `step` method should have return type `void`, and the `color` method should have return type `Color`.
  - You'll also need to write two concrete methods in this class. First, fill in the `moveTo` method to update the particle's `x` and `y` instance variables based on the corresponding parameters. This method is called whenever the particle is moved to keep these values accurate. Next, fill in the `swapWith` method. This method should call `grid.swap`, using the `grid` instance variable and `x` and `y` coordinates, to swap the particle with the particle at the coordinates `(x2, y2)`.
- `FixedParticle.java`
  - The `FixedParticle` class will be the base class of particles that never need to move. By writing intermediate abstract classes like this one, we'll add one layer of abstraction between the `Particle` class and the actual classes for different particle types in order to allow for shared behavior between similar particles.
  - A `FixedParticle` is a particle that doesn't move and doesn't do anything. This makes defining their behavior easy: all this class needs to do is do nothing when asked to `step`! Write an empty `step` method that implements the abstract one in `Particle.java`.
  - We won't need to implement `color` here, since that behavior will depend on the specific particle in question.
- `GravityParticle.java`
  - Things get slightly more complicated in the `GravityParticle` class, but it follows the same general approach. A "gravity particle" is a particle that, like sand, falls down into air below it. We'll simulate this behavior by swapping with any `Air` particle below the `GravityParticle` every time `step` is called.
  - Fill in the `step` method to check if the particle at `grid.get(x, y-1)` is an instance of `Air` (a class we'll write later). If so, swap the two particles using `swapWith`. If not, do nothing.
- `LiquidParticle.java`
  - Liquid particles are similar to `GravityParticle`s, but instead of only falling down, they can also flow side to side.
  - To simulate the behavior of liquids, write a `step` method in this class that picks a random direction (either down, right, or left) and then checks if the particle immediately to that direction is an instance of `Air`. If so, swap with it. If not, do nothing.
    - If you'd like, you can also allow swapping with other `LiquidParticle`s to simulate mixing.

Part 4: Concrete Particle Classes

Now that we've written the abstract classes for different types of particles, writing concrete implementations should be a breeze! We'll write particle classes for `Stone`, `Water`, and `Sand`, as well as `Barrier` and `Air`.

- First, open up the file for `Stone.java`. We'll only need to implement one method here, `color`, since the rest of the behavior is implemented in the base class. Fill in the `color` method to return whatever color you'd like. A good option for gray is `(100, 100, 100)`.
- The `Air` and `Barrier` classes will be very similar. In `Air.java`, fill in the color method to return a light-blue sky color, like `(200, 200, 255)`. In `Barrier.java`, fill in the color method to just return `(0, 0, 0)`.
- Since we've already written the behavior for liquids and gravity-particles in the corresponding abstract classes, writing `Water` and `Sand` is just as simple. In `Water.java`, fill in the color method to return a shade of blue, like `(0, 0, 255)`. In `Sand.java`, return a sand-like color like `(150, 150, 50)`.

Part 5: Creative Portion

You now have a fully functional particle simulation! Try compiling all of your files and running `Simulator.java` to try it out. For the final portion of this assignment, you'll have the opportunity to extend the behavior of this simulation however you see fit! The following steps are minimum guidelines, but feel free to add any features you'd like in addition to these.

- Write a new concrete particle class of your choice by extending one of the abstract base classes we wrote. To add it to the simulation, insert the line `classes.add("<your-class-name-here>");` after the existing particle types in `Simulator.java`. Try running the simulation with your new particle type added.
- Write a new abstract particle class for a different type of material, and follow the steps above to write at least one concrete implementation of it. This is your chance to add completely novel behavior: you could add different states of matter, particles that react with other particles, particles that can grow or spread, or even particle robots!
- For inspiration, check out some other versions of the Falling Sand game:
- https://www.silvergames.com/en/falling-sand
- https://boredhumans.com/falling_sand.php

Thank you to Jake North, OHS class of 2022, for developing this lab and allowing me to make edits to it. ~Mr. Polacco