CS102 – Algorithms and Programming II
Programming Assignment 2
Spring 2026

ATTENTION:

- Compress all of the Java program source files (.java) files into a single zip file.
- The name of the zip file should follow the below convention:
  CS102_Sec1_Asgn2_YourSurname_YourName.zip
- Replace the variables “Sec1”, “YourSurname”, and “YourName” with your actual section, surname, and name.
- Upload the above zip file to Moodle by the deadline (if not, significant points will be taken off). You will have the opportunity to update and improve your solution by consulting with the TAs and tutors during the lab. You have to make the preliminary submission before the lab day. After the TA checks your work in the lab, you will make your final submission. Even if your code does not change in between these two versions, you should make two submissions for each assignment.

GRADING WARNING:

- Please read the grading criteria provided on Moodle. The work must be done individually. Code sharing is strictly forbidden. We are using sophisticated tools to check the code similarities. The Honor Code specifies what you can and cannot do. Breaking the rules will result in disciplinary action.

Interactive Sandbox Simulation

In this assignment, you will build an interactive sandbox simulation, often called a “Falling Sand” game. You will create a graphical application with a 2D grid-based world, where each cell may contain a material such as sand, water, or smoke. The user will use their mouse to place or remove particles in the world. Each particle will follow simple movement rules and interact only with its immediate neighbors. Using these simple movement rules, particles will produce behaviors such as sand piling up, water flowing and spreading across surfaces, and smoke drifting upward and dispersing.

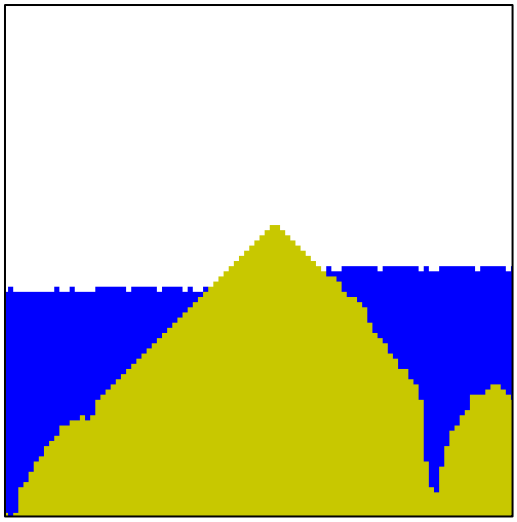
This type of simulation is a specific instance of a **Cellular Automaton**. A Cellular Automaton consists of a rectangular grid of cells, each in a particular state, which evolves over time according to fixed rules based on the states of neighboring cells. The most famous example of this concept is **Conway's Game of Life**. In the Game of Life, cells are either dead or alive. Each turn, the state of each cell changes according to the states of its eight neighbors. Despite having very simple rules, the Game of Life produces incredibly complex, emergent patterns showing how simple local interactions can give rise to rich global behaviors.

World

While the Game of Life simulates biological rules, your assignment will simulate physical ones. The world consists of a nxm grid of rectangular Cells. Each cell represents a specific coordinate (x, y) in the world. Each cell can be either empty (`null`) or contain a particle.

Your program will update the particles in regular periods. At each update step, your program will iterate over all cells and update the particles based on their update rules. Particles can stay in place, move to an empty cell, displace (swap with) another particle, or be removed from the world.

An example sandbox simulation



Particle Types

You will implement the following particle types with their update rules:

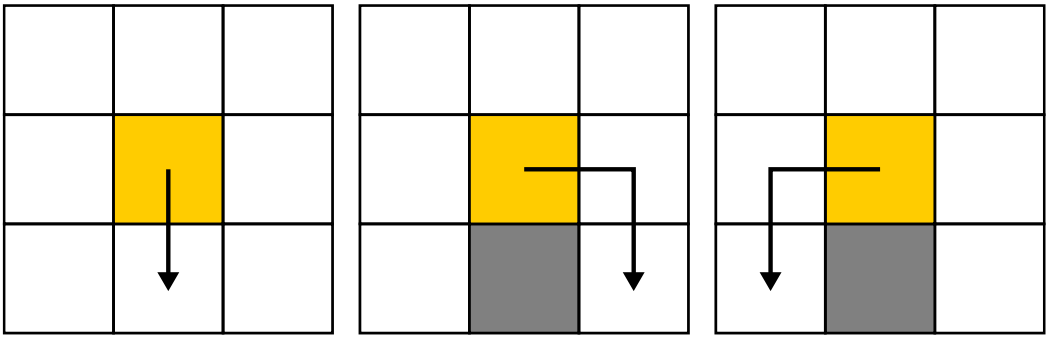
Stone

- Stone does not move.
- Cannot be displaced by any other particle

Sand

- Sand is affected by gravity and will attempt to fall whenever possible.
- If the cell directly below the sand particle is empty or contains a particle that sand can displace (such as water or smoke), the sand particle will move down into that cell.
- If downward movement is blocked, sand will try to slide diagonally: either down-left or down-right.
    - For sand to slide down-left, both the cell to the left and the cell down-left must be empty or contain a particle that sand can displace.

    - For sand to slide down-right, both the cell to the right and the cell down-right must be empty or contain a particle that sand can displace.
    - Apply the left and right diagonal rules in random order to avoid a directional bias in your simulation.
    - For sliding, the left or right cells must also be empty (or contain a particle that can be displaced by sand).
- If down and diagonals are blocked sand particle will stay in place.



Down, down-left, down-right rules of the sand particle

Water

- Water will be affected by gravity, falling downward when possible, and will spread horizontally to find its level when blocked.
- Similar to sand, water will check downwards first, then diagonals in random order.
- Water has a preferred horizontal direction, stored in a direction variable.
    - Direction is initialized randomly for each water particle.
    - It will try to move horizontally (spread) in this direction when downward and diagonal movement is blocked.
    - If movement in that direction is blocked, the direction flips, and water will attempt movement in the opposite direction on the next update.
- If water cannot move in any direction, it will stay in place.

Smoke

- Smoke will be affected by buoyancy, rising upward when possible, and will drift horizontally when blocked.
- Smoke has a lifetime:
    - After 100 steps, the smoke particle will disappear from the world.
    - As the particle ages, its color will fade from dark gray to white.
- With the following probability,

$$risePrabability = 1 - 0.8 \times \frac{Age}{Maximum\ Age}$$

smoke will attempt to rise upward. Similar to sand, when rising, smoke will first try to move up; if it fails, it will move to the diagonals (up-left, up-right) in random order.

- With the following probability,

$$spreadProbability = 0.2 + 0.5 \times \frac{Age}{Maximum\ Age}$$

smoke will attempt horizontal drift. Similar to water, smoke will have a preferred horizontal direction it will try to move. If movement towards the preferred direction is blocked, smoke will swap its preferred direction.

- If no movement is possible, smoke will stay in place.

Cell-Based Particle Simulation

Your program must update the particles regularly to simulate movement over time. An update step is a single cycle in which the program visits each particle at least once and applies its movement rules. During an update step, a particle can move to an empty cell, displace another particle, or stay in place. To maintain consistent behavior, each particle can move at most once per update step. At the start of the step, every particle is reset to an unupdated state. Once a particle successfully moves or displaces another particle, it is marked as updated and will not move again until the next step.

In the real world, particles interact and change positions simultaneously, but in your simulation, particles will be processed sequentially. One simple way to update particles is to visit all cells in a fixed order, such as top-to-bottom and left-to-right. While easy to implement, this method produces unrealistic behavior. For example, if the order is strictly top-to-bottom, particles that are falling cannot move as expected, because the top cells are updated before the bottom cells have moved. As a result, falling particles may be blocked, sideways or rising movements may favor one direction, and the overall motion does not reflect the true interactions that would occur if all particles were updated simultaneously.

To avoid these problems, your program will use a multi-pass, randomized update method. During each update step, the program will perform multiple passes over the grid, visiting cells in a random order. Once a particle successfully moves or displaces another, it is marked as updated so it will not move again until the next step.

Algorithm for a single update step is as follows:

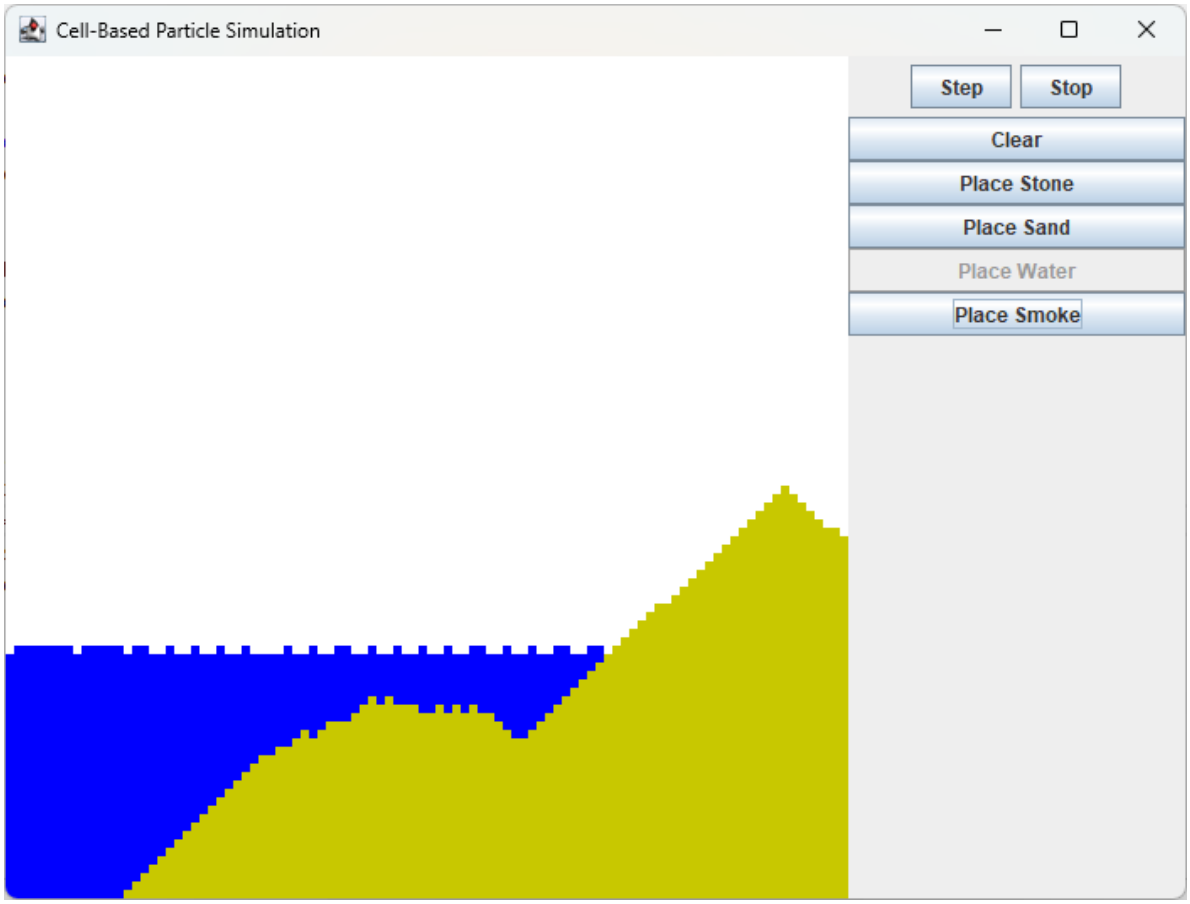
1. Reset all particles to an unupdated state.
2. Perform 8 passes over the grid:
    - Visit each particle in a randomized order.
    - If a particle is unupdated:
        - Apply its movement rule.
        - If it moves or displaces another particle, mark it as updated.

To visit the cells in a random order during each pass, you can either create an array of all particles and shuffle it, or use an `ArrayList` and apply `Collections.shuffle()`.

User Interface

Design and implement a user interface for your simulation. Your user interface should display the current state of the world by coloring each cell according to its particle type. Provide a button to update the simulation for a single step. Also, include a button to enable or disable automatic, timer-based updates. When enabled, your simulation should update every 35 milliseconds using the `javax.swing.Timer` class.

Using your program, a user should be able to add and remove particles using their mouse. Add buttons for selecting which particle to add or remove. The user should be able to add particles while pressing and dragging the mouse. Particles should be added within a circular radius. The program will iterate over the cells surrounding the cursor and add a particle to a cell only if its distance from the mouse position is less than the specified radius. Use `MouseListener` and `MouseMotionListener` to detect mouse presses and dragging.



An example user interface

You are free to design the user interface layout. Try to experiment with different layout options and organize your interface elements using panels. Use object-oriented principles in your implementation. Use inheritance and polymorphism to manage different particle types.

Preliminary Submission: You will submit an early version of your solution before the final submission. This version should at least include the following:

- Implementation of Sand.
- World update logic. For preliminary submission you can implement a simple bottom to top update
- A basic user interface showing the animation of the sand.

Not completing the preliminary submission on time results in a 50% reduction of this assignment's final grade.