Cell Society Simulation Plan

Team Four:

Davis Gossage, Wenjun Mao, Will Chang

netID: dcg13 wm56 wkc10

Introduction

In creating our program, we attempt to design a flexible, experimentable, and observable simulation of the behavior of Cellular Automata (CA). We focus the extendability of our design into our Cell hierarchy model as a way of allowing for a variety of CA behaviors under different settings and environments.

Our program architecture overall follows the Model-View-Controller design of programming, involving a main controller which handles the primary management operations of the simulation window in conjunction with a user interface as the view, or display controls. This dual-system allows for concurrent execution and modification of the simulation's settings in a closed architecture, meanwhile allowing it to remain open to parameter changes imported through an XML input file. Meanwhile, our Cell hierarchy model is based on an open architecture, allowing for other users to add additional behaviors and experiments.

Overview

In this section we give an overview of the main components of our design and how they work together to generate our simulation. The program overall (Figure 1) is split into two primary packages, the controller classes and the simulation object classes. Categorizing the classes in this way is intuitive because our program serves two purposes–simulation of CA behavior and experimentation on said simulation through controlling its variables and parameters.

The controllers include the MainController, GridManager, and UserInterface classes. To achieve the dual-system of control and display described in the introduction, the MainController and the

UserInterface classes communicate with each other to provide the core functionality of the simulation. The MainController imports the simulation parameters through an XML file, then calls the UserInterface to intialize the simulation window. Next, the MainController constructs a GridManager, which holds a 2D array that is populated with the simulation object data from the XML file. The references to the simulation objects are fed back to the UserInterface to be displayed. The MainController then initializes the gameloop and waits for input from the UserInterface. At this point, progression of the simulation depends on the user's needs and desires as processed through the UserInterface's many buttons and sliders. User input is fed back to the MainController and processed as the simulation progresses, updating the objects through the GridManager.

While the controllers handle the progression, process, and display of the simulation, the simulation object package contains the Cell Class hierarchy modeling the actual behavior of particular CAs. Currently, we have three premade simulation objects to serve the immediate needs of the user–the ForestCell, SegregationCell, and PredatorCell Subclasses. Each Cell subclass defines its own unique functions to model behavior as CA. Except for the inherited method conventions from Cell, each subclass is its own independent unit, and new subclasses of Cell may be created for user-defined simulations. More detailed information on the controller package, all Cell subclasses, and their implementations may be found in the Design Details section below.

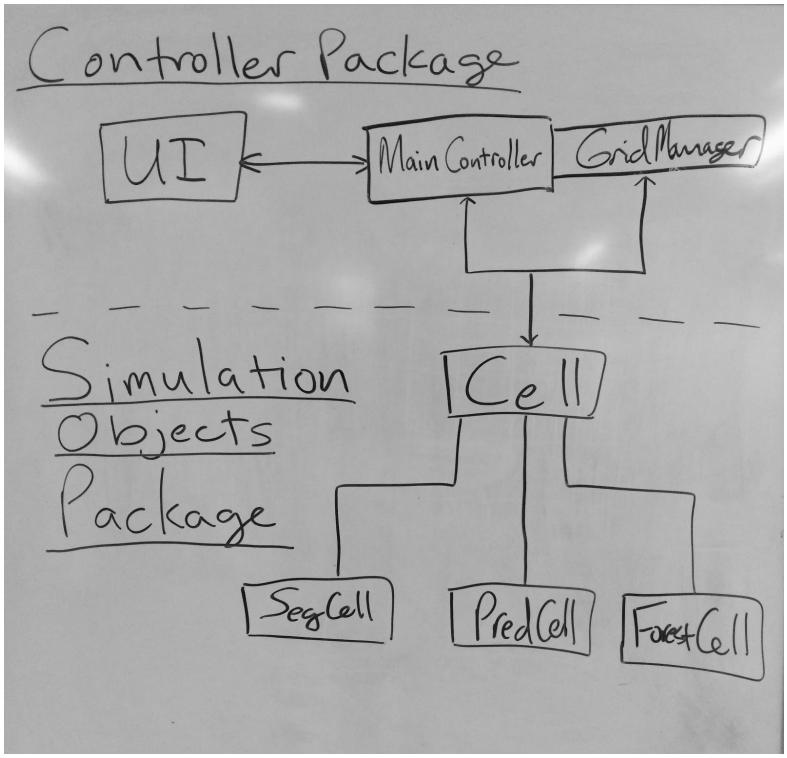

Figure (1) Program Abstraction

User Interface

The user interface of our program is a single view which is broken into two sections. The top section of this view is the simulation grid, this contains the cells for the simulation that was selected by the user. The bottom section of the view contains a settings panel for the simulation. The user can start/stop the simulation using a JavaFX Button, change the speed of the simulation using a JavaFX Slider, and choose an XML file to load data from. The file selection dialog will be shown using Java's FileDialog class. If an invalid file is received from the user, an alert will be shown using JOptionPane asking the user to choose another file. If other forms of bad input are received, such as the user clicking start without specifying a file, an alert will be shown asking the user to choose a file.

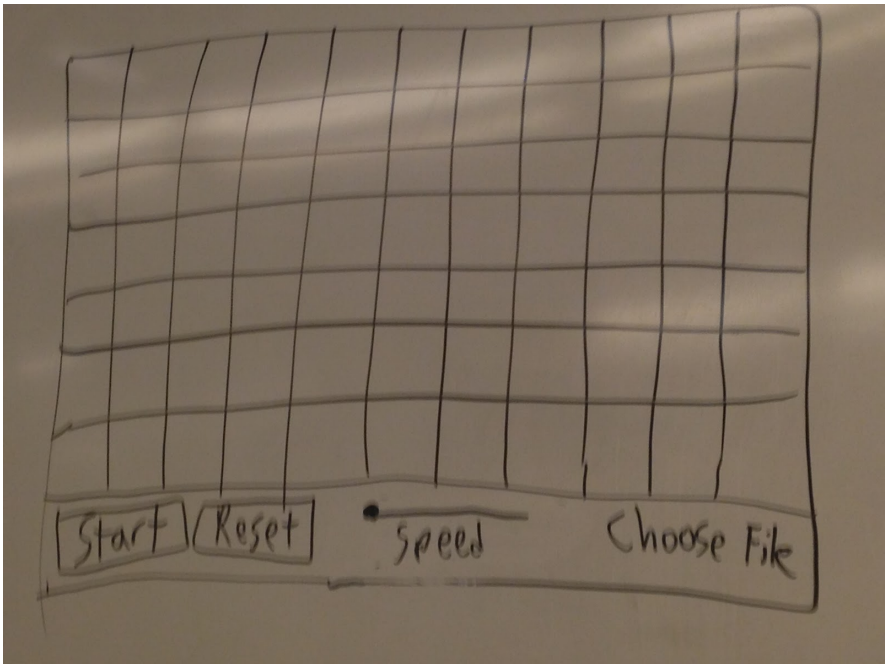

Figure (2) User Interface Sketch

Design Details

On startup, the MainController creates an instance of the User Interface class and waits for user input. After the MainController receives the XML input, the data is parsed using Java's DocumentBuilder class. The XML file contains an ASCII map of the initial simulation data, this means that each character represents a cell with a state. The ASCII map method was chosen because cells states can be represented simply and graphically in the XML data file. As the simulation data is parsed, objects are created and added to the grid using the GridManager class.

The MainController is where the gameloop happens, meaning the MainController is responsible for updating all of the cells on each frame. The MainController will query each cell object to see if it's rules are satisfied for a state change. If the cell needs to change state, then the prepareToUpdate method is invoked on the cell. When the MainController has finished looping through and determining all of the cell's future positions, the MainController invokes update on all of the cells.

The User Interface class handles any input from the user. It is discussed in detail in the User Interface section. When the User Interface handles a button press or receives a file, it sends that result of the action on to the MainController. For example, when a file is selected the UserInterface class calls the MainController method processXMLFile. This is possible because the UserInterface class keeps a reference to the instance of the MainController.

The GridManager is responsible for making updates to and interpreting the 2D array that contains cell objects. The GridManager takes a grid size as the constructor, which, in turn, determines the size of the 2D array. The GridManager handles any and all requests related to cell positioning. The GridManager is responsible for filling the array with objects as they are received from MainController using the method addCellAtPoint. Methods that the GridManager implements include getCellAtPoint, getCellsAroundPoint, and getEmptyCellsAroundPoint, all of which accept a Point object as a paramater which corresponds to a point in the 2D array. The GridManager takes this Point object and returns the single cell object or an array of adjacent cell objects based on simple array math.

The simulation objects include the abstract cell class which is extended into SegregationCell, PredatorCell, and ForestCell. Each cell is one object in our grid of cells. All cells implement two methods, prepareToUpdate and update. The prepareToUpdate method is used to prepare the cell for changing state without actually executing the changes. This becomes important in simulations like Schelling's model of segregation, and is discussed more in design considerations. The cells maintain state using enumeration. Enumeration allows states to be simply referenced and compared. To give an example, the states of the ForestCell would be EMPTY, ON_FIRE, or TREE. The corresponding integers for this enumeration would be 0, 1, or 2.

The PredatorCell subclass is unique in that the cell can either be a fish or a shark. Both of these cells have custom behaviors, so it was decided to further subclass PredatorCell into a FishCell and a SharkCell. The SharkCell keeps a track of how long it has before it starves. Both the SharkCell and the FishCell keep track of time before it can breed as well as specific rules for how each cell moves around the grid.

Design Considerations

Stationary vs. Moving blocks

Our design is to build a 2D array to hold and update the simulation of cells. Different projects given, although observed to be different, all requires the movement of certain blocks to some randomly decided or another adjacent blocks. Our main problem is how we should deal with the movement.

The first and simpler approach is to keep every cell blocks in the grid stationery, and just modify the state of the cells to make it "seems" to moved to certain locations. Another way is that since we keeps the location for each block in the cell hierarchy, we can change the point location of a cell to actually "move" the cell block to a new location.

The pro of the former choice is the stationary blocks are easier to keep track with, and the cons are that it needs to create new cells blocks for every single cells even it is empty. For a large grid with scarcely distributed non-empty blocks, the second approach can leads to a huge difference in the space complexity.

And the cons of the latter is for some simulation cases (e.g. the Segregation Model), we need to move all the blocks at the same round which would lead to a space conflict problem if we attempted to move multiple blocks to the same location.

Our basic design is to use the stationary blocks since it is simpler to handle, and we may implement the latter to deal with certain cases during further implementation period.

Cell location conflicts

As mentioned in the above paragraph, while we move the cell blocks we may encounter the location conflicting issues. Moreover, since the moving conditions of the cells are interdependent on each other, we have to perform the move "at the same round". We would need to perform a check before the actual moving operation. The way we deal with this problem is adding a prepareToUpdate method and loop through the whole grid to check whether the block needs to be moved and setting a boolean flag for each cell blocks and set a future state where we want the cell to move.

Then the update method performs the movement for each block needs moving based on the movement rules provided by different simulations.

GridManager.getCellsAroundPoint

This method of GridManager is a helper method used by prepareToUpdate method to check whether the cell is satisfied with certain conditions by counting the states of the cells around. It is also utilized by update method for the selection of an empty block for the block to perform the move.

The method returns an ArrayList of Cell blocks around a certain point. One thing to notice is that for different simulations the rules we define “around” differs. (e.g. in Segregation the around cells include all eight cells around:S,SW,W,NW,N,NE,E,SE; while in Watorworld grid, the around is just the four adjacent blocks: S,E,N,W).

The different rules are given by the input XML files and processed during cell initialization. And the cases that the block is on the side or corner of the grid should be dealt during the initialization also.

Team Responsibilities

- User Interface Class
    - 10%
    - Primary Responsibility Davis
    - Secondary Responsibility Will
- Main Controller Class
    - 25%
    - Primary Responsibility Wenjun
    - Secondary Responsibility Davis
- Grid Manager Class
    - 15%
    - Primary Responsibility Will
    - Secondary Responsibility Wenjun
- Abstract Cell Class
    - 10%
    - Primary Responsibility Davis
    - Secondary Responsibility Will, Wenjun
- Predator Cell Class
    - 20%
    - Primary Responsibility Will
    - Secondary Responsibility Davis
- Forest Cell Class
    - 10%
    - Primary Responsibility Wenjun
    - Secondary Responsibility Will
- Segregation Cell Class
    - 10%
    - Primary Responsibility Davis
    - Secondary Responsibility Wenjun

Because our team has a complete understanding of the system and thorough planning was done before hand, we chose a top-down approach for development. We will start by creating the User Interface, Main Controller, and Grid Manager. The functionality of these classes will be tested by using extremely basic cell objects. Additional cells with custom behavior (Predator, Forest, Segregation) will then be added.