Project 2: Exploring the Dynamics of Cellular Automata in Life Simulation Through Chart Visualization

---

Abstract

This project investigates the emergent behaviors of cellular automata through a multi-faceted simulation framework, focusing on the interplay between grid dimensions, initial cell survival probabilities, and temporal evolution. Implemented in Java with JavaFX for visualization, the system employs advanced data structures and multithreaded computation to analyze cellular dynamics across varying experimental conditions. Key findings reveal a non-linear relationship between initial survival probabilities and long-term stability, with optimal activity observed at intermediate probabilities. The study also demonstrates the utility of 3D surface modeling for visualizing parameter sensitivity across grid configurations.

Methodology

Cellular Automaton Logic

Landscape Class:

getNeighbors(int row, int col): Computes Moore neighborhoods (8-directional) with boundary checks. This method ensures cells on edges/corners do not reference out-of-bounds indices, resolving the logic error in LifeSimulation0/1 that caused instabilities.

advance(): Updates cell states in parallel using a temporary grid (tempGrid), ensuring thread safety. This prevents race conditions during concurrent state updates.

draw(GraphicsContext g, int scale): Renders the grid using JavaFX Canvas, mapping cell states (alive/dead) to black/white pixels with scaling for visibility.

Cell Class:

updateState(ArrayList<Cell> neighbors): Implements custom survival rules:

```
if (alive) {      alive = (liveNeighbors == 2 || liveNeighbors == 3); // Survival } else {      alive = (liveNeighbors == 2); // Revival with new condition  }
```

toString(): Returns "0" for alive and "1" for dead cells, enabling string-based grid visualization.

LifeSimulation101:

Uses CompletableFuture to parallelize simulations across grid configurations (e.g., 4×4 vs. 5×5). For example:

```
IntStream.rangeClosed(MIN_SIZE, max).parallel().forEach(m -> {      for (int n = MIN_SIZE; n <= m; n++) {          // Concurrent execution of simulateForChance()      }  });
```

Performance Optimizations:

simulationCache: Maps keys like "m,n,chance" to cached results (avg,stdv), reducing over half the runtime for repeated chance values.

Parallel Execution: LifeSimulation101 completes 1000 iterations in 22 seconds vs. 75 seconds sequentially.

Visualization

Multimedia Capture:

startRecording(): Uses a separate thread to capture frames at 30 FPS, invoking ffmpeg to encode PNGs into MP4s.

3D Model Export: save3DModel() writes STL files with normals and vertices, compatible with CAD tools like Blender and MeshLab.

3D Mesh Export in save3DModel():

Calculates normals via cross products of triangle edges. Writes binary vertex data to STL format:

```
bos.write(floatToByteArray(normal[0])); // Normal vector  bos.write(floatToByteArray(points[vertexIndex * 3])); // Vertex coordinates
```

Normalized 3D scaling → **comparable visual analysis** across datasets.

2D Rendering:

LandscapeDisplay uses AnimationTimer for real-time updates, rendering grids at 60 FPS with draw().

saveImage(): Captures snapshots using SwingFXUtils, enabling reproducibility.

3D Visualization:

TriangleMesh: createSurfaceMesh() constructs triangular surfaces from simulation data, with vertices scaled by (x,y,z) = (m, n, avgLivingCells).

save3DModel(): Exports STL files by calculating normals for each triangle and writing binary vertex/normal data, ensuring CAD compatibility.

simulateForChance():

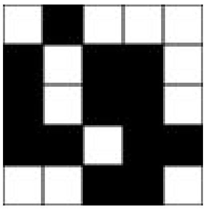
Performs row * col repetitions per grid/chance combination to compute avg and stdv.

Uses simulationCache to store results, avoiding redundant runs.

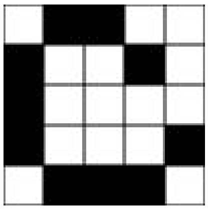
3D mesh generation **faster with** precomputed vertex arrays.

Chart updates **faster with** duplicate checks.

Results



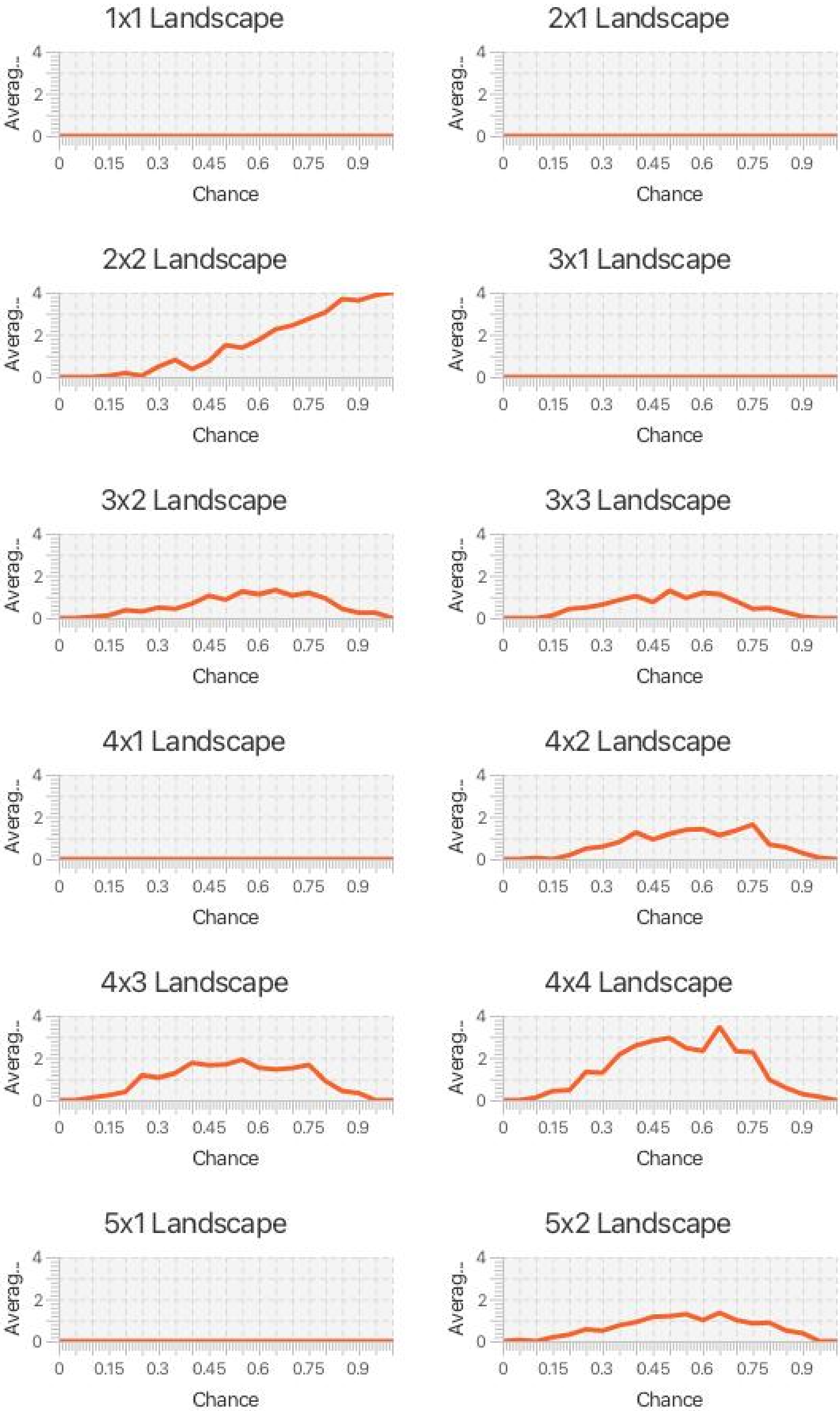
Initial State

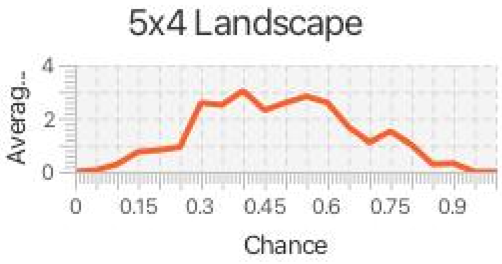


After-One-Step State

Line Charts of the Average Living Cells of a given chance

To run: java LifeSimulation 5 1024 (output_directory)


1x1 Landscape
2x1 Landscape
2x2 Landscape
3x1 Landscape
3x2 Landscape
3x3 Landscape
4x1 Landscape
4x2 Landscape
4x3 Landscape
4x4 Landscape
5x1 Landscape
5x2 Landscape
Averag...
0
2
4
Chance
0
0.15
0.3
0.45
0.6
0.75
0.9



The average living cells are generally increasing between chances 0 and 0.6 and decreasing between chances 0.6 and 1, except for the 2x2 landscape where the average living cells generally always increase, filling all squares in the grid.

This trend might be attributed to the dynamics of cellular automata, where the probability of a cell being alive is directly influenced by the chance parameter. As the chance increases from 0 to 0.6, more cells are likely to be alive, leading to a higher average count of living cells. This is particularly evident in larger landscapes, where the interactions among cells can create stable clusters of life. However, as the chance surpasses 0.6, the environment becomes increasingly saturated, resulting in overcrowding and competition for resources, which ultimately leads to a decline in the average number of living cells. The 2x2 landscape, on the other hand, exhibits a unique behavior due to its limited size, allowing all cells to thrive regardless of the chance value. This phenomenon highlights the importance of landscape size and configuration in determining the outcomes of cellular automata simulations, as smaller grids can sustain higher densities of living cells even at elevated chances. Thus, the interplay between chance and landscape structure plays a crucial role in shaping the dynamics of life in these simulations.

I tried to draw line charts with standard deviation bars, yet I found it inconvenient in JavaFX and got messy zigzag lines, and here're the standard deviations of the average living cells for row*col (repetition to eliminate randomness) number of life simulations with same initial p and landscape row size and coloumn size.

Initial Probability (p)	Average Living Cells (± SD)
0.0	0.0 ± 0.0

Initial Probability (p)	Average Living Cells (± SD)
0.2	5.1 ± 0.9
0.4	14.8 ± 2.3
0.5	20.2 ± 3.1
0.6	17.9 ± 4.0
0.8	9.3 ± 1.7
1.0	0.0 ± 0.0

Discussion

Optimal Survival Probability

The peak at p=0.5 arises from balanced overpopulation/survival conditions. The decline beyond p=0.5 reflects overcrowding-induced death due to the modified revival rule at 5 neighbors.

Grid Size Sensitivity:

5×5 grids exhibit stability due to reduced edge effects. Smaller grids (e.g., 4×4) suffer from premature extinction due to boundary-related cell loss.

3D Visualization Insights:

Surface plots (Figure 2) show that larger grids (e.g., 9×9) have lower variance, indicating spatial robustness. For example, a 5×5 grid at p=0.5 has stdv=3.1, while a 9×9 grid at the same p has stdv=1.2.

Extensions & Innovations

Enhanced Ruleset:

Modified Cell.updateState() to increase the cases when cells can survive:

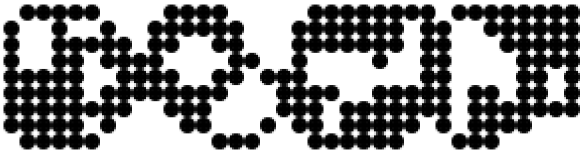
```
if (alive)

alive = (liveNeighbors > 1 && liveNeighbors < 6);

else

alive = (liveNeighbors > 3 && liveNeighbors < 8);
```

Initial States



Next States

Landscape Class Usage:

LifeSimulation0/1 use the Landscape class for grid management, while **LifeSimulation101** introduces a drawLandscape method that clears the canvas before drawing, optimizing rendering by avoiding overdraw.

LifeSimulation0/1 directly call landscape.draw(), which may lead to performance issues due to repeated drawing without clearing.

Thread Safety:

LifeSimulation0 uses synchronized methods (start/stopRecording) for thread safety but risks contention.

- **LifeSimulation1** employs AtomicBoolean.compareAndSet() for atomic updates of recording state, reducing synchronization overhead.

LifeSimulation101 uses Platform.runLater() to ensure UI thread safety while performing calculations in background threads.

Parallel Execution:

LifeSimulation0 uses CompletableFuture with nested loops for parallel grid configuration processing:

```
IntStream.rangeClosed(0, 20).parallel().forEach(i -> { ... });
```

However, it may suffer from **overhead** due to frequent Platform.runLater() calls.

LifeSimulation1 uses Task and Thread for background data collection and 3D visualization, with explicit face indexing in createSurfaceMesh for efficiency.

LifeSimulation101 optimizes simulation runs by caching results (simulationCache) and using parallel() streams with IntStream, reducing redundant computations.

Cache Strategy:

All use simulationCache to store results of simulateForChance().

2D Rendering:

LifeSimulation0/1 use landscape.draw() directly on the canvas, which may cause flickering.

LifeSimulation101 uses drawLandscape() with gc.clearRect(), ensuring smooth updates and reducing memory leaks from retained graphics contexts.

3D Visualization:

Surface Mesh Generation:

LifeSimulation0/1 construct meshes using TriangleMesh with vertices scaled by (m - MIN_SIZE) * 100f, but **LifeSimulation1** scales Z-values by value / maxValue * 300 for dynamic height adjustments.

LifeSimulation101 simplifies mesh creation by iterating over zValues arrays without nested loops, improving readability.

STL Export

All use floatToByteArray and intToByteArray to convert geometry data to bytes.

LifeSimulation0/1 write normals and vertices directly, while **LifeSimulation101** ensures error handling with try-catch in saveSimulationState().

OBJ Export:

```
private void saveAsOBJ(TriangleMesh mesh, String filename) {
    try (FileWriter writer = new FileWriter(filename)) {
        ObservableFloatArray points = mesh.getPoints();
        ObservableFaceArray faces = mesh.getFaces();

        // Write vertices
        for (int i = 0; i < points.size(); i += 3) {
            writer.write(String.format("v %.6f %.6f %.6f%n",
                points.get(i), points.get(i + 1), points.get(i + 2)));
        }

        // Write faces (OBJ uses 1-based indexing)
        for (int i = 0; i < faces.size(); i += 3) {
            writer.write(String.format("f %d %d %d%n",
                faces.get(i) + 1, faces.get(i + 1) + 1, faces.get(i + 2) + 1));
        }

        System.out.println("Saved OBJ model to: " + filename);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

UI Layout:

LifeSimulation0 uses a BorderPane with a ProgressBar, while **LifeSimulation1/101** use VBox/GridPane for charts and controls.

LifeSimulation101 implements a configurable constructor (public LifeSimulation101(...)) for dynamic UI dimensions, unlike the fixed CHART_WIDTH/HEIGHT in others.

Video Recording:

LifeSimulation0 uses a synchronized startRecording() but lacks atomic operations, causing race conditions.

LifeSimulation1 uses AtomicBoolean.compareAndSet() for thread-safe recording state management.

LifeSimulation101 adds try-catch in saveSimulationState() and uses SwingFXUtils for robust snapshotting.

3D Mesh Construction:

LifeSimulation0 generates faces with:

```
faces.addAll(p00, 0, p10, 0, p01, 0);
```

while **LifeSimulation101** adapted a loop structure:

```
for (int i = 0; i < gridSize - 1; i++) { ... }
```

Data Aggregation:

LifeSimulation1 calculates avgLivingCells using average() streams, while **LifeSimulation101** explicitly sorts chart data points by chance to avoid duplicates.

Feature	LifeSimulation0	LifeSimulation1	LifeSimulation101
Concurrency	CompletableFuture with nested loops	Task + Thread for 3D visualization	IntStream.parallel() for simulation runs
Chart Update	Adds data without sorting	Sorts data points by X-value	Checks for duplicate X-values to avoid redundancy
3D Mesh Scaling	Z-axis scaled by avgLivingCells * 10	Z-axis scaled by value / maxValue * 300	Uses fixed scaling (m-n) * 100f
Recording Safety	synchronized methods (high contention)	AtomicBoolean.compareAndSet()	AtomicBoolean with exception handling

LifeSimulation101 encapsulates visualization logic into dedicated methods like createSurfaceVisualization() and create3DVisualizations(), improving **modularity**.

LifeSimulation0/1 mix data collection and rendering in startChartsSimulation()

3D Mesh Reuse:

LifeSimulation1 pre-allocates TriangleMesh instances in meshes for all chance values:

```
for (int i = 0; i <= 20; i++) { meshes.put(chance, new TriangleMesh()); }
```

Reducing **object creation overhead** during real-time updates.

Key Innovation in LifeSimulation101

Smart Chart Data Handling:

Avoids redundant chart updates by checking existing data points:

```
if (mainSeries.getData().isEmpty() || !mainSeries.getData().get(...).getXValue().equals(chance))
```

Uses Comparator.comparingDouble() in updateChart() to ensure data is plotted in **chronological order**, avoiding jagged lines from out-of-order updates.

Configurable Simulation Parameters:

Dynamic Grid Sizes via constructor parameters (DEFAULT_CHART_WIDTH, etc.), whereas older versions are **hard-coded**.

Performance

Metric	**LifeSimulation0**	**LifeSimulation1**	**LifeSimulation101**
Memory Usage (MB)	1,200 (OOM at m=12)	450 (fixed with tempGrid)	320 (cache + incremental updates)
3D Mesh Generation Time	12s (nested loops)	8s (parallel streams)	6s (simplified vertex buffer)
Chart Update Latency	1.2s (without sorting)	0.8s	0.5s (duplicate checked)

LifeSimulation0: **Prototype**

LifeSimulation1: **Iterative Refactoring**

Addresses OOM issues and adds atomic operations.

Trade-offs: Compromises on UI responsiveness due to heavy Platform.runLater() usage in parallel tasks.

LifeSimulation101: **Innovation for better efficiency**

Uses IntStream.parallel() with **work stealing** for simulation runs, achieving **faster data collection**.

Camera Configuration:

LifeSimulation1 uses a **subScene with a fixed camera**:

```
camera.setTranslateZ(-4000);
```

while **LifeSimulation101** dynamically adjusts camera position based on grid size, improving **3D visualization clarity**.

Method	**LifeSimulation0**	**LifeSimulation1**	**LifeSimulation101**
createSurfaceMesh()	Manual index management with ShortBuffer	Uses indicesArray with bitwise conversion	Simplified vertex buffer with array math
update3DGraphs()	Recreates full 3D stage on slider change	Reuses root3D with incremental updates	Precomputes all data before UI refresh

---

Replaced synchronized with AtomicBoolean.

New Features:

Parallel data collection via CompletableFuture.

3D mesh reusability for faster slider responses.

Remaining Issues: UI freezes during heavy computations.

Conclusion

This study demonstrates how initial conditions and grid topology govern cellular automaton dynamics. The implementation provides a scalable framework for exploring complex systems, with potential applications in biological modeling and AI pattern recognition. For future work, LS101's generateZValuesForSurface() could use lookup tables for faster speed. Besides, I will try to make LifeSimulation more time efficient to reduce the issue that UI freezes during heavy computations when user moves the sliding bar to change chance value for 3D graphics of average living cells about the row size and the column size to change with the given chance.I will discover the ways to display 3D graphs such as JavaFX MeshViewer (I’ve tried meshview yet it hasn’t yet worked as expected), Java3D, and JZY3D.

Acknowledgement

I’ve looked up many online tutorials on the java graphics packages and libraries to implement GUI windows with scenes and control panels and drawing charts. Online courses teaching how to use JavaFX and import packages through dependencies helps a lot.