A Cellular Automata Based Model of Urban Development

Matthew Hefner

May 8, 2019

Background

Cities are immensely complex, dynamic structures. Attempting to mathematically model the organization of cities is an endeavor that is equally as challenging as it is fascinating. The models developed can help shed light on the mechanics of sociological, political, economic, and environmental systems and peoples' roles within them.

The study of these models was generally left to the social sciences before the mid-twentieth century, before which time most mathematical and natural science based models centered around deterministic systems that reached equilibrium fairly quickly [1]. As such, these models generally failed to reflect the greater reality of the constant flux of real-world cities beyond the scope of their specific purpose for short time-scales.

As the proliferation of increasingly powerful computing systems allowed for efficient simulation, models of urban development bloomed from the minds of scientists eager to take advantage of the new technology. From this emerged the notion of computable static structures that act within an overall dynamic system. One of these structures came to dominate the early study of these models thanks to its elegant nature and the ease of simulation: Cellular Automata [1].

Cellular Automata are a collection of entities, or cells, that are in some state within a state space. As an example, a cell may be "Alive" or "Dead." Cells are generally contained within a grid and each reflects the state of their location within that lattice. These cells are stationary with respect to an overall system. Cells within Cellular Automata change state synchronously in reaction to the states of their *neighborhood,* the cells that surround it, and their current state. The size and shape of this neighborhood may vary from model to model, along with the cells' state space and the rules that govern the reaction. Though patterns of state change may represent movement of populations, they are particularly useful for the modeling of the physical environment and structures within a city [2].

Development of the Model

There are numerous drivers of change within an urban system. Not all drivers are pertinent in all models; their inclusion or exclusion depends on the purpose of a model and is a balance between desired fidelity and how well their causal relationships are understood. One excellent categorization scheme for various modeling purposes is provided by Batty in *Cities and Complexity* [1]:

- *Physical Determinism* - The literal physical landscape and climate of an area.
- *Comparative Advantage* - Economic incentives that move populations.
- *Natural Advantage* - Natural resources that exist within the area.
- *Historical Accident* - Happenstance, often simply expedient development choices of populations.

As previously mentioned, Cellular Automata are particularly useful in modeling the change of physical structures within an urban environment. As such, the model developed here will seek to represent changes in the physical structures of a city based on physical determinism and historical accident. Comparative and natural advantages are perhaps better represented with Agent-Based models [1].

The neighborhood utilized for this model will be the *Moore Neighborhood*, one of the two most commonly used neighborhoods for Cellular Automata based models, a diagram of which is provided in Figure 1 [2].

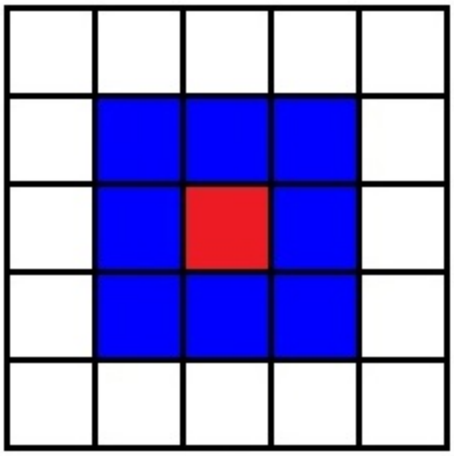

Figure 1: The Moore neighborhood, shown in blue, of a given cell, shown in red.

This Cellular Automata will be on (and affected by) a surface that represents the terrain of the area for which urban development is being modeled. This surface will be represented notationally by the function $f(x, y)$. Each cell within this model represents a lot of space. The time-step of this model will be 5 years, selected as a baseline realistic timespan for which buildings may be built or destroyed. Each cell will have three state characteristics.

The first of these characteristics is simply whether a building is built at that lot's location within the cellular automata - a binary *built* or *vacant*. The rule-set for a construction and destruction is dependent on the slope ($\|\nabla f(x, y)\|$) of the surface terrain at a cell location. A steeper gradient will be less hospitable to city life; it is by this mechanism that this model incorporates physical determinism. There are three levels of gradient, with varying levels of hospitality to buildings, defined by the constants g_1 and g_2. These levels of gradient are named in the table below for simplicity of explanation.

Gradient	Level Name
$\|\nabla f(x, y)\| < g_1$	**Flat**
$g_1 \leq \|\nabla f(x, y)\| \leq g_2$	**Sloped**
$\|\nabla f(x, y)\| > g_2$	**Steep**

A cell is *built* when:

- the cell is located at *any* gradient terrain and has exactly 3 neighboring built buildings.

A cell *remains* built when:

- the cell is located at *any* gradient terrain and has 2 or 3 neighboring built buildings, *or*
- the cell is located on a **Sloped** gradient terrain and has greater than 1 but less than 5 neighbors, *or*
- the cell is located on a **Flat** gradient terrain and has greater than 1 neighbors and less than 8 neighbors.

If these conditions are not met, cell becomes or remains *vacant*.

The second cell characteristic is its **urban density**. This is defined to be its exact number of built neighboring cells.

The third characteristic is a cell's **ruler**, represented by r, $r \in \mathbb{N}$. Represented with color in the interactive Java implementation of this model below, the ruler represents a discrete state for which majority is the deciding factor when the cell is built. When a cell is built, the majority value of the rulers of the three neighboring cells is passed to the cell being built. *If all three cells have different ruler values, the cell is not built.*

By intention, this may be interpreted in various ways, including political and social characteristics such as government administration or language.

The final consideration of the model is *historical accident*. Each year, a building may be destroyed by fire, torn down due to age, or collapse in a tragic accident involving the weight of 4,000 cats in the apartment of a hoarding, otherwise well-intentioned cat man. This is incorporated into the model simply by randomly destroying buildings within the automata. The destruction probability, D, is constant for the duration of the model and for all cells. All cells thus have a D probability of being destroyed every 5 years within the model.

Results

For experiment and simulation, this model was implemented in the programming language Java; the code for this implementation is provided in the code appendix.

The surface chosen for this simulation was $f(x, y) = sin(\frac{x}{40}) \cdot sin(\frac{y}{40})$ with cell indexes $x \in \{0, 1, ..., 249\}$, $y \in \{0, 1, ..., 249\}$. The values of $g_1 = 0.3$ and $g_2 = 0.8$ were determined experimentally to yield excellent results for this simulation.

Four rulers, $r \in \{0, 1, 2, 3\}$, were used, represented by red, purple, yellow, and green respectively within the simulation. The initial configuration was chosen to be four random straight lines within the automata, each line set to one of the four rulers.

These results are surprising in both their beauty and realism.

Global-Level System Behavior

Globally, as predicted in the development of the model, urban density is greatest at **Flat** gradients, and more dense at **Sloped** gradients than **Steep** gradients. This behavior is illustrated in the screen capture of the simulation shown in Figure 2.

City-Level System Behavior

At the city-level, the rule-set for the density of **Sloped** gradients yields a maze-like road structure of vacant cells. **Steep** gradients sustain relatively small, village-like auxiliary communities. The dense **Flat** gradients produce vibrant locales of surprising beauty and organized complexity. This behavior is illustrated in the screen capture of the simulation shown in Figure 3. The city center has developed a main thoroughfare and a large, open green space.

Discussion

While the urban development that arose from the simulation yielded cities with highly artificial, diamond-like shapes, this is only a result of the shape of the chosen surface. Real terrain data could be interpolated and used, resulting in a simulation that better reflects reality. Still, this model only incorporates physical determinism and historical accident. It is possible to interactively pair this model with an Agent-Based model that represents the movement of populations within these cities, which would represent comparative and natural advantage as well.

It may also be possible to better reflect not only the location and size of physical structures, but their type as well. This could perhaps be achieved manipulating the ruler characteristic interactions to reflect urban zoning - such as governmental, commercial, industrial, and residential zones - and manipulating the cellular neighborhoods to incorporate these changes.

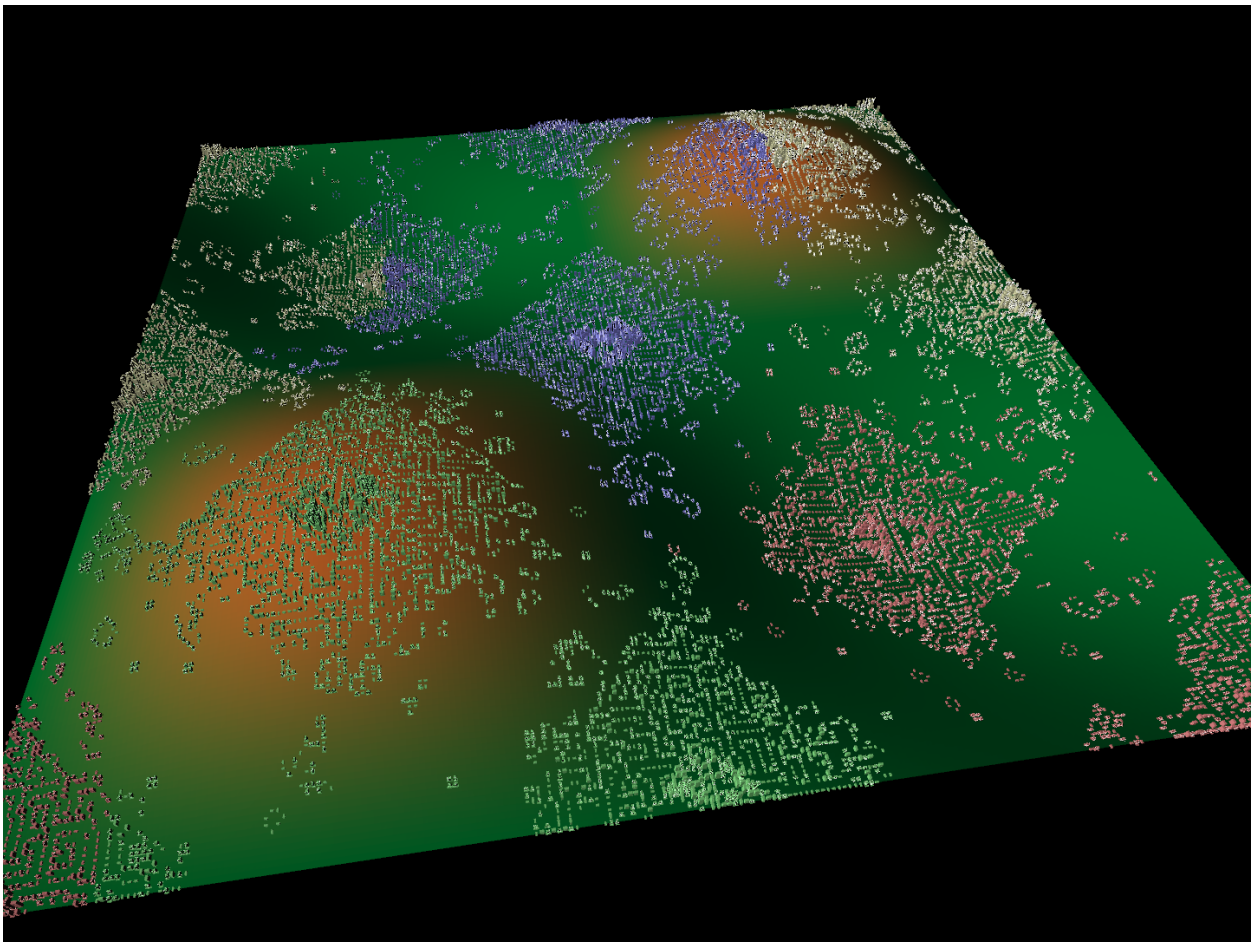

Figure 2: Simulation results of global-level system behavior.

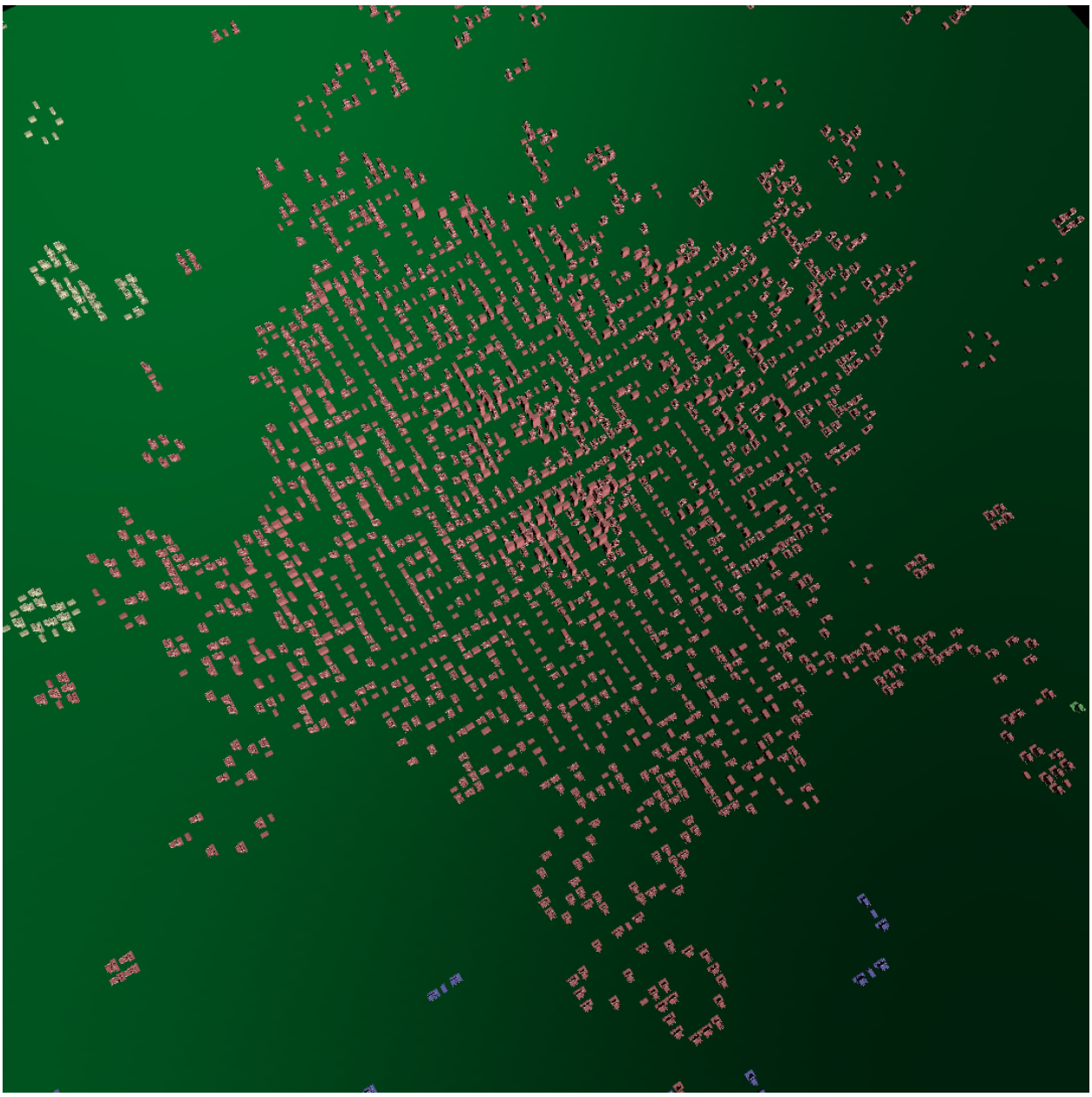

Figure 3: Simulation results of city-level system behavior.

References

1. Michael Batty. *Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals.* The MIT Press, 2007. Print.

2. Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics.* Boca Raton, Fla: CRC Press, 1999. Print.

Code Appendix

```
import processing.core.*;
import processing.event.MouseEvent;
import java.util.Arrays;
import entity.*;


/**
 * MultiAutomata3d is a 3-Dimensional, interactive cellular automata based
 * model of urban development utilizing the Processing library for Java.
 *
 * @author Matthew Hefner
 *
 */
public class MultiAutomata3D extends PApplet{
    private boolean[][] active; //2d boolean array of iteration
    private boolean[][] newActive; //2d boolean array of next iteration
    private boolean[][] alive; //2d boolean array of iteration
    private boolean[][] newAlive; //2d boolean array of next iteration
    private int rows; //int # of rows
    private int cols; //int # of cols
    private boolean paused; //boolean toggles resume/pause
    private int[][] neighbors; //2d int array of neighbors
    private int[][] rulers; //2d int array of rulers
    private int[][] last; //2d int array storing the last ruler
    private float[][] rotation;
    private int scalar; //int size of each cell when drawn to screen
    //
    private Navigation nav;
    private PShape[][] buildings;
    private PShape[][] road;
    private PShape ground;

    /**
     * Runs initial setup and initializes the cellular automata model.
     */
    public void setup() {
        background(0);
        loadObjects();
        paused = true;
        scalar = 30;
        rows = 250;
        cols = 250;
        active = new boolean[rows][cols]; //2d boolean array of iteration
        newActive = new boolean[rows][cols]; //2d boolean array of next iteration
        alive = new boolean[rows][cols]; //2d boolean array of iteration
        newAlive = new boolean[rows][cols]; //2d boolean array of next iteration
        neighbors = new int[rows][cols]; //2d int array of neighbors
        rulers = new int[rows][cols]; //2d int array of rulers
        last = new int[rows][cols];
        rotation = new float[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
```

```
                int r = (int) random(4, 2*PI);
                rotation[i][j] = r * PI / 2;
            }
        }
        shapeMode(CENTER);
        ground = ground();
        nav = new Navigation(this, rows * scalar, cols * scalar);
    }

    /**
     * Loads the OBJ files for buildings displayed in the interactive
     * environment.
     */
    public void loadObjects() {
        //2 Neighbors
        buildings = new PShape[4][8];
        buildings[0][2] = loadShape("Residential Buildings 001.obj");
        buildings[1][2] = loadShape("Residential Buildings 001.obj");
        buildings[2][2] = loadShape("Residential Buildings 001.obj");
        buildings[3][2] = loadShape("Residential Buildings 001.obj");
        buildings[0][2].setFill(color(150, 255, 150));
        buildings[1][2].setFill(color(150, 150, 255));
        buildings[2][2].setFill(color(255, 150, 150));
        buildings[3][2].setFill(color(255, 255, 200));
        buildings[0][2].scale((float) .8);
        buildings[1][2].scale((float) .8);
        buildings[2][2].scale((float) .8);
        buildings[3][2].scale((float) .8);
        //3 Neighbors
        buildings[0][3] = loadShape("Tower-3-1.obj");
        buildings[1][3] = loadShape("Tower-3-1.obj");
        buildings[2][3] = loadShape("Tower-3-1.obj");
        buildings[3][3] = loadShape("Tower-3-1.obj");
        buildings[0][3].setFill(color(150, 255, 150));
        buildings[1][3].setFill(color(150, 150, 255));
        buildings[2][3].setFill(color(255, 150, 150));
        buildings[3][3].setFill(color(255, 255, 200));
        buildings[0][3].scale((float) 0.3);
        buildings[1][3].scale((float) 0.3);
        buildings[2][3].scale((float) 0.3);
        buildings[3][3].scale((float) 0.3);
        //4 Neighbors
        buildings[0][4] = loadShape("Residential Buildings 005.obj");
        buildings[1][4] = loadShape("Residential Buildings 005.obj");
        buildings[2][4] = loadShape("Residential Buildings 005.obj");
        buildings[3][4] = loadShape("Residential Buildings 005.obj");
        buildings[0][4].setFill(color(150, 255, 150));
        buildings[1][4].setFill(color(150, 150, 255));
        buildings[2][4].setFill(color(255, 150, 150));
        buildings[3][4].setFill(color(255, 255, 200));
        buildings[0][4].scale((float) 1);
        buildings[1][4].scale((float) 1);
        buildings[2][4].scale((float) 1);
```

```
        buildings[3][4].scale((float) 1);
        //5 Neighbors
        buildings[0][5] = loadShape("Tower-3-2.obj");
        buildings[1][5] = loadShape("Tower-3-2.obj");
        buildings[2][5] = loadShape("Tower-3-2.obj");
        buildings[3][5] = loadShape("Tower-3-2.obj");
        buildings[0][5].setFill(color(150, 255, 150));
        buildings[1][5].setFill(color(150, 150, 255));
        buildings[2][5].setFill(color(255, 150, 150));
        buildings[3][5].setFill(color(255, 255, 200));
        buildings[0][5].scale((float) 0.3);
        buildings[1][5].scale((float) 0.3);
        buildings[2][5].scale((float) 0.3);
        buildings[3][5].scale((float) 0.3);
        //6 Neighbors
        buildings[0][6] = loadShape("Residential Buildings 007.obj");
        buildings[1][6] = loadShape("Residential Buildings 007.obj");
        buildings[2][6] = loadShape("Residential Buildings 007.obj");
        buildings[3][6] = loadShape("Residential Buildings 007.obj");
        buildings[0][6].setFill(color(150, 255, 150));
        buildings[1][6].setFill(color(150, 150, 255));
        buildings[2][6].setFill(color(255, 150, 150));
        buildings[3][6].setFill(color(255, 255, 200));
        buildings[0][6].scale((float) 1.1);
        buildings[1][6].scale((float) 1.1);
        buildings[2][6].scale((float) 1.1);
        buildings[3][6].scale((float) 1.1);
        //7 Neighbors
        buildings[0][7] = loadShape("Tower-3-3.obj");
        buildings[1][7] = loadShape("Tower-3-3.obj");
        buildings[2][7] = loadShape("Tower-3-3.obj");
        buildings[3][7] = loadShape("Tower-3-3.obj");
        buildings[0][7].setFill(color(150, 255, 150));
        buildings[1][7].setFill(color(150, 150, 255));
        buildings[2][7].setFill(color(255, 150, 150));
        buildings[3][7].setFill(color(255, 255, 200));
        buildings[0][7].scale((float) 0.4);
        buildings[1][7].scale((float) 0.4);
        buildings[2][7].scale((float) 0.4);
        buildings[3][7].scale((float) 0.4);
    }

    /**
     * Sets the size of the screen to fullscreen.
     */
    public void settings(){
        fullScreen(P3D);
    }

    /**
     * Main loop of the program.  Ran at each frame, this method drives
     * the generational development of the cellular automata model and
     * draw the major elements - buildings and the ground - to the screen,
```

```
     * including lighting.
     */
    public void draw(){
        background(0);
        directionalLight(255, 255, 255, 1, (float) -0.75, (float) 0.5);
        ambientLight(15, 15, 15);
        nav.movement();
        nav.draw();
        translate(0, 0, 0);
        drawGround();
        generation();
    }

    /**
     * Called by draw() - draws the surface of the ground to the screen.
     */
    public void drawGround() {
        pushMatrix();
        translate((rows * scalar / 2),
                550,
                0);
        shape(ground);
        popMatrix();
    }

    /**
     * Builds the ground surface.
     *
     * @return the PShape that represents the ground surface.
     */
    public PShape ground() {
        PShape ps = createShape(GROUP);
        road = new PShape[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                PShape child = createShape();
                road[i][j] = createShape(GROUP);
                child.beginShape();
                child.noStroke();
                child.fill(map((int) terrain(i, j), 0, 550, 0, 255),
                        120,
                        45);
                child.vertex(i * scalar - (rows * scalar / 2),
                        terrain(i,j),
                        j * scalar - (cols * scalar / 2));
                child.vertex((i + 1) * scalar - (rows * scalar / 2),
                        terrain(i + 1,j),
                        j * scalar - (cols * scalar / 2));
                child.vertex((i + 1) * scalar - (rows * scalar / 2),
                        terrain(i + 1,j + 1),
                        (j + 1) * scalar - (cols * scalar / 2));
                child.vertex(i * scalar - (rows * scalar / 2),
                        terrain(i,j + 1),
```

```
                        (j + 1) * scalar - (cols * scalar / 2));
                child.endShape();
                ps.addChild(child);
                PShape child2 = createShape(LINE, i * scalar - (rows * scalar / 2),
                        terrain(i,j + (float) 0.5),
                        (j + (float) 0.5) * scalar - (cols * scalar / 2),
                        (i + 1) * scalar - (rows * scalar / 2),
                        terrain(i + 1,j + (float) 0.5),
                        (j + (float) 0.5) * scalar - (cols * scalar / 2));
                child2.setStrokeWeight(3);
                PShape child3 = createShape(LINE, (i + (float) 0.5) * scalar - (rows * scalar / 2),
                        terrain(i + (float) 0.5,j),
                        (j) * scalar - (cols * scalar / 2),
                        (i + (float) 0.5) * scalar - (rows * scalar / 2),
                        terrain(i + (float) 0.5,j + 1),
                        (j + 1) * scalar - (cols * scalar / 2));
                child3.setStrokeWeight(3);
                road[i][j].addChild(child2);
                road[i][j].addChild(child3);
            }
        }
        return ps;
    }

    /**
     * The mathematical function that defines the ground shape.
     * @param i the x coordinate of the ground
     * @param j the y coordinate of the ground
     * @return the z coordinate of the ground
     */
    public float terrain(float i, float j) {
        return 550 * sin((float) i / 40) * sin((float) j / 40);
    }

    /**
     * The mathematical function that defines the slop of the ground shape.
     * @param i the x coordinate of the ground
     * @param j the y coordinate of the ground.
     * @return the slope of the ground at (x,y)
     */
    public float terrainDiff(float i, float j) {
        return (abs(sin(i / 40) * cos(j / 40)) + abs(sin(j / 40) * cos(i / 40)));
    }

    /**
     * Dictates the keyboard controls of the interactive model.
     */
    public void keyPressed() {
        if (key == ' ') {
            pauseProcess();
        } else if (key == 's') {
            startMoves();
        } else if (key == 'p') {
```

```
            saveFrame(frameCount + ".png");
        }
        if (keyCode == DELETE) {
            active = new boolean[rows][cols]; //2d boolean array of iteration
            newActive = new boolean[rows][cols]; //2d boolean array of next iteration
            alive = new boolean[rows][cols]; //2d boolean array of iteration
            newAlive = new boolean[rows][cols]; //2d boolean array of next iteration
            neighbors = new int[rows][cols]; //2d int array of neighbors
            rulers = new int[rows][cols]; //2d int array of rulers
            last = new int[rows][cols]; //2d int array of rulers
        }
    }

    /**
     * Allows the model to be paused at the will of the user.
     */
    public void pauseProcess() {
        paused = !paused;
        if (paused) {
            newActive = new boolean[rows][cols];
            newAlive = new boolean[rows][cols];
            for (int k = 0; k < rows; k++) {
                newActive[k] = Arrays.copyOf(active[k], active[k].length);
                newAlive[k] = Arrays.copyOf(alive[k], alive[k].length);
            }
        } else {
            for (int k = 0; k < rows; k++) {
                active[k] = Arrays.copyOf(newActive[k], newActive[k].length);
                alive[k] = Arrays.copyOf(newAlive[k], newAlive[k].length);
            }
            newActive = new boolean[rows][cols];
            newAlive = new boolean[rows][cols];
            fullCheck();
        }
    }

    /**
     * Draws an initial configuration to the cellular automata model.
     *
     * Currently set to draw straight lines of each of the colors of rulers.
     */
    public void startMoves() {
        for (int u = 0; u < rows; u++) {
            for (int v = 0; v < cols; v++) {
                if (u == (rows / 4) && v < cols / 2) {
                    mouseHelper(u, v, 1);
                }
                if (u == (rows / 2) && v > cols / 2) {
                    mouseHelper(u, v, 2);
                }
                if (u == (3 * rows / 4) && v < cols / 2) {
                    mouseHelper(u, v, 3);
                }
```

```
            if (u == (rows - 1) && v > cols / 2) {
                mouseHelper(u, v, 4);
            }
        }
    }
}

/**
 * private helper procedure for the startMoves() method.
 * @param row the row of the cell to be created
 * @param col the col of the cell to be created
 * @param rul the ruler of the cell to be created
 */
public void mouseHelper(int row, int col, int rul) {
    rulers[row][col] = rul;
    last[row][col] = rul;
    newActive[row][col] = true;
    newAlive[row][col] = true;
    newLife(row, col);
}

/**
 * "Historical Accident" - gives a 0.01% chance of a cell
 * randomly dying at each time step of the model.
 */
public void noise() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (active[i][j]) {
                if (random(1) > 0.9999) {
                    //Death; still active
                    newActive[i][j] = true;
                    newAlive[i][j] = false;
                }
            }
        }
    }
}

/**
 * Calculates the next time step of the Cellular Automata model.
 *
 * Here rules of the cellular automata are implemented, and cells
 * die, are born, or remain alive based on their neighbors and the
 * terrain of the ground calculated in terrain() and terrainDiff().
 */
public void generation() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (active[i][j]) {
                fill(255);
                if ((neighbors[i][j] == 2 || neighbors[i][j] == 3 ||
                        (abs(terrainDiff(i, j)) < 0.80 && (neighbors[i][j] == 4))
```

```
                        || (abs(terrainDiff(i, j)) < 0.30 && (neighbors[i][j] > 4 && neighbors[i][j]
                    if (rulers[i][j] == 0 && neighbors[i][j] > 3) {
                        //Tie death, stays active
                        newActive[i][j] = true;
                        newAlive[i][j] = false;
                    } else {
                        //Stays alive
                        newActive[i][j] = true;
                        newAlive[i][j] = true;
                        drawEntity(i, j);
                    }
                } else if ((neighbors[i][j] == 3) & !alive[i][j]) {
                    //New life.
                    newLife(i,j);
                    newActive[i][j] = true;
                    newAlive[i][j] = true;
                    drawEntity(i, j);
                } else if ((neighbors[i][j] >= 9) & alive[i][j]) {
                    if (rulers[i][j] == 0) {
                        //Tie death, stays active
                        newActive[i][j] = true;
                        newAlive[i][j] = false;
                    } else {
                        //Stays alive
                        newActive[i][j] = true;
                        newAlive[i][j] = true;
                        drawEntity(i, j);
                    }
                } else if (neighbors[i][j] == 0){
                    //No longer active
                    newActive[i][j] = false;
                    newAlive[i][j] = false;
                } else {
                    //Death; still active
                    newActive[i][j] = true;
                    newAlive[i][j] = false;
                }
            }
        }
    }
    //Historical oopsies time!
    noise();
    if (!paused) {
        for (int k = 0; k < rows; k++) {
            active[k] = Arrays.copyOf(newActive[k], newActive[k].length);
            alive[k] = Arrays.copyOf(newAlive[k], newAlive[k].length);
        }
        newActive = new boolean[rows][cols];
        newAlive = new boolean[rows][cols];

        fullCheck();
    }
}
```

```
/**
 * Draws a given building/cell to the screen.
 * @param i the row of a building/cell
 * @param j the col of a building/cell
 */
public void drawEntity(int i, int j) {
    pushMatrix();
    translate(i * scalar - (rows * scalar / 2),
            terrain(i, j),
            j * scalar - (cols * scalar / 2));
    noStroke();
    if (rulers[i][j] > 0) {
        rotateY(rotation[i][j]);
        shapeMode(CORNER);
        shape(buildings[rulers[i][j] - 1][neighbors[i][j]]);
        shapeMode(CENTER);
    }
    popMatrix();
    pushMatrix();
    translate(scalar / 2, 5, 0);
    //shape(road[i][j]);
    popMatrix();
}

/**
 * Ran when a cell is born.
 *
 * @param i
 * @param j
 */
public void newLife(int i, int j) {
    //Top Left
    if (i > 0 && j > 0) {
        newActive[i - 1][j - 1] = true;
    } else if (j > 0){
        newActive[rows - 1][j - 1] = true;
    }
    //Left
    if (i > 0) {
        newActive[i - 1][j] = true;
    } else {
        newActive[rows - 1][j] = true;
    }
    //Bottom Left
    if (i > 0 && j < cols - 1) {
        newActive[i - 1][j + 1] = true;
    } else if (j < cols - 1){
        newActive[rows - 1][j + 1] = true;
    }
    //Top
    if (j > 0) {
        newActive[i][j - 1] = true;
    }
```

```
        //Bottom
        if (j < cols - 1) {
            newActive[i][j + 1] = true;
        }
        //Top Right
        if (i < rows - 1 && j > 0) {
            newActive[i + 1][j - 1] = true;
        } else if (j > 0) {
            newActive[0][j - 1] = true;
        }
        if (i < rows - 1) {
            newActive[i + 1][j] = true;
        } else {
            newActive[0][j] = true;
        }
        if (i < rows - 1 && j < cols - 1) {
            newActive[i + 1][j + 1] = true;
        } else if (j < cols - 1) {
            newActive[0][j + 1] = true;
        }
    }

    /**
     * Runs neighborCheck() for each cell within the Cellular Automata.
     */
    public void fullCheck() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (active[i][j]) {
                    neighborCheck(i, j);
                }
            }
        }
    }

    /**
     * Checks for the number of neighbors and the ruler of a cell.
     * @param i the row of the cell
     * @param j the col of the cell
     */
    public void neighborCheck(int i, int j) {
        neighbors[i][j] = 0;
        int[] ruler = {0, 0, 0, 0, 0};
        //top left
        if (i > 0 && j > 0 && alive[i - 1][j - 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[i - 1][j - 1]] += 1;
        } else if (i == 0 && j > 0 && alive[rows - 1][j - 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[rows - 1][j - 1]] += 1;
        }
        //left
        if (i > 0 && alive[i - 1][j]) {
```

```
            neighbors[i][j] += 1;
            ruler[rulers[i - 1][j]] += 1;
        } else if (i == 0 && alive[rows - 1][j]) {
            neighbors[i][j] += 1;
            ruler[rulers[rows - 1][j]] += 1;
        }
        //bottom left
        if (i > 0 && j < cols - 1 && alive[i - 1][j + 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[i - 1][j + 1]] += 1;
        } else if (i == 0 && j < cols - 1 && alive[rows - 1][j + 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[rows - 1][j + 1]] += 1;
        }

        //top
        if (j > 0 && alive[i][j - 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[i][j - 1]] += 1;
        }
        //bottom
        if (j < cols - 1 && alive[i][j + 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[i][j + 1]] += 1;
        }

        //top right
        if (i < rows - 1 && j > 0 && alive[i + 1][j - 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[i + 1][j - 1]] += 1;
        } else if (i == rows - 1 && j > 0 && alive[0][j - 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[0][j - 1]] += 1;
        }
        //right
        if (i < rows - 1 && alive[i + 1][j]) {
            neighbors[i][j] += 1;
            ruler[rulers[i + 1][j]] += 1;
        } else if (i == rows - 1 && alive[0][j]) {
            neighbors[i][j] += 1;
            ruler[rulers[0][j]] += 1;
        }
        //bottom right
        if (i < rows - 1 && j < cols - 1 && alive[i + 1][j + 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[i + 1][j + 1]] += 1;
        } else if (i == rows - 1 && j < cols - 1 && alive[0][j + 1]) {
            neighbors[i][j] += 1;
            ruler[rulers[0][j + 1]] += 1;
        }
        int max = 0;
        int maxIndex = 0;
        int maxCount = 2;
```

```
        for (byte k = 1; k < 5; k++){
            if (ruler[k] > max) {
                max = ruler[k];
                maxIndex = k;
                maxCount = 1;
            } else if (ruler[k] == max) {
                maxCount++;
            }
        }
        if (maxCount == 1) {
            rulers[i][j] = maxIndex;
            last[i][j] = maxIndex;
        } else {
            rulers[i][j] = 0;
        }
    }

    /**
     * Used for interactive navigation purposes.
     */
    public void mousePressed(){
        nav.mousePressed();
    }

    /**
     * Used for interactive navigation purposes.
     */
    public void mouseWheel(MouseEvent event) {
        nav.mouseWheel(event);
    }

    /**
     * Used for interactive navigation purposes.
     */
    public void mouseDragged(){
        nav.mouseDragged();
    }

    /**
     * The main driver of the program; creates processing sketch and runs it.
     * @param args not used.
     */
    public static void main(String[] args){
        String[] processingArgs = {"MySketch"};
        MultiAutomata3D sketch = new MultiAutomata3D();
        PApplet.runSketch(processingArgs, sketch);
    }
}
```