Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Cellular Automata

Introduction

We illustrate two-dimensional arrays, and enumerations over them, using the examples of Cellular Automata and the Game of Life.

Cellular Automata model the Universe as a rectangular grid of *cells*, each in a given *state*. Time progresses in discrete steps. On each clock tick, each cell simultaneously decides what state to enter based on its current state and the current states of its neighbors. Each cell makes its decision independently, but all cells follow the same rules.

The Game of Life is a particular Cellular Automaton that models birth and death.

Systematic top-down development of an entire program is illustrated. Deeply-nested **for**-statements in the code arise naturally as a consequence of stepwise refinement, but are readily understood.

Top-level Code Structure

Class `CellularAutomaton` models the notion of a Cellular Automaton.

```
/* A cellular automaton. */
class CellularAutomaton {
   } /* CellularAutomaton */
```

☞ **Program top-down, outside-in.**

The simulation is implemented as method `main`.

```
/* A cellular automaton. */
class CellularAutomaton {
   /* Simulate a cellular automaton. */
   static void main() {
      } /* main */
   } /* CellularAutomaton */
```

☞ **Program top-down, outside-in.**

Adopt the general-iteration pattern that first initializes, and then iteratively updates.

```
/* A cellular automaton. */
class CellularAutomaton {
   /* Simulate a cellular automaton. */
   static void main() {
      /* Create the initial Universe and display it. */
      /* Simulate and display remaining generations. */
      } /* main */
   } /* CellularAutomaton */
```

☞ **Master stylized code patterns, and use them.**

☞ **Many short procedures are better than large blocks of code.**

Opt to simulate a finite number of generations.

```
/* A cellular automaton. */
class CellularAutomaton {
   /* Simulate a cellular automaton. */
   static void main() {
      /* Create the initial Universe and display it. */
         Initialize();
         Display();
      /* Simulate and display remaining generations. */
         for (generation=1; generation<=LAST_GEN; generation++) {
            NextGeneration();
            Display();
            }
      } /* main */
   } /* CellularAutomaton */
```

☞ **Don't type if you can avoid it; clone. Cut and paste, then adapt.**

Create stubs for methods on autopilot.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Create the initial Universe. */
   static void Initialize() { } /* Initialize */
   /* Display the Universe. */
   static void Display() { } /* Display */
   /* Update Universe to be the next generation. */
   static void NextGeneration() { } /* NextGeneration */
   ...
   } /* CellularAutomaton */
```

☞ **Defer challenging code for later; do the easy parts first.**

Declare and specify the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 6;                    // Height of Universe.
   static final int N = 20;                   // Width of Universe.
   static int old[][] = new int[M][N];        // old Universe.
   static int next[][] = new int[M][N];       // next Universe.
   static final int LAST_GEN = 50;            // Last generation.
   static int generation;                     // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Minimize use of literal numerals in code; define and use symbolic constants.**

Declare and specify the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 6;                    // Height of Universe.
   static final int N = 20;                   // Width of Universe.
   static int old[][] = new int[M][N];        // old Universe.
   static int next[][] = new int[M][N];       // next Universe.
   static final int LAST_GEN = 50;            // Last generation.
   static int generation;                     // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Introduce program variables whose values describe "state".**

Output is displayed by a standard row-major-order traversal, with newlines at row ends.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++) System.out.print( old[r][c] + " " );
      System.out.println();
      }
   } /* Display */
```

☞ **Master stylized code patterns, and use them.**

Instance of the standard compute-use pattern.

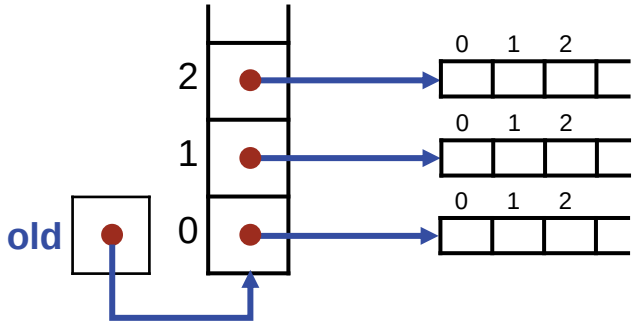
```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as F(old[][] states). */
   /* Swap old[][] and next[][] Universes. */
   } /* NextGeneration */
```

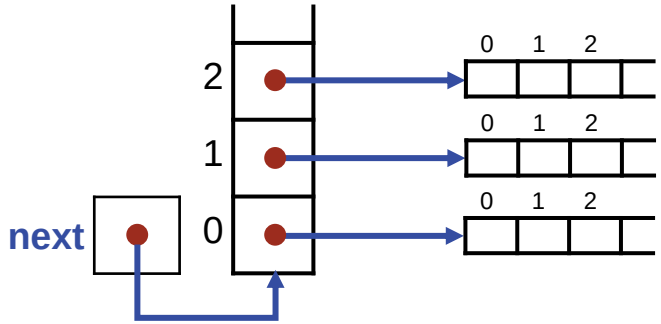
☞ **Master stylized code patterns, and use them.**

☞ Master stylized code patterns, and use them.

Standard row-major-order traversal for determining new states of each cell of next.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            /* next[r][c] = F( old[r][c] and its neighbors ); */
   /* Swap old[][] and next[][] Universes. */
   } /* NextGeneration */
```

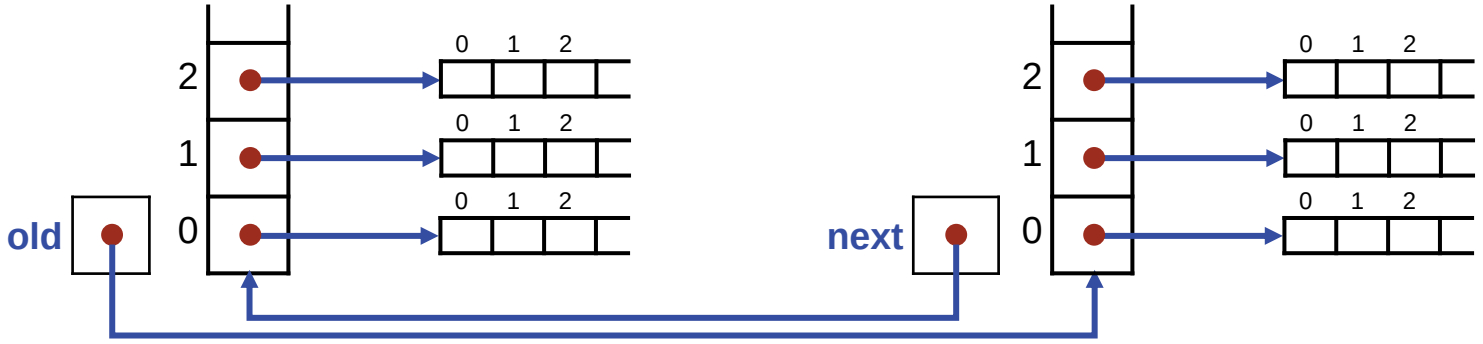




☞ Master stylized code patterns, and use them.

Standard code for swap, albeit whole Universes `old` and `next` are swapped in constant time.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            /* next[r][c] = F( old[r][c] and its neighbors ); */
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old=next; next=temp;
   } /* NextGeneration */
```



Game of Life. A Cellular Automaton in which each cell is either dead or alive.

In each generation:

- Each live cell with 2 or 3 live neighbors lives on to the next generation (life) otherwise it dies (death).
- Each dead cell with 3 live neighbors comes alive in the next generation (birth) otherwise it remains dead.

Each cell is either dead or alive, so specialize the Universes as Boolean 2-D arrays.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 6;                          // Height of Universe.
   static final int N = 20;                         // Width of Universe.
   static boolean old[][] = new boolean [M][N];  // cell true iff alive.
   static boolean next[][] = new boolean [M][N]; // cell true iff alive.
   static final int LAST_GEN = 50;                  // Last generation.
   static int generation;                           // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Choose representations that by design don't have nonsensical configurations.**

Specialize the output by compactly rendering dead as “_” and alive as “X“.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++)
         if ( old[r][c] ) System.out.print( "X" );
         else System.out.print( "_" );
      System.out.println();
      }
   } /* Display */
```

Specialize the cell update as an instance of the standard compute-use pattern.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
            /* Set next[r][c] according to the birth and death rules. */
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old=next; next=temp;
   } /* NextGeneration */
```

☞ **Master stylized code patterns, and use them.**

Game of Life

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( (dr!=0||dc!=0) && old[r+dr][c+dc] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old=next; next=temp;
   } /* NextGeneration */
```

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( (dr!=0||dc!=0) && old[r+dr][c+dc] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
               if ( old[r][c] ) /* Currently live. */
                  if ( liveNeighbors==2 || liveNeighbors==3 )
                     next[r][c] = true;
                  else next[r][c] = false;
               else /* Currently dead. */
                  if ( liveNeighbors==3 ) next[r][c] = true;
                  else next[r][c] = false;
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old=next; next=temp;
   } /* NextGeneration */
```

Game of Life

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( (dr!=0||dc!=0) && old[r+dr][c+dc] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
               if ( old[r][c] ) /* Currently live. */
                  if ( liveNeighbors==2 || liveNeighbors==3 )
                     next[r][c] = true;
                  else next[r][c] = false;
               else /* Currently dead. */
                  if ( liveNeighbors==3 ) next[r][c] = true;
                  else next[r][c] = false;
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old=next; next=temp;
   } /* NextGeneration */
```

To prevent the subscripts going out of bounds

Game of Life

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine each state of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( (dr!=0||dc!=0) && old[(r+dr)%M][(c+dc)%N] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
               if ( old[r][c] ) /* Currently live. */
                  if ( liveNeighbors==2 || liveNeighbors==3 )
                     next[r][c] = true;
                  else next[r][c] = false;
               else /* Currently dead. */
                  if ( liveNeighbors==3 ) next[r][c] = true;
                  else next[r][c] = false;
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old=next; next=temp;
   } /* NextGeneration */
```

To prevent the subscripts going out of bounds, simulate on a torus.

Create some life for gliding diagonally down and to the right …

```
/* Establish original configuration in old. */
static void Initialize() {
   /* Glider */
      old[0][1] = old[1][2] = old[2][3] = old[2][1] = old[2][2] = true;
   } /* Initialize */
```

old

	0	1	2	…
0		T		…
1			T	…
2	T	T	T	…
	…	…	…	…

… and let it rip.

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
____________________
____________________
```

Game of Life

… and let it rip.

```
Generation: 1
____________________
X_X_________________
_XX_________________
_X__________________
____________________
____________________
```

Game of Life

… and let it rip.

```
Generation: 2
____________________
__X_________________
X_X_________________
_XX_________________
____________________
____________________
```

Game of Life

… and let it rip.

```
Generation: 3
_____________________
_X___________________
__XX_________________
_XX__________________
_____________________
_____________________
```

… and let it rip.

```
Generation: 4
____________________
__X_________________
___X________________
_XXX________________
____________________
____________________
```

Back to the same configuration as Generation 0, but shifted down and one cell to the right.

Render output superimposed, like a movie, and slow it down.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "\u000CGeneration: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++)
         if ( old[r][c] ) System.out.print( "X" );
         else System.out.print( "_" );
      System.out.println();
      }
   /* Sleep for 300 milliseconds. */
      try { Thread.sleep(300); } catch (InterruptedException ie) { }
   } /* Display */
```

☞ **Ignore fussy details for as long as possible.**