CSc 227 — Program Design and Development
Spring 2014 (McCann)

http://www.cs.arizona.edu/classes/cs227/spring14/

# Program #5: Demons of Cyclic Space

*Due Date: Tuesday, March 4th, 2014, at 9:00 p.m. MST*

**Overview (Part I):** The magazine Scientific American used to run a monthly column called "Computer Recreations," written for several years by A. K. Dewdney, now Professor Emeritus of Computer Science at the University of Waterloo. The August 1989 column discusses and provides a rough algorithm for a two-dimensional cellular automaton (2D CA) Dewdney named "cyclic space." As older back-issues of Scientific American aren't available on-line, I've acquired scans of the article's four pages and have compiled them into a PDF document named `dewdney0889.pdf` available from the class web page in a password-protected area. See Piazza for the username and password. Because the bulk of the implementation detail is on the article's third page (p. 104), I've attached it to this handout. You are encouraged to read the complete article to better understand the ideas.

**Overview (Part II):** The textbook "Introduction to Programming in Java" by Robert Sedgewick and Kevin Wayne uses a library of classes called `stdlib` to support their example programs. Part of the library is a `Picture` class that was designed for doing simple image manipulation. We'll be using it to display a visualization of cyclic space. See the "Want to Learn More?" section, below, for the link to the `stdlib` documentation.

**Assignment:** You will create two Java classes, each in its own `.java` file. `Prog5` is the application; its job is to create the automaton (2D CA) object, make it advance through time, and display the result of each advance to the screen. This is the class that will deal with the graphics. The second class, `Demon`, manages the automaton and its functionality, implementing the logic described in Dewdney's column.

Before you can implement `Demon`, you'll need to know how it works. Start by reading at least the attached page, preferably the entire column. As you read, keep in mind that this was published in 1989, the year the Intel 486 CPU was introduced and three years before Microsoft released Windows 3.1, the first successful version of Windows. Technology advances rapidly in this business.

Dewdney's audience consisted mainly of amateur computer programmers who were using the language BASIC, so he doesn't put his description in object-oriented terms. We expect that you will, and here's how we want you to do it.

`Demon` has just one constructor:

- `public Demon (int, int, int)` — The first two parameters are the height and width of the universe, in cells, respectively. The third specifies the number of states; that is, each cell of the universe will hold a value drawn from the domain $[0 \ldots \text{states} - 1]$. With this information, the constructor builds two 2D arrays of `byte`s, one representing the current version of the universe, and the other the next version. Also, the constructor sees to it that the initial current version is populated with randomly generated `byte` values from the domain of possible state values (see `populate ()`, below).

For our needs, this small set of `Demon` methods is sufficient:

- `private void populate ()` — Called by the constructor to fill the current version of the universe with the random state values. The `stdlib` class `StdRandom` has a static method `uniform` that can be used to generate (pseudo-)random numbers, or you can use the generator provided in Java's `Math` class.
- `public byte[][] getUniverse ()` — The getter; it returns a copy of the current universe to the caller.
- `public void setUniverse (byte[][])` — The setter; it replaces the current universe's content.

- `public void advance ()` — This uses the content of the current universe to determine the content of the next universe. The details are spelled out in Dewdney's column, mostly on page 104. Note that Dewdney left out a key detail, probably on purpose: If a cell isn't "eaten," its value remains unchanged.

  To help you with your debugging, here's a sample 3x3 universe consisting of just 8 states (that is, each cell has a value from 0 through 7). If you're applying Dewdney's rules correctly, you should get the following sequence of universes:

  ```
            0  1  2             1  2  3             2  3  4
            7  4  3             0  5  4             1  6  5
            7  5  6             0  6  7             1  7  0

        Initial Content     After 1st Pass     After 2nd Pass
  ```

  Note that this is a contrived example; in general, not every cell will change from one version of the universe to the next.

  Also note that we're treating the universe as if it were wrapped around a torus (think 'donut' . . . mmmmm, donuts . . . ). For example, in the first universe above, 2's right neighbor is 0 and its top neighbor is 6. As Dewdney explains, this can be achieved by using modular arithmetic (that is, Java's % operator).

Now, what about the `Prog5` class? Start by carefully examining the `Pattern.java` example, thinking about the representation of the board. It's a 15x15 board, but we're not using just 225 pixels to display it. Each square is represented by a large rectangle. You are to follow this same idea: Here, each cell of the universe is to be displayed using a 3x3 rectangle. You'll have to do some calculations to determine where to place each cell's rectangle. With `Pattern.java` as your guide, it shouldn't be hard to figure out.

To keep things manageable, you are to use a 100x100 universe. Limit your program so that it asks the `Demon` object to generate only 250 new versions of the universe. As for the number of available states, use 13. Why 13? That matches the pre-defined color names in Java's `Color` class. You can assign the colors to states in any order you wish. In fact, if you'd like to, you can even define your own colors, but you'd have to dig into Java a bit to find out how to do this. We recommend that you stick to the pre-defined colors, at least until you get everything working correctly.

After you have the `Picture` object created, you can instantiate the `Demon` object and start generating new versions of the universe to be displayed. After each new version has been displayed, use Java's `TimeUnit.MILLISECONDS.sleep()` method to wait a fifth of a second (200 milliseconds) before moving on to the next version. This will keep the program moving slowly enough that the user can watch the show.

Keep this in mind: `Demon` objects don't care about graphics. To debug `Demon`, you can just print the cells' state values to the text output window, similar to the small example above. Using a smaller universe than 100 x 100 for initial testing is a good idea, too.

**Data:** You will be "hard-coding" (defining constants for) all of the necessary information (universe size, number of states, etc.). The user isn't expected to provide any information to the program.

**Output:** The major product of your program is the display of the sequence of versions of the automaton's universe. In addition, we want you to display to the text window (using `System.out.println()`) the version number of the universe that was just displayed. When you finish displaying a universe, print the line "Just displayed Universe Version #", where # = 0 for the initial random universe.

**Turn In:** Use the 'turnin' page to electronically submit your `Prog5.java` and `Demon.java` files to the `cs227p05` directory any time before the stated due date and time.

## Want to Learn More?

- The Wikipedia page on cellular automata is a good starting point:
  `http://en.wikipedia.org/wiki/Cellular_automata`

- The `stdlib` page has links to the documentation, plus much more:
  `http://introcs.cs.princeton.edu/java/stdlib/`
- Working at home but connecting remotely to `lectura`? To see the graphics, try using one of the Remote Access machines:
  `http://faq.cs.arizona.edu/index.php?action=artikel&cat=4&id=15`

**Hints, Reminders, and Other Requirements:**

- To use `TimeUnit`, import `java.util.concurrent.TimeUnit`.
- Your program needs just one `Picture` object. Update its content with each new version of the universe.
- **START EARLY!** This assignment asks you to deal with 2D arrays, wrap-around neighborhoods, and graphics, all for the first time this semester and all in the same assignment. You should expect to encounter several problems that will require time to think through. Remember, section leaders don't have lab hours between Thursday night and Monday morning.
- As usual, we'll be looking for good documentation, indentation, variable names, etc. Please allocate enough time to do a good job with these. We'll be looking for all the usual stuff: block comments, appropriate naming, consistent indentation, etc.
- Even a tiny error in your implementation of the cyclic space algorithm can cause your visualization to be incorrect. Pay close attention to detail. If it doesn't behave like the demo I showed in class, it's not correct.
- Having trouble installing `stdlib.jar` at home? Remember that you can still write and test `Demon` without it. You can then come to the lab to work on the graphics.

---

**Using the `stdlib.jar` file:** Collections of related Java classes are often packaged into `.jar` (Java ARchive) files for easy distribution and inclusion into programs. So that you don't need to do a lot of messing around with this if you are using lab workstations, we've installed the file `stdlib.jar` for you. What you need to do is tell `javac` and `java` where to find it.

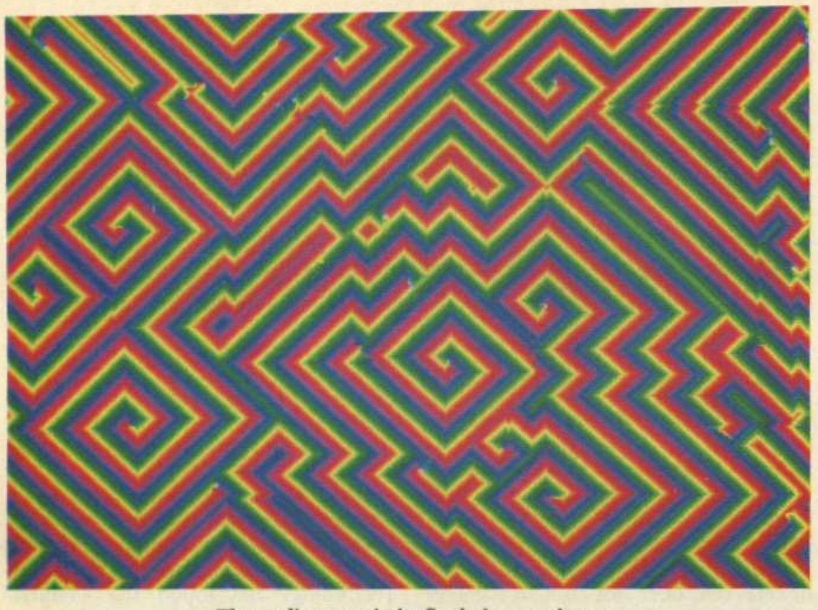
*Option 1: Compiling from the Command Line:* Java makes use of an environment variable called `CLASSPATH` to find Java packages. Your `CLASSPATH` lists names of directories to be searched for packages, but it doesn't include the path to `stdlib.jar`. You could monkey around with your environment to add the correct path, or you can do it the easy way, by providing the path when you compile and run. Here's how to do that with the `Pattern.java` example:

```
$ javac -cp .:/home/cs227/spring14/stdlib.jar Pattern.java
$ java  -cp .:/home/cs227/spring14/stdlib.jar Pattern
```

The 'dot colon' is important! Note that opening a `Picture` object won't work if you simply SSHing into lectura from home, unless you're running X11 at home and using X11 forwarding on your SSH login, which you probably aren't. Instead, you can use one of the Remote Access servers, if you have a fast internet connection.

*Option 2: Compiling in Eclipse:* If you know about the Eclipse IDE, you may be planning to use it on this assignment. If so, here's how to get Eclipse to associate the `stdlib.jar` file with your project.

1. If you're not in the lab, get the `stdlib.jar` file from the `stdlib` page. Remember where you stored it! (If you're in the lab, you can use the copy already installed in `/home/cs227/spring14`.)
2. Create an Eclipse project for this assignment.
3. Right-click on the project in the Package or Project Explorer, move the cursor over the Build Path entry until another pop-up menu appears, then click "Add External Archives . . . ". You should see a "JAR Selection" window. Use it to find the `stdlib.jar` file you downloaded, and click "Open."
4. Compile and run your project as usual. You should only need to perform steps 1-3 once each.


*The cyclic space in its final, demon phase*

and *old* to hold the current and previous states of the cells in cyclic space.

```
repeat until key is pressed
for $i \leftarrow 1$ to 100
    for $j \leftarrow 1$ to 100
        for each neighbor $(k,l)$ of $(i,j)$
            if $old(k,l) = old(i,j)+1$
                then $new(i,j) \leftarrow old(k,l)$
for $i \leftarrow 1$ to 100
    for $j \leftarrow 1$ to 100
        display $new(i,j)$
        $old(i,j) \leftarrow new(i,j)$
```

The outer loop, which ends when the user of DEMON presses a certain key, controls repetition of the main checking and displaying cycles. (Of course, there are other ways to construct such a loop.) Readers who have computers with small memories are advised to limit themselves to a smaller array of cells, say one that is 50 by 50. The two inner loops used here presuppose a 100-by-100 cellular array. The cell $(i,j)$ sits at the intersection of the $i$th row and $j$th column.

The innermost loop of DEMON simply checks the state of each of the four cells adjacent to cell $(i,j)$. (Readers must therefore include instructions in their versions of the algorithm that assign the values $i-1$ and $i+1$ to the index $k$ while $l$ equals $j$. Similarly, the values $j-1$ and $j+1$ must be assigned to $l$ while $k$ equals $i$.) If the state number of a neighboring cell happens to be 1 more than the state number of the cell at $(i,j)$, then the cell at $(i,j)$ is eaten: it has its state number changed to that of the neighboring cell.

DEMON must perform modular arithmetic when it calculates the value of $old(i,j)+1$. In other words, if $old(i,j)$ happens to be $n-1$, the highest state number, then $old(i,j)+1$ is equal to 0. Hence, if one has specified, say, 10 states in the cellular space, the state numbers will be 0, 1, 2..., 9 and $9+1=0$. If the value of a cell changes, the new value is assigned to $new(i,j)$.

The double loop that follows the checking loop displays all the cells in *new* and then updates the *old* array by replacing all its values by the corresponding values in *new*. Readers can consult previous columns, including last month's, to figure out how to display the cells on the computer screen.

Modular arithmetic must be applied not only with cell-state numbers but also with the indexes $i$ and $j$ themselves. The simplest way to achieve the illusion of an infinite space is to endow one's screen with the "wraparound" property. Cells on the extreme right-hand margin are considered to be adjacent to those on the left, and cells at the bottom of the display are effectively adjacent to those at the top. The effect is created by using index values from 0 to 99 instead of 1 to 100. The cell to the immediate right of (23,99) is in fact (23,0). Hence, the numbers $i-1$, $i+1$, $j-1$ and $j+1$, which are index values of the neighbors of cell $(i,j)$, must all be expressed in modular form. Most programming languages have instructions that do that automatically.

Naturally, DEMON must allow a user to "initialize" the cellular space under its control. This step can be done by including a provision in the algorithm for eliciting the desired number of states from the user, as well as a loop that gives every cell in the space an initial, randomly chosen state within the allowed range of numbers.

What number of states works best? It all depends on how long the reader is willing to wait for the four phases to succeed one another. When the number of states is very large, say more than 25, a 100-by-100 random cellular array is likely to remain fixed forever. On the other hand, when the number of states is small, the stages succeed each other too quickly to be appreciated. Griffeath recommends between 12 and 16 states.

Some months ago I mentioned that I would visit the University of Wisconsin to report on the activities of the so-called particle mafia, of which Griffeath is a member. (The name comes from an early association of the general field of particle systems with Frank Spitzer, a mathematician at Cornell University in upstate New York, where—according to Hollywood legend—the Mafia used to hang out.) Spitzer's articles and talks on particle systems first popularized the subject in North America, just as the work of his colleague R. L. Dobrushin did in the Soviet Union. Several of Spitzer's students at Cornell as well as his followers elsewhere took up the work of controlling the particle-systems numbers game. They include not only Griffeath but also Maury Bramson (also at Madison), Richard Durrett of Cornell and Thomas Liggett of the University of California at Los Angeles.

What exactly is a particle system? It is usually a cellular automaton in which only one cell changes at a time, often in a random fashion. The diffusion-limited aggregation algorithm described in last December's column is an example of a particle system. There, a single particle wanders randomly across a grid of cells until it encounters a growing aggregate. Its position is then frozen as another particle begins to wander. In time, the aggregate invariably develops a treelike shape.

A number of important but difficult problems are raised by particle systems, not the least of which are questions concerning the long-term stability of certain phases that arise when the systems are set running. Although cyclic space is not a traditional particle system, Griffeath thinks it nonetheless provides a model for locally periodic wave formations that, because