

CS 205 - Programming for the Sciences

Spring 2008 - In-class Exercise for 02/21/08 and 02/26/08

Today's exercise is to introduce and work with two-dimensional arrays to model cellular automata.

Use a Web browser go to the course webpage <http://csserver.evansville.edu/~hwang/s08-courses/cs205.html>. Under today's date, save the compressed folder GameOfLifeInClass.zip. Extract the solution folder. Double-click into the folder GameOfLifeInClass, then double-click on GameOfLifeInClass.sln (the Visual Studio solution file). This will launch Visual Studio with the solution loaded.

The GUI design for the program has been completed. To see it, right-click on Form1.cs in the Solution explorer window and select View Designer. We spend the next couple of days completing the program.

Notes on Cellular Automata and Conway's Game of Life

A cellular automaton is used in discrete modeling. It consists of a regular grid of *cells*, each of which can be in one of a finite number of *states*. Time is discrete, so the state of cell at time t is a function of the states of a finite number of *neighboring* cells at time $t-1$. This function (also called a *rule*) is applied every cell at each time step and all updates happen at once creating a new *generation*.

Conway's Game of Life is one well-known example of a two-state, two-dimensional cellular automaton. The cells are square, and the neighborhood is the 8 cells directly above, below, to the right, to the left, diagonally above to the left and right, and diagonally below to the left and right. Each cell may be live (represented by a ***) or dead (represented by a blank). The original rule posed by Conway is:

- If a live cell has fewer than 2 live neighbors, the cell dies of loneliness
- If a live cell has more than 3 live neighbors, the cell dies of overcrowding
- If a dead cell has exactly 3 live neighbors, the cell becomes alive by birth
- Otherwise, the cell state is unchanged

The starting state of a cellular automaton is called a *configuration*, e.g. as shown in Figure 1. Figure 2 shows the number of live neighbors in each cell from the state in Figure 1. Figure 3 shows the next generation.

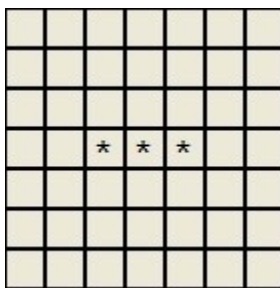


Figure 1

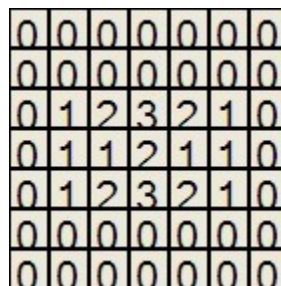


Figure 2

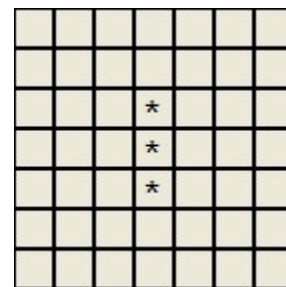


Figure 3

If you are interested in knowing more about cellular automata and Conway's Game of Life, see the short list of on-line references at the end of this assignment. These give a good overview and links to more in-depth references.

Notes on two-dimensional arrays in C#

A two-dimensional (2D) array is a data structure that is collection of elements of the same type that are accessed by giving an index for each of two dimensions. Conceptually, they may be thought of as similar to matrices.

The declaration of 2D arrays in C# has the following syntax:

```
<element type> [,] <name>;
```

For example, the Game of Life program, the cell grid is implemented as a 2D array of Label objects, so the declaration for the grid variable is:

```
Label [,] grid;
```

As with one dimensional arrays, this is only a variable that references a 2D array object. We still have to create the object itself, which is done using the following syntax:

```
<2D array name> = new <element type> [<# rows>, <# columns>;
```

So for the Game of Life, if we want a 10x20 grid, we would say:

```
grid = new Label [10, 20];
```

In order to implement the Game of Life, we also need a backing grid that will store the results of applying the rule to the grid for the next generation, since we must change all of the cells simultaneously each generation. This grid only needs to be a 2D array of strings, so we can declared and create the grid using:

```
string [,] nextGrid;  
nextGrid = new nextGrid[10,20];
```

To access an individual element of a 2D array, we need to give an index for each dimension. Like one-dimensional arrays, the indexes start at 0 for each dimension. If we want to access the element at row index *i* and column index *j* in the 2D array **grid**, then we would write **grid[i , j]**. Since each element of **grid** is a Label, we can access properties of this label using the dot notation as before. In particular, we will be accessing and setting the Text property of each Label. E.g., if we want to set the Text property of the element at row index *i* and column index *j* of **grid** to **"*"**, we would write:

```
grid[i, j].Text = "*";
```

Since we often want to do something with each element of a 2D array, it is very common to see the following nested loop pattern for a 2D array. Unlike a one-dimensional array, a 2D array does not have a Length property. Instead, it has a method GetUpperBound(x) where x is the dimension. Like indexing, dimensions start at 0, so 0 is the row dimension and 1 is the column dimension. Unlike the Length property of one-dimensional arrays, this method returns the highest index of the dimension, which is one

less than the total number in that dimension. For example, if we have 2D array **grid**, then we often see code like:

```
int i, j,
    numRows = grid.GetUpperBound(0),
    numColumns = grid.GetUpperBound(1);
for (i = 0; i <= numRows; i++)
    for (j = 0; j <= numColumns; j++)
    {
        // Do something with grid[i, j]
    }
```

Assignment (10 points each day)

For today's and next Tuesday's assignment we will complete the Game of Life project. You are given the code that creates the grid of Labels. (It is encapsulated in the **MakeGrid** method so that it may be used in other handlers besides when the Form loads. E.g., if you wanted to be able to resize the grid during execution.) The following Form1 properties were set in the Designer: `AutoSize` to true, `AutoSizeMode` to `GrowAndShrink`, and the `Padding` to 20,20,20,20. This is so that after the grid labels are added, the form resizes to fit around the grid and leaves space around it and the other controls.

Here are the steps we will do:

- Write a handler for the Click event on a Label that will toggle the Text property of the Label between a * and a blank. Unlike our previous handlers, this one is not attached to one of the controls we placed with the Designer. Also, it will be shared by all of the Labels making up the grid.
- Write a handler for the Clear button that will reset all of the grid Labels to blank.
- Write a method **CountNeighbors** to count the number of live neighbors of a grid cell at row *i* and column *j*.
- Declare and create a 2D array of strings **nextGrid** for use as a backing grid.
- Write a method **ComputeNextGeneration** to fill the backing grid with the state of the next generation. This method will use **CountNeighbors**.
- Write a method **DisplayNextGeneration** to copy the backing grid state into the grid of Labels for display.
- Finally, write a handler for the Next button. This handler will loop computing the next generation and displaying for as many times as input by the user.

References

- "Cellular automaton" from Wikipedia, URL: http://en.wikipedia.org/wiki/Cellular_automaton
- "Conway's Game of Life" from Wikipedia, URL: http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- "Cellular Automaton" from Wolfram Mathworld, URL: <http://mathworld.wolfram.com/CellularAutomaton.html>
- "Life" from Wolfram Mathworld, URL: <http://mathworld.wolfram.com/Life.html>