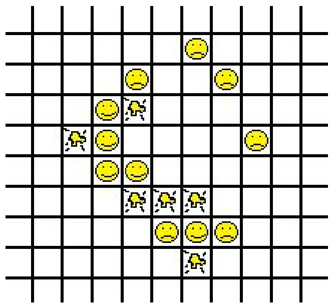
Game of Life

This document will show you how to create a graphical version of the “Game of Life” as outlined in the project description.



Classes/Structs

Form1
Generation (Represents one iteration of the game)
Cell (Represents one square on the board)

The Cell struct

The Cell represents one square on the board. The cell needs to know if it has a life form in it or not (a bool). If you want to draw the states of the lifeforms before changing them, you’ll need another variable, an int, to store the state (what’s going to happen to that life form).

Note that is a **struct** and not a class. This will make cloning boards easier because structs are *value* types.

```
struct Cell
{
    public bool hasOrganism;      //does this square have a life form?
    public int state;             //what will happen to this life form (if present)?

    public const int EMPTY=0;
    public const int SURVIVES=1;
    public const int DEATH_BY_OVERCROWDING=2;
    public const int DEATH_BY_LONELINESS=3;
    public const int SPAWNING=4;
    public const int UNMARKED = 5;

    public Cell(bool hasOrganism)
    {
        this.hasOrganism = hasOrganism;
        this.state = EMPTY;
    }
}
```

The Generation Class

The generation class wraps the board (a 2d array of Cell structs). It has public functions to draw the board and mark the board. Marking the board means examining each cell, counting its neighbors, and setting the *state* variable to indicate what will happen to that cell at the end of the current generation.

Hint: Remember, since this class will be drawing images, you'll need to add *using System.Drawing* to the top of your file.

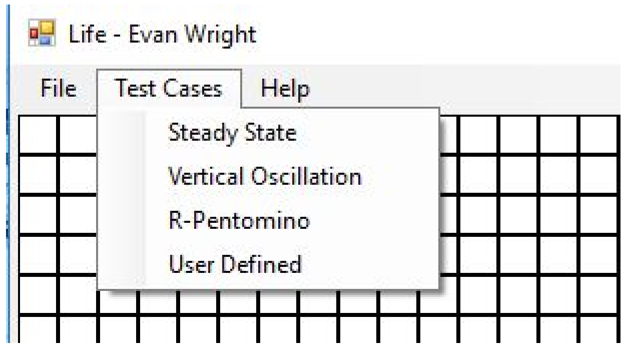
Generation	
variables/constants	
const int BOARD_WIDTH=60;	How many Cells wide the board is (you can make this whatever you want)
const int BOARD_HEIGHT=40;	How many Cells tall the board is (you can make this whatever you want)
const int CELL_SIZE = 20;	How big a cell is in pixels. The Draw function will need this for computing the coordinate to draw the tiles at.
Cell[,] board	A 2d array of Cell structs. Allocate this using the *new* operator. To see how to allocate this array, see the FloodIt code.
functions	
void Mark()	Loop over all the rows and columns in the board and call MarkCell() on each of them. The FloodIt game contains numerous examples of nested loops.
void Draw(Graphics g, bool showInMarkedState)	Loop over all the rows and columns (i,j) using a nested loop. Look at the *state* flag in cell i,j and draw the appropriate icon. If the *showInMarkedState* is false, just draw the generic organism icon. If *showInMarkedState* is true, draw the icon that represents what is going to happen to that organism. You can either build a lookup table using an array or Dictionary or use a series of *if/else if* statements. **Hint:** This will be very similar to the draw code in SlidingPuzzle.
bool IsExtinct()	This function is called by Form1 to check if the game is

	over. return *true* if any square on the board has an organism in it. Otherwise, return false. Use nested loops to do this.
void MarkCell(int x, int y)	This function sets the *state* flag on a Cell based on whether it is going to live or die (so the appropriate icon can be drawn) Count up the number of organisms in the surrounding 8 cells using CountOrganism(x,y) passing in the coordinates of the squares surrounding x,y. Based on that number, set the *state* flag in the cell to the appropriate constant (defined in the Cell struct) base on the rules of the game outlined in the instructions. **Hint:** This is similar to FloodIt, where each square calls Fill() on its neighbors.
int CountOrganism(int x, int y)	If x or y is outside the 2d array bounds, return 0. Otherwise if the Cell at x,y has a lifeform return 1, otherwise return 0.
Generation Update()	This function is where the *state* flag is applied to produce the next iteration of the game. Create a new Generation object (nextGen) Loop over all the rows and columns (i,j) of the new board using nested loops. Set the *marked* flag at i,j to false. Set the *state* flags of the cell at i,j to EMPTY If the old board at i,j has an organism and is marked SURVIVES, set the new board's cell at i,j to UNMARKED and its *hasOrganism* variable to *true.* If the board at i,j has an organism and is marked SURVIVES, set the new board's cell at i,j to UNMARKED and its *hasOrganism* variable to *true.* If the old board at i,j 's *state* is SPAWNING, set the new board's cell at i,j to UNMARKED and its *hasOrganism* variable to *true.* return the new Generation object
void AddOrganism(int x, int y)	Set the *hasOrganism* variable in the cell at x,y to *true.*

bool Equals(Generation g)	This function is used by the form to determine if the current state is identical to a previous state. Use nested loops to compare the *hasOrganism* variables in the two board arrays. If you find cells that don't match, return false. After the loops, return true.

The main form

The main form, news a timer, a picture box and a start button. It should also have menu strip with options which allow you to set up the test cases.



Consider adding :"user defined states" to be optional.

Form1.cs	
Variables/Constants	

const int MARK=0;
const int UPDATE=1;
	The game will either be in a marking state or an updating state. These constants define those two states.

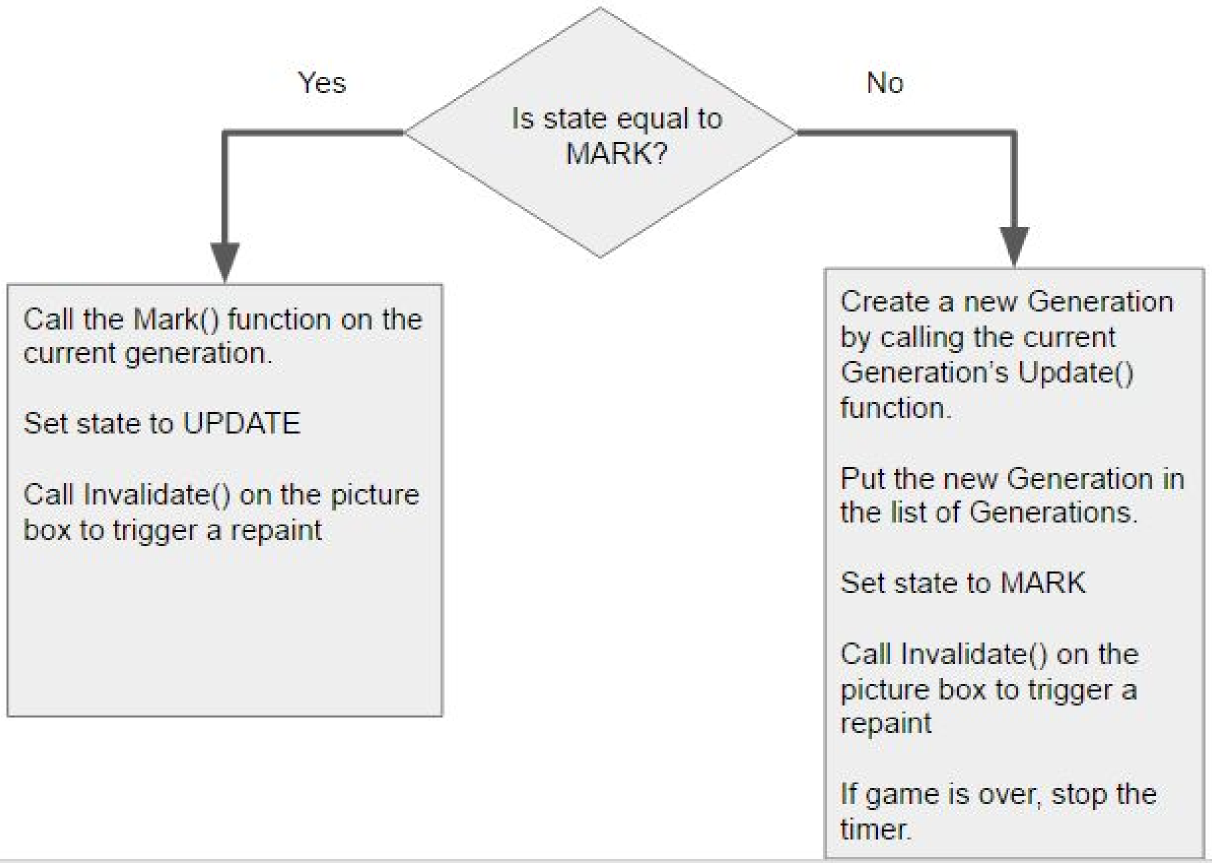
int state = MARK;	Which state the game is currently in. This variable is used by the timer to decide whether call the current generation's Mark() or Update() functions.
List<Generation> generations	A list of the generations the board has gone through. Allocate this using the *new* operator.
Functions	
Form1 (Constructor)	You don't really have to put anything here. You could call a function to set up one of your

	test cases.
bool GameOver()	Return true if the current generation is extinct (remember the Generation has an Extinct function). Return true if the max number of turns has been reached. Return true if the current Generation is equal to any of the previous Generations. (Remember the Generation class has an Equals() function). If none of the previous cases were hit, return *false.*
Paint event handler for picture box	Add the paint event to the picturebox using the lightning bolt icon on the properties panel at the bottom right of Visual Studio. Call Draw() for Generation object at the end of the Generations list. Pass in e.Graphics as the first parameter. If the game state is MARK, pass in as the second parameter. If the game state is UPDATE, pass in as the second parameter.
The timer tick event function	This function is the :"heart beat" for the game. See the flowchart below..
Menu Event Handlers for Test Cases	These functions set up a test case. Note: You could also doe as button events. Setting up a test case goes as follows… Stop the timer Clear the generations list Create a new generation object

	Add the lifeforms to specific squares on it. Add the new generation to the list Call Invalidate() on the picture box to trigger a repaint. Start the timer.
Go button event handler	Called when the Go button is clicked. Call picturebox1's Invalidate() function to redraw the board. Start timer1 to make the game run. (timer1.Start()) Note: You could also put this code in a function called Go() and call it after a test case is selected from the menu bar.

Driving the game

The form should have a timer and an event to handle when it "ticks". In the timer tick function, call the current generation's Update() function. The "current" generation will be the last one in the list. Once you have called Update(), call the picture box's Invalidate() method to trigger repaint.


Yes
No
Is state equal to MARK?
Call the Mark() function on the current generation.
Set state to UPDATE
Call Invalidate() on the picture box to trigger a repaint
Create a new Generation by calling the current Generation's Update() function.
Put the new Generation in the list of Generations.
Set state to MARK
Call Invalidate() on the picture box to trigger a repaint
If game is over, stop the timer.