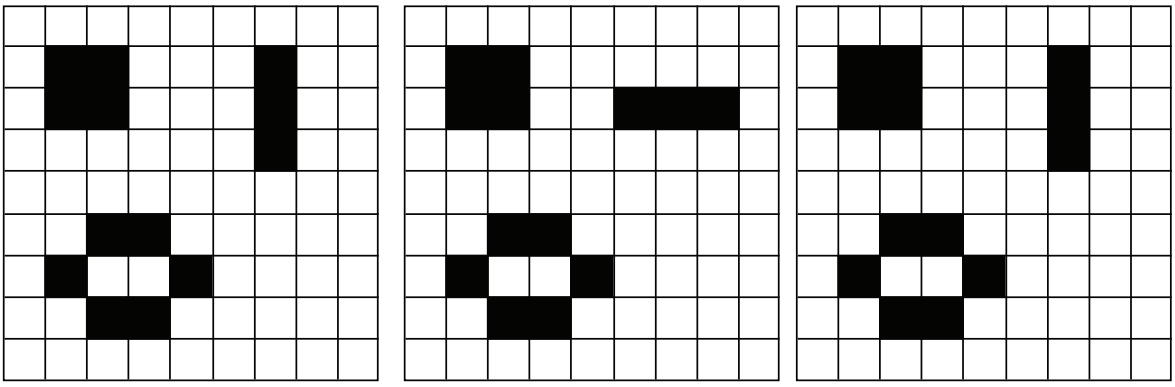
# ECE-C301 Programming for Engineers
# Instructor: James A. Shackleford
# Programming Assignment 1

## 1 The Big Idea

In this assignment you will be programming a simulation known as Conway's Game of Life. As shown in Fig. 1, the simulation consists of a *world* defined by an $N \times M$ pixel grid. In this world, pixels represent *cells* that can take on one of two different states: *living cells* are drawn as black pixels and *dead cells* are drawn as white pixels. The state of each cell within the world may change as the simulation runs forward in time. We determine if a particular cell will be alive or dead in the next frame by examining the current frame and applying an *update rule*. In Conway's Game of Life, this update rule consists of inspecting the state of the eight cells immediately surrounding the cell being updated. Living cells with too many neighbors die due to *starvation*, living cells with too few neighbors die due to *under population*, and dead cells with neither too many nor too few neighbors come to life as a result of reproduction within a hospitable environment.



(a) Frame $n$ (b) Frame $n+1$ (c) Frame $n+2$

**Fig. 1:** Three frames from a running simulation of Conway's Game of Life. Each pixel represents a living cell (black) or a dead cell (white). The number of living cells immediately adjacent to a cell determines if it is alive or dead in the next frame of the simulation. Each cell has eight immediately adjacent neighbors. The particular pattern shown here oscillates between two states.

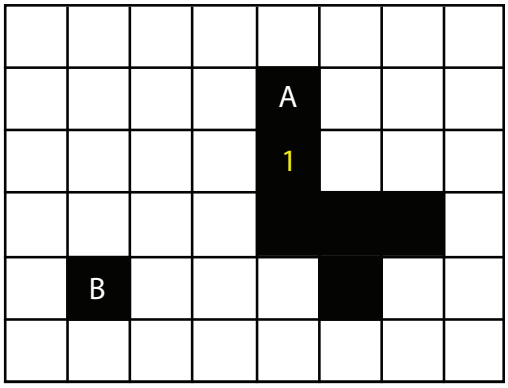
These simple biologically inspired rules for determining if a single cell should be alive or dead lead to some pretty astounding results when simulated. You will notice that certain stable and oscillatory stable patterns that tend to behave like cellular organisms emerge from the chaos.

The process of building a world grid, implementing an update rule, and managing boundary conditions are all fundamental elements of designing and implementing a broad class of computer simulations you will encounter within engineering. This project will expose you to these concepts and their implementation.

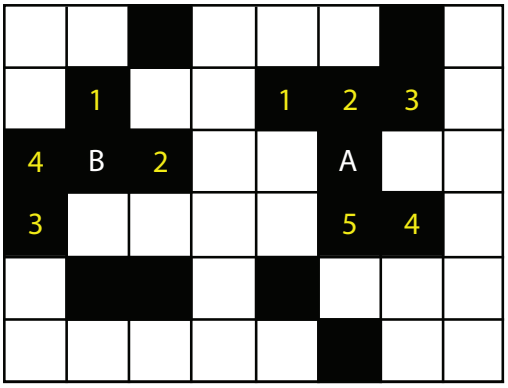
## 2 The Update Rule

Simulations step through *simulation time* (usually at a rate that is different than real time) and produce a result at each step. The results produced at each step in the simulation are generated by applying an *update rule*. The update rule is applied to the results of the previous step in order to generate the results at the next step. The update rule for Conway's Game of Life is simple:

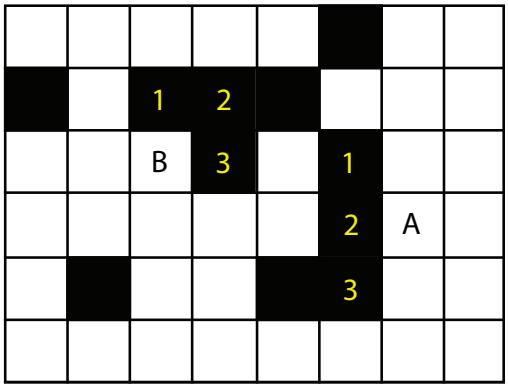
- If a cell is alive and has less than 2 living neighbors, it dies due to underpopulation. See Fig. 2(a)
- If a cell is alive and has more than 3 living neighbors, it dies due to overpopulation. See Fig. 2(b)
- If a cell is dead and has exactly 3 living neighbors, it comes to life due to reproduction. See Fig. 2(c)
- Otherwise, cells maintain their current state. See Fig. 2(d)



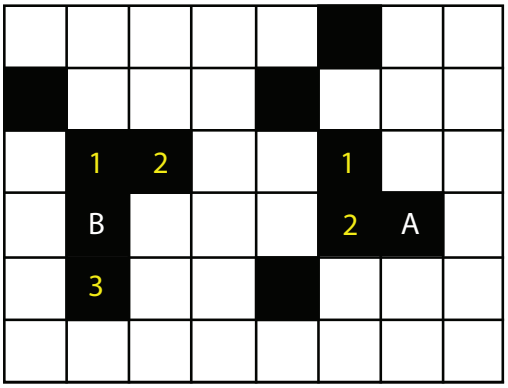

(a) **Under Population** – Both cells A and B will be dead in the next frame because they have less than 2 living neighbors.




(b) **Over Population** – Both cells A and B will be dead in the next frame because they have more than 3 living neighbors.




(c) **Reproduction** – Both cells A and B become alive in the next frame because they have exactly 3 living neighbors.




(d) **Stable Population** – Both cells A and B will stay alive in the next frame because the number of neighbors is either 2 or 3.

**Fig. 2: Update Rule Examples** – The number of living cells surrounding any given living cell determines if it will be dead (a,b) or alive (c) in the next frame. Likewise, the number of living cells surrounding a dead cell determines if it will come back to life (d) in the next frame.

# 3 Getting Ready: Basic Program Organization

The simulation program begins by running the function `main()`, which performs 5 simple operations:

1. Gets the command line options provided by the user
2. Generates the simulation world
3. Prints the user supplied options to the screen
4. Adds a known pattern to the simulation (in this case a *glider*)
5. Starts the simulation loop

```
def main():
    """
    The main function -- everything starts here!
    """
    opts = get_commandline_options()
    world = generate_world(opts)
    report_options(opts)

    blit(world, patterns.glider, 20, 20)

    run_simulation(opts, world)

if __name__ == '__main__':
    main()
```

Let's first focus on getting command line options from the user. We will take a look at each of the other operations in later sections.

This program uses a standard Python module called argparse. The `gameoflife.py` skeleton code provided to you dedicates an entire function for dealing with getting commandline options that have been supplied by the user: `get_commandline_options()`:

```
def get_commandline_options():
    parser = argparse.ArgumentParser()

    parser.add_argument('-w', '--world',
                        help='type of world to generate',
                        action='store',
                        type=str,
                        dest='world_type',
                        default='empty')

    opts = parser.parse_args()

    return opts
```

The code shown above is an abridged version of the actual function in the `gameoflife.py` skeleton in that it only defines a single option, which is used to set the type of world to be used in the simulation. This option allows the user to set the `string` stored in a variable named `world_type`. For example, if the user wanted your program to start with a world populated by randomly set pixels, they might supply the following argument at the commandline:

```
$ python gameoflife.py --world random
```

Once your program executed the line:

```
    opts = parser.parse_args()
```

the value supplied by the user, in this case: 'random', would be stored in an attribute of the `opts` object named `world_type`. So, if you read the value of `opts.world_type`, you would get the value of 'random' supplied by the user at the command line. Later in your program were the world is generated, you would first read `opts.world_type` to determine the type of world the user asked for and generate accordingly. This brings us to your first programming task.

# 4 Task 1: Creating the World

As mentioned previsouly, for this assignment you are provided with incomplete skeleton code for Conway's game of life. Your first task will be to generate the 2D matrix that will hold the state of the world. This is done in the function `generate_world()`:

```
def generate_world(opts):
    """
    Accepts: opts  -- parsed command line options
    Returns: world -- a list of lists that forms a 2D pixel buffer

    Description: This function generates a 2D pixel buffer with dimensions
                 opts.cols x opts.rows (in pixels).  The initial contents
                 of the generated world is determined by the value provided
                 by opts.world_type: either 'random' or 'empty'  A 'random'
                 world has 10% 'living' pixels and 90% 'dead' pixels.  An
                 'empty' world has 100% 'dead' pixels.
    """
    world = []

    ## TASK 1 ###################################################################
    #
    #                     [ YOUR CODE GOES HERE ]
    #
    ############################################################################

    return world
```

As you can see by inspecting the `main()` function of the simulation, the function `generate_world()` will only be called once at the start of the simulation. This same world will be used throughout the entire simulation and will be updated at every simulation step.

For this task your program should generate an empty 2D world if the user specifies the commandline option `--world empty` and a world with a randomly distributed 10% of the pixels being alive and 90% dead if the user specifies `--world random`

Upon sucessfully completing this task, you should be able to run `gameoflife.py` and the simulation world will be displayed.

# 5 Task 2: Writing a Blitter

A *blitter* (short for **bl**ock **i**mage **t**ransferer) copies a smaller block of pixel data into a larger pixel buffer at a particular x, y coordinate. You are going to write a blitting function in order to help you test out the simulation – this will allow you place specific patterns of cells into the simulation, which will make testing the correctness of your update rule implementation (i.e. Task 3) much easier.

Again refer to the code in the function `main()`. You will see that once the world is created, we attempt to copy in the pattern of a *glider* into the world at coordinates (20, 20):

```
    blit(world, patterns.glider, 20, 20)
```

Notice that patterns is the name space given to our `import`ed file `patterns.py`. In this file, I have provided some common patterns with known behaviors. If you open this file, you will see that I have divided the different patterns into groups:

- **Stills** – These are patterns that do not change when the update rule is ran. They are statically stable.
- **Oscillators** – These are patterns that don't move around within the world (i.e. their coordinates don't change) but they are in some way animated with a set period.
- **Spaceshipts** – These are patterns that do move around within the world.
- **Generators** – These are patterns that are capable of producing other independent patterns.

In the `gameoflife.py` skeleton provided, the function `blit()` doesn't do anything:

```
def blit(world, sprite, x, y):
    """
    Accepts: world  -- a 2D world pixel buffer generated by generate_world()
             sprite -- a 2D matrix containing a pattern of 1s and 0s
             x      -- x world coord where left edge of sprite will be placed
             y      -- y world coord where top edge of sprite will be placed

    Returns: (Nothing)

    Description: Copies a 2D pixel pattern (i.e sprite) into the larger 2D
                 world.  The sprite will be copied into the 2D world with
                 its top left corner being located at world coordinate (x,y)
    """
    ## TASK 2 ###################################################################
    #
    #                    [ YOUR CODE GOES HERE ]
    #
    ############################################################################
```

It is your job to populate this function. Test your implementation by generating an empty simulation world and placing a `glider` pattern at coordinates (20, 20).

# 6 Task 3: Implementing the Update Rule

Once the world has been generated and you have added a pattern or two using the blitter, you are ready to start the simulation—this is done by calling `run_simulation()`.

```
    run_simulation(opts, world)
```

This function creates a `figure` using matplotlib and plots each pixel in the world grid using the imshow() function. Subsequently, `FuncAnimation()` is called:

```
    fig = plt.figure()
    img = plt.imshow(world, interpolation='none', cmap='Greys', vmax=1, vmin=0)
    ani = animation.FuncAnimation(fig,
                                  update_frame,
                                  fargs=(opts, world, img),
                                  interval=opts.framedelay)

    plt.show()
```

The fourth argument supplied to `FuncAnimation()` specifies how frequently the plot should be automatically updated (in milliseconds). As you can see, this is set on the command line using the `--framedelay` option (See Section 3 and `get_commandline_options()` in `gameoflife.py`). The second argument supplied to `FuncAnimation()` specifies the function that is called for each frame update. The third argument allows you to pass function parameters to the update function in the form of a tuple. In this case we are passing the `opts`, `world`, and `img` parameters to the `update_frame()` function[1]. This means that all you need to do to implement the update rule is implement the function `update_frame()`:

```
def update_frame(frame_num, opts, world, img):
    """
    Accepts: frame_num  -- (automatically passed in) current frame number
             opts       -- a populated command line options instance
             world      -- the 2D world pixel buffer
             img        -- the plot image
    """

    img.set_array(world)

    new_world = []
    for row in world:
        new_world.append(row[:])

    ## TASK 3 ##################################################################
    #
    #                    [ YOUR CODE GOES HERE ]
    #
    ############################################################################

    # Copy the contents of the new_world into the world
    # (i.e. make the future the present)
    world[:] = new_world[:]
    return img,
```

[1] If you look at the definition of the update function `update_frame()`, you will notice that it accepts 4 parameters instead of 3. This is because the frame number is automatically passed in as the first argument when the update function is called. Any arguments passed in as a tuple through the third argument of `FuncAnimation()` are passed in after the frame number argument.

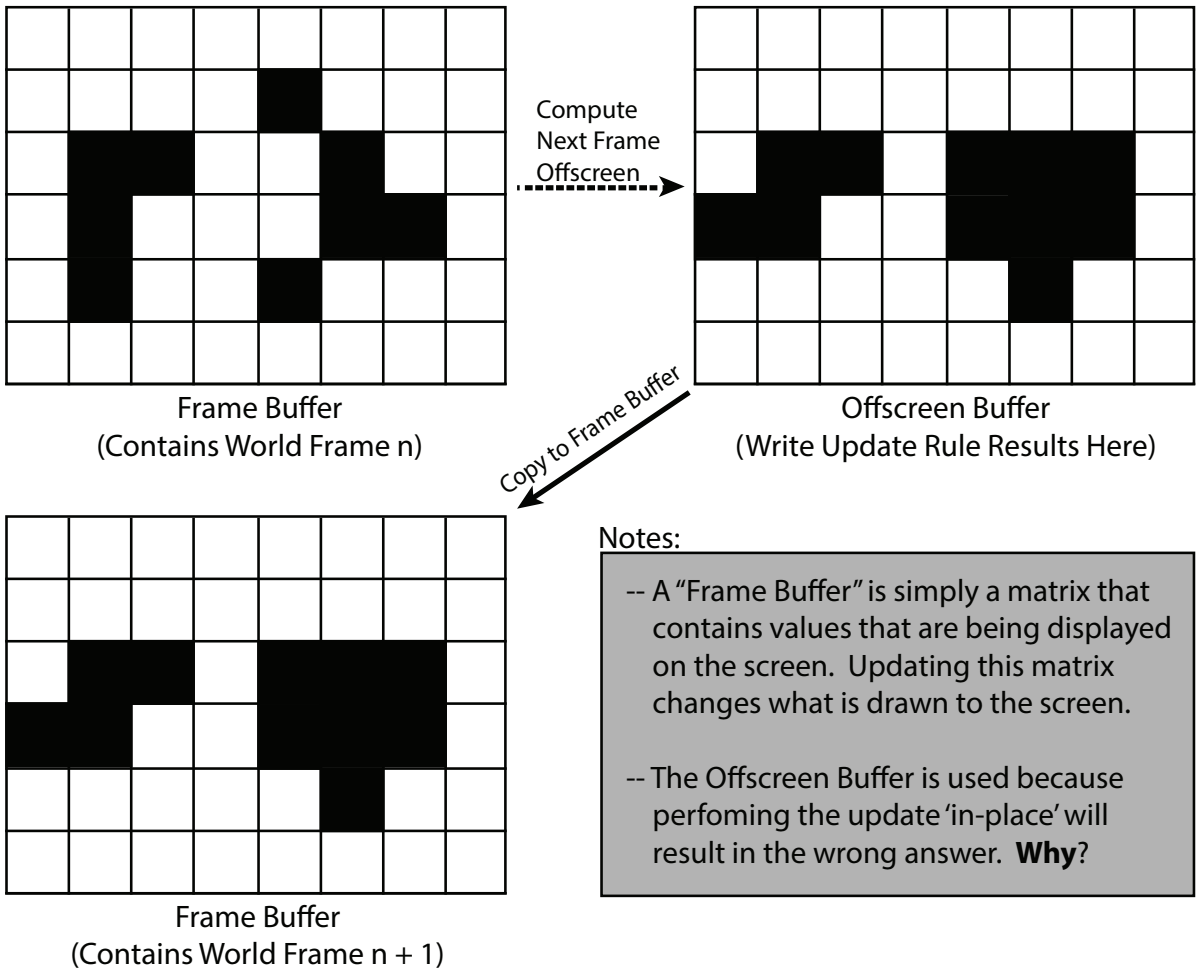



**Fig. 3:** Graphical depiction of the frame update process.

Here, `world` contains the current world state, and `new_world` will be used to hold the solution to applying the update rule. Once the next frame has been generated, the solution is copied back into `world` for display. This process is visually depicted in Fig. 3.

## 6.1 Imposing Boundary Conditions

In Conway's Game of Life, the number of neighbors surrounding a cell determine if it lives or dies. Generally, cells have 8 surrounding neighbors. However, cells lying on the border of the image only have 5 neighbors, and corner cells only have 3 neighbors.

We can make all cells have 8 neighbors by implementing the boundary condition that the "world wraps around." In other words, all cells on the left edge of the world should be considered to be neighbors to the right of all cells on the right edge of the world. A similar condition should be implemented for the top and bottom of the world.

## 6.2 How to Test it

Generate an empty world using `--world empty` and place a static pattern into the world using your `blit()` function. The static pattern should not change if your update function is correct.

Next, test your update function using a more complex oscillating pattern. Check the Game of Life entry on Wikipedia for animations that show what the oscillating patterns should look like if your update rule is correct.

Finally, test your update function using the `glider` spaceship pattern. If your update rule and boundary conditions are correct, the glider should move diagonally towards the bottom right of the world and wrap around to the top left of the world when it hits the edge.

Once your simulation if finally verified as working, try out the `gosper` pattern.

# 7 Deliverables

Your deliverables for this project are a working implementation of Conway's Game of Life that *you* wrote as well as a report detailing the project, the theory, your implementation, and your testing results including screenshots. Please do not submit other students work.

Submit your report in PDF format.

Submit your complete source code as a zip file. (Do *NOT* include your report in this zip file.)

**NOTE: In order for the matplotlib windows to open, you must enable X11 tunneling! The TA will explain how to do this in the beginning of lab in Week 1. If you are connecting to thanos.ece.drexel.edu on Windows, make sure X-Win32 is running (it renders the graphical windows for you!). Additionally, using X11 tunneling to display windows will probably be incredibly slow if you are working from home. I suggest that you work on this assignment in the labs. You can also install Python 2.7 and matplotlib on your personal computer as an alternative, which doesn't require you to connect to thanos.**