Name : Kwok Leung Liu

ECE-C301

Progect_1

Theory of Conway's Game of Life

World:

Pixels represent cells that can take on one of two different states:

Living cells are drawn as black pixels

Dead cells are drawn as white pixels.

Update Rules:

1. If a cell is alive and has less than 2 living neighbors, it dies due to underpopulation

2. If a cell is alive and has more than 3 living neighbors, it dies due to overpopulation.

3. If a cell is dead and has exactly 3 living neighbors, it comes to life due to reproduction.

4. Otherwise, cells maintain their current state.

Implementation Code

Method:

1. get_commandline_options()

```
parser = argparse.ArgumentParser()

parser.add_argument('-r', '--rows',
                    help='set # of rows in the world',
                    action='store',
                    type=int,
                    dest='rows',
                    default=100)

parser.add_argument('-c', '--columns',
                    help='set # of columns in the world',
                    action='store',
                    type=int,
                    dest='cols',
                    default=100)

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

parser.add_argument('-d', '--framedelay',
                    help='time (in milliseconds) between frames',
                    action='store',
                    type=int,
                    dest='framedelay',
                    default=40)

opts = parser.parse_args()

return opts
```

This command will use to get the input from the user in order to set the property of the world.

2. 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 = [[0 for x in range(opts.cols)]for x in range(opts.rows)];           # make the world size
    if opts.world_type=='empty':                                               # make empty world
        return world;
    if opts.world_type=='random':                                              # make random world
        for i in range(opts.rows):
            world[i]=numpy.random.choice([0, 1],size=opts.cols,p=[0.9, 0.1]);
        return world;
```

This command will use to generate random or empty world

3. report_options()

```
def report_options(opts):
    """
    Accepts: opts  -- a populated command line options class instance

    Returns: (Nothing)

    Descrption: This function simply prints the parameters used to
                start the 'Game of Life' simulation.
    """

    print "Conway's Game of Life"
    print "====================="
    print "    World Size: %i x %i" % (opts.rows, opts.cols)
    print "    World Type: %s" % (opts.world_type)
    print "  Frame Delay: %i (ms)" % (opts.framedelay)
```

This command will print the parameters used to start the 'Game of Life' simulation.

4. blit()

```
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)
    """
    for rowcount in range(y,len(sprite)+y):
        for colcount in range(x,len(sprite[1])+x):
            world[rowcount][colcount]=sprite[rowcount-y][colcount-x];
```

This command will applied the pattern.py file to the created world.

5. run_simulation()

```
def count(rowcount,colcount,world,opts):
    temp=0;
    rowcount2=rowcount+1;
    colcount2=colcount+1;
    if rowcount==opts.rows-1:rowcount2=0;
    if colcount==opts.cols-1:colcount2=0;
    if world[rowcount2][colcount-1]==1:temp=temp+1;
    if world[rowcount2][colcount]==1:temp=temp+1;
    if world[rowcount2][colcount2]==1:temp=temp+1;
    if world[rowcount][colcount-1]==1:temp=temp+1;
    if world[rowcount][colcount2]==1:temp=temp+1;
    if world[rowcount-1][colcount-1]==1:temp=temp+1;
    if world[rowcount-1][colcount]==1:temp=temp+1;
    if world[rowcount-1][colcount2]==1:temp=temp+1;
    return temp;
```

```
def run_simulation(opts, world):
    """
    Accepts: opts  -- a populated command line options class instance
             world -- a 2D world pixel buffer generated by generate_world()

    Returns: (Nothing)

    Description: This function generates the plot that we will use as a
                 rendering surfance.  'Living' cells (represented as 1s in
                 the 2D world matrix) will be rendered as black pixels and
                 'dead' cells (represetned as 0s) will be rendered as
                 white pixels.  The method FuncAnimation() accepts 4
                 parameters: the figure, the frame update function, a
                 tuple containing arguments to pass to the update function,
                 and the frame update interval (in milliseconds).  Once the
                 show() method is called to display the plot, the frame
                 update function will be called every 'interval'
                 milliseconds to update the plot image (img).
    """
    if not world:
        print "The 'world' was never created.  Exiting"
        sys.exit()

    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()
```

This command will utilize matplotlib function to product the animation.

6. Count()

```
def count(rowcount,colcount,world,opts):
    temp=0;
    rowcount2=rowcount+1;
    colcount2=colcount+1;
    if rowcount==opts.rows-1:rowcount2=0;
    if colcount==opts.cols-1:colcount2=0;
    if world[rowcount2][colcount-1]==1:temp=temp+1;
    if world[rowcount2][colcount]==1:temp=temp+1;
    if world[rowcount2][colcount2]==1:temp=temp+1;
    if world[rowcount][colcount-1]==1:temp=temp+1;
    if world[rowcount][colcount2]==1:temp=temp+1;
    if world[rowcount-1][colcount-1]==1:temp=temp+1;
    if world[rowcount-1][colcount]==1:temp=temp+1;
    if world[rowcount-1][colcount2]==1:temp=temp+1;
    return temp;
```

This command will count the neighbor cell of each lived cell.

7. 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
    """

    # set the current plot image to display the current 2D world matrix
    img.set_array(world)

    # Create a *copy* of 'world' called 'new_world' -- 'new_world' will be
    # our offscreen drawing buffer.  We will draw the next frame to
    # 'new_world' so that we may maintain an in-tact copy of the current
    # 'world' at the same time.
    new_world = []
    for row in range(opts.rows):
        new_world.append(numpy.zeros(opts.cols));
    for rowcount in range(opts.rows):
        for colcount in range(opts.cols):
            if world[rowcount][colcount]==1:
                if count(rowcount,colcount,world,opts)<2:
                    new_world[rowcount][colcount]=0;
                if (count(rowcount,colcount,world,opts)==2)or(count(rowcount,colcount,world,opts)==3):
                    new_world[rowcount][colcount]=1;
                if count(rowcount,colcount,world,opts)>3:
                    new_world[rowcount][colcount]=0;
            else:
                if count(rowcount,colcount,world,opts)==3:
                    new_world[rowcount][colcount]=1;



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

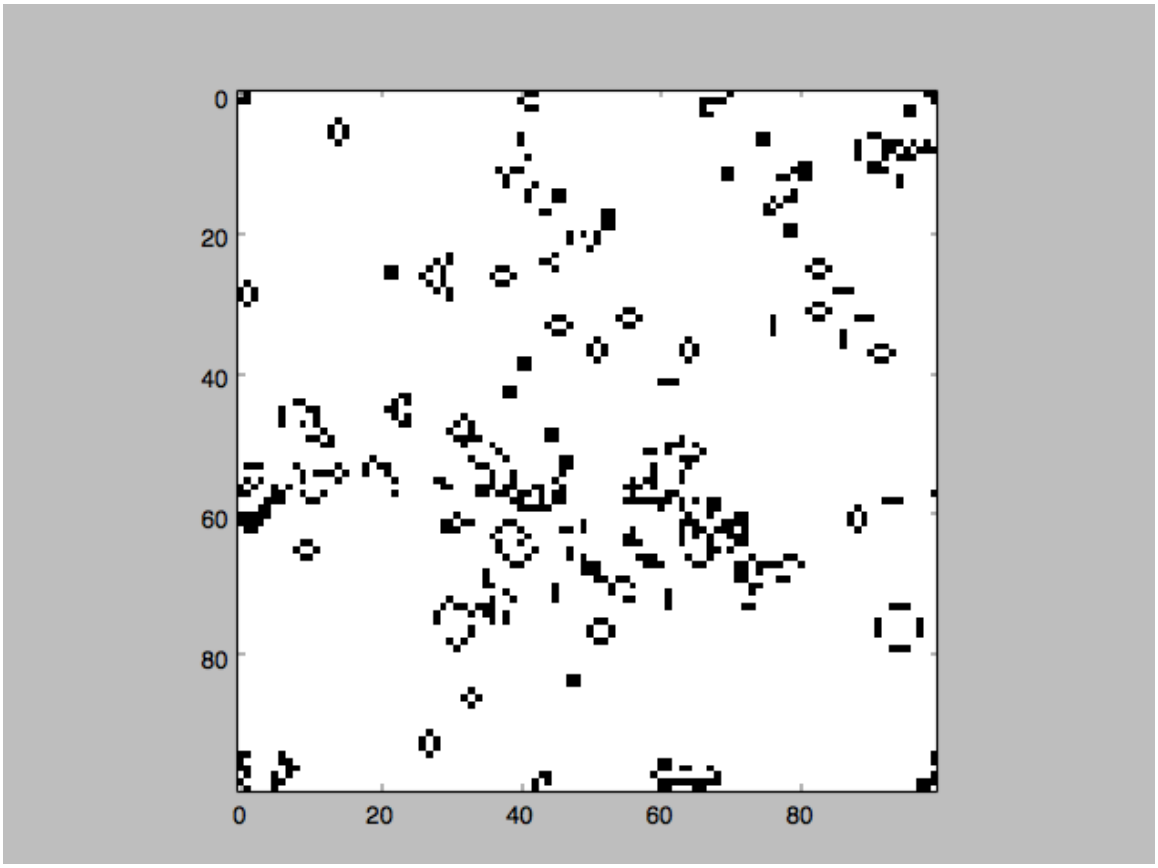
This command will applied above 4 rules to decide cell is died or lived.

Testing results

1. This is when I run the random world's code.

```
TiGeRs-MacBook-Air:~ TiGeR$ python gameoflife.py -w random
```

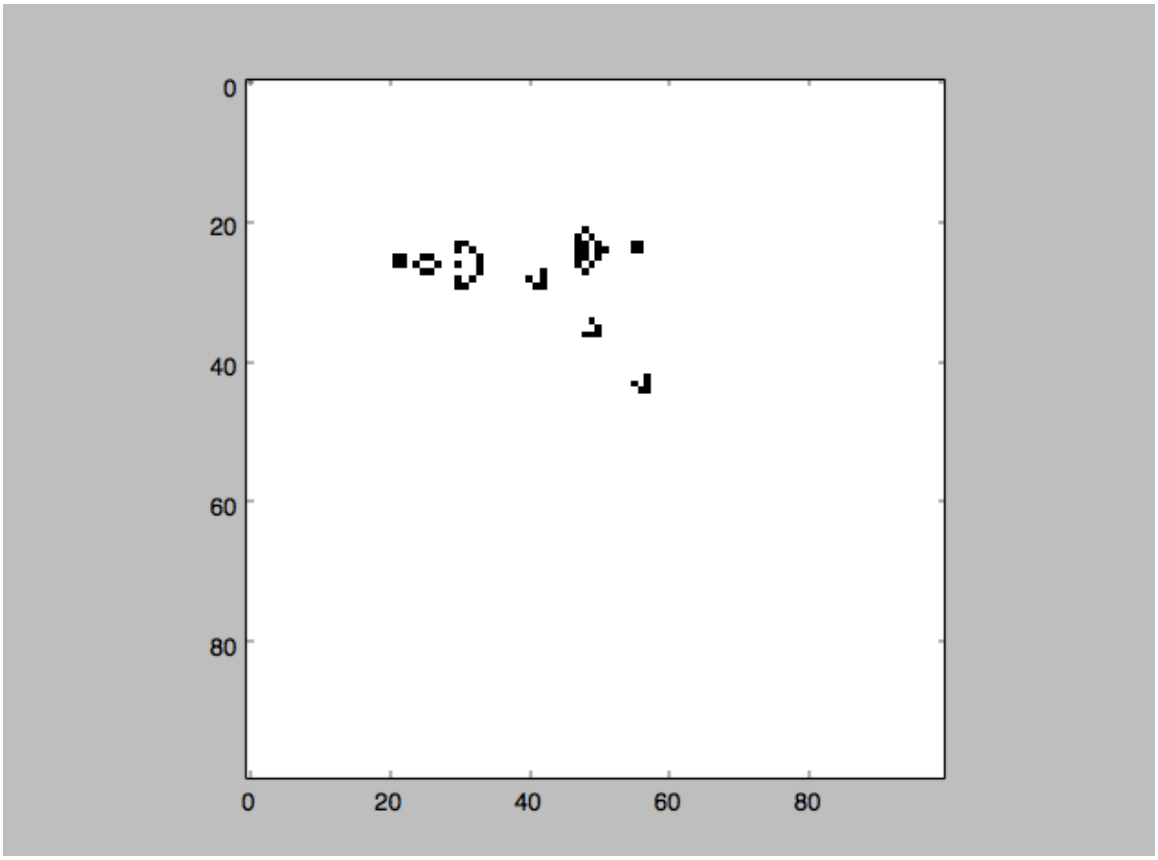
This is matplot picture for random world.



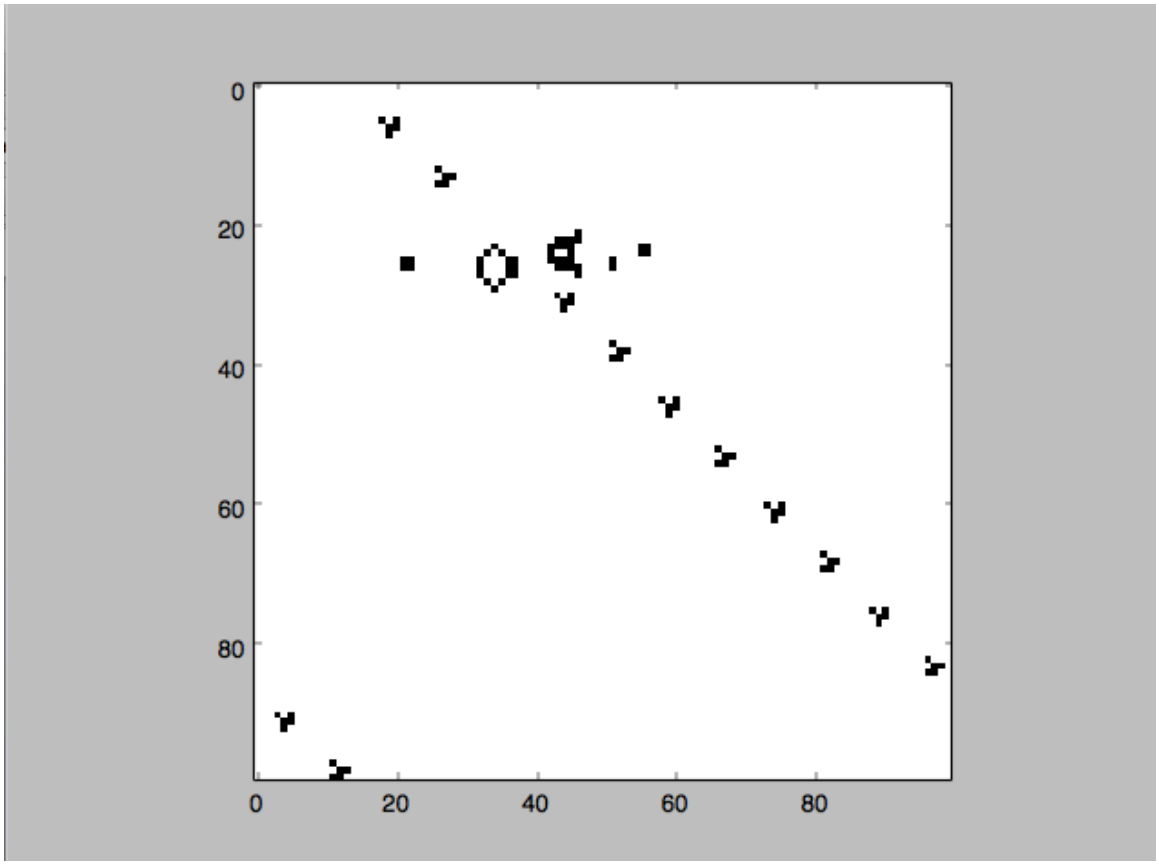
2. This is when I run the empty world's code.

```
TiGeRs-MacBook-Air:~ TiGeR$ python gameoflife.py -w empty
```

This is matplot picture for empty world.



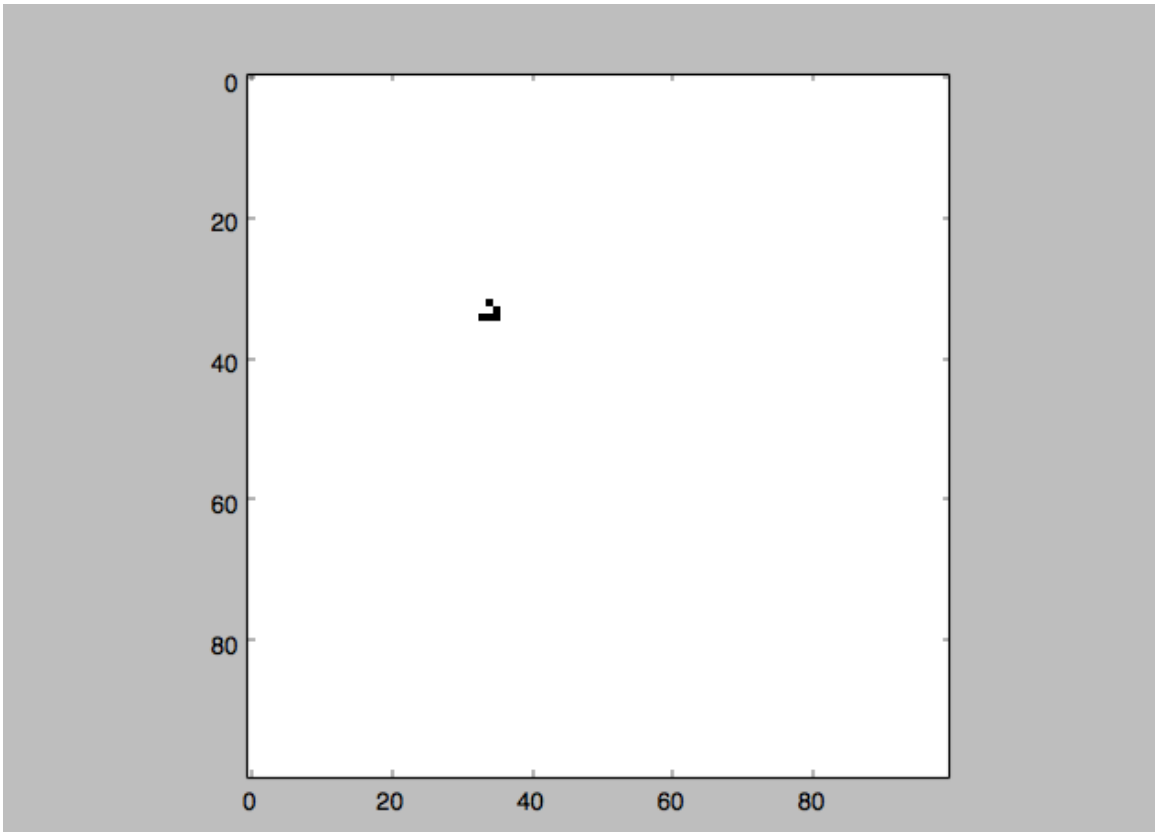
While counting the cell at the edge of the world, the program will wrap around and count the other side of the matrix.s



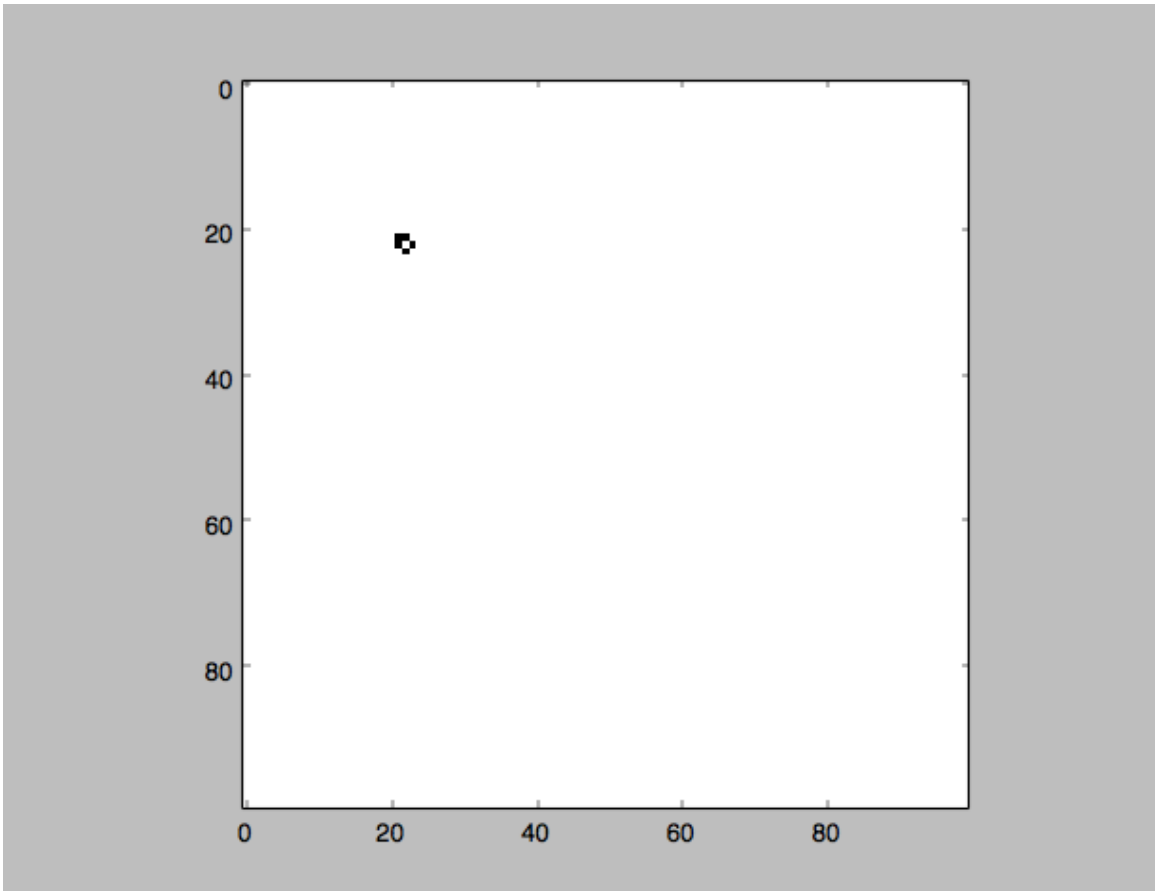
3. The user can choose pattern while calling the blit method.

```
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)    # here is for change the patterns

    run_simulation(opts, world)


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

This is the matplot when I change the glider patterns.



4. This is the boat pattern example, the world is in stationary state.



Conclusion

The program contain eight main methods. When the program is executed the main() will be ran. main() will use get_commandline_options() to get the setting of the simulation from user. If user does not provide any input. Default value in will be used for the run, the setting will be print by report_option() to the console. After getting the setting of the simulation, generate_world() will be called and the setting will be sent to the method. generate_world() will take create an empty or random double array which depend on input of the setting. The size of the double array is also depend on the input setting. Then

blit() method will be called. This method will take the world that was just created by generate_world(), and an pattern array from the provided patterns.py file, and also the x, y coordinate corresponded to the left and top most position where user want the pattern be placed in the world array. run_simulation() will be called after the pattern is placed into the 2D world pixel buffer (which is the world). And this method will use matplotlib function to create an image and do the animation rendering. It also populate the world array in graph form. After that, update_frame() will be called. This method accepts current frame number, populated command line options instance, the 2D world pixel buffer (which is the world), and the plot image generated by run_simulation. And then this method will create an empty new_world array which has a same size of the world. And will use it to store the status of all the cell for the next frame. Then this method will loop through every cell in the current world and get the number of neighbor from count_neighbor() method and determent the cell should be alive or dead by checking all the following rules. 1. Any live cell with two or three live neighbors lives on to the next generation. 2. Any live cell with fewer than two live neighbors dies, as if caused by under-population. 3. Any live cell with more than three live neighbors dies, as if by over-population. 4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction When the checking is done. This method will put the new_world the image buff for the next frame. count_neighbors() this method will be called when update_frame() is checking the number of each cell. This method will take the x, y, coordinate that is checking and the current world array. And it will return the number of neighbor around the (x,y) cell. Notice that in fig2 the edge cell checking has set to wrap