Programmers Guide to "Game of Life by John Conway"

Introduction

Welcome to the Programmers Guide for "Game of Life by John Conway"! This guide will provide you with an overview of the program's structure, functionality, and how to make modifications or enhancements to the code.

Program Overview

The "Game of Life" is a cellular automaton devised by the British mathematician John Horton Conway. It consists of a grid of cells that can be either alive or dead. The cells evolve over time based on a set of rules, creating interesting patterns and behaviors. The program allows you to interact with the game by changing the state of the cells and observing their evolution.

Dependencies

To run the program, you need to have the `pygame` library installed. You can install it using the following command:

```
pip install pygame
```

Program Structure

The program is written in Python and consists of the following components:

1. **Constants**: There are several constant variables defined at the beginning of the code, such as the grid size, colors, delay, and cell size.

```
DELAY_MS = 100
WIDTH, HEIGHT = 50, 50
COLOR_ALIVE_NOT_STATIC = (255,255,255)
```

```
COLOR_DEAD_NOT_STATIC = (0,0,0)
COLOR_ALIVE_STATIC = (0,255,0)
COLOR_DEAD_STATIC = (255,0,0)

COLOR_GRID = (40, 90, 0)

CELL_SIZE = 10
# if cell min >= and <= max it will be alive on the next move
MIN_NEIGHBOURS_FOR_ALIVE = 2
MAX_NEIGHBOURS_FOR_ALIVE = 3
MIN_NEIGHBOURS_FOR_DEAD = 2
MAX_NEIGHBOURS_FOR_DEAD = 2
```

2. `class Cell`

This class represents a single cell in the grid. It has methods to change the state(is cell alive or dead) and static (a cell that does not change its state) properties of the cell, as well as getter and setter methods. It also has its own "id" if you want to do some other cell behavior.

```
def __init__(self, id_height:int, id_width:int, state:bool = False, static:bool = False) -> None:
        self.id_height = id_height
        self.id_width = id_width
        self.state = state
        self.static = static
    def changeState(self) -> None:
        self.state = not self.state
    def changeStatic(self) -> None:
        self.static = not self.static
    def setState(self, flag) -> None:
        self.state = flag
    def getState(self) -> bool:
        return self.state
    def getStatic(self) -> bool:
        return self.static
```

The class has two similar functions that change the status of the cell. The first option is made to change the cell, when we click on it, we do not need to look at any properties of the cell.

```
def setState(self, flag) -> None:
        self.state = flag
def changeState(self) -> None:
        self.state = not self.state
```

```
def get_color(self): # return rgb (int, int, int) by state and status
        if not self.state and not self.static:
            color = COLOR_DEAD_NOT_STATIC
        elif not self.state and  self.static:
            color = COLOR_DEAD_STATIC
        elif self.state and  not self.static:
```

```
            color = COLOR_ALIVE_NOT_STATIC
        else: # state and static
            color = COLOR_ALIVE_STATIC
        return color
```

3. **Update Function**: The `update` function is responsible for rendering the grid on the screen based on the current state of the cells.

```
def update(screen, grid, cell_size=CELL_SIZE):
    #iterate every cell
    for  col in range(HEIGHT):
        for row in range(WIDTH):
            cell = grid[col][row]
            color = cell.get_color()
            #cell_size-1, to see the grid (background)
            pygame.draw.rect(screen, color, (row*cell_size, col*cell_size,\
cell_size-1, cell_size-1))
```

Called every time, regardless of whether the game is stopped or not

```
def live_or_die(is_alive:bool, alive_neighbours:int)->bool:
```

Return true/false if cell should be alive or dead on the next move.

4. **Game of Life Function**: The `game_of_life` function implements the rules of the "Game of Life" and updates the state of the cells for each generation.

Input is our grid of cells

The output data is the same size as the grid table, which contains True if the cell is alive, False - dead, None - unchanged.

5. Main Function: The `main` function is the entry point of the program. It initializes the pygame library, sets up the screen, handles user input (to start the game), and controls the game loop.

```
    # Initialize pygame
    pygame.init()

    #setting screen
    screen = pygame.display.set_mode(size=(WIDTH * CELL_SIZE, HEIGHT *
CELL_SIZE))
    pygame.display.set_caption("Game of Life   (paused)")

    grid = list()
```

```
    for i in range(HEIGHT):
        rows = list()
        for j in range(WIDTH):
            rows.append(Cell(i,j))
        grid.append(rows)

    #grid color
    screen.fill(COLOR_GRID)
```

Initializing our grid and screen

```
    grid = list()
    for i in range(HEIGHT):
        rows = list()
        for j in range(WIDTH):
            rows.append(Cell(i,j))
        grid.append(rows)

    #grid color
    screen.fill(COLOR_GRID)

    #set empty cells over the background
    update(screen, grid, CELL_SIZE)


    last_col_lmb = last_row_lmb  = -1
    last_col_rmb = last_row_rmb = -1
```

We set the last mouse position to -1 so that the user can click on the cell without moving from it and it changes its state

In the loop, we look at the user's input (if he pressed the cross to close the program), pressed or released LMB/RMB

```
 while True:
      for event in pygame.event.get():
```

```
if event.type == pygame.QUIT:
      …
elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                  # pause unpause the game
```

```
if event.type == pygame.MOUSEBUTTONUP and event.button == 1: # LEFT MB
      …
      The user releases the button and we forget the mouse position
```

```
      Similar to RMB
```

```
col, row = pygame.mouse.get_pos()[0]//CELL_SIZE, \
               pygame.mouse.get_pos()[1]//CELL_SIZE
           if pygame.mouse.get_pressed()[0] == 1: #LEFT MB
               if col != last_col_lmb or row != last_row_lmb:
                   grid[row][col].changeState()
                   last_col_lmb, last_row_lmb = col, row
```

By the coordinates of the cursor, we change the cell when the button is pressed on it.When the button is pressed, we change the state of the cell that the mouse is pointing at, but so that it does not change every cycle, we check the previous mouse coordinates.

```
update(screen, grid, CELL_SIZE)
pygame.display.update()
```

The user sees the changes instantly

```
if pause == False and pygame.time.get_ticks() - timer >= DELAY_MS:
```

Every „DELAY loop“ we „play“ game of life, in other words we iterate though „game_of_life“ algorithm.

```
new_grid = game_of_life(grid)
for col in range(HEIGHT):
      for row in range(WIDTH):
            if new_grid[col][row] is not None:
                  grid[col][row].setState(new_grid[col][row])
update(screen, grid, CELL_SIZE)
pygame.display.update()
# next iteration time
timer = pygame.time.get_ticks()
```

Modifying the Code

If you want to make modifications or enhancements to the code, here are a few areas you can consider:

- **Customize Grid Size**: You can adjust the `WIDTH` and `HEIGHT` constants to change the size of the grid.

- **Modify Cell Colors**: The program uses predefined colors for different cell states. You can modify the RGB values of the color constants (`COLOR_ALIVE_NOT_STATIC`, `COLOR_DEAD_NOT_STATIC`, `COLOR_ALIVE_STATIC`, `COLOR_DEAD_STATIC`, `COLOR_GRID`) to customize the colors.

- **Change Update Frequency**: The `DELAY_MS` constant determines the delay between each grid update. You can adjust this value to change the speed of the game.

- **Implement Additional Rules**: The `game_of_life` function currently implements the standard rules of the "Game of Life." You can add or modify the rules to create new behaviors or patterns.

By changing MIN (MAX) NEIGHBOURS FOR ALIVE (DEAD) for example.

- **Enhance User Interface**: You can improve the user interface by adding buttons or menu options to control various aspects of the game.

- **Save/Load Grid**: Implement functionality to save the current grid state to a file and load it later.

These are just a few ideas to get you started. Feel free to explore and experiment with the code to create your own version of the "Game of Life."

Conclusion

Congratulations! You now have a good understanding of the "Game of Life by John Conway" program and how to modify it. Enjoy exploring the

world of cellular automata and have fun experimenting with different patterns and rules. Happy coding!