Interactive Conway's game of life

April 2, 2024

1 Conway's Game of Life

1.0.1 Description

The Conway's Game of Life is a classic example of cellular automaton devised by mathematician John Conway in 1970. It's a zero-player game, meaning its evolution is determined by its initial state, requiring no further input. The game is played on a two-dimensional grid where each cell can be either alive or dead. The state of each cell evolves over discrete time steps based on simple rules:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

These rules create complex patterns from simple initial configurations. The game has found applications in various fields such as computer science, biology, and even art due to its emergent properties and ability to model complex systems.

1.0.2 Package contents

Main file:

- Interactive_Conways_game.py

Supplementary scripts:

- Objects.py
- glider_gun_script.py
- pulsar_script.py
- board_script.py
- bar_oscillator_script.py
- soba_script.py #### Other files:
- requirements.txt
- Minecraft.ttf

See further description in text below

1.0.3 Prerequisites

First of all, necessary libraries need to be installed. This can be done from requirments.txt (running pip install -r requirements.txt). Make sure to have the following libraries installed: - numpy -

pygame - matplotlib - sys - datetime - screeninfo - importlib

Also (albeit optionally) install the following fonts (or have the font files in the same directory as this file): - Minecraft, which can be found e.g. at: https://www.dafont.com/minecraft.font

1.0.4 Program overview

The core of the program is a 2D playing field, called 'board', which consists of black and white squares: white ones denote living cells and the black ones represent dead cells. The board is coded as a 3D numpy array with the following shape: (x, y, 2). The first two dimensions represent the board's size and the third dimension stores the current state and the neighbours count. The state is stored in the first layer and the neighbours count is stored in the second layer. The state of a cell can be either 0 (dead) or 1 (alive). The neighbours count is an integer that represents the number of alive cells around a given cell. When the board is initialized with random values, with 80% of the cells being dead and 20% being alive. This proportion can be changed by modifying the p parameter in the np.random.choice function. The cells on the edges of the board are initialized as dead (boundary condition). The function create_board() receives two parameters, x and y, which represent the size of the board. Size in the pygame build is determined given the screen resolution or manually at user's discretion in a simple version.

[ ]:
```
def create_board(x,y):
    """
    Function to create a random board for the Conway's game of life.
    Args:
        x (int): the width of the board
        y (int): the height of the board
    Returns:
        board (numpy.ndarray): the initialized board
    """

    states = 0
    counts = 1
    board = np.zeros((x, y, 2), dtype=int)
    board[:,:,states] = np.random.choice([0, 1], size=(x, y), p=[0.8, 0.2])
    board[0,:,states] = 0
    board[x-1,:,states] = 0
    board[:,0,states] = 0
    board[:,y-1,states] = 0
    return board
```

Given the current state of the board, in order to determine the next generation, the number of neigbors is counted and then the Conway's game rules, described above, are implemented by the update() function.

[ ]:
```
def update(self):
    """
    Update the board state given the number of neighbors.
    """
    # Count the neighbours
```

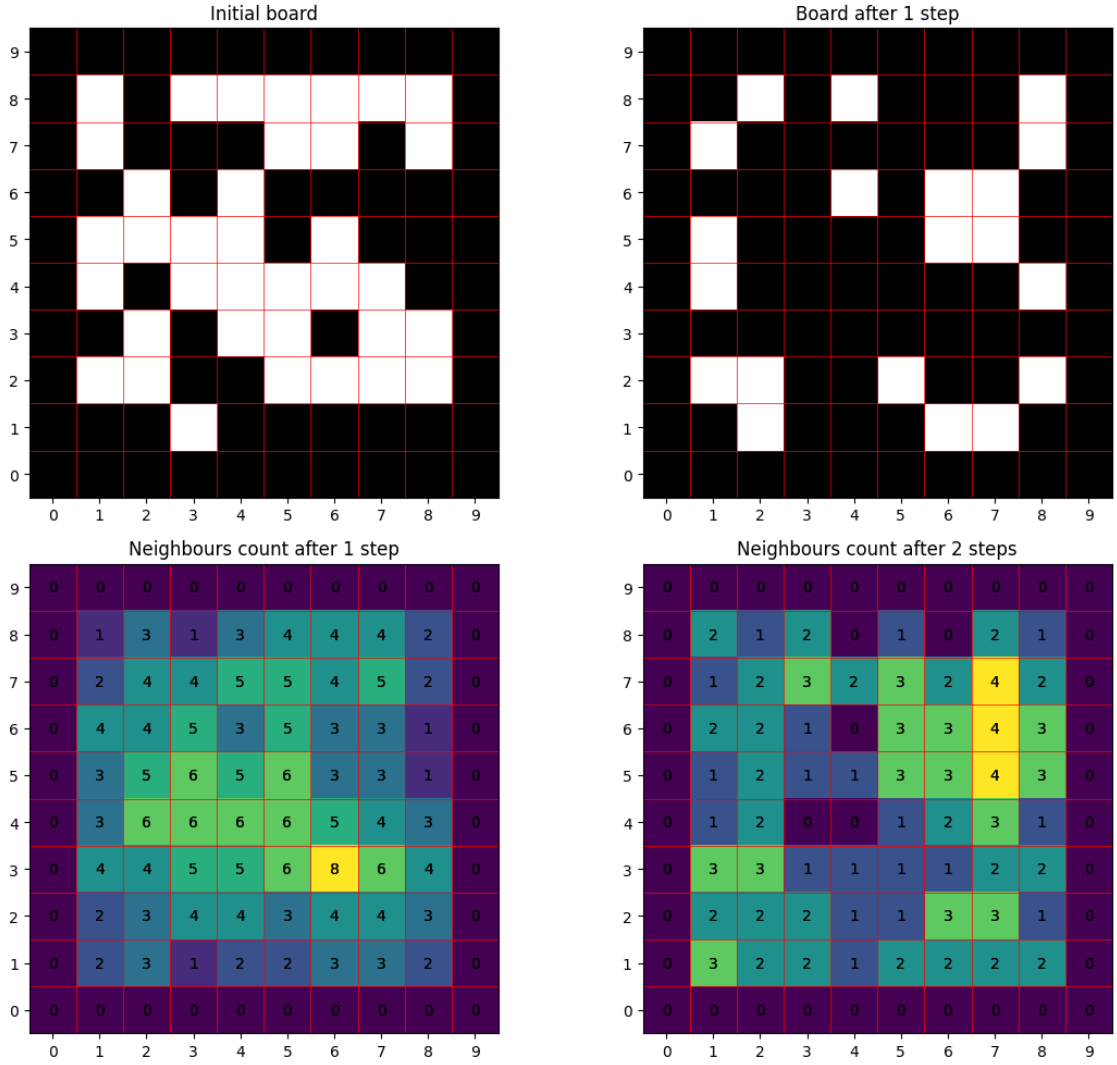
```
    self.board[:, :, counts] = np.zeros((x, y), dtype=int)
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di != 0 or dj != 0:
                self.board[1:x - 1, 1:y - 1, counts] += self.board[1 + di:x - 1␣
↪+ di, 1 + dj:y - 1 + dj, states]
    time.sleep(0.1)  # to slow down the game

    # Update states based on number of neighbours
    self.board[np.where(self.board[1:x - 1, 1:y - 1, counts] > 3)[0] + 1,
               np.where(self.board[1:x - 1, 1:y - 1, counts] > 3)[
                   1] + 1, states] = 0

    self.board[np.where(self.board[1:x - 1, 1:y - 1, counts] < 2)[0] + 1,
               np.where(self.board[1:x - 1, 1:y - 1, counts] < 2)[
                   1] + 1, states] = 0

    self.board[np.where(self.board[1:x - 1, 1:y - 1, counts] == 3)[0] + 1,
               np.where(self.board[1:x - 1, 1:y - 1, counts] == 3)[
                   1] + 1, states] = 1
```

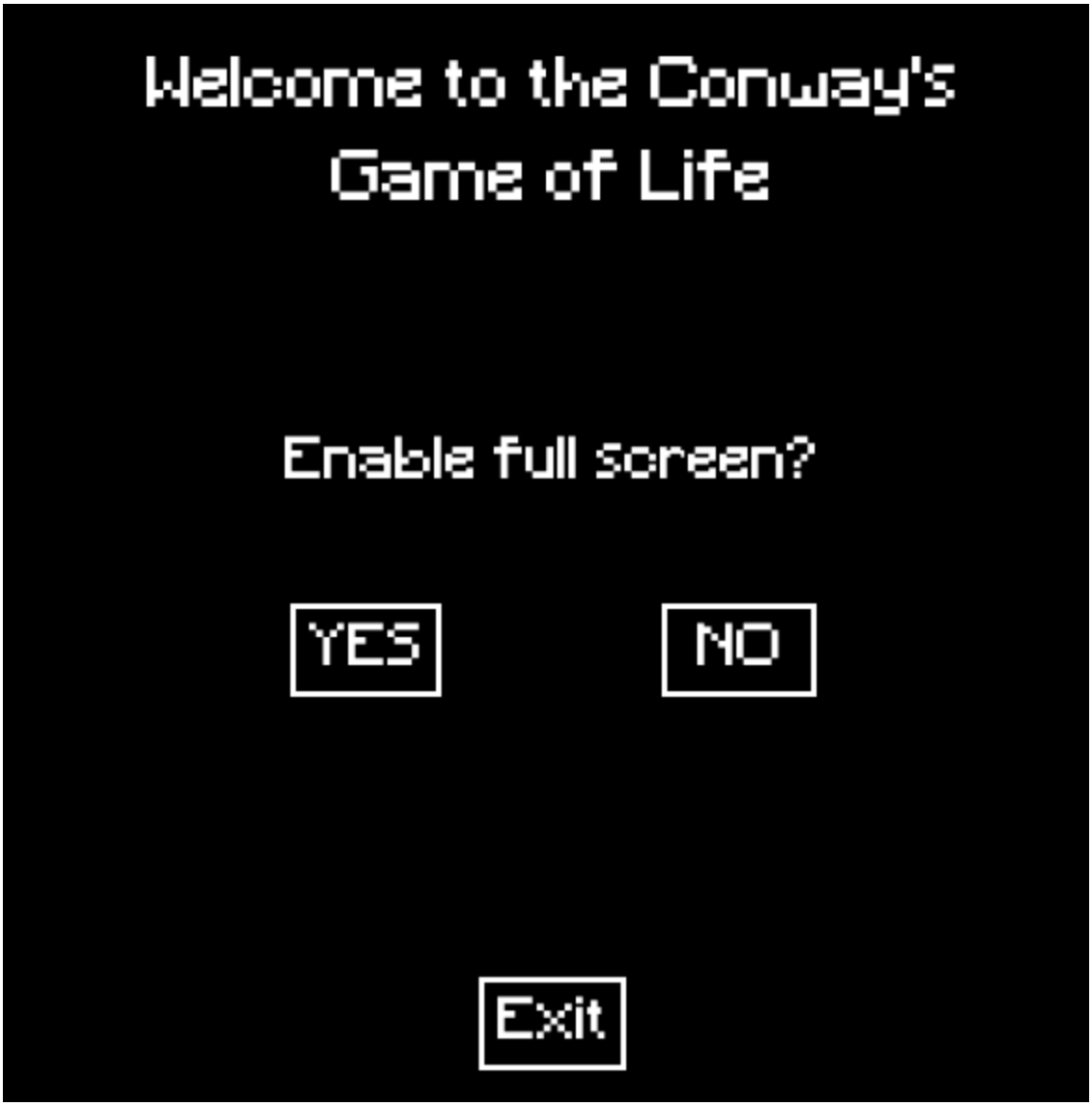
Illustration of the game working principle using matplotlib tools below:



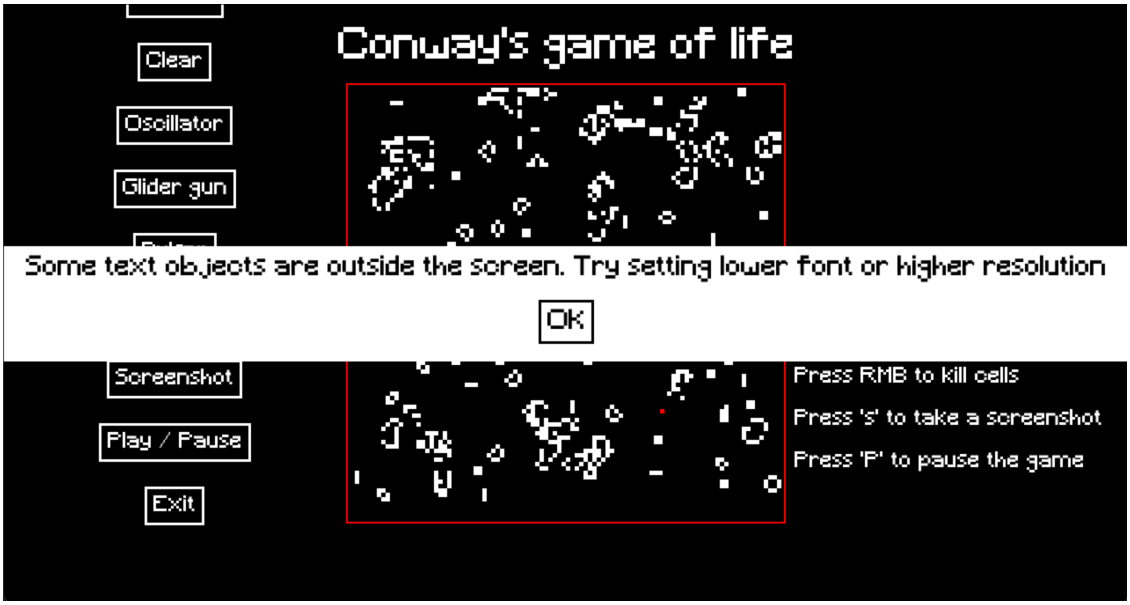
1.0.5 Interactive Conway's Game

The build consits of several files. Main program is represented by **Interactive_Conways_game.py** file. Additionally there is a supplementary file, called Objects.py, which contains classes and functions, needed to build interactive in-game menu, dialog boxes, prompts and so on. Finally, there are several scripts, containing numpy arrays or strings, which initialize the board randomly as well as some interesting patterns of the Conway's game, namely: Bar oscillator, Glider gun, Pulsar, Soba Spaceship and so on. The scripts are named correspondingly. In the main file user may adjust the user_screen variable according to his/her screen resolution or leave it as None – then the program will detect the resolution automatically. Then the program will promt a user to choose if he/she wants to run the program in full screen or absolutely ruin the ultimate gaming experience by running the program in the window mode. Albeit, make sure to reserve at least 1450 pixels for width, or reduce the prompts font size in the code, as the prompts will not fit the screen and the corresponding error will popup.

Welcome screen below:

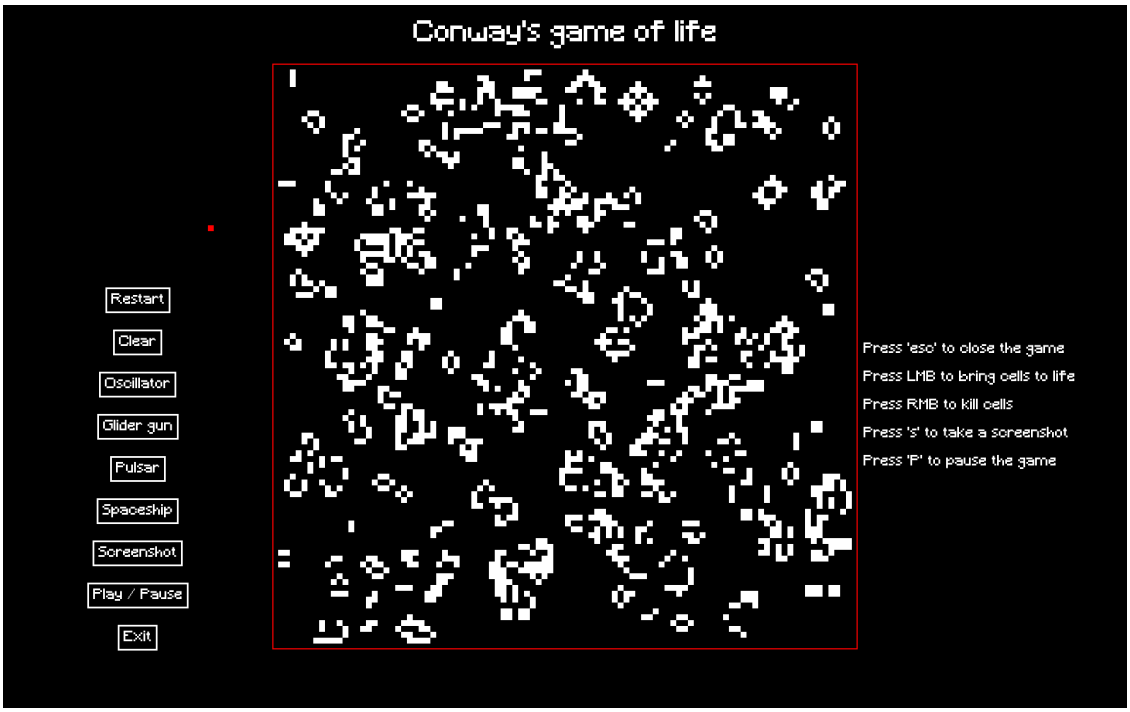


Example of inappropriate resolution below:



Based on the resolution, the program will create an appropriate playing field with red boundaries, initialize the board randomly the same way as above, but from a separate board_script.py script, display the cursor as a red dot, the interactive menu on the left-hand side and some prompts on the right-hand side

Main screen of the game below:

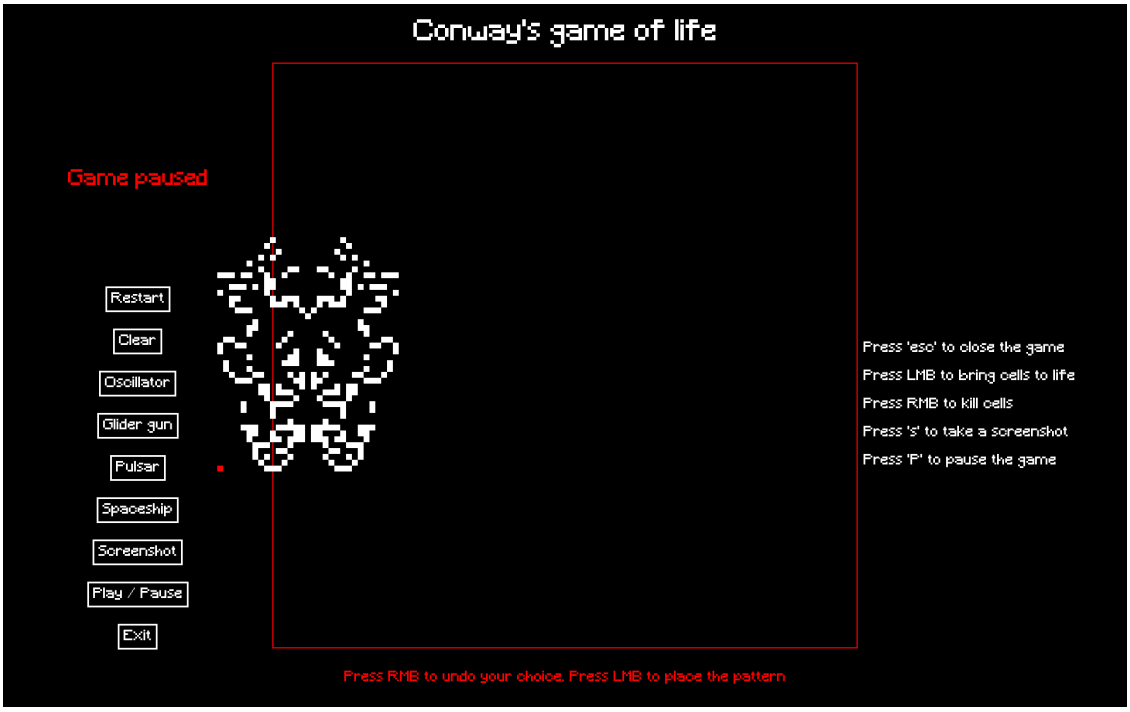


Menu and prompts are quite self-explanatory. User can interact with the game by clicking the left mouse button to bring cells to life and the right mouse button to kill cells within the playing field.

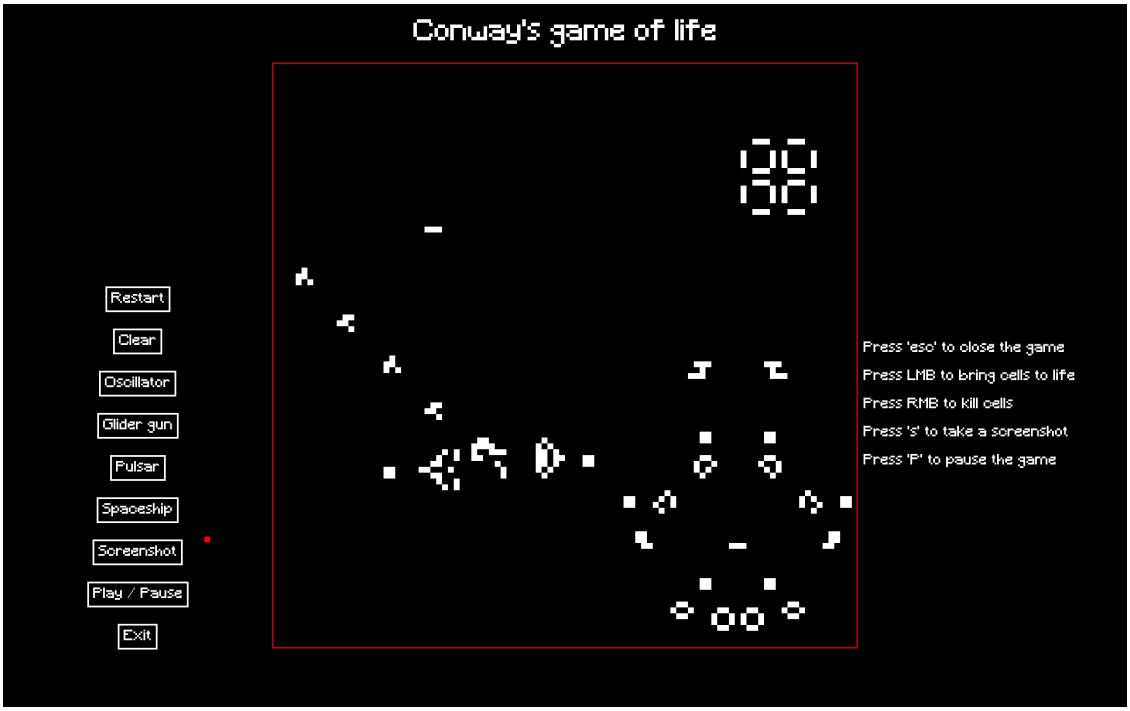
Note, that attempting to place a cell in dead space if the game is not paused will not yield any visible effect as it will die straight away, however placing another cell adjacent to static patterns can produce interesting effects. Menu options: 1. The game can be restarted by clicking the 'Restart' button in the menu. 2. The board can be cleared by clicking the 'Clear' button in the menu. Clearing the playing field is recommended if user wants to try out the patterns below. 3. The user can draw a bar oscillator by clicking the 'Oscillator' button in the menu. 4. The user can draw a glider gun by clicking the 'Draw gun' button in the menu. 5. The user can draw a pulsar by clicking the 'Pulsar' button in the menu. 6. The user can draw a spaceship by clicking the 'Spaceship' button in the menu 7. The user can take a screenshot of the game by clicking the 'Screenshot' button in the menu. The screenshot will be saved in the same directory as this file. Additionally, an imshow matplotlib plot of the current board state will be saved in the same directory as this file. 8. The user can pause updating the board. At this point user can safely add living cells or whole patterns to the board 9. The user can exit the game by clicking the 'Exit' button in the menu or pressing esc button on the keyboard. A dialog box will appear to confirm if the user wants to exit the game.

The game will also be closed if the 'x' button is clicked.

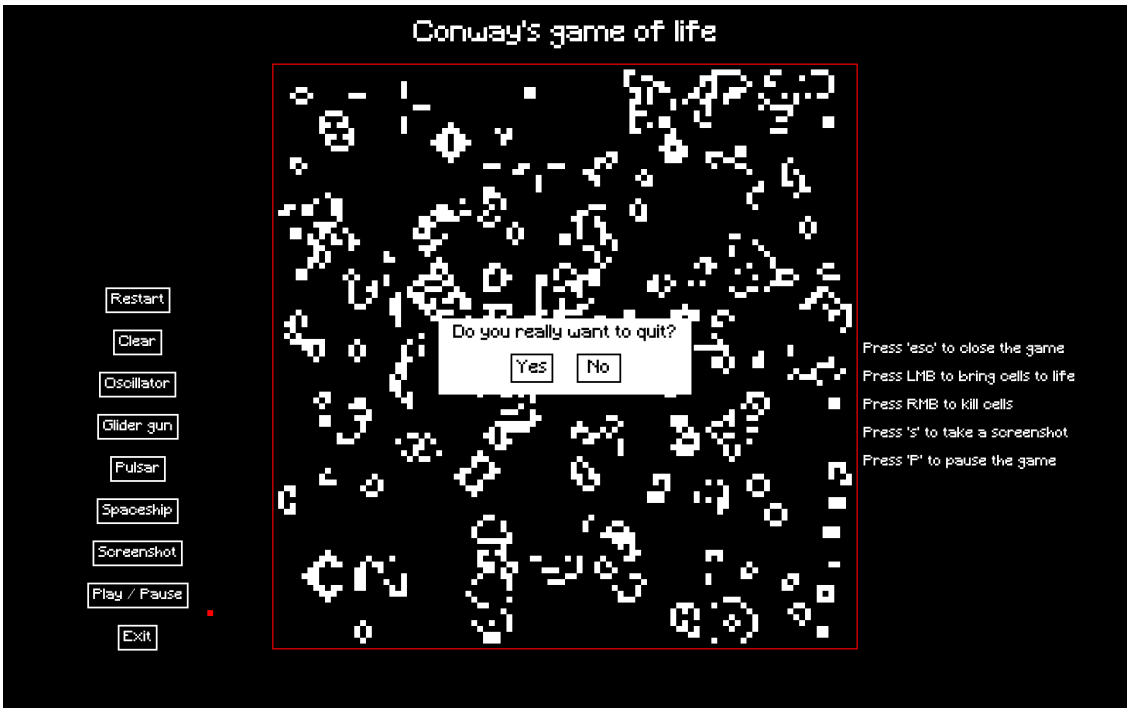
When choosing the patterns, their mockup will be drawn at the cursor position and placed in playing field once LMB is clicked within its boundaries



Examples of using menu options below (note that some screenshots are from earlier versions of the program):



Attempting to exit the game pops up a corresponding dialog window:



User can safely add living cells and whole patterns, while the game is paused:

These interesting patterns are initialized in supplementary scripts. In order to use more patterns in the game one would implement the following two easy steps: 1. create such a script file with an initialization of the corresponding pattern. The pattern can be given as strings of dots and 0 (i.e. taken from https://conwaylife.com/wiki/, see example below) or in form of a numpy array itself. The script must contain the function create_pattern() which will return the pattern (str or np.array). If the pattern is str, the pattern_from_str() function of the main program will convert it to an array anyway. 2. add an additional entry in the Menu dictionary, analogous to other patterns.

After that the option should display in the menu in the game and be able to load just like other patterns.

Example of a str type pattern of a Soba spaceship from https://conwaylife.com/wiki/Soba below:

```
pattern = """
........OOO.........OOO........
.......O...O.......O...O.......
......O.O...O.....O...O.O......
......O...OO.......OO...O......
......OOO...O.....O...OOO......
.....O...OO.OOO.OOO.OO...O.....
.....OO..O...OO.OO...O..OO.....
....OOO...OOOOO.OOOOO...OOO....
............O.....O............
.........O...........O.........
....O....O...........O....O....
....O....OOOO.....OOOO....O....
........O....O...O....O........
.......OO.OOO.....OOO.OO.......
.......OO.O...O.O...O.OO.......
...OOO.......OO.OO.......OOO...
..O...OO....O.O.O.O....OO...O..
.O...O...................O...O.
.O.........OOO...OOO.........O.
.....O......OO...OO......O.....
O............O...O............O
O..OO.....O.........O.....OO..O
.OO........O.......O........OO.
.....O......O.....O......O.....
.....OO.................OO.....
......O.................O......
...............O...............
...OOO........O.O........OOO...
..O......OOO.O...O.OOO......O..
..OO.....O.OOO...OOO.O.....OO..
O.......O.............O.......O
..OOO.O.OO...........OO.O.OOO..
........OO...........OO........
OOO.OO..O..O.......O..O..OO.OOO
.......O....OO...OO....O.......
.....O...................O.....
.......O.O...........O.O.......
..........O.........O..........
........O.............O........
.........O...........O.........
"""
```

The Menu dictionary is given below. First element of the list for each key is used to render the corresponding button on the display and is set to None by default. The second entry defines if the button is clicked (and therefore remains clicked, until its function is done or selection is undone by RMB). The third entry contains the name of the corresponding script (where applicable) and should be accurate.

[ ]:
```
# MENU DICTIONARY #
self.Menu = {'Restart': [None, False],
        'Clear': [None, False],
        'Oscillator': [None, False, "bar_oscillator_script"],
        'Glider gun': [None, False, "glider_gun_script"],
        'Pulsar': [None, False, "pulsar_script"],
        'Spaceship': [None, False, "soba_script"],
        'Screenshot': [None, False],
        'Play / Pause':[None, False],
        'Exit': [None, False],
        }
```

The functions to convert the pattern from str to an array and draw the pattern are given below:

[ ]:
```
def pattern_from_str(self, pattern_str):
    """
    Convert the pattern string to a numpy array.
    Args:
        pattern_str (str): the pattern string
    Returns:
        pattern_array (numpy.ndarray): the pattern as a numpy array
    """
    # Split the pattern string into lines and remove empty lines
    pattern_lines = pattern_str.strip().split('\n')
    for i in range(len(pattern_lines)):
        pattern_lines[i] = pattern_lines[i].strip()  # Remove leading and␣
 ↪trailing whitespaces in each line
    # Determine the size of the pattern
    rows = len(pattern_lines)
    cols = len(pattern_lines[0])

    # Create a numpy array to store the pattern
    pattern_array = np.zeros((rows, cols), dtype=int)

    # Convert the pattern string to a numpy array
    for i, line in enumerate(pattern_lines):  # Loop through the pattern lines
        for j, char in enumerate(line):  # Loop through the characters in the␣
 ↪line
            if char == 'O':  # Check if the character is 'O'
                pattern_array[i, j] = 1  # Set the corresponding element in the␣
 ↪numpy array to 1
```

```
    return np.rot90(pattern_array)

# draw the pattern on the board

def draw_pattern(self):
    """
    Checks if any pattern was selected from the menu and draws it.
    The pattern is drawn by calling the create_pattern function from the␣
 ↪corresponding script.
    The pattern is displayed near the mouse position and can be placed on the␣
 ↪playing field by clicking the left mouse
    button within the playing field.
    The choice can be undone by clicking the right mouse button.
    The pattern is placed on the board when the left mouse button is clicked␣
 ↪within the board borders.
    """
    restricted = ['Clear', 'Restart', 'Exit', 'Screenshot',
                  'Play / Pause']  # List of commands that are not patterns
    for key in self.Menu.keys():  # Loop through the Menu dictionary keys
        if self.Menu[key][
            1] and key not in restricted:  # Check if any pattern is selected␣
 ↪(i.e. second element in the list is True)
            script = importlib.import_module(self.Menu[key][
                                                 2])  # import the␣
 ↪corresponding script from the Menu dictionary (using the third element in␣
 ↪the list)

            pattern = script.create_pattern()  # Call the create_pattern␣
 ↪function from the script

            # Check if the pattern is given as string lines
            if type(pattern) == str:
                pattern = self.pattern_from_str(pattern)  # convert the pattern␣
 ↪string to a numpy array

            # Display the pattern mockup near the mouse position
            pos = pygame.mouse.get_pos()
            for i in range(0, len(pattern)):
                for j in range(0, len(pattern[0])):
                    if pattern[i][j] == 1:
                        pygame.draw.rect(screen, WHITE,
                                         (pos[0] + i * cell_size, pos[1] - j␣
 ↪* self.cell_size, self.cell_size,
                                         self.cell_size))

            undo_prompt = Objects.Objects(text_color="RED",
```

```
                                                font_size=24)  # load the undo␣
↪prompt class from the archive_objects.py with specific options
            undo_prompt.draw_text_box(screen, "Press RMB to undo your choice.␣
↪Press LMB to place the pattern",
                                      WIDTH // 2, HEIGHT - 50, align='center',
                                      frame_width=-1)  # Display the undo␣
↪prompt
            # Place the pattern on the board
            if border.collidepoint(pygame.mouse.get_pos()):  # Check if the␣
↪mouse is within the board
                if self.LMB:  # Check if the left mouse button is pressed

                    # convert the mouse position to the board coordinates
                    x, y = pos
                    x = (x - zero_x) // cell_size
                    y = (y - zero_y) // cell_size

                    # place the pattern on the board
                    for i in range(0, len(pattern)):
                        for j in range(0, len(pattern[0])):
                            self.board[x + i, y - j, states] = pattern[i][j]

            if self.RMB:  # Check if the right mouse button is pressed
                self.Menu[key][1] = False  # Undo the choice
```

Please see the whole code if further interested