Import Libraries

Python

```
import pygame

import numpy as np
```

- import pygame: This line imports the Pygame library, which is a set of Python modules designed for writing video games. It provides functionalities for graphics, sound, and user input.[1]
- import numpy as np: This line imports the NumPy library, which is essential for numerical operations in Python. Here, it's used to efficiently create and manipulate the grid for the Game of Life due to its fast array operations.

Pygame Initialization

Python

```
pygame.init()
```

- pygame.init(): This function initializes all the Pygame modules required to get everything running. It's usually the first Pygame function you call in your program.

Grid Configuration

Python

```
# Grid config

ROWS, COLS = 50, 50

button_height = 50

fullscreen = False

screen_width, screen_height = 800, 850
```

- ROWS, COLS = 50, 50: These variables define the dimensions of the Game of Life grid. The grid will have 50 rows and 50 columns.
- button_height = 50: This sets the height in pixels reserved for the control buttons at the top of the display.
- fullscreen = False: A boolean flag to track the current fullscreen status of the display. Initially set to False.

- screen_width, screen_height = 800, 850: These set the initial width and height of the Pygame display window in pixels. The height includes space for the buttons.

Display Setup

Python

```
# Display setup
screen = pygame.display.set_mode((screen_width, screen_height), pygame.RESIZABLE)
pygame.display.set_caption("Conway's Game of Life")
```

- screen = pygame.display.set_mode((screen_width, screen_height), pygame.RESIZABLE): This creates the Pygame display surface (the window where everything will be drawn).
    - (screen_width, screen_height): Specifies the initial dimensions of the window.
    - pygame.RESIZABLE: This flag allows the user to resize the window, which is handled in the event loop.
- pygame.display.set_caption("Conway's Game of Life"): This sets the title that appears in the window's title bar.

Colors

Python

```
# Colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GRAY = (30, 30, 30)
BLUE = (100, 149, 237)
YELLOW = (255, 255, 0)
RED = (200, 50, 50)
GREEN = (0, 200, 0)
```

- These lines define RGB (Red, Green, Blue) color tuples that will be used for drawing various elements on the screen, such as the background, grid cells, and buttons.

Framerate Control

Python

```
# Framerate control
framerates = [5, 10, 15, 30]
current_framerate_index = 1  # Default to 10 FPS
```

- framerates = [5, 10, 15, 30]: A list of available frames per second (FPS) options for the simulation speed.
- current_framerate_index = 1: Initializes the index to 1, meaning the default framerate will be framerates[1], which is 10 FPS.

Initialize Grid

Python

```
# Initialize grid
grid = np.zeros((ROWS, COLS), dtype=int)
```

- grid = np.zeros((ROWS, COLS), dtype=int): This creates a 2D NumPy array representing the Game of Life grid.
    - np.zeros((ROWS, COLS)): Initializes all cells in the grid to 0.
    - dtype=int: Specifies that the elements of the array should be integers (0 for dead, 1 for alive).

Preset Patterns

These functions define various common patterns in Conway's Game of Life. They take the grid, a row, and a column as input and set specific cells to 1 (alive) to create the pattern at that location.

Python

```
def place_glider(grid, r, c):
    for dr, dc in [(0, 1), (1, 2), (2, 0), (2, 1), (2, 2)]:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1
```

```
def place_blinker(grid, r, c):
    for dr, dc in [(0, 0), (0, 1), (0, 2)]:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_gosper_gun(grid, r, c):
    coords = [
        (5, 1), (5, 2), (6, 1), (6, 2),
        (5, 11), (6, 11), (7, 11), (4, 12), (8, 12),
        (3, 13), (9, 13), (3, 14), (9, 14), (6, 15),
        (4, 16), (8, 16), (5, 17), (6, 17), (7, 17), (6, 18),
        (3, 21), (4, 21), (5, 21), (3, 22), (4, 22), (5, 22),
        (2, 23), (6, 23), (1, 25), (2, 25), (6, 25), (7, 25),
        (3, 35), (4, 35), (3, 36), (4, 36)
    ]
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_toad(grid, r, c):
    # 2 rows, period 2
    coords = [(0,1), (0,2), (0,3), (1,0), (1,1), (1,2)]
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_beacon(grid, r, c):
    coords = [(0,0), (0,1), (1,0), (1,1), (2,2), (2,3), (3,2), (3,3)]
```

```
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_pulsar(grid, r, c):
    # Period 3 oscillator, centered at r, c
    base = [
        (2,4), (2,5), (2,6), (2,10), (2,11), (2,12),
        (7,4), (7,5), (7,6), (7,10), (7,11), (7,12),
        (4,2), (5,2), (6,2), (10,2), (11,2), (12,2),
        (4,7), (5,7), (6,7), (10,7), (11,7), (12,7),
        (4,9), (5,9), (6,9), (10,9), (11,9), (12,9),
        (4,14), (5,14), (6,14), (10,14), (11,14), (12,14),
        (14,4), (14,5), (14,6), (14,10), (14,11), (14,12),
        (9,2), (9,7), (9,9), (9,14)
    ]
    for dr, dc in base:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_lwss(grid, r, c):
    # Lightweight spaceship, moves right
    coords = [(0,1), (0,4), (1,0), (2,0), (3,0), (3,4), (4,0), (4,1), (4,2), (4,3)]
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_and_gate(grid, r, c):
```

```
    # Simplified glider-based AND gate demo (not functional unless input gliders stream in correctly)
    coords = [
        (0, 0), (0, 1), (1, 0), (1, 1),  # Static block
        (4, 4), (4, 5), (5, 4), (5, 5),  # Static block
        (2, 10), (3, 11), (4, 9), (4, 10), (4, 11)  # Glider toward blocks
    ]
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_or_gate(grid, r, c):
    coords = [
        (0, 0), (0, 1), (1, 0), (1, 1),
        (0, 8), (1, 8), (1, 9), (2, 9),
        (3, 11), (4, 11), (4, 12), (5, 12)  # two gliders merging
    ]
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
            grid[r + dr][c + dc] = 1

def place_not_gate(grid, r, c):
    coords = [
        (0, 0), (0, 1), (1, 0), (1, 1),  # Block (acts as input)
        (4, 3), (5, 4), (6, 2), (6, 3), (6, 4)  # Glider that hits block
    ]
    for dr, dc in coords:
        if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:
```

```
            grid[r + dr][c + dc] = 1
```

- Each place_* function iterates through a list of relative coordinates (dr, dc) to define the shape of the pattern.
- if 0 <= r + dr < ROWS and 0 <= c + dc < COLS:: This crucial check ensures that the pattern is placed within the grid boundaries, preventing errors if a pattern extends beyond the edges.
- grid[r + dr][c + dc] = 1: Sets the cell at the calculated position to alive.

Presets Dictionary

Python

```
presets = {
    "Start/Stop": None,
    "Reset": None,
    "Framerate": None,
    "Glider": place_glider,
    "Blinker": place_blinker,
    "Gosper Gun": place_gosper_gun,
    "Toad": place_toad,
    "Beacon": place_beacon,
    "Pulsar": place_pulsar,
    "LWSS": place_lwss,
    "AND Gate": place_and_gate,
    "OR Gate": place_or_gate,
    "NOT Gate": place_not_gate,
}
```

- This dictionary maps button names (strings) to their corresponding functions.
- "Start/Stop", "Reset", and "Framerate" have None as their value because their actions are handled directly in the event loop rather than by calling a separate pattern-placement function.

- Other entries like "Glider", "Blinker", etc., point to the functions defined above, allowing the program to call the correct function when a specific preset button is clicked.

Game State Variables

Python

```
selected_preset = None
running_sim = False
clock = pygame.time.Clock()
```

- selected_preset = None: Keeps track of which preset button, if any, is currently selected by the user.
- running_sim = False: A boolean flag that controls whether the Game of Life simulation is actively running (updating generations). Initially set to False.
- clock = pygame.time.Clock(): Creates a Pygame Clock object, which is used to control the framerate of the simulation, ensuring it doesn't run too fast.

Fullscreen Toggle Function

Python

```
def toggle_fullscreen():
    global fullscreen, screen, screen_width, screen_height
    fullscreen = not fullscreen
    if fullscreen:
        screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
    else:
        screen_width, screen_height = 800, 850
        screen = pygame.display.set_mode((screen_width, screen_height), pygame.RESIZABLE)
    screen_width, screen_height = screen.get_size()
```

- global fullscreen, screen, screen_width, screen_height: Declares that these variables, when modified inside this function, refer to the global variables defined outside the function, not local ones.

- fullscreen = not fullscreen: Toggles the fullscreen flag.
- if fullscreen:: If entering fullscreen mode, it sets the display mode to pygame.FULLSCREEN which typically uses the highest available resolution. (0, 0) as dimensions in fullscreen mode often means "use current desktop resolution".
- else:: If exiting fullscreen, it reverts the display to the initial windowed size (800, 850) and makes it pygame.RESIZABLE again.
- screen_width, screen_height = screen.get_size(): Updates the screen_width and screen_height variables to reflect the actual current dimensions of the display surface, which is crucial for correct drawing calculations, especially after resizing or toggling fullscreen.

Drawing Function

Python

```
def draw(win, grid):
    cell_w = screen_width // COLS
    cell_h = (screen_height - button_height) // ROWS

    win.fill(BLACK)

    # Draw buttons
    bw = screen_width // len(presets)
    for i, name in enumerate(presets):
        rect = pygame.Rect(i * bw, 0, bw, button_height)
        color = RED if name == "Reset" else GREEN if name == "Start/Stop" and running_sim else BLUE
        if selected_preset == name:
            color = YELLOW
        pygame.draw.rect(win, color, rect)
        font = pygame.font.SysFont(None, 24)
```

```
        label = f"{name} ({framerates[current_framerate_index]}fps)" if name == "Framerate" else name
        text = font.render(label, True, BLACK)
        win.blit(text, text.get_rect(center=rect.center))



    # Draw grid
    for row in range(ROWS):
        for col in range(COLS):
            color = WHITE if grid[row][col] == 1 else GRAY
            pygame.draw.rect(
                win,
                color,
                pygame.Rect(
                    col * cell_w,
                    row * cell_h + button_height,
                    cell_w - 1,
                    cell_h - 1
                )
            )
    pygame.display.flip()
```

- def draw(win, grid):: This function is responsible for drawing everything on the screen.
    - win: The Pygame display surface (our screen object).
    - grid: The current state of the Game of Life grid.
- cell_w = screen_width // COLS and cell_h = (screen_height - button_height) // ROWS: Calculate the width and height of each individual grid cell based on the current screen dimensions and the number of rows/columns. Note that button_height is subtracted from screen_height to allocate space for the grid below the buttons.

- win.fill(BLACK): Fills the entire screen with black, effectively clearing the previous frame.
- # Draw buttons: This section iterates through the presets dictionary to draw each button.
    - bw = screen_width // len(presets): Calculates the width of each button, dividing the screen width equally among the buttons.
    - rect = pygame.Rect(i * bw, 0, bw, button_height): Creates a Rect object for each button, defining its position and size.
    - color = ...: Sets the button color. "Reset" is red, "Start/Stop" is green if the simulation is running, otherwise blue. If a preset is selected_preset, it turns yellow.
    - pygame.draw.rect(win, color, rect): Draws the rectangular button on the screen.
    - font = pygame.font.SysFont(None, 24): Creates a font object for rendering text on the buttons.
    - label = ...: Generates the text label for the button. For "Framerate", it includes the current FPS.
    - text = font.render(label, True, BLACK): Renders the text onto a new surface. True enables anti-aliasing for smoother text.
    - win.blit(text, text.get_rect(center=rect.center)): Draws the text surface onto the button, centering it within the button's rectangle.
- # Draw grid: This section iterates through each cell in the grid to draw it.
    - color = WHITE if grid[row][col] == 1 else GRAY: Sets the cell color: white if alive (1), gray if dead (0).
    - pygame.draw.rect(...): Draws each cell as a small rectangle.
        - col * cell_w, row * cell_h + button_height: Calculate the top-left coordinates of the cell on the screen. button_height is added to row * cell_h to offset the grid below the buttons.
        - cell_w - 1, cell_h - 1: Draws the cell with a 1-pixel border (by subtracting 1 from width and height) to create a visible grid effect.
- pygame.display.flip(): Updates the entire screen to show what has been drawn. This is essential to make the changes visible.

Neighbor Calculation Function

Python

```
def get_neighbors(grid, row, col):
    return sum(grid[(row + dr) % ROWS][(col + dc) % COLS]
               for dr in (-1, 0, 1) for dc in (-1, 0, 1)
               if not (dr == 0 and dc == 0))
```

- def get_neighbors(grid, row, col):: This function calculates the number of live neighbors for a given cell at (row, col).
- sum(...): Sums up the values of the neighbor cells. Since live cells are 1 and dead cells are 0, the sum directly gives the count of live neighbors.
- for dr in (-1, 0, 1) for dc in (-1, 0, 1): This nested loop iterates through all 8 surrounding cells (and the cell itself). dr and dc are "delta row" and "delta column" respectively.
- if not (dr == 0 and dc == 0): This condition excludes the current cell itself from the neighbor count.
- grid[(row + dr) % ROWS][(col + dc) % COLS]: This uses the modulo operator (%) for **toroidal wrapping**. This means if a cell is at the edge of the grid, its neighbors "wrap around" to the opposite side, simulating an infinitely connected grid. For example, if row + dr is -1, (-1) % ROWS will give ROWS - 1, effectively wrapping to the last row.

Grid Update Function

Python

```
def update(grid):
    new_grid = np.copy(grid)
    for r in range(ROWS):
        for c in range(COLS):
            n = get_neighbors(grid, r, c)
            if grid[r][c] == 1 and (n < 2 or n > 3):
                new_grid[r][c] = 0
            elif grid[r][c] == 0 and n == 3:
```

```
            new_grid[r][c] = 1
    return new_grid
```

- def update(grid):: This function implements the core rules of Conway's Game of Life.
- new_grid = np.copy(grid): Creates a **copy** of the current grid. It's crucial to work with a copy because all cell updates in a generation must be based on the *previous* generation's state. Modifying the grid in place would lead to incorrect calculations for subsequent cells in the same generation.
- for r in range(ROWS): for c in range(COLS):: Iterates through every cell in the grid.
- n = get_neighbors(grid, r, c): Gets the number of live neighbors for the current cell.
- **Conway's Game of Life Rules:**
    - if grid[r][c] == 1 and (n < 2 or n > 3): new_grid[r][c] = 0:
        - **Underpopulation:** A live cell with fewer than two live neighbors dies.
        - **Overpopulation:** A live cell with more than three live neighbors dies.
    - elif grid[r][c] == 0 and n == 3: new_grid[r][c] = 1:
        - **Reproduction:** A dead cell with exactly three live neighbors becomes a live cell.
    - Implicitly, if a live cell has 2 or 3 neighbors, it lives on to the next generation. If a dead cell has anything other than 3 neighbors, it remains dead.
- return new_grid: Returns the newly calculated grid for the next generation.

Main Loop

Python

```
# Main loop
running = True
while running:
    clock.tick(framerates[current_framerate_index])
    cell_w = screen_width // COLS
    cell_h = (screen_height - button_height) // ROWS
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_f:
                toggle_fullscreen()

        elif event.type == pygame.MOUSEBUTTONDOWN:
            x, y = pygame.mouse.get_pos()

            # Check if a button was clicked
            if y < button_height:
                idx = x // (screen_width // len(presets))
                btn_name = list(presets.keys())[idx]
                if btn_name == "Start/Stop":
                    running_sim = not running_sim
                elif btn_name == "Reset":
                    grid = np.zeros((ROWS, COLS), dtype=int)
                    running_sim = False
                    selected_preset = None  # Also clear preset selection
                elif btn_name == "Framerate":
                    current_framerate_index = (current_framerate_index + 1) % len(framerates)
                else:
                    selected_preset = btn_name

            else:
                # Clicked on the grid
```

```
            col = x // cell_w
            row = (y - button_height) // cell_h
            if selected_preset in presets and presets[selected_preset]:
                presets[selected_preset](grid, row, col)
            else:
                grid[row][col] = 1 - grid[row][col]

    if running_sim:
        grid = update(grid)

    draw(screen, grid)

pygame.quit()
```

- running = True: A boolean flag that keeps the main game loop running as long as it's True.
- while running:: The main game loop. Everything inside this loop runs continuously until running becomes False.
- clock.tick(framerates[current_framerate_index]): This line limits the frame rate of the simulation to the currently selected FPS. It ensures the game doesn't run faster than intended on powerful machines.
- cell_w = screen_width // COLS and cell_h = (screen_height - button_height) // ROWS: These lines recalculate cell dimensions in each iteration. This is important for when the window is resized, as screen_width and screen_height would have updated via pygame.event.VIDEORESIZE (though implicitly handled by toggle_fullscreen here).
- for event in pygame.event.get():: This loop processes all pending events (like keyboard presses, mouse clicks, window closing) that have occurred since the last frame.
    - if event.type == pygame.QUIT:: If the user clicks the 'X' button to close the window, the running flag is set to False, exiting the main loop.
    - elif event.type == pygame.KEYDOWN:: Checks if a key was pressed.

        - if event.key == pygame.K_f:: If the 'F' key is pressed, it calls toggle_fullscreen() to switch between windowed and fullscreen modes.
    - elif event.type == pygame.MOUSEBUTTONDOWN:: Checks if a mouse button was clicked.
        - x, y = pygame.mouse.get_pos(): Gets the current mouse cursor's coordinates.
        - if y < button_height:: Checks if the click occurred within the button area at the top of the screen.
            - idx = x // (screen_width // len(presets)): Determines which button was clicked based on the x-coordinate and button width.
            - btn_name = list(presets.keys())[idx]: Gets the name of the clicked button from the presets dictionary keys.
            - if btn_name == "Start/Stop": running_sim = not running_sim: Toggles the simulation's running state.
            - elif btn_name == "Reset": ...: Resets the grid to all dead cells, stops the simulation, and clears any selected preset.
            - elif btn_name == "Framerate": ...: Cycles through the available framerates. The modulo operator ensures it wraps back to the beginning of the framerates list.
            - else: selected_preset = btn_name: If any other preset button (like "Glider", "Blinker") is clicked, it sets selected_preset to that button's name.
        - else:: If the click was not on a button, it means it was on the grid area.
            - col = x // cell_w and row = (y - button_height) // cell_h: Calculates the grid col and row where the click occurred. button_height is subtracted from y to correctly map the click from screen coordinates to grid coordinates.
            - if selected_preset in presets and presets[selected_preset]:: If a preset pattern is currently selected (and it's a function, not None), it calls that preset function to place the pattern at the clicked (row, col).

        - else: grid[row][col] = 1 - grid[row][col]: If no preset is selected, it simply toggles the state of the clicked cell (alive to dead, or dead to alive).

- if running_sim: grid = update(grid): If the running_sim flag is True, it means the simulation is active, so the update function is called to calculate the next generation of the grid.
- draw(screen, grid): Calls the draw function to render the current state of the grid and buttons on the screen.
- pygame.quit(): After the main loop finishes (when running becomes False), this function uninitializes all Pygame modules, cleaning up resources before the program exits.